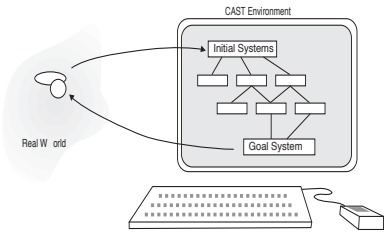

Intelligent unterstütztes systemtheoretisches Problemlösen

Dissertation

zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften

eingereicht von

Rudolf Mittelmann

angefertigt am
Institut für Systemwissenschaften,
Abteilung für Systemtheorie und Informationstechnik
der Technisch-Naturwissenschaftlichen Fakultät
der Johannes Kepler Universität

Linz, Dezember 1995

o. Univ.-Prof. Ing. Dr. Franz Pichler	1. Gutachter
Hon.-Prof. Dr. Heinz Schwärtzel	2. Gutachter

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubter Hilfsmittel bedient habe.

Linz, am 1. Dez. 1995 *Rudolf Mittelmann*

Kurzfassung:

Nach einer kompakten Darstellung der Systemtheorie als Wissenschaft werden die Besonderheiten systemtheoretischen Problemlösens erläutert. Um zu einer klaren Sichtweise zu gelangen, werden wichtige Begriffe wie System, Systemtyp, Systemtransformation, Systemproblem, u.a. formalisiert. Die formale Grundlage einer CAST (Computer Aided Systems Theory) Umgebung wird in einer neuen Form vorgestellt.
Als möglicher Weg zur Automatisierung des Systemproblemlösens wird STA eingeführt, eine Maschine, die Systemalgorithmen ausführen kann. In weiterer Konsequenz wird eine Art Expertensystem STX für das intelligent unterstützte Problemlösen vorgeschlagen. Dabei sind die Besonderheiten des Problemlösens im Bereich der Systemtheorie zu berücksichtigen. Architekturvarianten für ein solches STX werden eingeführt und gegenübergestellt. Implementierte Beispiele für Anwendungen von logikorientierten und produktionsregelbasierten Systemen werden beschrieben.

Stichworte:

Systemtheorie, Systemtypen, mathematische Systembeschreibungen, Systemtransformationen, Systemalgorithmen, automatisiertes Problemlösen, regelbasierte Programme.

Intelligent Support for Systems Theory Instrumented Problem Solving

Abstract:

The field of Systems Theory is presented as a science with a special approach to problem solving. The important basic notions of system, system type, system transformation, systems problem etc. are introduced as mathematical constructions. The introductional part of the work is completed by a new description of the principal construction of a CAST (Computer Aided Systems Theory) environment.
The development of a more automated way of systems problem solving starts with the introduction of STA, a system which stores and applies system algorithms to given initial systems. In consequence, this leads to a kind of expert system for intelligent support of problem solving (STX). To be considered are the special aspects of problem solving in the context of systems sciences, where the objects are mathematically defined exact systems. Several competing variants of architectures for STX are introduced and compared. Implemented prototypes of logic-oriented and rulebased systems and applications to example problems are shown.

Keywords:

Systems Theory, system types, mathematical system descriptions, system transformations, system algorithms, automated problem solving, rule based systems.

Dank:

Bei Professor F. Pichler möchte ich mich ganz besonders für die ursprüngliche Idee der STIPS-Maschine und viele weitere Anregungen, die zu dieser Arbeit führten, bedanken, sowie ganz allgemein für die wohlwollende Unterstützung meiner Tätigkeit in diesen Jahren am Institut für Systemwissenschaften. Bei Prof. H. Schwärtzel bedanke ich mich für die ideelle und finanzielle Unterstützung im Rahmen mehrerer Zusammenarbeitsprojekte zwischen der Forschungsabteilung ZTI der Siemens AG, München, und der Abteilung für Systemtheorie und Informationstechnik der Universität Linz.
Ferner möchte ich mich bei allen Kollegen für zahlreiche, anregende Diskussionen bedanken, vor allem bei Dr. Thomas Müller-W. und Alfred Spalt. Die Implementierung der derzeitigen CAST-FSM-Version ist zum großen Teil das Verdienst von Dr. Thomas Müller-W., ohne die die in der vorliegenden Arbeit vorgestellte STX kaum möglich gewesen wäre. (Für weitere Autoren von CAST-FSM siehe Kap. 5.4.)
Schließlich möchte ich mich bei meiner Frau Angelika für die aufgebrachte Geduld bedanken.

Inhaltsverzeichnis:

1. Kapitel - Mathematische Systemtheorie und ihre Anwendung in der Technik 1
1.1. Wissenschaftstheoretische Einordnung der Mathematischen Systemtheorie 1
1.2. Problemlösen im technisch-wissenschaftlichen Bereich 2
1.3. Mathematische Systemtheorie 4
1.3.1. Klassifizieren von gebräuchlichen Modellen 5
1.3.2. Abstrahieren zu mathematischen Systemen 5
1.3.3. Einordnen in ein einheitliches Gebäude von Systemtypen 6
1.3.4. Ausbau der Theorie zu diesen Systemen (Systemtypen) 7
1.3.5. Aufstellen der Theorie der Systemtransformationen 7
1.3.6. Bereitstellen von Methoden für wichtige Systemtransformationen 8
1.3.7. Aufbereiten der Methoden in algorithmische, anwendungsnahe Form 8
1.3.8. Zusammenfassen von Systemtransformationen zu Systemalgorithmen 9
1.4. Anwendung der Systemtheorie in der Technik 10
1.5. Anmerkungen zum 1. Kapitel 10

2. Kapitel - Systemtypen und Systemtransformationen 12
2.1. Mathematische Systeme und Systemtypen 12
2.1.1. Systemtyp Black Box 14
2.1.2. Systemtyp Dynamik 14
2.1.3. Systemtyp Generator 16
2.1.4. Systemtyp Algorithmus 17
2.1.5. Systemtyp Netzwerk 18
2.2. Speziellere Systemtypen und Vererbung 19
2.3. Systemspezifikation 22
2.4. Allgemeine Systemspezifikation 22
2.5. Die RES Notation 23
2.6. Systemtransformationen 26
2.7. Anmerkungen zum 2. Kapitel 28

3. Kapitel - Allgemeine Systemprobleme und Systemalgorithmen 29
3.1. Systemtheorie-unterstütztes Problemlösen 29
3.2. Formalisierung des Begriffs Problem 31
3.3. Systemprobleme 31
3.4. Allgemeine Systemprobleme 32
3.5. Typische Systemprobleme 32
3.5.1. Analyseprobleme 32
3.5.2. Syntheseprobleme 33
3.5.3. Realisierungsprobleme 33
3.6. Systemalgorithmen 33
3.7. Anmerkungen zum 3. Kapitel 35

4. Kapitel - Formale Methodenbanken für Systemtheorie: STIPS 36
4.1. Methodenbanken 36
4.2. Die STIPS-Maschine 36
4.2.1. Abstrakte Beschreibung eines interaktiven Computersystems 38
4.2.2. Definition der STIPS-Maschine 39
4.3. Steuerung der STIPS-Maschine 43
4.4. STIPS-Maschine als Systemtyp 44
4.5. STIPS-Trajektorien und Systemalgorithmen 44
4.6. Anmerkungen zu 4. und den folgenden Kapiteln 44

5. Kapitel - Implementierung einer systemtheoretischen Methodenbank: CAST-FSM 46
5.1. Frühe CAST Methodenbanken 46
5.2. Lisp als Implementierungssprache und als Entwicklungsumgebung 48
5.3. Die Bedeutung von Browsern für interaktive CAST-Umgebungen 48
5.4. CAST-FSM - Ein portierbares CAST System 49
5.4.1. Programmautorenschaft 53
5.4.2. Der Aufbau von CAST-FSM 54
5.4.3. CAST Kernel: Objekte und Methoden der Systemtheorie 56
5.4.4. CAST User Interface 71
5.4.5. CAST-FSM als Anwendungsprogramm 75

6. Kapitel - Automatisierung des Problemlösungsprozesses mit Systemalgorithmen 82
6.1. Algorithmen zur Lösung von Systemproblemen 82
6.2. Die Maschine STA 83
6.3. Erstellung von Systemalgorithmen für STA 83
6.4. Beispiele für fertige Systemalgorithmen 84
6.4.1. pdecomp-realizations 84
6.4.2. sdecomp-realizations 85
6.4.3. shiftreg-realizations 86
6.5. Eine einfache STA 87

7. Kapitel - Wissensbasierte Systeme als Programmierparadigma 89
7.1. Wissensbasierte Methoden 89
7.2. Navigation bei der Lösungssuche im Zustandsraum 92
7.2.1. Symbolische Datenverarbeitung 92
7.2.2. Zusätzliche Elemente wissensbasierter Systeme 106
7.3. Interaktive und heuristische Entscheidungsfindung: STX 108
7.3.1. Prinzipielle Architektur von STX 108
7.3.2. Implementierungsaspekte: Heterogene Programmierung 109
7.3.3. Planung von Systemalgorithmen 110
7.3.4. Ausführung von Systemalgorithmen 113
7.4. Bewertung wissensbasierter Systeme für die Implementierung von STX 113

8. Kapitel - Beispiel einer STX für Automaten und Schaltwerke 115
8.1. Trennung von Daten, Wissensbasis, und Steuerung 115
8.2. Objektorientierte Programmierungstechniken 116
8.2.1. Wichtige objektorientierte Programmiersprachen 116
8.2.2. Vererbungsstrategien 119
8.2.3. Algorithmus für Vererbungsstrukturen 121
8.2.4. Information Hiding 123
8.2.5. Prototyping 125
8.2.6. Zusammengesetzte Objekte 125
8.2.7. Design von User Interface Systemen 127
8.2.8. Einbindung von Systemen in eine Umgebung 129
8.3. Implementierung in Common Lisp 129
8.3.1. Realisierungsgraphen 130
8.3.2. Quantifizierte Ausdrücke als Prädikate 132
8.3.3. Quantifizierte Ausdrücke als Programmstücke 134
8.3.4. Systemprobleme 136
8.3.5. Regeln 138
8.3.6. Inferenzmaschinen 140
8.3.7. STX 141
8.4. Beispielanwendungen 142
8.4.1. Die erste Regelbasis 142
8.4.2. Andere Regelbasen 150

9. Kapitel - Verallgemeinerungen und Ausblick 154
9.1. Unabhängigkeit von einem bestimmten Anwendungsgebiet 154
9.2. CAST-CRYPTO 155
9.2.1. Beispiele für graphische Analysemethoden 155
9.2.2. Eine STX für die Kryptographie 157
9.3. Vertiefungsthemen und offene Probleme 158
9.3.1. Sprachen für Constraints und Bewertungsmetriken 158
9.3.2. Anwendungsnahe Heuristiken 158
9.3.3. Systemspezifikationen und Problemklassen 158

A. Anwendung von CAST-FSM in der Lehre 159
A.1. CAST-Programmierpraktikum 159
A.2. Programmiertechniken der KI 159
A.3. Übungen mit macCASTfsm 160
A.3.1. "Diskrete Strukturen"- und "Systemtheorie"-Übungen 160
A.4. Übungsaufgabensammlung 162
A.4.1. CAST-FSM 162
A.4.2. CAST-CRYPTO 167

B. Literaturangaben 170

Abbildungsverzeichnis

Bild 1.1 Die Modellmethode naturwissenschaftlichen Problemlösens ... 3
Bild 1.2 Modell eines Transistorschalters ... 3
Bild 1.3 Phasendiagramm ... 4
Bild 1.4 Der Graph der prinzipiellen Systemtransformationsarten ... 8
Bild 1.5 CAST als Problemlösungswerkzeug am Arbeitsplatz ... 10
Bild 2.1 System und Systemtyp ... 12
Bild 2.2 Systemtyp "Black Box" ... 14
Bild 2.3 Systemtyp Algorithmus ... 17
Bild 2.4 Ein System vom Typ Netzwerk mit drei Komponenten ... 19
Bild 2.5 Vererbung bei Systemtypen ... 19
Bild 2.6 Vererbungsgraph ... 20
Bild 2.7 Die Funktion ink ... 21
Bild 2.8 Die Funktion typ ... 21
Bild 2.9 RES für ein Automatennetzwerk ... 23
Bild 2.10 Beispiel einer interpretierten RES ... 26
Bild 2.11 Systemtransformationen ... 27
Bild 3.1 Systemmethode des Problemlösens ... 30
Bild 3.2 Systemtheoretisches Problemlösen ... 30
Bild 3.3 Systemproblem P = (I, C, F) ... 32
Bild 4.1 Lösungsgraph ... 37
Bild 4.2 Synthese, Analyse, Projektion und Kombination ... 38
Bild 4.3 Erweiterung des Lösungsgraphen ... 40
Bild 4.4 STIPS-Maschine als Serienschaltung ... 41
Bild 5.1 CAST-FSM - Kernel und Basics ... 51
Bild 5.2 CAST-FSM Anwendungsprogramm für Macintosh Computer ... 52
Bild 5.3 Common Lisp - eine Sprachfamilie für portierbare Programmentwicklungen ... 53
Bild 5.4 Schichtenmodell von CAST-FSM - allgemeines Prinzip ... 54
Bild 5.5 Schichtenmodell von CAST-FSM - Implementierungsbeispiele ... 54
Bild 5.6 Diverse CAST Architekturvarianten ... 56
Bild 5.7 Datenstrukur der Klasse two-into-one-fun ... 63
Bild 5.8 Datenstruktur einer Funktion der Klasse two-into-one-fun-large ... 64
Bild 5.9 Lisp Listener als Kommandozeileninterpreter ... 72
Bild 5.10 Experimentelles CLIM User Interface für CAST-FSM ... 75
Bild 5.11 macCASTfsm - Algebraische Strukturtheorie, Verbände, Parallelzerlegung ... 78
Bild 5.12 macCASTfsm - V-Graphen, Schieberegisterzerlegung ... 79
Bild 5.13 macCASTfsm - Systemalgorithmus-Entwicklung ... 80
Bild 6.1 STA vor Anwendung des Systemalgorithmus Serienzerlegung ... 87
Bild 6.2 Zustand der STA nach Ausführung des Algorithmus ... 88
Bild 6.3 Endzustand der STA ... 88
Bild 7.1 "Herkömmliche" EDV-Programmstrukur ... 89
Bild 7.2 Programmstruktur eines Expertensystems ... 90
Bild 7.3 Abhängigkeit von Anwendungsgebiet und Problem ... 90
Bild 7.4 Test eines eingebetteten Automaten ... 95
Bild 7.5 Vereinfachte Testumgebung mit FSMs g, s, a ... 95
Bild 7.6 Die Regel TPL-1 ... 103
Bild 7.7 Lösungssuche ... 105
Bild 7.8 Bidirektionale Suche im Lösungsraum ... 105
Bild 7.9 STX Architektur (schematisch) ... 109

Bild 8.1. Aufbau von STXFSM 115
Bild 8.2 Systeme vom Typ Generator als semantisches Netz mit ISA-Kanten 120
Bild 8.3 Berechnete Vererbungsstruktur 122
Bild 8.4 Ergebnis der Berechnung "kleine binäre Komponenten" 144
Bild 8.5 Ergebnis von "kleine binäre Komponenten" mit Goal und Constraint 147
Bild 8.6 Ergebnis für "fsm7" 148
Bild 8.7 Schieberegisterdarstellung 149
Bild 8.8 Lösung für "fsm7" mit parallelen Regeln 151
Bild 9.1 Ergebnis einer Analysemethode in CAST-CRYPTO 156
Bild 9.2 Ergebnis einer weiteren Analysemethode mit graphischer Ausgabe 157

Tabelle

Tabelle 5.1 Datentypen für Mengen und binäre Tupel 61

1. Kapitel - Mathematische Systemtheorie und ihre Anwendung in der Technik

1.1. Wissenschaftstheoretische Einordnung der Mathematischen Systemtheorie

Die *Systemtheorie* ist eine Wissenschaft, die allgemeine Zusammenhänge darzustellen versucht und so aufbereiten will, daß die Ergebnisse in den Ingenieurdisziplinen direkt angewendet werden können. In der Systemtheorie sind daher einerseits sehr abstrakte Denkweisen üblich, wie in anderen "Metawissenschaften", etwa der Philosophie. Andererseits soll die Systemtheorie aber immer auch sehr praxisbezogen sein.

Diese Beschreibung der Wissenschaft Systemtheorie trifft allerdings genausogut auf die Mathematik zu. Ein wesentlicher Unterschied zwischen der Mathematik und der Systemtheorie, und somit zwischen mathematischem Problemlösen und systemtheoretischem Problemlösen, liegt in der Komplexität der bearbeiteten Objekte. Mathematische Ansätze zum Problemlösen beziehen sich auf mathematische Objekte, wie Zahlen, Matrizen, Gleichungssysteme, Gruppen, logische Aussagen etc. Um mathematische Methoden anwenden zu können, muß ein Problem aus einer praktischen, konkreten Situation erst entsprechend in die Sprache der Mathematik übersetzt werden. Dieser Vorgang ist im allgemeinen aber alles andere als trivial, vielmehr häufig der schwierigste Schritt des gesamten Prozesses.

In der Systemtheorie wird versucht, ähnlich exakte und allgemeingültige Methoden zu erarbeiten und anderen Fachgebieten zur Verfügung zu stellen, wie in der Mathematik. Jedoch sind die betrachteten Objekte nicht Zahlen, Matrizen, Gruppen, ... sondern komplexere Einheiten, die direkte Abstraktionen realer Objekte darstellen. Insbesondere werden die dem Ingenieur vertrauten Modelle im systemtheoretischen Ansatz direkt verwendet. Diese abstrakten, aber mit Ingenieurmodellen korrespondierenden Objekte werden *Systeme* genannt.

Das gesamte Wissensgebiet Systemtheorie kann untergliedert werden in die Richtungen Allgemeine Systemforschung, Mathematische Systemtheorie, Modellierung und Simulation, und weitere. In der Allgemeinen Systemforschung (GST - General Systems Theory) wird bewußt auf eine direkte Bezugnahme auf bestimmte Anwendungen verzichtet, statt dessen wer-

den nur die generellen Eigenschaften von Systemen und sich daraus ergebende Zusammenhänge dargestellt, um allgemeingültige Problemlösungsstrategien zu erarbeiten. Dazu wird in der GST eine ganzheitliche Sicht bevorzugt. Obwohl die GST [Bert68], [Klir69], [Wymo67] so alt ist wie die Systemtheorie überhaupt, sind ihre Ergebnisse bisher nicht allzu erfolgreich anwendbar gewesen. Andererseits kann sie als Grundlagenforschung angesehen werden und stellt sicher einen wissenschaftstheoretisch wertvollen Baustein im Gebäude der Wissenschaft dar.

In der *Mathematischen Systemtheorie* [Pich75] werden Systeme als mathematische Konstruktionen dargestellt, was eine strenge Behandlung mit formalen Methoden erlaubt. Mathematische Methoden können also hier im Problemlösungsprozeß als Baustein verwendet werden. Die behandelten Objekte und Problemstellungen sind allerdings meist ungeeignet für eine direkte Übertragung in die mathematische Sprache.

Wichtige *systemtheoretische Arbeitsweisen* sind das Überführen von anwendungsbezogenen Modellen in mathematisch definierte Systeme, die *Analyse* des Verhaltens und der Struktur der Systeme, die Berechnung von *Realisierungen* eines Systems mit bestimmten Eigenschaften, die Strukturierung (*Dekomposition*) von komplexen Systemen in Teilsysteme und die *Synthese* von komplexen Systemen aus einfachen Teilen.

Ein Untergebiet ist die Theorie der dynamischen Systeme [Pich83-1]. Hier liegt das Gewicht auf der Veränderlichkeit eines Systems (meist entlang einer Zeitachse, aber auch anderer Koordinaten). Wesentlich ist dafür der Begriff des Zustandes. Die zentrale Bedeutung des Zustandsbegriffes wurde schon in den frühen Arbeiten der Kybernetik erkannt [Ashb56].

Je nach den besonderen mathematischen Eigenschaften der Zeitmenge und der Zustandsmenge ergeben sich spezielle Systemtheorien. Beispiele sind die Lineare Systemtheorie, die Automatentheorie, die Theorie der Differentialsysteme, und andere.

Die Systemtheorie will den Anwendern jedoch nicht einfach einen Katalog von Methoden zur Verfügung stellen, sondern vielmehr einen allgemeingültigen Rahmen erarbeiten, in welchem solche Methoden verwendet werden können. Dazu ist der Begriff Problemlösungsprozeß fundamental. Die konkrete Arbeit eines Ingenieurs an seinem Arbeitsplatz wird als ein Prozeß der Lösung eines gestellten Problems aufgefaßt. Die Systemtheorie versucht diesen Prozeß möglichst weitgehend zu unterstützen.

1.2. Problemlösen im technisch-wissenschaftlichen Bereich

Zunächst soll der Begriff Problem eingegrenzt werden. Im Kontext dieser Arbeit wird ein Problem verstanden als eine Fragestellung an eine materielle oder ideelle Realität, die noch nicht beantwortet ist. Meist wird der Begriff hier noch enger interpretiert werden als eine (ungelöste) Frage im Bereich der Technik. Aber auch in anderen Bereichen kann die Systemtheorie Beiträge liefern, Beispiele sind aus der Ökonomie [Sydo80] bekannt, aus den Sozialwissenschaften [Cava79], weitere in der Ökologie [Sydo86].

Zwei wichtige Klassen von Problemen sind zu unterscheiden: Bei Analyseproblemen wird untersucht, ob bestimmte Eigenschaften vorliegen. Bei Syntheseproblemen wird eine Konstruktion gesucht, die vorgegebene Eigenschaften hat. Die Problemlösung ist dann die Antwort auf die behandelte Frage: Im Falle der Analyseprobleme wird die Lösung durch eine Menge von Eigenschaften angebbar sein, bei Syntheseproblemen dagegen durch eine konkrete Konstruktion.

Der Problemlösungsprozeß ist dann der Vorgang, zu einem gestellten Problem die Lösung zu finden. In diesem Prozeß werden Methoden angewendet und Werkzeuge benutzt. In der Wissenschaft wird meist ein Vorgehen gewählt, das als Modellmethode bezeichnet wird. Der für das Problem relevante Realitätsausschnitt wird auf ein Modell abgebildet (Bild 1.1).

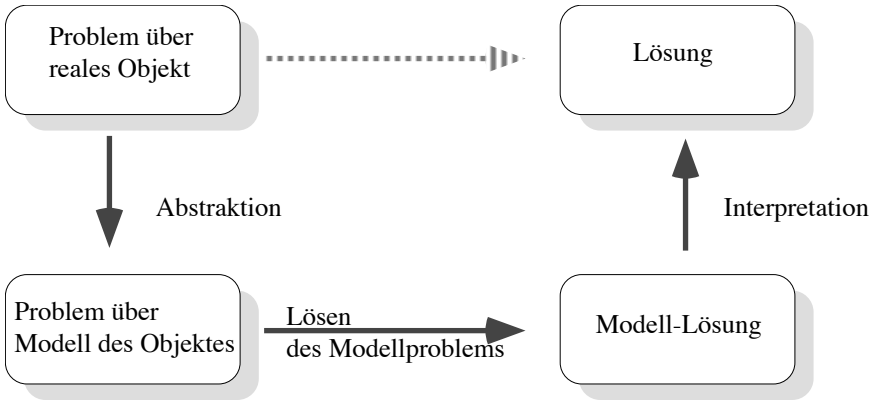

Bild 1.1 Die Modellmethode naturwissenschaftlichen Problemlösens

Das Modell muß so gewählt werden, daß einerseits die Problemfrage ohne Beschränkung der Gültigkeit auf das Modell übertragen werden kann, das Modell muß also reichhaltig genug sein, andererseits soll das Modell von aller für das Problem unwichtigen Komplexität der Realität abstrahiert sein, das heißt es soll so einfach wie möglich sein. Abstraktion von der Realität kann erstens durch weglassen (Ausschnitt bilden) erfolgen und zweitens durch vereinfachen (annähern) und idealisieren. Zum Beispiel ist für viele Fragestellungen in der Informationstechnik eine Transistor-Schaltstufe (Bild 1.2) durch eine einfache Ersatzschaltung ausreichend genau beschrieben [Mösc71].

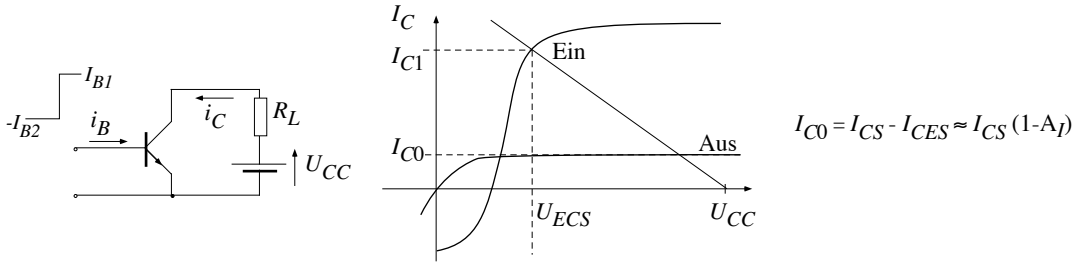

Bild 1.2 Modell eines Transistorschalters

Weggelassen in diesem Modell sind der mechanische Aufbau, die thermischen Effekte, Alterung, sogar das zeitliche Verhalten, und vieles andere. Ein weiteres Beispiel wäre eine Verstärkerstufe modelliert durch einen linearen Vierpol. Die Übertragungsfunktion wird ver-

einfacht zu einer linearen Funktion, obwohl sie sicher nichtlinear ist. Die lineare Charakteristik ist aber für den gewünschten Betrieb eine in weiten Grenzen gute Annäherung.

Nachdem ein passendes Modell gewählt oder konstruiert wurde, wird das ursprüngliche Problem auf das Modell übertragen und versucht, die Lösung für dieses Modell-Problem zu finden. Im technisch-wissenschaftlichen Bereich werden dazu meist mathematische Werkzeuge verwendet werden. Wenn die Lösung des Modell-Problems gefunden wurde, muß diese Modell-Lösung in die Realität abgebildet werden. Anders ausgedrückt, die Lösung des Modell-Problems muß in die Realität rückinterpretiert werden. Nach den einzelnen Problemlösungsschritten, oder nach der Anwendung der Lösung in der Realität, kann sich herausstellen, daß das erreichte Resultat nicht akzeptiert werden kann, und es muß zu einem früheren Schritt zurückverzweigt werden [Schw81]. Eine geeignete detailliertere Darstellung des Problemlösungsprozesses ist das Phasendiagramm (Bild 1.3).

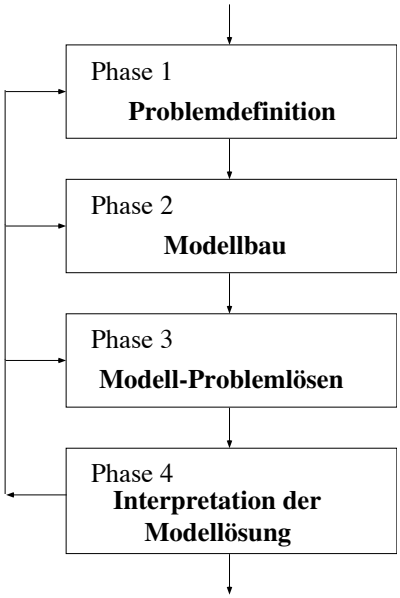

Bild 1.3 Phasendiagramm nach [Pich90-1]

Hier wird der zeitliche Ablauf der Problemlösungsphasen in vertikaler Richtung dargestellt, aber auch die Verzweigungspunkte, an denen Rückführungen in frühere Phasen möglich sind.

Der systemtheoretische Ansatz zum Problemlösen baut auf der Modellmethode auf, indem er dieses allgemeingültige Konzept erweitert.

1.3. Mathematische Systemtheorie

Die Vorgehensweise der Mathematischen Systemtheorie soll nun genauer dargestellt werden. Dieser Abschnitt 1.3. könnte als *8-Punkte-Programm der mathematischen Systemtheorie* bezeichnet werden:

1. Klassifizieren von gebräuchlichen Modellen aus den Anwendungsgebieten
2. Abstrahieren zu mathematischen Systemen
3. Einordnen in ein einheitliches Gebäude von Systemtypen
4. Ausbau der Theorie zu diesen Systemen (Systemtypen)
5. Aufstellen der Theorie der Systemtransformationen
6. Bereitstellen von Methoden für wichtige Systemtransformationen
7. Aufbereiten der Methoden in algorithmische, anwendungsnahe Form
8. Zusammenfassen von Systemtransformationen zu Systemalgorithmen

1.3.1. Klassifizieren von gebräuchlichen Modellen

In den Anwendungsgebieten der Systemtheorie übliche Modelle werden auf Gemeinsamkeiten und Unterschiede untersucht und klassifiziert. In der Informationstechnik wird der Aufbau von Geräten meist durch netzwerkartige Modelle beschrieben, die Module als Zusammenschaltung von kleineren Modulen in hierarchischer Form erlauben. Bauteile und Baugruppen werden in der Analogtechnik durch ihre Dynamik oder ihr Übertragungsverhalten modelliert, in der Digitaltechnik durch schaltalgebraische Formen und Wahrheitstafeln.

1.3.2. Abstrahieren zu mathematischen Systemen

Die anwendungsspezifischen Modelle werden in mathematische Systeme abstrahiert. Ein *mathematisches System* ist dabei eine mathematische Konstruktion, meist ein n-Tupel. Jede Komponente dieses Systems kann ein einfaches mathematisches Objekt sein, wie Menge, Relation, etc. oder auch ein anderes System. Die mathematischen Systeme enthalten die für die Anwendung wichtigen Eigenschaften der Modelle in einer rein formalen Art, und sind damit von jeder konkreten Anwendung völlig unabhängig, wie alle mathematischen Konzepte. Andererseits gibt es meist zu einem abstrakten System eine Interpretation, die eine technischen Systemen nahestehende Semantik angibt.

1.3.3. Einordnen in ein einheitliches Gebäude von Systemtypen

Die Aufstellung einer Ordnung, einer Systematik in einem Wissensgebiet ist ein allgemeines wissenschaftliches Verfahren, um solche Ziele wie einheitliche Sprache, Übersicht, Konsistenz und Zugreifbarkeit auf die Ergebnisse zu ermöglichen. In der Systemtheorie werden die vielfältigen Arten von Systemen deshalb in ein einfaches, aber effektives Schema von *Systemtypen* eingeteilt. Nach Pichler [Pich81] wird hier der folgende Aufbau verwendet: Die Vielfalt aller mathematischen Systemtypen wird in fünf elementare Systemtypen gruppiert. Es sind dies die Typen Black Box (abgekürzt B), Algorithmus (A), Dynamik (D), Generator (G), und Netzwerk (N). Dieser allgemeine Aufbau wird nun in der folgenden Weise konkretisiert:

Der Systemtyp *Black Box* ist der einfachste. Hier wird das System nur durch die Abgrenzung von oder die Wechselwirkung mit der Umgebung dargestellt. Dagegen bleibt die innere Struktur und Funktionsweise offen. Ein einfaches Beispiel für eine Black Box ist eine (mathematische) Funktion. Die Abgrenzung des Systems zur Umgebung ist durch die Angabe der Definitionsmenge und des Wertebereiches gegeben, die Wechselwirkung durch die Abbildungsvorschrift. Wie diese erreicht wird, ist dagegen nicht beschrieben und für dieses System auch ohne Bedeutung.

Ein besonders wichtiger Systemtyp ist der *Generator*. Ein als Generator beschriebenes System ist außer durch Input- und Outputmengen durch sein Verhalten gegeben. Das Verhalten des Systems wird durch einen Zustand und die möglichen Änderungen des Zustandes als Reaktion auf den Input definiert. Diese Beschreibung der Zustandsveränderung ist lokal, in einem zeitlichen, räumlichen oder anderen Sinne. Diese Lokalität der Beschreibung macht es oft möglich, komplexes Systemverhalten kompakt zu definieren. Es wird nur das Verhalten im Kleinen (zu einem Zeitpunkt, an einem Ort, etc.) in generischer Weise angegeben. Das globale Verhalten ist durch das lokale vollständig bestimmt, es ist bei vielen Systemen allerdings nur durch Simulation (Durchrechnung von I/O-Experimenten) berechenbar. Meist wird die Zustandsveränderung entlang der Zeitachse modelliert, dann ist der Zustand zu einem Zeitpunkt zusammen mit dem Input zu diesem Zeitpunkt für die Zustandsüberführung verantwortlich.

Eine *Dynamik* ist eine globale Systembeschreibung, die den zulässigen Inputtrajektorien die entsprechenden Zustands- und Outputtrajektorien zuordnet. Wenn sich ein zu modellierendes System in geschlossener Form als Dynamik darstellen läßt, lassen sich viele Systemprobleme direkt lösen. Häufig ist daher die Berechnung der Dynamik-Form eines vorgegebenen Systems Bestandteil des Problemlösungsprozesses. Systeme vom Typ Generator werden durch eine Form von "Integration" in den Typ Dynamik überführt. Beispiele für Dynamiken sind die Lösungen von Differentialgleichungssystemen, und die Wortüberführungsfunktionen λ^* von linearen Automaten. Die allgemeine Theorie der dynamischen Systeme (Dynamical Systems Theory) hat viele wichtige Resultate erarbeitet und wird auch als die Grundlage für einen systematischen Aufbau der Systemtheorie angesehen.

Der *Algorithmus* ist ein besonderer Systemtyp zur Beschreibung von Systemen, die als Verarbeitungsvorschrift in einer formalen Sprache vorliegen. Natürlich können alle Computerprogramme als Beispiel herangezogen werden. Genauer ist ein Algorithmus eine Arbeitsvorschrift von endlicher Länge, die in exakter, unmißverständlicher Form abgefaßt ist und den gesamten Arbeitsvorgang auf die Anwendung gewisser vorausgesetzter Primitiv-Operatoren zurückführt. Jede Anwendung eines Primitivs heißt dabei ein Schritt. In der klassischen Definition [z.B. Herm61] wird weiter verlangt, daß die Abarbeitung der Vorschrift in jedem

Falle nach endlich vielen Schritten abbricht. Meist läßt man diese Forderung jedoch fallen, da viele für die Anwendung interessante "Algorithmen" nicht in jedem aller möglichen Fälle terminieren.

Die Dekomposition komplexer Systeme in kleinere, einfachere Teilsysteme ist ein Grundprinzip der Wissenschaft schlechthin. Es sind dann aber nicht nur die Teile zu betrachten, sondern auch das ganze System als Zusammenschaltung seiner Komponenten mit geeigneten Kopplungen. Dieser Systemtyp, der bei der Zerlegung von Systemen entsteht, heißt *Netzwerk*. Umgekehrt können auch mehrere, getrennt vorliegende Systeme zu einem Netzwerk verkoppelt werden, dieser Vorgang heißt dann Komposition.

In Erweiterung dieses Aufbaus auf die fünf grundlegende Systemtypen B, G, D, A und N werden die einzelnen Systemtypen in ein hierarchisches Spezialisierungsschema eingefügt. Die Spezialisierung von Systemtypen wird dabei als Halbordnung auf der Menge aller Systemtypen eingeführt. Damit erhält jedes mathematische System einen festen Platz im Gebäude der Systemtheorie.

1.3.4. Ausbau der Theorie zu diesen Systemen (Systemtypen)

Zu den einzelnen Systemtypen sind in verschiedenen, der Systemtheorie verwandten Wissensgebieten mehr oder weniger umfangreiche Theorien vorhanden. Diese werden in der Systemtheorie gesammelt und weiter ausgebaut. Als Beispiele können hier aufgeführt werden: Die Automatentheorie, die Schaltwerktheorie, die Theorie orthogonaler Transformationen (z.B. Fourier-, Walsh-, Laplace-, D-Transformationen), die Theorie der Differentialgleichungssysteme, etc.

1.3.5. Aufstellen der Theorie der Systemtransformationen

Unter einer Systemtransformation versteht man eine Übersetzung eines Systems auf ein anderes System. In der mathematischen Systemtheorie kann eine Systemtransformation formal als Abbildung eines mathematischen Systemtyps auf einen anderen definiert werden [Pich85-1]. Für die fünf grundlegenden Systemtypen B, A, D, G und N gibt es daher 25 verschiedene Arten von Systemtransformationen (Bild 1.4).

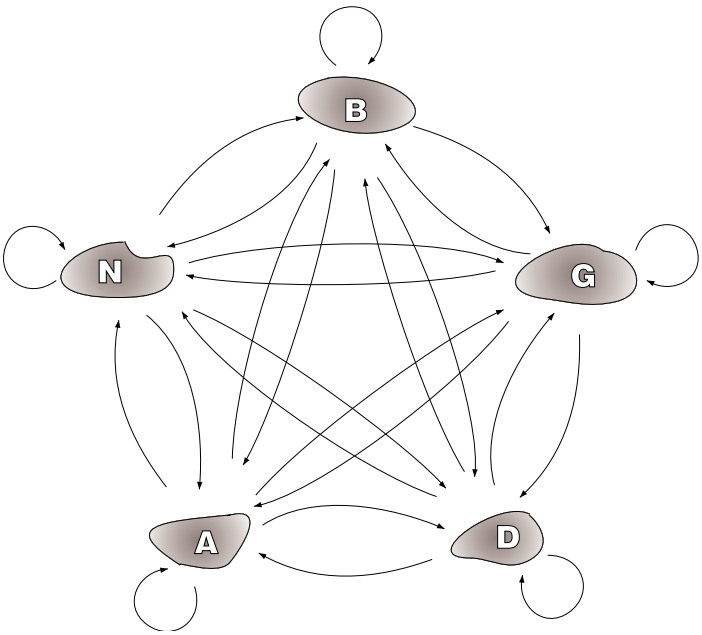

Bild 1.4 Der Graph der prinzipiellen Systemtransformationsarten

Wählt man eine Sprache als Metapher für die Systemtheorie, so stellen die Systemtypen die Substantive dar, aus denen Subjekte und Objekte gebildet werden; die Systemtransformationen sind die Verben, die zur Bildung der Prädikate dienen.

1.3.6. Bereitstellen von Methoden für wichtige Systemtransformationen

Eine (systemtheoretische) Methode wird im folgenden als eine mathematische Vorschrift zur Ausführung einer Systemtransformation verstanden. Solche Methoden gehören zu den Resultaten des Ausbaues der Theorien der Systeme.

1.3.7. Aufbereiten der Methoden in algorithmische, anwendungsnahe Form

Um mathematisch definierte Methoden in der Praxis anwenden zu können, ist häufig die Verwendung von leistungsfähigeren Werkzeugen als Papier und Bleistift notwendig. In vielen Fällen ist aber die direkte Ausführung von Methoden selbst auf modernen Rechnern für nichttriviale Beispiele nicht möglich, weil ihre Formulierung zwar im mathematischen Sinne konstruktiv ist, die Probleme der algorithmischen Komplexität aber nicht berücksichtigt wer-

den (z.B. NP-Vollständigkeit). Aus Sicht der Mathematiker waren lange Zeit alle "endlichen" Probleme trivial, und es ist eine bemerkenswerte Tatsache, daß gerade die Entwicklung schnellster Rechner die Entwicklung der Komplexitätstheorie gefördert hat. Die entscheidende Frage nach der Verwendbarkeit von mathematischen Methoden ist nicht mehr die der Berechenbarkeit (Turing-berechenbar, μ-rekursiv berechenbar etc.) schlechthin, sondern mit welchem Aufwand sie berechenbar sind. Ein im Sinne der Church-Turing These berechenbares Problem, das mit jedem Algorithmus exponential viele Schritte zur Lösung braucht, ist de facto unlösbar für mittlere und große Beispiele, und ein Warten auf schnellere Technologien hilft nicht.

> Ein kleines Zahlenbeispiel kann dies verdeutlichen: Ein NP-vollständiger Algorithmus kann etwa in einer praktischen Situation 10^{100} Rechenschritte benötigen. Ein sehr schneller Rechner kann etwa 100 Millionen Schritte pro Sekunde ausführen. Für das Beispielproblem ergäbe sich hier eine Rechenzeit von 10^{92} Sekunden, d.h. etwa $3*10^{84}$ Jahre. Bei einer millionenfach schnelleren Technologie würde sich diese Zahl nur auf 10^{78} Jahre reduzieren. Gegen exponentielle Komplexität ist mit technologischen Verbesserungen nichts zu machen.

Eine oft angewendete Lösung für diesen Problemkreis ist die Verwendung von sogenannten heuristischen Lösungsmethoden. Diese versuchen, durch Vereinfachungen und Anwendung von Faustregeln schneller zu Problemlösungen zu kommen, die vielleicht nicht optimal, aber akzeptabel sind.

Die Aufbereitung von "Lehrbuchmethoden" und mathematischen Algorithmen zu praktisch anwendbaren, das heißt auf einem Computer effizient implementierbaren Algorithmen oder heuristischen Methoden ist ein eigenes, für CAST sehr wichtiges Thema [Müll91-1] und wird deshalb hier nicht weiter untersucht.

1.3.8. Zusammenfassen von Systemtransformationen zu Systemalgorithmen

Viele Systemprobleme lassen sich zurückführen auf die Frage nach der Anwendung einer bestimmten Systemtransformation auf ein vorgegebenes System. Jedoch wird für diese (gewissermaßen gesuchte) Systemtransformation im allgemeinen keine fertige Berechnungsvorschrift vorliegen. Obwohl die Systemtheorie und die anderen Wissenschaften natürlich daran arbeiten, die Theorien immer weiter auszubauen und Lücken zu schließen, kann zumindest auf absehbare Zeit sicher nicht damit gerechnet werden, daß für alle in der Praxis relevanten Probleme im vorhinein fertige Lösungen vorhanden sein werden.

Oft kann ein Systemproblem, für das keine direkte Methode vorliegt, oder das sich nicht durch eine einzige Systemtransformation lösen läßt, in kleinere Unterprobleme zerlegt werden, die dann mit vorhandenen Methoden gelöst werden können. Die Lösung des gesamten Problems wird durch die Abarbeitung eines Algorithmus erreicht, der als Elementaroperationen systemtheoretische Methoden verwendet. Ein solcher Algorithmus wird im folgenden *Systemalgorithmus* (nach [Pich82-2]) genannt. Der Begriff Algorithmus wird dabei nicht zu eng gefaßt,

beispielsweise können auch nichtdeterministische Vorschriften und interaktive Abfragen zugelassen werden [Mitt87-1].

1.4. Anwendung der Systemtheorie in der Technik

Im allgemeinen Fall kann der Prozeß der Problemlösung folglich gesehen werden als die Ausführung eines für das gestellte Problem passenden Systemalgorithmus. Im Zuge der Abarbeitung des Algorithmus werden gegebene und errechnete Systeme in andere Systeme transformiert, indem durch konkrete Methoden definierte Systemtransformationen ausgeführt werden.

Die Arbeit eines Ingenieurs an einem konventionellen Arbeitsplatz oder an einem modernen CAD-Arbeitsplatz kann nun mit den eingeführten Begriffen beschrieben und als Beispiel für systemtheoretisch unterstütztes Problemlösen verstanden werden. Allerdings wird sich der Vorteil einer systemtheoretischen Denkweise nur dann voll auswirken können, wenn die Arbeitsschritte selbst durch entsprechende Werkzeuge ausgeführt oder erleichtert werden. Dies ist die Idee von *STIPS* (Systems Theory Instrumented Problem Solving) und führt in Konsequenz zu *CAST* (Computer Aided Systems Theory), also zu Computersystemen am Arbeitsplatz, die systemtheoretische Verfahren und Methoden in ähnlich selbstverständlicher und bequemer Weise zur Verfügung stellen, wie ein "wissenschaftlicher" Taschenrechner Gleitpunktarithmetik und transzendente Funktionen anbietet.

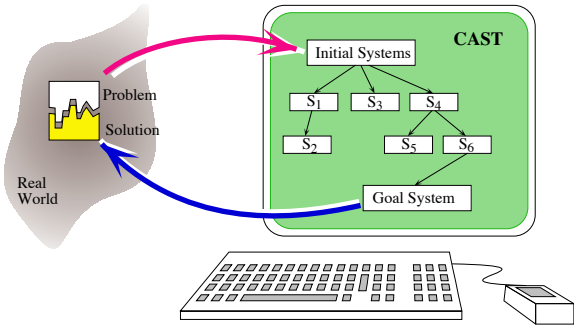

Bild 1.5 CAST als Problemlösungswerkzeug am Arbeitsplatz

1.5. Anmerkungen zum 1. Kapitel

Die hier gebrachte kurze Darstellung der mathematischen Systemtheorie geht auf die langjährige Mitarbeit am Institut für Systemwissenschaften zurück. Sie orientiert sich an [Pich75] und

den jeweils genannten Quellen, ist in dieser kompakten, doch geschlossenen Form, insbesondere dem "8-Punkte-Programm der mathematischen Systemtheorie" (1.3.1 - 1.3.8.), aber eigenständig erstellt. Ein Anstoß, diese Sicht einmal niederzuschreiben, waren die immer wiederkehrenden Diskussionen unter den Kollegen, was denn das Besondere an systemtheoretischer Denkweise und Methodik ("systems approach") sei.

2. Kapitel - Systemtypen und Systemtransformationen

2.1. Mathematische Systeme und Systemtypen

Es soll nun eine formale Definition der Objekte aller systemtheoretischen Untersuchungen, der Systeme, versucht werden. Es gibt in der Literatur sehr verschiedene Definitionen, von völlig abstrakten Systembegriffen ("ein System ist eine Liste von Variablen", [Ashb56]) bis zu sehr komplexen Definitionen ([Wymo67], [Klir69]). Die folgende Darstellung, die erstmalig streng unterscheidet zwischen Systemtypen und Mengen von Systemen, baut auf der Idee der fünf grundlegenden Systemtypen von [Pich85-1] auf.

Ein System ist eine organisierte Anordnung von Teilen, zu einem sinnvollen Ganzen. Weiterhin ist ein System von seiner Umgebung eindeutig abgegrenzt. Die Wechselwirkung eines Systems mit seiner Umgebung erfolgt einerseits durch die "Eingabe" von der Umgebung an das System und andererseits durch "Ausgabe" des Systems an die Umgebung. Eingabe (Input) und Ausgabe (Output) stellen somit die Schnittstelle zwischen System und Umgebung dar. Das Verhalten eines Systems ist der Zusammenhang von Input und Output und daher als eine Form von Input-Output-Relation beschreibbar. Ein innerer "Zustand" ist nach der Input-Output Schnittstelle ein weiterer wichtiger Aspekt eines Systems. Der Zustand eines Systems ist nicht direkt erkennbar, aber er beeinflußt wie die Eingabe ebenfalls die Ausgabe, und damit das Verhalten. Bei einem deterministischen System ist die Ausgabe durch Eingabe und Zustand eindeutig bestimmt. Der Zustand enthält die gesamte Information über die Vergangenheit des Systems, soweit sie für die Zukunft relevant ist. Bei nichtdeterministischen Systemen ist auch bei Kenntnis des Zustandes und des Inputs das weitere Verhalten nicht eindeutig festgelegt. Die Behandlung vieler realer Probleme verlangt diesen Systemtyp, zum Beispiel bei unvollständig definierten Systemen, bei stochastischen Systemen, bei sehr komlexen Systemen.

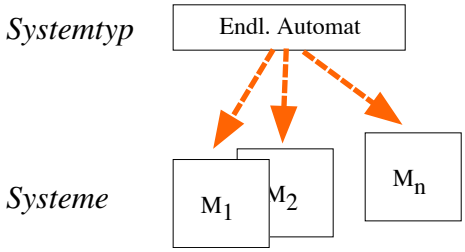

Bild 2.1 System und Systemtyp

Jedes System ist von einem bestimmten Systemtyp:

Definition Systemtyp

> Ein *Systemtyp* S ist ein n-Tupel (S_1,S_2, ... ,S_n). Die Bestandteile S_i eines Systemtyps werden auch Komponenten genannt. Jede Komponente von S ist entweder ein mathematischer Objekttyp, oder ein Systemtyp.

Mathematische Objekttypen sind etwa die Typen Menge, Relation, Funktion, Mengensystem, Gleichung, Ordnung, Verband etc., wie sie in der Mathematik behandelt werden.

Die Komponenten eines Systems sind demnach alle von einem bestimmten Typ (mathematischer Typ oder Systemtyp):

Definition System

> Ein (mathematisches) *System* s ist ein n-Tupel (s_1,s_2, ... ,s_n). Die Teile s_i eines Systems heißen auch seine Komponenten. Jede Komponente von s ist entweder ein mathematisches Objekt, oder ein System.

Ein *System* ist eine Inkarnation (Ausprägung) eines Systemtyps. Ein Systemtyp hat beliebig viele Ausprägungen. Viele Systeme können von einem gemeinsamen Systemtyp sein. Die Unterscheidung von Systemtypen und Systemen ist an vielen Stellen wichtig und entspricht der Unterscheidung von Klassen und Inkarnationen in objektorientierten Programmiersprachen oder der Unterscheidung von Mengen und Elementen in der Mathematik. Der Systemtyp ist hier als eine explizite Konstruktion eingeführt worden, um gemeinsame Eigenschaften einer Menge von Systemen darstellen und manipulieren zu können. Ein Systemtyp kann als eine Abstraktion einer Menge von Systemen aufgefaßt werden. Die Abbildungen "Abstraktion von System zu Systemtyp" und "Inkarnation von Systemtyp zu System" werden später vertieft. Zunächst sollen die wesentlichen Arten von Systemtypen definiert werden, auf die das Systemtheoriegebäude aufgebaut wird.

Es gibt fünf grundlegende Systemtypen, nämlich Black Box, Algorithmus, Generator, Dynamik und Netzwerk (siehe 1.3.3.). Alle weiteren Systemtypen sind *Spezialisierungen* dieser allgemeinen Typen.

Der einfachste Systemtyp ist die Black Box. Dieser Systemtyp läßt sich auf sehr vielfältige Weise definieren. In diesem Kontext soll die folgende Definition verwendet werden:

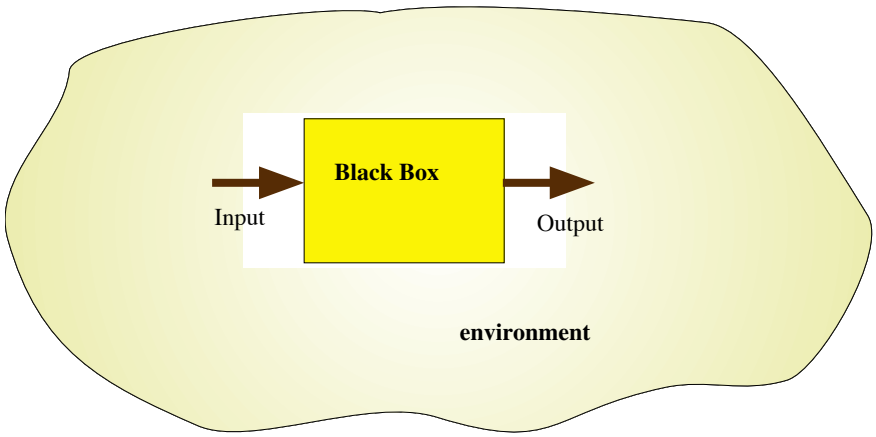

Bild 2.2 Systemtyp "Black Box"

2.1.1. Systemtyp Black Box

Definition Black Box

> Eine *Black Box* ist ein Tripel (A, B, R), mit R ⊆ AxB. Dabei ist A die Inputmenge, B die Outputmenge und R eine Relation, die auch Input-Output-Relation genannt wird. R beschreibt den Zusammenhang zwischen Input und Output des Systems.

Als einfaches und für das Thema FSM typisches Beispiel sei eine (mathematische) Funktion $f:X \rightarrow Y$, $x \mapsto y=f(x)$, angeführt. f kann durch das Tripel (X,Y,R_f) mit $R_f = \{(x,y) : x \in X, y \in Y, y=f(x)\}$ als Black Box angegeben werden.

Ein weiteres Beispiel stellen endliche Zeichenfolgen dar. Eine Zeichenfolge oder ein Wort u über dem Alphabet A von der Länge $l \in \mathbb{N}$ kann durch $u=(I,A,w)$ als Black Box dargestellt werden, wobei $I=1, ...,l$ und $\{w \subseteq IxA : iwa :\Leftrightarrow a$ ist das i-te Zeichen von $u\}$.

2.1.2. Systemtyp Dynamik

Definition Dynamik

Als dynamisches System oder *Dynamik* bezeichnat man ein System, in dem die Veränderlichkeit entlang einer Parametermenge in globaler Weise modelliert wird. Meist wird die Ver-

änderung über die Zeit betrachtet, weshalb diese Parametermenge auch Zeitmenge genannt wird. Wesentlich an der Notation der Zeitmenge, die allen dynamischen Konstruktionen zugrunde liegt, ist die Möglichkeit, "vorher" und "nachher" ausdrücken zu können. Die Zeitmenge muß deshalb eine vollständig geordnete Menge sein. Auch soll die Länge von Intervallen und der Abstand von Zeitpunkten ausdrückbar sein, wofür weitere einfache mathematische Eigenschaften gebraucht werden. Für die Modellierung diskreter Übergänge reicht daher die Menge der natürlichen oder ganzen Zahlen, sonst wählt man üblicherweise die reellen Zahlen als Zeitmenge. Für die Anwendung in der Praxis ist es fast immer notwendig, Systeme mit Input und Output zu definieren. Autonome Systeme können aber jederzeit als Spezialfall der folgenden Notation behandelt werden.

Eine Dynamik (mit I/O) ist ein 7-Tupel (A,B,U,Q,T,ϕ,β), mit

A ist die Menge der Eingabewerte, B die Menge der Ausgabewerte,

U ist die Menge der Eingabetrajektorien u: $[t, t') \rightarrow A$,

Q ist die Menge der Zustände, T ist die geordnete Zeitmenge,

ϕ: TxTxQxU $\rightarrow$ Q , $\quad \phi(t_2, t_1, q, u[t_1, t_2)) = q'$
ist die globale Zustandsüberführungsfunktion,

β: TxQxA $\rightarrow$ B, $\quad \beta(t, q, a) = b$
ist die Ausgabefunktion.

Zu einem gegebenen Anfangszustand q zum Zeitpunkt t_1 berechnet ϕ für jeden möglichen Input den Folgezustand q' zum Zeitpunkt t_2. Aus einem Zustand q und einem Inputzustand a bestimmt β die Ausgabe b.

Naheliegende Spezialisierungen von allgemeinen dynamischen Systemen erhält man durch Einführung von zeitinvarianten Funktionen für ϕ und β. Es wird zur Vereinfachung immer angenommen, daß die Zeitmenge die Menge der natürlichen Zahlen (diskreter Fall) oder die Menge der reellen Zahlen (kontinuierlicher Fall) ist.

Definition Zeitinvariante Überführungsfunktion

Die Überführungsfunktion ϕ wird zeitinvariant genannt, wenn für alle Zeitpunkte t_1, t_2, $t_1<t_2$, für alle Zustände $q \in Q$, und für alle Inputfunktionen $u \in U$ gilt:

$$\phi(t_2, t_1, q, u[t_1, t_2)) = \phi(t_2-t_1, 0, q, u[0, t_2-t_1))$$

Mit anderen Worten, die Zustandsüberführung ist von einer Verschiebung des Inputintervalls unabhängig. Man kann daher auf die Betrachtung beliebiger Intervalle verzichten und nur mehr die bei Null beginnenden Intervalle verwenden.

Definition Zeitinvariante Ausgabefunktion

Die Ausgabefunktion β ist zeitinvariant, wenn für alle Zeitpunkte t_1, t_2, für alle Zustände $q \in Q$ und für alle Inputzustände $a \in A$ gilt:

$$\beta(t_1, q, a) = \beta(t_2, q, a)$$

Die Ausgabe hängt nicht vom Zeitpunkt ab, daher kann man auf die einfachere Ausgabefunktion λ übergehen:

$$\lambda: TxQxA \rightarrow B$$

Definition Zeitinvariantes Dynamisches System

Ein allgemeines zeitinvariantes Dynamisches System ist eine Dynamik, deren Zustandsüberführungs- und Ausgabefunktionen zeitinvariant sind.

2.1.3. Systemtyp Generator

Definition Generator

Bei einem System vom Typ *Generator* gibt es wie bei der Dynamik Eingabewertemenge, Ausgabewertemenge und Zustandsmenge, sowie Zustandsüberführung und Ausgabefunktion. Die Zustandsüberführungsfunktion eines Generators ist aber nur lokal definiert, das heißt generisch für einen beliebigen Zeitpunkt. Ist man an der Ausgabe eines Systems nicht interessiert, spricht man von einer Zustandsmaschine, die dann nur aus Eingabemenge, Zustandsmenge und Zustandsüberführungsfunktion besteht.

Ein Generator ist ein 5-Tupel (A, B, Q, δ, λ), mit

A ist die Eingabemenge, B ist die Ausgabemenge, Q ist die Zustandsmenge

und den Funktionen:

$\delta: QxA \rightarrow Q$, $\delta(q, a) = q'$ ist die lokale (Einschritt-)Zustandsüberführung

$\lambda: QxA \rightarrow B$, $\lambda(q, a) = b$ ist die Ausgabefunktion

Für diskrete Eingabe-, Ausgabe- und Zustandsmengen nennt man A bzw. B Ein- bzw. Ausgabealphabet und das System einen Automaten. Wichtige Spezialfälle sind die Systemtypen der endlichen Automaten und der Schaltwerke.

Wenn die Mengen A, B und Q alle endlich sind, heißt der Generator endlicher Automat. Wenn die Mengen A, B und Q kartesische Produkte der Boolschen Menge {0, 1} sind, heißt das System Schaltwerk.

Weitere Spezialisierungen ergeben sich aus den Eigenschaften der Komponenten. So können die Funktionen partiell definiert sein (partielle Automaten), oder als Verteilungsfunktionen angegeben sein (stochastische Automaten). Ferner kommt man zu der wichtigen Klasse der li-

nearen Automaten, wenn die Mengen Vektorraumstruktur haben und die Funktionen linear sind.

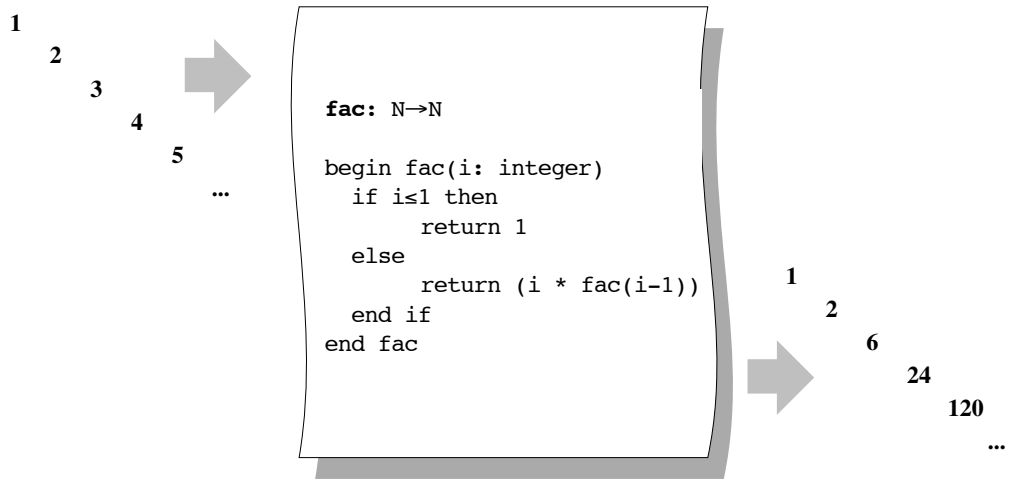

Bild 2.3 Systemtyp Algorithmus

2.1.4. Systemtyp Algorithmus

Definition Algorithmus

Der Algorithmus ist ein Systemtyp zur Beschreibung von Systemen, die als Verarbeitungsvorschrift in einer formalen Sprache definiert sind. Genauer ist ein Algorithmus eine Arbeitsvorschrift von endlicher Länge, die in exakter, unmißverständlicher Form abgefaßt ist und den gesamten Arbeitsvorgang auf die Anwendung gewisser vorausgesetzter Primitiv-Operatoren zurückführt. Jede Anwendung eines Primitivs heißt dabei ein Schritt. Wie bereits erwähnt, verlangt die strenge Definition, daß die Abarbeitung der Vorschrift in jedem Falle nach endlich viel Schritten abbricht. Für die Anwendungen ist meist ein weniger strenger Begriff von Algorithmus sinnvoller, bei dem auch Vorschriften zugelassen sind, deren Abarbeitung unter Umständen nicht abbricht.

Systemtheoretische Beispiele für Systeme vom Typ Algorithmus sind die Algorithmen für systemtheoretische Methoden, wie die Verbandsberechnung von Automaten, die Termminimierung nach Quine-McCluskey, u.s.w.. Einen später behandelten Untertyp stellen die Systemalgorithmen dar. In Kapitel 3 wird dazu auch eine konkrete Notation für Algorithmen angegeben werden.

Algorithmen können als Systeme mit Input und Output betrachtet werden. Der Input eines Algorithmus ist die Folge seiner Argumente, oder Eingangsdaten (Sprechweise der Informatik).

Der Output ist das Resultat der Berechnung. In Kapitel 6 wird das Ergebnis der Anwendung (Durchführung) des Algorithmus a auf die Argumentfolge *s* funktional geschrieben als: a(*s*).

2.1.5. Systemtyp Netzwerk

Definition Netzwerk

Komplexe Systeme werden meist als zusammengesetzte Systeme, bestehend aus einfacheren Teilsystemen, behandelt. Bei der "Bottom-Up"-Vorgangsweise werden bereits vorliegende Teilsysteme zu einem neuen System verkoppelt. Bei der "Top-Down"-Vorgangsweise wird ein System unter Beibehaltung seiner wesentlichen äußeren Eigenschaften zerlegt in kleinere Systeme mit einer Verbindungsstruktur.

Es gibt in der Literatur sehr verschiedene Definitionen für netzwerkartige Systeme. Die folgende, vom Konzept her einfache, aber für diesen Kontext ausreichende Definition soll hier genügen:

Ein *Netzwerk* N ist ein Quintupel (X, Y, K, P, p_o), wobei gilt
- X und Y sind Mengen (Ein- und Ausgabemengen von N)
- $K=\{k_i\}_i$ ist die Menge der Komponenten k_i von N
- $P=\{p_i\}_i$ ist die Menge der Port-Abbildungen p_i

und
für jedes $i = 1 \ldots |K|$ gilt
k_i ist ein System mit Inputmenge X_i und Outputmenge Y_i
$p_i: Y_{j1} x Y_{j2} x \ldots Y_{js} \rightarrow X_i$ ist eine Abbildung von Ausgabemengen anderer Komponenten in die Eingabemenge der Komponente *i* (p_i ist Input-Port der Komponente *i*)
dabei sind die $Y_{jl} \in \{X\} \cup \{Y_m \mid Y_m$ ist Ausgabem. der Komp. k_m für alle $m=1 \ldots |K|\}$
$p_o: Y_{j1} x Y_{j2} x \ldots Y_{jt} \rightarrow Y$ ist die Ausgabeport-Abbildung des Netzwerkes
dabei sind die Y_{jl} wieder Ausgaben der Komponenten oder der Netzeingang

Das verwendete Konzept der Darstellung der Kopplungen zwischen den Komponenten des Netzwerkes durch Port-Abbildungen erlaubt die Verwendung gewöhnlicher Blackbox-Systeme, Generatoren, usw. als Komponenten. Die Zusammenführung von Teilen des Output von mehreren Komponenten auf eine Komponente wird von der Port-Abbildungen durchgeführt, ohne daß die Komponenten selber in Ports strukturierte Ein-/Ausgabe haben müssen. Die Definition erlaubt auch zyklische Strukturen; wo dies nicht sinnvoll ist, müssen entsprechend speziellere Arten von Netzwerken eingeführt werden.

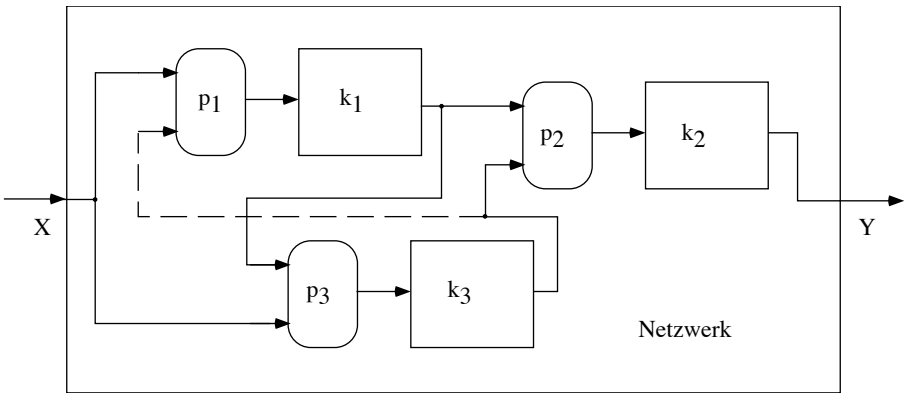

Bild 2.4 Ein System vom Typ Netzwerk mit drei Komponenten

2.2. Speziellere Systemtypen und Vererbung

Die bis hierher eingeführten Systemtypen teilen die Menge aller möglichen Systeme in fünf verschiedene Klassen. Natürlich ist diese Klassifizierung für die Verwendung im Kontext des systemtheoriegestützten Problemlösens noch viel zu grob. Viele speziellere Systemtypen müssen noch eingeführt werden. Zum Beispiel ist ein endlicher deterministischer Automat FSM ein System vom Typ Generator. Jede Methode, die auf Systeme vom Typ Generator angewendet werden kann, kann demnach auf FSM angewendet werden. Allerdings gibt es viele Methoden der Automatentheorie, die nur auf Automaten, oder nur auf endliche Automaten, oder nur auf endliche deterministische Automaten anwendbar sind, nicht aber auf allgemeinere Generatoren.

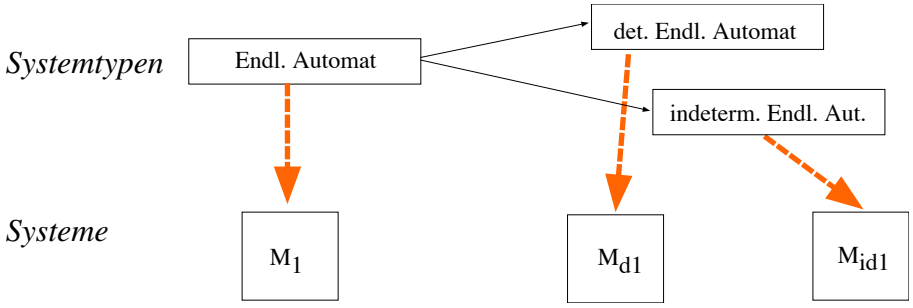

Bild 2.5 Vererbung bei Systemtypen

Die Menge der Systemtypen bildet daher eine Hierarchie, mit den fünf grundlegenden Systemtypen als Wurzeln. Diese Hierarchie läßt sich als einen nichtzyklischen gerichteten Graphen darstellen. Von den Wurzeln weg führen die Kanten in Richtung Spezialisierung von allgemeineren zu spezielleren Systemtypen. Die umgekehrte Richtung kann auch als Abstraktion oder Verallgemeinerung gesehen werden.

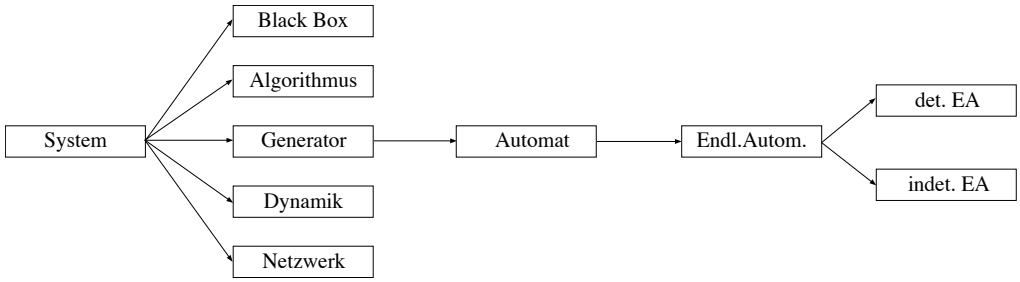

Bild 2.6 Vererbungsgraph

Die Operation Spezialisierung wird in der folgenden Definition eingeführt:

Definition Spezialisierung von Systemtypen

Wir unterscheiden einfache und mehrfache Spezialisierung:

> Bei der *einfachen Spezialisierung* eines Systemtyps S entsteht ein neuer Systemtyp S', der die Komponenten des Typs S enthält, sowie eine beliebige Anzahl zusätzlicher Komponenten. Jede von S stammende ("geerbte") Komponente k von S' kann von anderem Typ sein als die Komponente k von S. Die Komponente k von S' ist dann durch Überschreiben aus k von S entstanden.
>
> Bei der *mehrfachen Spezialisierung* einer Menge von Systemtypen $\{S_i\}_i$ entsteht ein neuer Systemtyp S', der die Vereinigung der Komponenten der Menge $\{S_i\}_i$ enthält, sowie eine beliebige Anzahl zusätzlicher Komponenten.

Jeder Systemtyp kann durch Einführung zusätzlicher Eigenschaften weiter spezialisiert werden. Die Spezialisierungsrelation "ist Spezialisierung von" wird aus der Menge der fünf grundlegenden Systemtypen durch Anwendung der Spezialisierungsoperation generiert. Sie ist transitiv und antisymmetrisch und definiert somit eine Halbordnung auf einer Menge von Systemtypen.
Schreibweise: $S_1 \leq S_2 :\Leftrightarrow$ S_1 ist Spezialisierung von S_2. Eine andere Sprechweise ist: S_1 erbt von S_2. Eine Spezialisierungsrelation wird auch Vererbungsrelation genannt. Eine Menge von Systemtypen bildet mit ihrer Halbordnung $\leq$ eine sogenannte Vererbungshierarchie (eine halbgeordnete Menge). Die nichtreflexive Variante von $\leq$ ("ist Spezialisierung von und ist nicht gleich" oder "echte Spezialisierung") wird wie üblich mit $<$ bezeichnet.

Definition der Funktionen ink und typ

> Für jeden Systemtyp S sei mit ink(S) die Menge der Systeme dieses Typs bezeichnet.

Ist ein System s vom Typ S, so ist es auch vom Typ V, wenn $S \leq V$. Dann gilt offensichtlich: $S_1 \leq S_2 \Leftrightarrow \text{ink}(S_1) \subseteq \text{ink}(S_2)$. Mit anderen Worten entspricht der Vererbungshierarchie der Systemtypen die Teilmengenbeziehung der Mengen der Systeme.

> Umgekehrt liefert typ(s) für ein System s den (speziellsten) Systemtyp von s.

Zusammenhang von ink und typ

Es gilt dann

$$s \in \text{ink}(\text{typ}(s))$$

und

$$\forall\, s \in \text{ink}(S) : S \geq \text{typ}(s).$$

Eine Visualisierung von ink zeigt das folgende Bild 2.7 für vier Beispiel-Systemtypen der Automatentheorie:

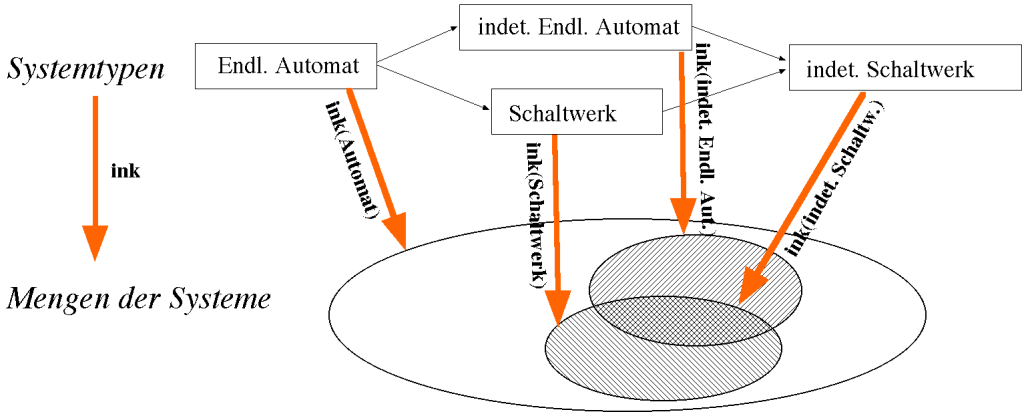

Bild 2.7 Die Funktion ink

Das nächste Bild 2.8 zeigt für vier Systeme M1, M2, M3 und M4, jeweils von den Systemtypen aus dem Bild 2.7, die Werte von typ:

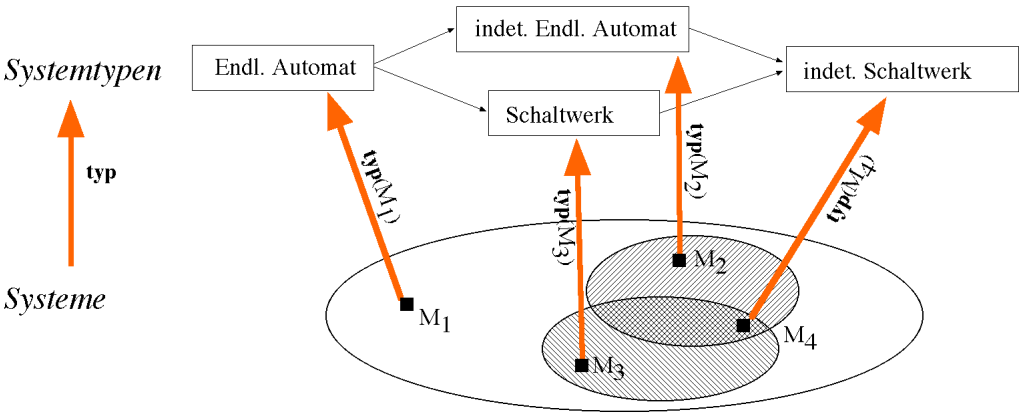

Bild 2.8 Die Funktion typ

2.3. Systemspezifikation

Zur Bearbeitung praktischer Aufgaben müssen Systeme auf vielfältige Weise spezifiziert werden können. Soll ein konkretes System analysiert werden, ist die Angabe aller seiner Komponenten in exakter Form notwendig. Das System wird also durch eine Inkarnation eines passenden Systemtyps voll spezifiziert. Soll dagegen ein Zielsystem einer durchzuführenden Synthese angegeben werden, so sind die Komponenten der erst zu berechnenden Instanz noch nicht bekannt. Nur der Systemtyp des gesuchten Systems ist evtl. schon festgelegt. Andererseits sind durch die Vorgabe von Werkzeugen oder anderen Bedingungen gewisse Eigenschaften der Struktur des gesuchten Systems bekannt und müssen in einer sinnvollen Weise formal ausgedrückt werden können. Um auch in allgemeinen Fällen Systeme spezifizieren zu können, wird die Notation der allgemeinen Systemspezifikation eingeführt.

2.4. Allgemeine Systemspezifikation

Eine *allgemeine Systemspezifikation* kann auf eine von drei verschiedenen Arten erfolgen. Die Liste ist nach Allgemeinheitsgrad aufsteigend sortiert:

(spezielle Sp.)	Durch Angabe eines Systems oder mathematischen Objektes *(special specification)*
(RES Sp.)	Durch Angabe einer RES (recursive entity structure) *(RES specification)*
(Typen Sp.)	Durch Angabe eines Systemtyps oder mathematischen Objekttyps mit Bedingungen *(constrained specification)*

Die erste Art ist die gewöhnliche konkrete Definition eines bestimmten Systems, dessen Komponenten vollständig angegeben werden. Die letzte Art ist die unbestimmteste, die außer dem Systemtyp nur eine (möglicherweise leere) Menge von Bedingungen festlegt, die das System einzuhalten hat. Dazwischen liegt die Spezifikation mittels einer RES-Struktur, die viele, mehr oder weniger konkrete Möglichkeiten bietet, Wissen über ein System auszudrücken. Die RES Notation stellt eine Verallgemeinerung der SES (system entity structure) für die Zwecke im Systemtheorie-unterstützten Problemlösen dar. Die SES wurde von B. Zeigler [Zeig84] für Anwendungen in dem Gebiet "Modellierung und Simulation" geschaffen.

Definition der Funktion ink*

> Für jede allgemeine Systemspezifikation G bezeichne ink*(G) die Menge aller möglichen (konkreten) Systeme, die die allgemeine Spezifikation G erfüllen.

Im Falle einer Typen-Spezifikation ist dann ink* = ink, im Falle einer speziellen Spezifikation sS enthält ink*(sS) nur das System selbst.

2.5. Die RES Notation

Ein (reales oder abstraktes) Objekt, welches beschrieben werden soll, wird in diesem Abschnitt Entität genannt. Eine Entität ist als ein Verweis auf eine (nicht unbedingt im Sinne von "real") existierende Einheit zu verstehen. Eine Entität hat einen Namen, sowie evtl. noch andere Eigenschaften. Entitäten sind Grundbausteine für jede RES Spezifikation.

Der zweite Baustein einer RES ist der Aspekt. Ein Aspekt verweist nicht auf ein (reales) Objekt, sondern stellt eine Betrachtungsweise für Entitäten dar. Wie bei Zeigler werden hier drei Arten von Aspekten unterschieden, aber es sind auch noch Erweiterungen denkbar, die noch andere Arten von Aspekten erlauben. Wichtig ist, daß das Wissen über die interessierenden Systeme in denkökonomischer [Mach82] Form dargestellt werden kann.

Die drei Arten von Aspekten sind

- Dekomposition, also Zerlegung in Teile, die wieder als Entität dargestellt werden
- Mehrfachzerlegung, eine Zerlegung in eine Vielfachheit gleichartiger Teile, deren eines wieder stellvertretend für alle als Entität dargestellt wird, und
- Spezialisierung, das bedeutet Übergang auf eine andere Systembeschreibung, etwa Darstellung durch einen anderen Systemtyp.

Am Beispiel einer digitalen Schaltung sollen die Begriffe veranschaulicht werden. Zunächst ist die Schaltung eine Entität. Sie kann unter verschiedenen Aspekten betrachtet werden. Beispielsweise kann der Aspekt Spezialisierung verwendet werden, um von der realen Schaltung zu der abstrakten Beschreibung als Schaltwerk zu gelangen. Das Schaltwerk kann unter dem Aspekt der Dekomposition in den speichernden Teil und den funktionalen ("kombinatorischen") Teil zerlegt werden. Das Schaltwerk kann aber auch unter dem Spezialisierungsaspekt Verhalten in einen Automaten transformiert werden, weiters in dessen I/O-Verhalten, etc. Der speichernde Teil des Schaltwerkes ist eine Menge gleichartiger Flipflops (1-Bit Speicherelemente) und kann daher als Mehrfachdekomposition bequem dargestellt werden (Bild 2.9).

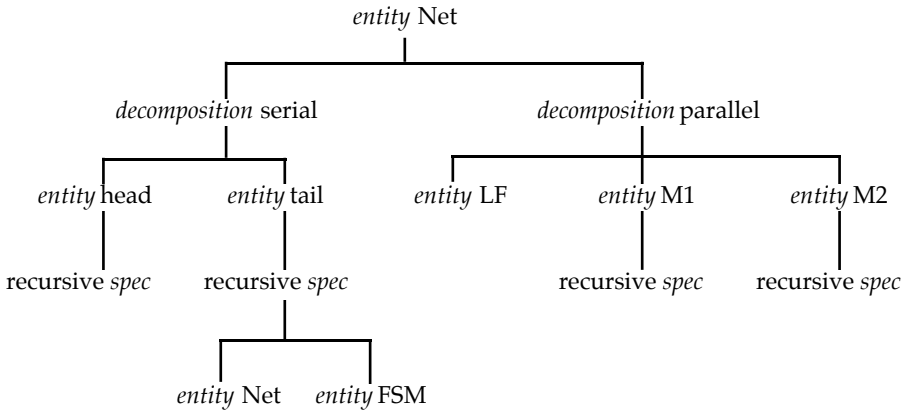

Bild 2.9 RES für ein Automatennetzwerk

Eine RES ist also ein bipartiter Baum, mit den beiden Knotenarten Entität und Aspekt. Die Wurzel ist eine Entität und steht für das zu modellierende reale "System". Jeder weitere Entitätsknoten steht für einen Teil des ganzen Objektes. Jeder Entitätsknoten, der Wurzel eines Unterbaumes der RES-Struktur ist, wird durch diesen genauer beschrieben. Auf einen Entitätsknoten, der kein Blatt ist, folgen immer ein oder mehrere Aspektknoten, die angeben, unter welchem Aspekt die weitere Spezifikation erfolgt.

Definition RES

Eine RES ist ein Quadrupel (E, A, F, e), mit
E ist eine endliche Menge, die Menge der Entitätsknoten
A=D+M+S ist die endliche Menge der Aspektknoten, wobei
"+" die disjunkte Vereinigung bezeichnet und die drei Teilmengen D, M und S von A die verschiedenen Arten von Aspekten bezeichnen,
(E, A, F) ist ein bipartiter gerichteter azyklischer Graph, der RES Graph,
$e \in E$ ist die Wurzel des RES Graphen.

Jeder RES-Graph hat die folgenden Bedingungen zu erfüllen:

(bipartitionedness) In jedem Pfad (= Weg, der die Richtung einhält) folgt auf einen Aspekt eine Entität und umgekehrt.

(unique names) Für zwei beliebige Knoten *h*,*k* einer RES gilt:
h und *k* haben denselben Namen $\Rightarrow h \in X \wedge k \in X \wedge X = \{E, D, S, M\}$.

(uniformity) Knoten mit denselben Namen repräsentieren gleiche (isomorphe) Unterbäume. Zwei Knoten, die im selben Pfad liegen und denselben Namen haben, müssen Entitätsknoten sein.

(recursive hierarchy) Wenn zwei Entity-Knoten *h* und *k*, die den gleichen Namen haben, im gleichen Pfad liegen, dann muß gelten: (der Knoten mit dem größeren Abstand zur Wurzel e sei *k*, o.B.d.A.)
(rh1) *k* ist ein Blatt der RES
(rh2) *k* folgt direkt auf einen Spezialisierungsknoten *s*
(rh3) *s* muß zumindest einen weiteren Nachfolger haben außer *k*
h und *k* werden *rekursive Entitätsknoten* genannt.

(brotherrule) Knoten mit dem gleichen Vater haben verschiedene Namen und repräsentieren verschiedene Strukturen.

(decomposition) Ein D-Knoten hat Entitätsknoten als Nachfolger, die die Teile der zerlegten Einheit repräsentiert. Die Kopplung ist als Eigenschaft des D-Knotens dargestellt.

(multiple decomposition) Ein M-Knoten hat nur eine Entität als Nachfolger, welche stellvertretend für alle Teile der Mehrfachzerlegung steht. Die Kopplung der Komponenten ist als Eigenschaft des M-Knotens dargestellt.

(specialization) Eine Spezialisierung hat Entitätsknoten als Nachfolger, die alternative Spezifikationen der Vorgänger-Entität der Spezialisierung darstellen.

Die Regeln garantieren, daß eine Interpretation der RES-Spezifikation eine sinnvolle Systembeschreibung ergeben kann.

Definition RES-Interpretation

Eine *Interpretation* einer RES-Systemspezifikation wird als Algorithmus definiert:

Input: Eine RES-Spezifikation mit Wurzelknoten *e*

Output: Ein interpretierter RES-Graph

(RES-interpr 0) Der Graph der gegebenen RES ist zu expandieren, indem alle nichtrekursiven, nach Regel (uniformity) abgekürzten Unterbäume eingesetzt werden.

(RES-interpr 1) Eine Kopie *e'* des Wurzelknoten *e* der RES ist in eine Menge "OPEN" zu legen. Ein neuer Graph RES-i wird gebildet, bestehend aus der Wurzel *e'*.

(RES-interpr 2) Wenn OPEN leer ist, weiter mit (RES-interpr 5). Sonst ist ein Knoten *n* aus OPEN herauszunehmen. k(*n*) bezeichne den Knoten in RES, dessen Kopie *n* ist.

(RES-interpr 3) Wenn k(*n*) keine Nachfolger hat in RES und nicht rekursiv ist, weiter bei (RES-interpr 2). Wenn k(*n*) keine Nachfolger hat und rekursiv ist, weiter bei Schritt (RES-interpr 4). Sonst ist einer der Nachfolger, ein Aspekt *a*, auszuwählen. Für $a \in D$ sind alle Nachfolger in den RES-i Graph zu kopieren. Für $a \in M$ ist eine Menge von Nachfolgern von *a* in RES-i zu kopieren. Für $a \in S$ ist ein Nachfolger von *s* auszuwählen und in RES-i zu kopieren. Gerichtete Kanten verbinden *n* mit den neuen Entitäten. Alle kopierten Entitäten werden noch in OPEN gestellt. Weiter mit Schritt (RES-interpr 2).

(RES-interpr 4) k(*n*) ist ein rekursiver Blattknoten: *n* wird durch eine Kopie eines gleichnamigen Knoten von k(*n*) ersetzt, der kein Blatt ist. (Die Existenz eines solchen folgt aus (recursive hierarchy)). Weiter bei Schritt (RES-interpr 3).

(RES-interpr 5) RES-i ist das Ergebnis.

Im resultierenden Graphen RES-i sind nur mehr Entity-Knoten vorhanden. Diese geben eine mögliche Interpretation der RES Struktur wieder. Von mehreren Alternativen der Spezialisierungen wurde dabei jeweils eine ausgewählt, bei Mehrfachdekompositionen wurde die Anzahl der Teile festgelegt. Die Bedingungen für die RES garantieren, daß auch für rekursive

Beschreibungen sinnvolle Interpretationen gefunden werden können. (Andererseits können durch Anwendung der Interpretationsregeln auch beliebig viele sinnlose Interpretationen erzeugt werden.)

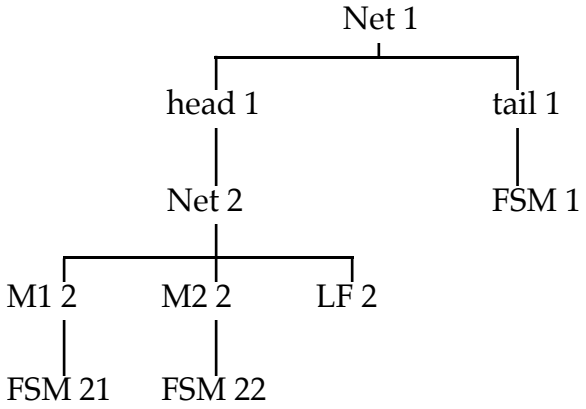

Bild 2.10 Beispiel einer interpretierten RES

Interpretationen in diesem Sinne sind noch keine fertigen Systeme und somit keine Lösungen für Systemprobleme. Andererseits können sie als Leitfaden zur Suche, als eine besondere Art von Heuristik, verwendet werden. Sie stellen eine Art Schema dar, nach dem Systemtransformationen ausgewählt werden können, um zu Zielsystemen mit bestimmten Eigenschaften zu kommen.

2.6. Systemtransformationen

Im Problemlösungsprozeß werden bereits vorliegende Systeme in andere Systeme umgewandelt. Diese Umrechnungsschritte, die aus Sicht des Problemlösers die Elementaroperationen des Prozesses sind, werden Systemtransformationen genannt. Mathematisch kann eine Systemtransformation als eine Abbildung von (Mengen von) Systemen in Systeme aufgefaßt werden.

1. Definition Systemtransformationen

> Eine *Systemtransformation* ist eine (partielle) Abbildung t: ink($(S_i)_{i\in I}$) → ink(S) mit: S_i sind Systemtypen und I eine beliebige nichtleere Indexmenge, mit anderen Worten: $(S_i)_{i\in I}$ ist eine Folge von Systemtypen, S ist ein Systemtyp oder ein mathematischer Objekttyp.

Eine wichtige Klasse von Systemtransformationen sind Dekompositionen. Eine Dekomposition eines Systems *s* ergibt eine Zerlegung in kleinere Systeme, die zusammengenommen das glei-

che Verhalten aufweisen wie *s*. Eine Dekomposition ist daher eine spezielle Syntheseoperation, die ein System in ein Netzwerk von kleineren Systemen überführt.

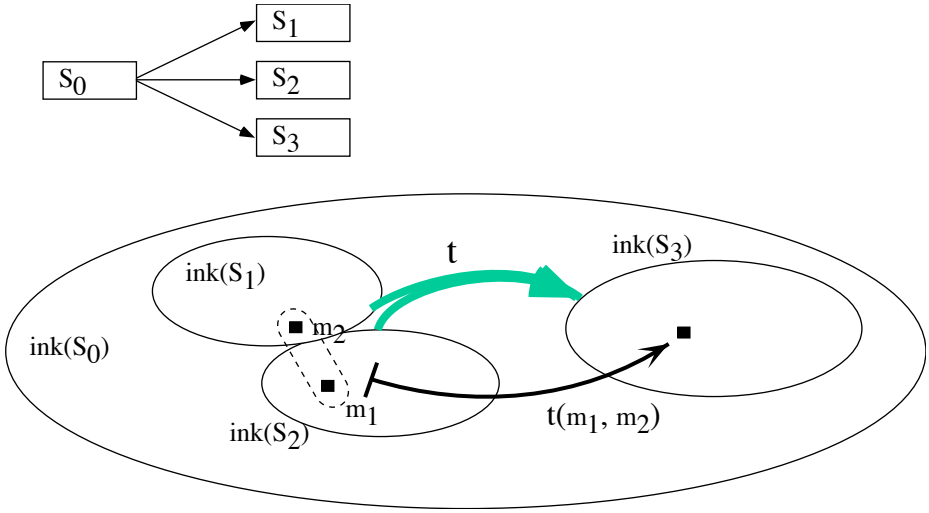

Bild 2.11 Systemtransformationen

Für die praktische Anwendung ist die Zusammenfassung von ähnlichen Transformationen, die sich nur durch Parameter unterscheiden, zu einer parametrisierten Systemtransformation sinnvoll. Beispielsweise wären nach obiger Definition alle Transformationen, die zu einem gegebenen Automaten den Quotienten nach einer Kongruenzrelation *r* bilden, verschiedene Systemtransformationen. Ökonomischer ist aber die Notation der Transformation "Division", die außer dem Ausgangsautomaten als Argument noch die Relation *r* als Parameter erhält.

Daher wird die Definition der Systemtransformationen noch um eine Parameterliste erweitert:

2. Definition Systemtransformationen, Synthese, Analyse, Projektion

Eine *Systemtransformation* ist eine (partielle) Abbildung
t: ink($(S_i)_{i\in I}$) x $(P_j)_{j\in J}$ → ink(S)
mit: $(S_i)_{i\in I}$ ist eine Folge von Systemtypen und $(P_j)_{j\in J}$ ist eine Folge von Parametermengen, S ist ein Systemtyp oder ein mathematischer Objekttyp.

Wenn S ein Systemtyp ist und I einelementig, spricht man von einer *Syntheseoperation*.

Wenn S ein Systemtyp ist und I mehrelementig, heißt t *Komposition*.

Sonst heißt t *Analyseoperation*.

Eine (System-)*Projektion* ist eine Abbildung von einem System auf eine seiner Komponenten.

Als weitere Beispiele seien erwähnt: Die Parallel- und Seriendekomposition von Automaten, die eine Menge von Partitionen als Parameter hat, die Schieberegisterdarstellung von

Automaten, die als Parameter u.a. die Wertigkeit der Registerzellen und die verwendete Liste von Schieberegisterpartitionen erhält, und die Gatterrealisierung von Schaltnetzen, die als Parameter etwa die zu verwendenden Logikbausteine (Zahl der Eingänge, Länge des kritischen Pfades, etc.) haben könnte.

2.7. Anmerkungen zum 2. Kapitel

Die Idee der fünf allgemeinen Systemtypen als universelle Klassifizierung der für die Ingenieurdisziplinen interessanten Objekte und ihrer gegenseitigen Umwandlungen (Systemtransformationen) wurde von [Pich85-1] übernommen. Die RES-Spezifikation ist eine Verallgemeinerung der SES von B. Zeigler. Der Hauptteil des Kapitels ist die Konkretisierung der Hierarchie der Systemtypen und ihrer Transformationen, inspiriert durch Mengenlehre und objektorientiertes Programmieren. Dieser Formalismus wird hier erstmalig vorgestellt.

3. Kapitel - Allgemeine System-probleme und System - algorithmen

3.1. Systemtheorie-unterstütztes Problemlösen

In Kapitel 1.2 wurde die Modellmethode des technisch-wissenschaftlichen Problemlösungsprozesses beschrieben. Dabei wird der interessierende Realitätsauschnitt in ein Modell abgebildet. Die ursprüngliche Fragestellung des eigentlichen Problems wird dann am Modell untersucht. Die so gefundene Lösung gibt dann die Antwort für das reale Problem, indem von der Modellwelt wieder auf die Wirklichkeit geschlossen wird.

Systemtheoretisches Problemlösen, oder die Systemmethode des Problemlösens, unterscheidet sich von der Modellmethode dadurch, daß hier nicht in einer speziell für das vorliegende Problem geschaffenen Modellwelt gearbeitet wird, sondern daß allgemeingültige Systemtypen verwendet werden. Das reale Problem wird also in die Welt der Systemtheorie übersetzt. Hier können die allgemeinen Methoden der Systemtheorie eingesetzt werden, um eine Lösung des (System-) Problems zu erreichen. Der Vorteil liegt also darin, daß vorhandene Theorien und Verfahren für neue Probleme eingesetzt werden können.

Die Transformation eines gegebenen Problems in die Welt der Systemtheorie kann entweder unter Verwendung eines Modells als Zwischenschitt erfolgen, oder direkt, wenn eine sinnvolle Korrespondenz zwischen realen Gegenständen und mathematischen Systemen bereits bekannt ist. Dann kann aber doch ein Modell identifiziert werden, nämlich der dem System entsprechenden, aus der Wirklichkeit herausgelöste Teil der Umgebung. Man kann daher im allgemeinen von der Vorstellung ausgehen, daß die Systemmethode zwei hintereinandergeschaltete Übersetzungen verwendet (Bild 3.1):

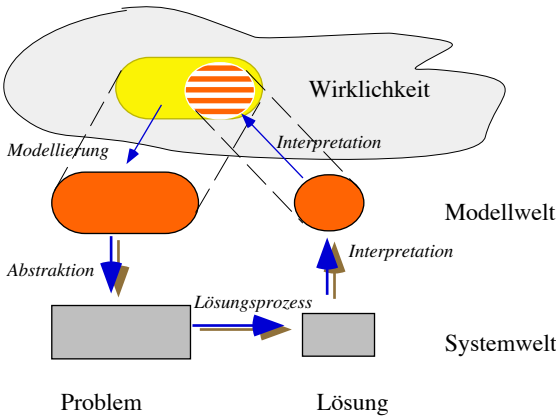

Bild 3.1 Systemmethode des Problemlösens

Zuerst die Abbildung von der Realität in das Modell, welches die nicht relevanten Teile der Realität ausblendet und die relevanten Teile in angemessenem Abstraktionsgrad enthält. Dann die Abbildung vom anwendungsfallabhängigen Modell in das allgemeingültige mathematische System. Zur Anwendung der in der Systemwelt gefundenen Lösung muß diese natürlich in umgekehrter Reihenfolge über beide Stufen zurücktransformiert werden.

Im weiteren wird der Vorgang der Übersetzung von realen Problemen in Systemprobleme nicht genauer untersucht. Es kann daher bei der einfacheren Sichtweise (Bild 3.2) geblieben werden.

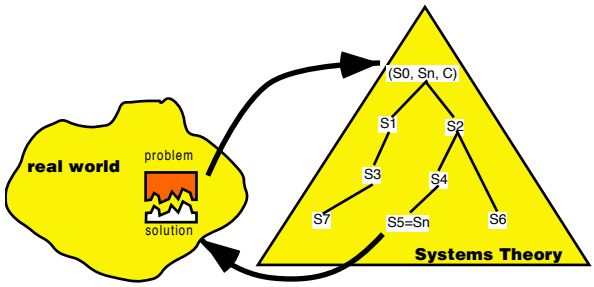

Bild 3.2 Systemtheoretisches Problemlösen

Wenn für ein Systemproblem keine Lösung gefunden wird, besteht noch die Möglichkeit, die zugehörige Systemtheorie entsprechend auszubauen, um das vorliegende Problem doch lösen zu können. Diese Erweiterung kommt dann aber nicht nur dem gerade vorliegenden Problem zugute, sondern auch anderen zukünftig bearbeiteten Fragestellungen. Es ist der Systemtheorie ja ein prinzipielles Anliegen (siehe Kapitel 1), spezielle Lösungen zu verallgemeinern und auf breiterer Basis nutzbar zu machen.

3.2. Formalisierung des Begriffs Problem

Für die Konstruktion von Werkzeugen zur Unterstützung von Problemlösungsprozessen ist eine passende Formalisierung des Begriffes Problem wesentlich. Die Definition muß weit genug sein, um alle Anwendungen systemtheoretischer Vorgangsweisen zu umfassen. Sie muß präzise genug sein, um eine Automatisierung zu ermöglichen. Mit heutigen Mitteln lassen sich nur solche Arbeiten maschinell ausführen oder zumindest unterstützen, die sich auf Symbolmanipulation und Berechnungen zurückführen lassen. Dies gilt insbesondere auch für die Arbeitsgebiete der Artificial Intelligence Forschung; inwieweit diese in Zukunft auch die direkte Arbeit auf semantischer Ebene automatisierbar machen kann, ist derzeit nicht abzusehen.

Für eine Formalisierung des Problembegriffes ist die Unterscheidung von semantischen und formalen Sichtweisen besonders wichtig. Ein am Arbeitsplatz zu lösendes reales Problem hat einen konkreten semantischen Inhalt. Ein formalisiertes Problem ist dagegen frei von jeder Bedeutung, wie alle rein formalen Objekte in der mathematischen Logik. Natürlich soll zwischen dem realen Problem und seiner Formalisierung eine enge Korrespondenz bestehen. Nur so kann erwartet werden, daß die in der formalen Welt erarbeitete Lösung des Problems in eine brauchbare Lösung in der realen Welt zurückübersetzt werden kann. Es ist aber andererseits prinzipiell unmöglich, diese Korrespondenz zwischen realen und formalen Objekten mit der von den mathematischen Disziplinen gewohnten Schärfe und Klarheit zu untersuchen. Die Notation der Homomorphismen ist auf semantische, reale Dinge nur als Metapher anwendbar, nicht aber in mathematisch-präziser Weise, weil die reale Welt eben nicht mathematisch-präzise ist.

Diese Problematik darf aber gerade in den Systemwissenschaften nicht zum Anlaß genommen werden, den Praktiker bei der Formalisierung seiner Aufgaben allein zu lassen, vielmehr ist ihre Unterstützung mit modernen Werkzeugen ein wesentliches Anliegen der CAST-Forschung.

3.3. Systemprobleme

Formalisierte Probleme im Sinne der Systemtheorie werden *Systemprobleme* genannt und werden als Tripel P=(I,C,F) definiert. F ist eine allgemeine Systemspezifikation und I ist eine Menge von speziellen Systemspezifikationen, und zwar beschreibt I die Ausgangssysteme (initial systems) und F das Zielsystem (goal system, final system). C ist eine Menge von Nebenbedingungen (constraints). Die Lösung Sol des Systemproblems ist das vollständig spezifizierte Zielsystem: Sol(P) := sS(F).

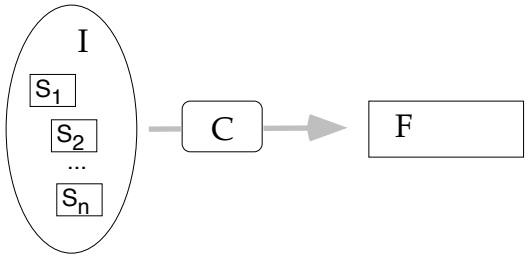

Bild 3.3 Systemproblem P = (I, C, F)

3.4. Allgemeine Systemprobleme

Ein *allgemeines Systemproblem* entsteht aus einem (konkreten) Systemproblem, indem eine oder mehrere speziellere Systemspezifikationen durch allgemeinere ersetzt werden. Ein allgemeines Systemproblem kann als eine Klasse von ähnlichen (konkreten) Systemproblemen aufgefaßt werden.

Ein noch offenes Problem ist die Behandlung von Verallgemeinerung und Inkarnation von Problemen und Problemklassen (siehe Kap. 8).

3.5. Typische Systemprobleme

Im folgenden sollen nun die typischen Arten von Systemproblemen vorgestellt werden, die für die Untersuchung von systemtheoretischen Vorgehensweisen interessant sind.

3.5.1. Analyseprobleme

Eine große Menge von Systemproblemen läßt sich dadurch charakterisieren, daß ein vorgegebenes System oder eine Menge vorgegebener Systeme auf bestimmte Eigenschaften hin untersucht werden soll. Dann liegt ein *Analyseproblem* vor.

Für ein Analyseproblem (I, C, F) ist daher I eine Menge von konkreten Systemspezifikationen, F ist meist ein mathematischer Systemtyp. Dieser beschreibt die gesuchten Eigenschaften der

Systeme in I. Zusätzliche Bedingungen, unter denen die Eigenschaften bestimmt werden sollen, können in C enthalten sein.

3.5.2. Syntheseprobleme

Eine andere große Gruppe von Systemproblemen verlangt die Konstruktion eines neuen Systems. Diese Systemprobleme werden *Syntheseprobleme* genannt. Für ein Syntheseproblem (I, C, F) wird das Verhalten des gewünschten Systems durch I ausgedrückt, seine Struktur durch eine allgemeine Systemspezifikation in F. Nebenbedingungen, die bei der Synthese einzuhalten sind, sowie Optimierungskriterien stehen in C.

Die typische Arbeit eines Ingenieurs im Großen kann meist als eine Art von Synthese aufgefaßt werden. Entwurf und Design im weitesten Sinne von technischen, ökologischen, oder ökonomischen Systemen können als Beispiele angeführt werden. Natürlich fallen bei diesen Arbeiten eine Fülle von kleineren Problemen an, die ihrerseits Analyseprobleme oder Syntheseprobleme sein können.

3.5.3. Realisierungsprobleme

Jedes Problem, bei dem eine Menge von vorgegebenen Systemen in ein Zielsystem zu transformieren ist, heißt *Realisierungsproblem*: Gesucht ist eine Realisierung der Ausgangssysteme. Wenn man diesen Begriff so allgemein faßt, können Analyseprobleme und Syntheseprobleme trotz ihrer doch sehr verschiedenen Semantik wieder einheitlich als Sonderfälle des Realisierungsproblems behandelt werden. Für den Fall der Synthese ist das leicht einzusehen. Bei Analyseproblemen wird in diesem Sinne die Realisierung des mathematischen Ergebnisobjektes gesucht. Dazu sind häufig einige Unterprobleme zu lösen, die noch offensichtlicher Realisierungsprobleme sind: Immer dann, wenn ein gewünschtes Ergebnis nicht direkt (in einem Schritt) erhalten werden kann, wird man versuchen, eine für die Fragestellung günstigere Darstellung der Ausgangssysteme zu erreichen, das heißt, diese durch Anwendung von Systemtransformationen in eine Form zu bringen, die die Beantwortung der Problemfrage ermöglicht oder erleichtert. Es wird also wieder eine Realisierung gesucht.

3.6. Systemalgorithmen

Viele Systemprobleme kommen in gleicher oder ähnlicher Form immer wieder vor. Für die Berechnung ihrer Lösung werden dann im wesentlichen die gleichen Folgen von Schritten durchzuführen sein. Genau wie die einzelnen Berechnungsschritte einer bestimmten system-

theoretischen Methode oder Systemtransformation als ein Algorithmus beschrieben werden, können auch die Schritte zur Lösungsfindung eines Systemproblems zu einem Algorithmus zusammengefaßt werden. Natürlich sind diese Schritte, die Elementaroperationen des Algorithmus, dann selber Methoden oder Systemtransformationen.

Definition Systemalgorithmus

> Ein Algorithmus, der zur Berechnung der Lösung eines Systemproblems oder einer Klasse von Systemproblemen dient, heißt ein *Systemalgorithmus*. Die im Algorithmus verwendeten Elementaroperationen sind systemtheoretische Methoden oder Systemtransformationen. Die Berechnungsvorschrift muß in einer formalen Sprache abgefaßt sein.

Als formale Sprache wird man sich normalerweise eine Programmiersprache vorstellen, die für die Spezifikation komplexer Ablaufstrukturen gut geeignet ist. Einfachere mathematische Algorithmensprachen sind aber auch zulässig.

Der bei dieser Definition zugrundegelegte Begriff des *Algorithmus* wird sehr locker gefaßt und soll nicht allzu exakt formalisiert werden, da dies zu mathematisch-philosophischen Komplikationen führt, die in diesem Kontext nicht relevant sind. Wichtig ist, daß der Algorithmus wenigstens im Prinzip auf einem Computer exekutierbar ist. Vorraussetzungen über Endlichkeit, Terminiertheit, etc. werden hier nicht gemacht, um heuristische und interaktive Implementierungen nicht auszuschließen.

Für die Zwecke in dieser Arbeit reicht folgende (stark vereinfachende) Darstellung von Algorithmen.

Notation Algorithmus:

Ein Algorithmus ist eine Vorschrift über die Bearbeitung bestimmter vorgegebener Objekte mit Hilfe von vorgegebenen Elementaroperationen und Prädikat-Funktionen. Sie besteht aus einer Abfolge von Schritten. Ein Schritt dient entweder zur Kontrolle des Ablaufes (Kontrollanweisung) oder zur Ausführung einer Elementaroperation (ausführbare Anweisung). Die folgende Beschreibung des Aufbaus eines Algorithmus hält sich an die althergebrachte Notation von Algorithmen aus der mathematischen Logik, und entspricht auch dem Modell von Maschinensprachen und einfachen prozeduralen Programmiersprachen. Natürlich könnte man diese Beschreibung ohne weiteres durch die Spezifikation einer beliebigen Programmiersprache ersetzen. (Für die Implementierung der ersten CAST-Systeme wurde etwa Lisp als Sprache, auch für Systemalgorithmen, verwendet.)

> Ein *Algorithmus* ist eine endliche Folge (vollständig geordnete Menge) von Anweisungen. Eine Anweisung ist entweder eine Kontrollanweisung oder eine ausführbare Anweisung. Jede Anweisung hat einen im Algorithmus eindeutigen Index.

Eine Kontrollanweisung bestimmt, mit welchem Schritt die Verarbeitung fortgesetzt wird (oder ob sie beendet wird). Dazu verwendet sie Prädikat-Funktionen, die Eigenschaften der berechneten Objekte anzeigen.

> Eine Kontrollanweisung ist ein Paar, bestehend aus einem Index und einer Prädikatklausel. Die Prädikatklausel enthält eine Prädikatsfunktion samt Argumenten und je einen Index für jeden möglichen Ausgang der Berechnung der Prädikatsfunktion. Diese Indizes bezeichnen die Anweisung, mit der die Abarbeitung fortgesetzt werden soll. Ein Sonderfall ist noch die Stopanweisung, die die Beendigung der Abarbeitung bewirkt.

Eine ausführbare Anweisung wendet eine Elementaroperation auf vorgegebene oder bereits berechnete Objekte an. Dabei entsteht i.a. ein neues Objekt.

> Eine ausführbare Anweisung besteht aus einem Index *i* und einem Aufruf (m, s_i) einer Elementaroperation m auf ihre Argumentfolge s_i.

Die Anweisungen eines Algorithmus werden also normalerweise der Reihe nach abgearbeitet, Kontrollanweisungen können Verzweigungen bewirken. Wird eine Stopanweisung erreicht, ist die Abarbeitung beendet.

Damit ist in einer wenig formalen Weise eine konkrete Algorithmennotation für eine Algorithmensprache mit ihrer dynamischen Interpretation festgelegt, die die gewünschten allgemeinen Eigenschaften hat. Zum Beweis wird diese Algorithmennotation einfach auf die klassische Turingmaschine abgebildet.

In Kapitel 6 werden Systemalgorithmen als Objekte behandelt, die Input oder Zustand einer Maschine sein können.

3.7. Anmerkungen zum 3. Kapitel

Aus [Pich82-4] sind die Begriffe Systemprobleme und -algorithmen entnommen. Die weitere Entwicklung ist hier angepaßt für die Verwendung in den folgenden Kapiteln.

4. Kapitel - Formale Methodenbanken für Systemtheorie: STIPS -

4.1. Methodenbanken

Unter einer *Methodenbank* wird im Rahmen dieser Arbeit eine Sammlung von formalisierten oder implementierten Methoden zu einem bestimmten Fachgebiet verstanden, wobei die Elemente der Methodenbank in einem einheitlichen Rahmen organisiert und auf einheitliche Weise zugreifbar sind. Eine computerbasierte Methodenbank, die systemtheoretische Methoden enthält, wird auch als CAST-System bezeichnet.

Als theoretisches Modell eines CAST-Systems zur Lösung von Systemproblemen wird in diesem Kapitel die STIPS-Maschine eingeführt.

Notation:

Wie üblich bezeichne $\{a_i\}_{i\in I}$ eine indizierte Menge und $(a_i)_{i\in I}$ eine Folge mit der Indexmenge I. Weiterhin sei mit seq(M) := $\{s : s=(a_i)_{i\in I}$, für I=[0, 1, ..., n] für ein n∈N, und $a_i\in M\ \forall i\in I\}$ die Menge aller endlichen Folgen von Elementen der Menge M bezeichnet.

4.2. Die STIPS-Maschine

Sei F ein systemtheoretischesThemengebiet. Beispiele für F sind die Automaten- und Schaltwerktheorie, lineare Systemtheorie, Signalverarbeitung, etc., oder Kombinationen von diesen.

Systemtypen:

Mit S_F sei die für das Thema F relevante Menge von Systemtypen bezeichnet. Dabei ist ein Systemtyp eine beliebige Spezialisierung eines der fünf grundlegenden Systemtypen (B, G, D, A, N). Mit M wird die Menge der benötigten mathematischen Objekte, wie Zahlen, Mengen, Matrizen und Verbände, bezeichnet. Ob ein Objekt wie etwa eine Folge oder eine Funktion als mathematische Hilfsgröße oder als System eingestuft wird, hängt vom Kontext ab. Die Zustandsüberführung eines Automaten beispielsweise wird bei der Definition des Systems als mathematisches Hilfsobjekt eingeführt. Durch die Anwendung einer Projektionsoperation wird sie aber zum weiterbearbeitbaren System aufgewertet, wenn das Thema F auch Funktionen umfaßt.

Die Systemtypen in S_F definieren implizit die Spezialisierungsrelation $<_{S_F}$ der STIPS-Maschine.

Systemtransformationen:

Die Menge aller Systemtransformationen von Systemtypen aus S_F heiße $T(S_F)$. Viele dieser Transformationen sind für das betrachtete Thema nicht bekannt, nicht sinnvoll oder stehen aus anderen Gründen nicht zur Verfügung. Daher wird die Menge T_F der für die Bearbeitung von F vorhandenen Transformationen als Teilmenge von $T(S_F)$ definiert und statt $T(S_F)$ der STIPS-Maschine zugrundegelegt.

Lösungsgraphen:

Ein *Lösungsgraph* (auch: *Realisierungsgraph*) ist ein gerichteter Graph , dessen Knoten Systeme, dargestellt durch allgemeine Systemspezifikationen, und Anwendungen von Systemtransformationen darstellen.

> Sei Sys eine Menge von Systemen, Tran eine Menge von Systemtransformationsanwendungen. Dann ist ein Lösungsgraph ein Tripel lg:=(Sys, Tran, Φ)
> mit $\Phi \subseteq (Sys \times Tran \cup Tran \times Sys)$
> und den Eigenschaften:
> (LG 1) $Sys \neq \emptyset$
> (LG 2a) $\forall t \in Tran\ \exists\ s_1, s_2 \in Sys : (s_1, t) \in \Phi \wedge (t, s_2) \in \Phi$
> (LG 2b) $\forall s \in Sys\ \exists\ t \in Tran : (s, t) \in \Phi \vee (t, s) \in \Phi$
> (LG 3) Sei $W = \{<s_0\ s_n> := <(s_0, t_1)\ (t_1, s_1), (s_1, t_2), \ldots, (t_n, s_n)> : n>0, (s_i, t_j) \in \Phi, (t_k, s_l) \in \Phi\}$ die Menge der Wege in lg.
> Dann gilt: $\forall w = <s_0\ s_n> \in W : s_0 \neq s_n$

Ein Lösungsgraph ist demnach ein (bzgl. Sys) azyklischer bipartiter Graph über einer nichtleeren Menge von Systemen Sys und einer Menge von angewendeten Transformationen Tran. (Bild 4.1)

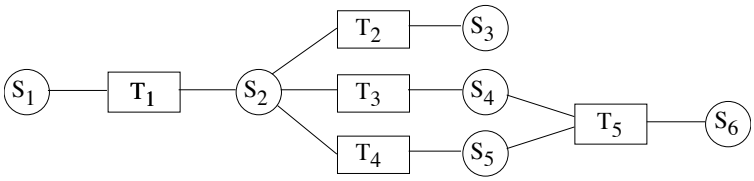

Bild 4.1 Lösungsgraph

Die Menge aller Lösungsgraphen LG(S,T) über den Systemtypen S und den Transformationen T ist dann
LG(S, T) := {(Sys, Tran, Φ):

$$Sys \subseteq ink^*(S) \wedge Tran \subseteq T \times seq(Sys)$$
$$\wedge\ F \subseteq (Sys \times Tran \cup Tran \times Sys)$$
$$\wedge\ (Sys, Tran, \Phi) \text{ erfüllt LG1, LG2, LG3}\}.$$

Vereinigung von Lösungsgraphen:

Auf der Menge LG(S, T) wird die Vereinigung ∪ als zweistellige Operation definiert:

$$\cup : LG(S, T) \times LG(S, T) \rightarrow LG(S, T)$$
$$(Sys_1, Tran_1, \Phi_1) \cup (Sys_2, Tran_2, \Phi_2) := (Sys_1 \cup Sys_2, Tran_1 \cup Tran_2, \Phi_1 \cup \Phi_2)$$

Bemerkung: (LG(S, T), ∪) bildet ein kommutatives Monoid. Beweis durch Rückführung auf die Vereinigung von Mengen.

Für eine übersichtliche Darstellung können für die Transformationsarten Synthese, Analyse, Projektion und Kombination verschiedene Kantensymbole verwendet werden. (Bild 4.2)

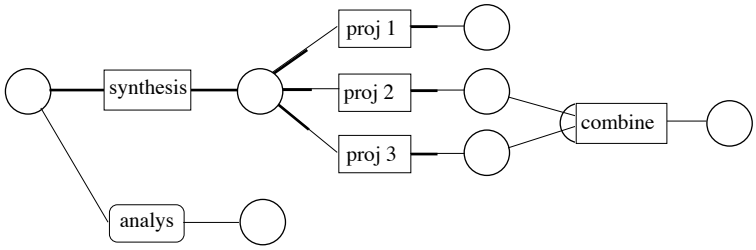

Bild 4.2 Synthese, Analyse, Projektion und Kombination

Projektionen von Lösungsgraphenelementen:

Auf der Menge LG(S, T) werden die Projektionen sys und tran als einstellige Operationen definiert:

Für einen Lösungsgraphen *lg*:=(Sys, Tran, Φ) sei
sys(*lg*) := Sys , d.h. die Menge der Systeme in *lg*, und
tran(*lg*) := Tran , d.h. die Menge der Transformationen in *lg*.

4.2.1. Abstrakte Beschreibung eines interaktiven Computersystems

Nach Hoare-Takahara läßt sich ein interaktives Computersystem als ein Automat C(M) beschreiben, der aus einem Inputalphabet A, einer Ausgabemenge B und einer Menge innerer Zustände Q besteht, und dessen Zustandsüberführung definiert ist als

δ_C: A*x Q x A → A*x Q , $\delta_C(w, q, i) = (w+i, \delta(q, i))$.

(Die Ausgabefunktion λ_C ist hier nicht interessant.)

Dabei ist M=(A, B, Q, δ, λ) eine Automatendarstellung der inneren Vorgänge des Computersystems in einer Auflösung, wie sie für die Behandlung der Interaktion mit dem Benutzer benötigt wird. Die Zustandsüberführung δ_C der Maschine C schreibt demnach die Folge von Inputwerten in der erweiterten Zustandsbeschreibung A*x Q mit. Dadurch wird beispielsweise die Realisierung von Rücknahmebefehlen ("Undo") ermöglicht. Die erste Komponente des Zustands, also das Inputwort, wird auch als "History" bezeichnet.

4.2.2. Definition der STIPS-Maschine

Definition der STIPS-Maschine

Die STIPS-Maschine STIPS-F wird jetzt durch das Paar (S_F, T_F) implizit definiert. STIPS-F wird als eine (partielle) Maschine (also ein System vom Typ Generator) interpretiert. Sie baut, während sie "läuft", einen Lösungsgraphen auf. Der Zustand der Maschine ist durch den bis zu dem betrachteten Zeitpunkt errechneten Lösungsgraphen (engl. "realizations computing graph") bzw. durch die in ihm enthaltene Menge von Systemen gegeben.

Der Anfangszustand ist ein Lösungsgraph ohne Kanten, die Knoten sind dann die Ausgangssystemspezifikationen des bearbeiteten Systemproblems. Der Input ist die Auswahl einer im gegebenen Zustand anwendbaren Systemtransformation *t* aus T_F und ihrer Argumente (s_i). Eine Transformation *t* ist anwendbar, wenn alle Systeme ihres Definitionsbereiches im Lösungsgraphen vorkommen. Der neue Zustand ergibt sich aus der Erweiterung des bestehenden Zustandes um das neue System *s*, welches das Resultat der Anwendung der Transformation *t* ist. Die Erweiterung des Lösungsgraphen erfolgt um die Knoten *t* und *s*, wobei *s* das Ergebnis der Transformation *t* ist, und den Kanten von den Systemen im Definitionsbereich von *t* zum Knoten *t* und den Kanten vom Knoten *t* zu dem neuen Knoten *s*. Der Knoten *t* steht dabei für die Anwendung der Transformation *t* auf (s_i).

Minimaldefinition der STIPS-Maschine

> Formaler definiert das Paar (S_F, T_F) die partielle Zustandsmaschine (A_F, Q_F, δ_F), mit $Q_F = \wp(\text{ink}^*(S_F))$

und $A_F = T_F \times seq(ink^*(S_F))$
und $\delta_F: Q_F \times A_F \rightarrow Q_F$, $\delta_F(q, t, q') = q \cup t(q')$ mit $t \in T_F$

Diese Zustandsmaschine produziert eine monoton wachsende Folge von Systemen als Zustand. Diese stellen die System-Knotenmengen des jeweiligen Lösungsgraphen dar, die Kanten und die Transformations-Knoten werden in dieser Minimalkonstruktion noch nicht beschrieben.

Eine umfassendere Modellierung der STIPS-Maschine wird daher auch die Darstellung des Lösungsgraphen enthalten müssen, ähnlich dem History-Mechanismus in der oben angegebenen Maschine C. Der Lösungsgraph ist dabei insofern allgemeiner als die bloße Folge von Inputwerten, als hier die logischen Abhängigkeiten von Inputs und Ergebnissen (neuen Systemen) festgehalten werden. Andererseits geht die Reihenfolge der Inputs verloren, was aber in diesem Fall kein Nachteil ist, da die Systemtransformationen neue Elemente produzieren, nicht aber bestehende verändern oder löschen, so daß eine "Undo"-Kommandofolge (zumindest für eine elementare STIPS-Maschine) überflüssig erscheint. (Zu den Eigenschaften echter Implementierungen von STIPS-Maschinen siehe Kapitel 5.)

Erweiterte Definition der STIPS-Maschine STIPSlg mit Lösungsgraph als Zustand

Das Paar (S_F, T_F) definiert die partielle Zustandsmaschine $STIPSlg = (A_F, Q_F, \delta_F)$,
mit $Q_F = LG(S_F, T_F)$
und $A_F = T_F \times seq(ink^*(S_F))$
und $\delta_F: Q_F \times A_F \rightarrow Q_F$, $\delta_F(lg, t, (s_i)_{i\in I}) = lg \cup lg'$
wobei $lg \in Q_F$, $t \in T_F$, $s_i \in ink^*(S_F)$
und $lg' = (S, \{t'\}, \{(s_i, t'): \forall i \in I\} \cup \{(t', t((s_i)_{i\in I}))\}$
mit $S = \{q : q \in \{s_i\}_{i\in I} \vee q = t((s_i)_{i\in I})\}$ und $t' = (t, (s_i)_{i\in I})$

Das heißt, der bestehende Lösungsgraph lg wird bei dem Input einer Transformation t und ihrer Argumente s_i um Kanten von jedem s_i zu t' und einer Kante von t' zu dem durch Anwendung von t entstehenden neuen System $t(\{s_i\}_{i\in I})$ erweitert (Bild 4.3).

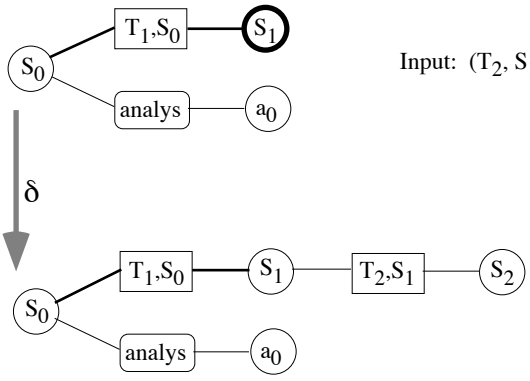

Bild 4.3 Erweiterung des Lösungsgraphen

Die Entwicklung des Lösungsgraphen wird durch die folgende Definition der Dynamik der STIPS-Maschine beschrieben:

Die Definition des Zustandsübergangs vom Zeitpunkt τ auf τ+1 der STIPS-Maschine M mit Lösungsgraphen L(M) kann in formaler Weise wie üblich durch die Einführung einer Phasenüberführungsfunktion δ^0_F erfolgen.

$$\delta^0_F: Z \times Q_F \times A_F \rightarrow Z \times Q_F ,$$

$$\delta^0_F(\tau , lg, t, (s_i)_{i\in I}) = (\tau +1, lg \cup lg')$$

wobei Z die diskrete Zeitmenge ist, und lg' der neue Teilgraph

(o.B.d.A.: Z = Menge der ganzen Zahlen)

Die Phasenüberführungsfunktion wird im folgenden nicht mehr explizit angegeben, da sie ja implizit durch die STIPS-Maschine schon festgelegt ist. Die *dynamische Interpretation* wird durch diese Funktion aber besonders deutlich: Jedem Zeitpunkt τ wird für den bestehenden Lösungsgraphen und eine gegebene Transformation mit ihrer Argumentfolge für den nächsten Zeitpunkt ein neuer Lösungsgraph festgelegt, der aus der Erweiterung des bestehenden um den neuen Teilgraphen besteht.

Andere Formalisierung einer STIPS-Maschine

Alternativ kann die Definition der STIPS-Maschine auch in einer anderen Weise erfolgen: Die jeweilige Struktur des Lösungsgraphen wird aus der Maschine herausgezogen und als externe Konstruktion (ähnlich dem Band-Speichermedium einer Turingmaschine) von der eigentlichen STIPS-Maschine manipuliert. Letztere ist dann ein System vom Typ Automat mit Ausgabe. Man kann die Aufteilung der STIPS-Maschine in den Verarbeitungsteil und den externen Speicher als *Serienschaltung* im Sinne der allgemeinen Systemtheorie deuten.

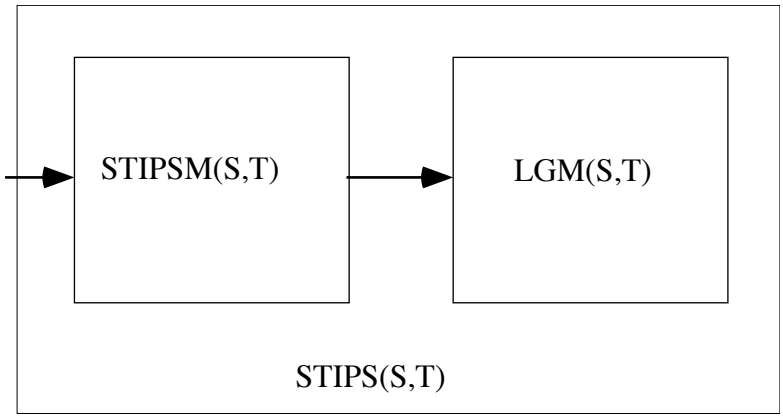

Bild 4.4 STIPS-Maschine als Serienschaltung

Definition der STIPS-Maschine mit Ausgabe und externem Lösungsgraph

Das Paar (S_F, T_F) definiert die STIPS-Maschine STIPSn(S_F, T_F) als Serienschaltung der Komponenten (1) und (2) gegeben durch:

(1) den partiellen Automaten $STIPSM(S_F, T_F) := (A_F, B_F, Q_F, \delta_F, \lambda_F)$,
mit $Q_F = \wp(ink^*(S_F))$
und $A_F = T_F \times seq(ink^*(S_F))$
und $B_F = LG(S_F, T_F)$
und $\delta_F: Q_F \times A_F \rightarrow Q_F$, $\delta_F(S, t, (s_i)_{i\in I}) = S \cup \{s' : s'=t((s_i)_{i\in I})\}$
falls t anwendbar, undefiniert sonst
wobei $S\in Q_F$, $t\in T_F$, $s_i\in ink^*(S_F)$
und $\lambda_F: Q_F \times A_F \rightarrow B_F$, $\lambda_F(S, t, (s_i)_{i\in I}) = f$
falls t anwendbar, undefiniert sonst
mit $f = ($ $\{q : q\in\{s_i\}_{i\in I} \vee q=t((s_i)_{i\in I}\}$,
$\{t'\}$,
$\{(s_i, t') : \forall i\in I\} \cup \{(t', t((s_i)_{i\in I}))\}$)
wobei $S\in Q_F$, $t\in T_F$, $s_i\in ink^*(S_F)$ und $t'=(t, (s_i)_{i\in I})$

und

(2) die partielle Zustandsmaschine $LGM(S_F, T_F) := (A_{LG}, Q_{LG}, \delta_{LG})$
mit $Q_{LG} = LG(S_F, T_F)$
und $A_{LG} = LG(S_F, T_F)$
und $\delta_{LG} : Q_{LG} \times A_{LG} \rightarrow Q_{LG}$
$\delta_{LG}(lg, f) := lg \cup f$

Die Serienschaltung der beiden Komponenten ergibt sich durch die Beschaltung:
$B_F(STIPSM(S_F, T_F)) = A_{LG}(LGM(S_F, T_F))$

Das heißt, bei dem Input einer Transformation t und ihrer Argumente s_i wird der Zustand von $STIPSM(S_F, T_F)$ um das neue System erweitert, und die neuen Kanten für den Lösungsgraphen werden ausgegeben. (Die neuen Kanten führen wieder von jedem s_i zu t und von t zu dem durch Anwendung von t entstehenden neuen System $t(\{s_i\}_{i\in I})$.)

Das Speichermedium $LGM(S_F, T_F)$ nimmt die neuen Kanten als Eingabe und erweitert seinen Lösungsgraphen entsprechend.

Die STIPS-Maschine $STIPSn(S_F, T_F)$ kann formaler als System vom Typ Netzwerk (2.1.5.) (hier ohne Ausgabe) beschrieben werden, in der folgenden Weise:

$STIPSn(S_F, T_F) = (An_F, \{\}, \{STIPSM(S_F, T_F) , LGM(S_F, T_F) \}, P,)$,

wobei An_F die Inputmenge des Netzwerkes ist und
die Portabbildungsmenge $P = \{p_1, p_2\}$ mit
$p_1: An_F \rightarrow A_F$, $p_1(x) = x$ und $p_2: B_F \rightarrow A_{LG}$, $p_2(x) = x$
die Kopplung als Serienschaltung definiert.

Bezeichnung:

Mit $lg(LGM(S_F, T_F))$ sei zu jedem Zeitpunkt der aktuelle Lösungsgraph des Speichermediums LGM bezeichnet, also sein Zustand.

Diesen Definitionen liegt die erste, einfachere Definition einer Systemtransformation zugrunde. Für die realistischere, zweite Definition, die die erforderlichen Parametermengen berücksichtigt, wären einige Erweiterungen notwendig, die jedoch hier nicht explizit ausgeführt werden sollen.

4.3. Steuerung der STIPS-Maschine

Der Input für die STIPS-Maschine ist die Auswahl einer im momentanen Zustand anwendbaren Transformation (und ihrer Argumente). Die STIPS-Maschine ist eine partielle Maschine, da im allgemeinen nicht immer jede Transformation aus T_F anwendbar sein wird, schon gar nicht auf alle möglichen Kombinationen von Systemen.

Die Steuerung einer STIPS-Maschine kann auf verschiedene Weise erfolgen. Im einfachsten Fall muß der Anwender die Auswahl der nächsten Transformation selbst vornehmen. Dies entspricht einem passiven Systemverhalten, wie der Bedienung eines Taschenrechners, wo die Entscheidung über den nächsten Schritt auch ganz beim Anwender liegt. Bei den Prototypen einer STIPS-Maschine "CAST:FSM" und "macCASTfsm" werden dem Anwender zwar *nur* die für ein selektiertes System *möglichen* Transformationen angeboten, die Auswahl einer zielführenden Operation wird aber ganz dem Benutzer überlassen.

Eine STIPS-Maschine kann aber auch als Herzstück einer komplexeren Konstruktion verwendet werden, bei der die Lösung des gestellten Problems selbsttätig gesucht wird. In Kapitel 6 und folgende wird auf Möglichkeiten eingegangen, wie das Problemlösen automatisiert werden kann. Den vorgeschlagenen Systemen wird immer eine STIPS-Maschine der hier angegebenen Art zugrunde liegen. Diese wird dann nicht mehr direkt über das User-Interface vom Benutzer gesteuert, sondern von anderen Komponenten des Systems. Bei der Realisierung einer STIPS-Maschine ist daher eine klare Ein-/Ausgabe-Schnittstellendefinition wichtig.

Bei der praktischen Ausführung der Methoden T_F muß insbesondere der Unterschied zwischen interaktiver und nicht-interaktiver Benutzung der Maschine genau beachtet werden. Die nicht-interaktive Betriebsweise wird hier auch "programmatische" Betriebsweise genannt (der Aufruf erfolgt von einem Programm, nicht direkt vom Benutzer, und das Ergebnis wird von dem aufrufenden Programm weiterverarbeitet). Der Unterschied kommt bei komplexeren Methoden zum Tragen, die bei interaktiver Arbeitsweise während ihrer Ausführung noch zusätzliche Angaben vom Benutzer erfragen können, bei programmatischem Aufruf jedoch ganz auf sich selbst gestellt sind. Bei der ersten Implementierung der STIPS-Maschine als CAST:FSM wurde dieses Problem an vielen Stellen noch nicht berücksichtigt.

4.4. STIPS-Maschine als Systemtyp

Hier wurden mehrere Definitionen der STIPS-Maschine angegeben. Betrachtet man diese Konstruktionen aus der Sicht der Systemtheorie, kann man jedenfalls den allgemeinen Systemtyp Generator ausmachen (die letzte Definition war eine Serienschaltung zweier Generatoren). Natürlich werden bei einer solchen Betrachtungsweise verschiedene Ebenen vermischt: Die STIPS-Maschine manipuliert Objekte der Systemtheorie, und ist andererseits selbst ein solches Objekt. Mit anderen Worten wird ein Element der Metasprache in der Sprache selbst verwendet, was im Prinzip zu Widersprüchen führen kann. In dem hier vorliegenden Fall scheint aber kein Problem zu entstehen.

In der Tat wäre es denkbar, mit einer STIPS-Maschine für Automatentheorie ($STIPS_{FSM}$), oder für allgemeine Systemtheorie, die Konstruktion von anderen STIPS-Maschinen und komplexeren Gebilden, die solche STIPS-Maschinen verwenden, zu untersuchen.

4.5. STIPS-Trajektorien und Systemalgorithmen

Betrachtet man die Änderungen des Zustandes einer STIPS-Maschine über die Zeit, kann man von STIPS-Trajektorien sprechen.

Für die oben eingeführte Zeitmenge Z kann die *Zustandstrajektorie* einer STIPS-Maschine angegeben werden als

$tr: Z \rightarrow Q_F$, $tr(t) := q_t$, wobei q_t der Zustand der STIPS-Maschine zum Zeitpunkt t ist.

Da die STIPS-Maschine ein zeitinvariantes System ist, reicht als Beschreibung einer Trajektorie die Angabe der Folge der erreichten Zustände. Für eine bestimmte Menge von Ausgangssystemen kann man die Menge aller möglichen Trajektorien der STIPS-Maschine betrachten. Diese bilden den Raum der Lösungssuche von Systemproblemen, der auch den Pfad der optimalen Lösung enthält. Die Navigation im Lösungsraum wird im Kontext der Automatisierung des Lösungsprozesses in den Kapiteln 6 bis 9 besprochen.

4.6. Anmerkungen zu 4. und den folgenden Kapiteln

Die STIPS-Maschine wurde von Prof. Pichler als Produktionsregelsystem eingeführt in [Pich86-2], erste Ansätze in [Pich84-1]. Die hier vorgestellte Konstruktion, die den Lösungsgraphen explizit als Zustandsdarstellung verwendet, ist eine eigenständige, wesentlich exaktere

Konstruktion. Sie hat andererseits auch einen direkteren Bezug zu praktischen Realisierungen (siehe Kap. 5).

In Kapitel 5 werden die an der Abteilung entstandenen Implementierungen von STIPS-Maschinen vorgestellt. Die Autoren der Programme werden in einem eigenen Abschnitt aufgeführt.

Die restlichen Kapitel 6 bis 9 sind eine eigenständige Entwicklung des Autors; verwendete Zitate sind gekennzeichnet.

5. Kapitel - Implementierung einer systemtheoretischen Methodenbank: CAST-FSM

In diesem Kapitel sollen anhand der ersten Prototypen von STIPS-Maschinen "CAST:FSM", "CAST-FSM" und "macCASTfsm" Implementierungsaspekte der CAST-Philosophie diskutiert werden.

5.1. Frühe CAST Methodenbanken

Einerseits für die Unterstützung der Forschungsarbeiten, insbesondere das Projekt "STI-DFT" (Systems Theory Instrumented Design For Testability oder Entwurf testbarer Schaltkreise), eine Forschungskooperation zwischen der Siemens AG München, ZTI, Prof. Schwärtzel, und dem Linzer Institut für Systemwissenschaften, andererseits als Werkzeug für die Lehre, wurde 1985 in unserer Abteilung mit der Implementierung von Methoden aus der Automatentheorie begonnen. Gegenüber früheren Automaten-Programmen, die in FORTRAN auf den Modcomp Minicomputern und dem Uni-Großrechner verwirklicht worden waren und isolierte Prozeduren in der Art von numerischen Routinen darstellten, wurden bei den neueren Arbeiten im Rahmen von STI-DFT neue Wege beschritten: Symbolisches Rechnen, objektorientierter Ansatz, Methodenbank, Arbeitsplatz-Rechner, graphische Benutzeroberfläche sind hier die wichtigsten Schlagworte.

Der Abteilung standen seit 1985 vier Interlisp-D Arbeitsplatzrechner "Dandelion 1108" oder "Siemens Bürosystem 5800" zur Verfügung, die beim Hersteller XEROX bereits seit den Siebzigern als Werkzeuge für modernste Forschungen in den Bereichen Artificial Intelligence und User Interface verwendet wurden. Obwohl die Hardware der Rechner 1985 als veraltet einzustufen war, ist die *Entwicklungsumgebung* Interlisp-D, erst recht mit ihrer objektorientierten Erweiterung LOOPS, in vieler Beziehung noch heute vorbildlich und teilweise unübertroffen.

Die erste Automaten-Software [Mitt85] in Interlisp-D war eine symbolische Beschreibung von Automaten mit graphischer Darstellung des Zustandsdiagramms und interaktiven I/O-Experimenten in diesem Diagramm (ähnlich dem "Tokenspiel" bei Petrinetzen). Wichtig war hier die Allgemeinheit der Darstellungsmöglichkeiten für die Zustandsmenge Q, die Input-

menge A und die Outputmenge B des Automaten: Hier konnten nicht nur Zahlen, sondern auch Symbole, Listen oder andere beliebige (Lisp-)Objekte verwendet werden, insbesondere etwa komplette Programme, was bei der Modellierung von Kommunikationsprotokollen sinnvoll sein kann. Tatsächlich waren die Mengen A, B und Q überhaupt nicht explizit abgespeichert, sondern implizit in einer einzigen Tabelle in Listenschreibweise enthalten, die die Zustandsüberführung und die Ausgabefunktion definierte.

Bald folgte die Implementierung [Präh86] der Verbandsberechnung als Grundlage der algebraischen Strukturtheorie. Die Kenntnis des Verbandes eines Automaten ist für die Zerlegung in Serien- oder Parallelschaltungen, also für Dekompositionsmethoden, notwendig und für die Zustandsreduktion hilfreich.

Parallel zu der grundlegenden theoretischen Konstruktion der STIPS-Maschine [Pich86-2] wurde von [Präh86] und [Mitt89-1] die kanonische Übersetzung der systemtheoretischen Philosophie der fünf Systemtypen und ihrer Spezialisierungen in die Welt der objektorientierten Programmierung und ihrer Klassensysteme entwickelt. Das experimentelle LOOPS Entwicklungswerkzeug unterstützte diese frühen Phasen der CAST-Idee in hervorragender Weise. Im Sinne von "Rapid Prototyping" und "explorativem Programmieren" entstand in kurzer Zeit ein Klassensystem für die direkte Modellierung von Systemtypen wie Automaten, lineare Automaten, deterministische endliche Automaten, Schaltwerke, lineare und allgemeine Schieberegister, Register-Fluß-Maschinen (= Informationsflußnetzwerke), Funktionen, boolsche Funktionen und Funktionsnetzwerke, algebraische Ausdrücke, diverse spezielle Netzwerktypen und andere mehr. Als wesentlichstes Element der Benutzeroberfläche dieser "CAST:FSM" genannten Methodenbank wurde der Realisierungsbaum eingeführt, der eine spezialisierte Form des LOOPS Instance-Browsers war. Die Browser dienten einerseits der Anzeige des Zustandes des Problemlösungsprozesses und andererseits als Eingabemedium, indem sie zu jedem angezeigten Objekt die spezifisch auf dieses anwendbaren Operationen (Systemtransformationen etc.) als Menu zur Verfügung stellten.

Zunächst parallel zu diesen Arbeiten entstand ein interaktives System[Mitt86-2], [Mitt88-1] zur Unterstützung der Arbeit mit Petrinetzen (zeitweise "CAST:PN" genannt). Hier war ein komfortabler graphischer Petrinetz-Editor mit einem Simulator integriert, so daß ein Netz schrittweise modifiziert und getestet werden konnte. An analytischen Methoden waren nur sehr einfache tatsächlich implementiert, eine Erweiterung auf die in der Petrinetz-Literatur üblichen Netzanalysen war geplant. Später wurde dieses Petrinetz-System in CAST:FSM integriert.

Eine weitere frühe CAST-Methodenbank war LISAS [Müll86], eine Umgebung zum Editieren, Testen, Debuggen und Simulieren von systolischen Arrays. In einer späteren Diplomarbeit [Gass89] wurde eine Anbindung von LISAS an CAST:FSM vorgenommen, um mit CAST:FSM entwickelte Automaten mit speziellen kryptologischen Eigenschaften als Knotenrechner in einem systolischen Array verwenden zu können.

Eine allgemein zugängliche Beschreibung dieser frühen CAST-Systeme findet sich in den Büchern [Pich90-1], [Pich92].

5.2. Lisp als Implementierungssprache und als Entwicklungsumgebung

Die Objekte der mathematischen Systemtheorie sind vornehmlich als n-Tupel definiert, bestehend aus Mengen, Relationen und Funktionen. Analyse- und Syntheseoperationen ergeben neue Objekte, die ihrerseits wieder aus Mengen, Relationen etc. bestehen können. Die prinzipiellen Aufgabenstellungen eines CAST-Systems sind demnach durch "symbolisches Rechnen" zu lösen. Unter symbolischem Rechnen versteht man die Verarbeitung von nicht-numerischen Daten, eben etwa von beliebigen Mengen etc.

Natürlich lassen sich die Algorithmen der Systemtheorie auch in herkömmliche Programmiersprachen, wie C oder Modula, übersetzen. Die Arbeit der Kodierung eines abstrakt, mathematisch formulierten Algorithmus in eine Programmiersprache wie C ist allerdings erheblich schwieriger, da viel mehr Aufwand in die Entwicklung von geeigneten Datenstrukturen gesteckt werden muß. In Lisp, das von McCarthy von vorneherein für das symbolische Rechnen ausgelegt wurde, lassen sich mathematische Konzepte vergleichsweise direkt hinschreiben. Auch die Kontrollstrukturen sind in Lisp bedeutend flexibler und somit bequemer zu verwenden. Insbesondere ist das leistungsfähige Prinzip der Rekursion in Lisp keine Ausnahme, sondern naheliegender Programmierstil, der für die Verwendung der Sprache typisch ist und von den Compilern im allgemeinen auch effizient übersetzt wird.

Für aufwendige Algorithmen, die auch für größere Problemstellungen akzeptable Rechenzeiten liefern sollen, muß aber auch in Lisp genau überlegt werden, welche Daten- und Kontrollstrukturen gewählt werden, um die Anforderungen zu erfüllen. Zumindest in inneren Schleifen und an anderen kritischen Stellen ist die Programmierarbeit daher ähnlich aufwendig wie in konventionelleren Sprachen. Andererseits stehen auch für die Definition komplizierter Datenstrukturen bereits sehr leistungsfähige Primitive zur Verfügung, wie mehrdimensionale Arrays (Felder) beliebigen Typs und Hash-Tabellen.

5.3. Die Bedeutung von Browsern für interaktive CAST-Umgebungen

In den frühen objekt-orientierten Systemen wie Smalltalk und LOOPS war ein wichtiges Element der Benutzerschnittstelle der sogenannte "Browser". Ein Browser ist ein als Fenster angezeigtes Element der graphischen Oberfläche, das eine Menge von (prinzipiell beliebigen) Objekten und Beziehungen zwischen diesen darstellt. Ein Browser ist deshalb ein (graphisches) Ausgabemedium. Andererseits reagieren die Objekte auf typische GUI-Ereignisse, wie zeigen (anklicken) mit dem Zeigegerät (Maus), durch Präsentation von objektspezifischen Operationen, aus denen der Anwender die gewünschte auswählen kann. In anderen Worten: Die Objekte werden im Browser nicht nur angezeigt, sondern sind auch (Maus-)sensitiv. Der Browser kann dann als Werkzeug zur Befehlsauswahl und -ausführung dienen. In diesem Sinn ist der Browser das typische Eingabemedium objektorientierter Umgebungen.

Begriff Browser: Graphisches Ein/Ausgabemedium.

(B-Output) Sensitive Darstellung einer Menge von Objekten und einer Relation zwischen diesen.

(B-Input) Präsentation und Ausführung von objektspezifischen Operationen auf benutzer-selektierten Objekten.

In LOOPS wurden solche Relationen zwischen Objekten für die Darstellung in Browsern gut unterstützt, die sich als DAG (gerichtete azyklische Graphen) darstellen lassen. Primäres Beispiel ist hier die Klassen-Vererbungshierarchie von LOOPS selbst. Andere Relationen waren möglich, erforderten aber mehr Programmieraufwand (SES-Browser [Mura89], Blockdiagramm-Browser [Schö91]).

Wird in einem LOOPS Browser ein Objekt angeklickt, wird eine Reaktion des Systems hervorgerufen. Im einfachsten Fall ist die ausgeführte Funktion bereits vordefiniert und wird direkt exekutiert. Um dem Benutzer mehr als eine Funktion pro Objekt anbieten zu können, wurden in LOOPS zwei Verfahren verwendet: Zunächst kann ein Menu von mehreren Funktionen angezeigt werden, aus denen der Anwender eine auswählt; zweitens gibt es nicht nur eine, sondern mehrere "Gesten" (= Aktionen des Benutzers), die auf ein Objekt gezielt werden können. Je nach geklicktem Mausknopf (links, Mitte, rechts) und zusätzlich verwendeten "modifier-keys" (Shift, Control, Shift+Control) standen 12 verschiedene Gesten zur Unterscheidung des gewünschten Verhaltens zur Verfügung. Natürlich wurden in den meisten Kontexten nur wenige dieser Varianten wirklich belegt, um Verwirrung zu vermeiden. Konsistent wurde in den meisten Programmen die rechte Maustaste zur Anzeige eines Menus von allgemeinen Fensterbefehlen verwendet. Die linke Maustaste (ohne Modifier) war die Hauptgeste, die die jeweils vom Endbenutzer erwartete Reaktion auslöste. In Browsern, die objektspezifische Operationen anzubieten hatten, war die für LOOPS naheliegende Reaktion auf einen Mausklick mit der linken Taste das Präsentieren eines Pop-Up Menus mit den für das ausgewählte Objekt möglichen Kommandos.

5.4. CAST-FSM - Ein portierbares CAST System für Anwendungen der Automatentheorie

Von der CAST (oder CAST-FSM) "Idee an sich" sind die konkreten Realisierungen in Form von Software-Systemen zu unterscheiden. Die älteste CAST-FSM Implementierung hieß "CAST:FSM", war in Interlisp-D / LOOPS geschrieben und ist in [Präh86] beschrieben. CAST:FSM war nur auf XEROX Dandelion bzw. Siemens 5815 Rechnern lauffähig, die längst das Ende ihres Lebenszyklus erreicht haben und von den Herstellern nicht mehr unterstützt werden. Daher ist CAST:FSM heute nicht mehr einsetzbar. Charakteristisch für CAST:FSM waren die Eigenschaften:

+ großer, ständig (1985-1990) erweiterter Funktionsumfang
+ graphische Oberfläche für Anwender und Entwickler mit Browsern und Pop-Up-Menus

+ graphische Editoren für alle behandelten Systemtypen im von Systemtheorie-Lehrbüchern gewohnten Format

- Implementierung der mathematischen Grundlagen großteils ineffizient

- Datentypen uneinheitlich

- geschlossenes System, Datenaustausch mit anderen Systemen kaum möglich

- abhängig von bestimmten Rechnern und der Interlisp-D Umgebung, beides wird von den Herstellern nicht mehr unterstützt

1989/90 wurde mit einer neuen Implementierung von CAST-FSM begonnen [Müll91-1]. Die Gründe waren

(1) das absehbare Ende des Lebenszyklus der XEROX/Interlisp Maschinen,

(2) die Notwendigkeit der Anwendung der Methoden der algebraischen Strukturtheorie der Endlichen Automaten auf Beispiele, die für CAST:FSM zu groß waren [Müll91-2] und

(3) das Ziel, eine portable Version von CAST-FSM zur Verfügung zu haben, die streng auf Standards oder Industriestandards beruht und die daher problemlos auf der jeweils neuesten/schnellsten vorhandenen Maschine mit Lisp-Umgebung eingesetzt werden kann.

Die bekannten Algorithmen der algebraischen Strukturtheorie (Hartmanis/Stearns, Booth, Kohavi) zur Verbandsberechnung und Dekomposition wurden dazu sehr sorgfältig aufbereitet und auf neuartige Weise implementiert. Die meistens "CAST2" genannte neue CAST-FSM Version war zunächst in Common Lisp [Stee84] und Flavors geschrieben. Einige wenige, besonders zeitkritische Unterprogramme wurden in C geschrieben, um die volle Leistung der (Unix-)Rechner nutzen zu können. Diese C-Funktionen wurden durch das Lucid-Foreign-Function-Interface in die Lisp-Umgebung eingehängt. [Müll91-1]

CAST2 hatte zum Benutzer nur das Lisp-typische zeichenorientierte Kommando-Interface. (Natürlich war die Kommandosprache selbst nichts anderes als Lisp.) Für die Darstellung des Verbandes der Maschinenkongruenzen war aber eine graphische Darstellung unumgänglich. Ein standardisiertes GUI (graphical user interface) für Common Lisp war zu dieser Zeit noch nicht verfügbar. Daher wurde die graphische Darstellung des Automatenverbandes von CAST2 zunächst auf CLIMT aufgesetzt. CLIMT ist ein Fenstersystem für Apollo Workstations und wurde ebenfalls an dieser Abteilung für die Computer Vision Forschung entwickelt [Burg91].

1990 wurde CAST2 von Flavors auf CLOS übersetzt. CLOS ist die standardisierte objektorientierte Erweiterung von Common Lisp. Damit war bis auf die Verbandsanzeige und die externen C-Routinen das Ziel der Portabilität erreicht.

Die wesentlichen Eigenschaften des frühen CAST2 sind denen von CAST:FSM diametral entgegengesetzt:

- stark eingeschränkter Funktionsumfang: Verbandsberechnung, Quotientenmaschine, r-Quotient

- statt vollem graphischem User Interface nur Lisp Listener als Kommandointerpreter

+ hocheffiziente Implementierung der zugrundeliegenden Algorithmen

+ portabler Common Lisp Code (außer Graphik und C-Funktionen)
+ graphische Anzeige des Verbandes in CLIMT
+ Datenaustausch mit anderen Systemen über Dateien im Berkeley File Format (auch "Kiss"-Format genannt)

Die Portierung von CAST2 von Flavors nach CLOS wurde vom Autor zunächst auf dem Apple Macintosh in Procyon Common Lisp, der ersten auf dem Markt erhältlichen CLOS-Umgebung, durchgeführt. (Die Flavors-Version dagegen lief auf Apollo Workstations mit Unix und Lucid Common Lisp.) Dabei mußten die verwendeten C-Routinen in Lisp übersetzt werden (das Foreign-Function-Interface von Procyon wurde hier nicht verwendet) und die CLIMT-Graphik-Funktionalität mußte in die Graphiksprache von Procyon (Common Graphics) übertragen werden. Die meiste Arbeit bereitete jedoch die Übersetzung der objektorientierten Konstrukte von Flavors auf CLOS. Als Hilfe wurde eine halbautomatische Syntaxumwandlung erstellt. Etwa 5% des Source-Codes mußte nach dem automatischen Übersetzungslauf noch manuell nachgebessert werden.

Im Rahmen dieser Übersetzung entstand auch die erste Version 0.1 der macCASTfsm genannten CAST Implementierung, eine sogenannte doppelklickbare Macintosh-Applikation (also unabhängig von einer Lisp-Umgebung zu verwenden).

Als mit einer neueren Version von Lucid Common Lisp (4.0) dann CLOS auch auf den Workstations zur Verfügung stand, wurde CAST2 endgültig in CLOS übersetzt. Etwa gleichzeitig wurde eine Trennung zwischen den mathematischen Grundlagen und den eigentlichen Automatentheorie-spezifischen Teilen von CAST2 eingeführt. Ersterer Teil wurde "CAST-Basics" genannt und auch von einem anderen Projekt der Abteilung, nämlich STIMS, verwendet. Beide Teile, also Basics und der Automatentheorieteil, hießen zusammen von da an "CAST-Kernel".

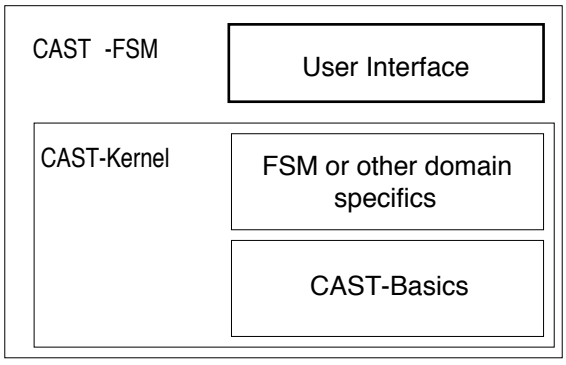

Bild 5.1 CAST-FSM - Kernel und Basics

Eine Diplomarbeit [Scha92] über die Böhlingsche Methode der Schieberegisterdarstellung von Automaten lieferte einen wesentlichen neuen Beitrag für CAST2: Einerseits ist die Transformation von beliebigen Schaltwerken in ein Schieberegisternetzwerk schon für sich eine wertvolle Systemtransformation mit Anwendung für Testbarkeit etc., andererseits kann sie auch als eine Methode der Zustandskodierung gesehen werden und ist auch für nicht vollständig definierte Automaten anwendbar.

Weitere Ergänzungen für CAST2 [Auer92] sind Inversion von Automaten, Überprüfung der Finite-Memory Eigenschaft (endliches Gedächtnis), Konvertierung von Mealy-FSM in Moore-FSM, und die Testbarkeitsanalysen Homing, Diagnose und Synchronisierung.

Parallel zu der Arbeit am CAST-Kernel, dessen Source-Code inzwischen voll portabel geschrieben ist, wurde in Procyon Common Lisp die Entwicklung der CAST-Anwendung "macCASTfsm" weitergetrieben [Mitt92-3]. Die GUI-Komponenten von macCASTfsm enthalten Browser und Icons, Menus und Dialogboxen, und ein interaktives Protokollfenster, in welchem diverse Meldungen der gerade laufenden Programme ausgegeben werden. Außerdem kann der Benutzer hier Befehle eingeben, zum Beispiel wenn in besonderen Fällen die Menukommandos nicht die gewünschten Optionen anbieten. Der Kommandointerpreter ist als einfacher Lisp-Listener geschrieben, so daß dem Benutzer eine leistungsfähige Sprache zur Entwicklung komplexer Systemalgorithmen zur Verfügung steht. (In CAST2 hat der Benutzer ja ohnehin die komplette Lisp-Umgebung zur freien Verfügung - in der "standalone Mac-Application" macCASTfsm dagegen fehlt der dem Lisp-Hersteller gehörende Interpreter/Compiler und mußte daher durch einen eigenen Interpreter erstzt werden.)

macCASTfsm wurde ab der Version 1.0 als Public-Domain Programm (korrekter: FreeWare) freigegeben und im Internet auf dem FTP-Server "ftp.cast.uni-linz.ac.at" plaziert.

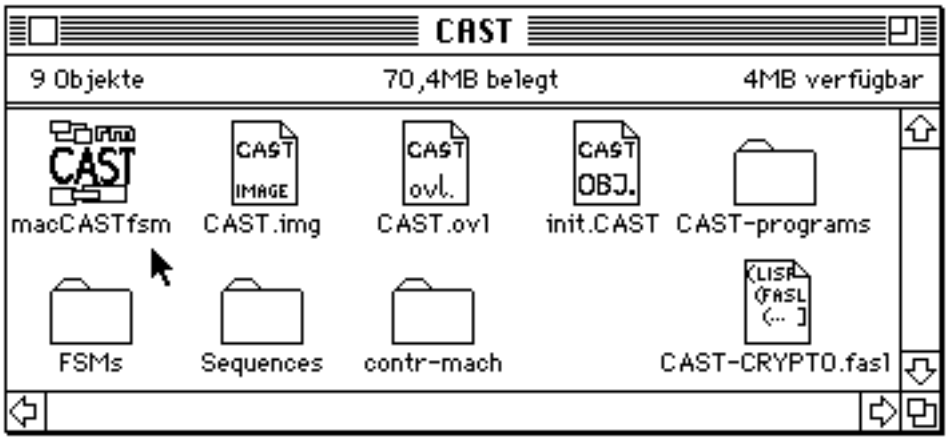

Bild 5.2 CAST-FSM Anwendungsprogramm für Macintosh Computer

Die wesentlichen Eigenschaften des heutigen CAST-FSM:

+ großer Funktionsumfang, vergleichbar mit CAST:FSM
+ verschiedene User Interface Lösungen je nach Plattform
+ hocheffiziente Implementierung der zugrundeliegenden Algorithmen
+ portabler Common Lisp Code für den gesamten CAST-Kernel
+ Datenaustausch mit anderen Systemen über Dateien im Berkeley Standard PLA Format (Kiss, Jedi, mv)
+ für Macintosh existiert Lizenz-freie Applikation "macCASTfsm"

Seit 1992 steht der Abteilung eine frühe Version von CLIM (nämlich 1.1 Lucid CLIM für Sun Sparc und 1.1 für Macintosh Common Lisp) zur Verfügung. CLIM ist die von den meisten Lisp-

Anbietern propagierte Erweiterung für portable graphische Benutzerschnittstellen. Ein logischer Schritt für die CAST-FSM Entwicklung ist natürlich, nach dem kompletten Kernel nun auch die Oberfläche von CAST neu zu entwerfen und mithilfe von CLIM voll portabel zu halten, so daß CAST-FSM auch auf den nächsten Generationen von Hard- und Software mit kleinem Aufwand gewartet und genutzt werden kann.

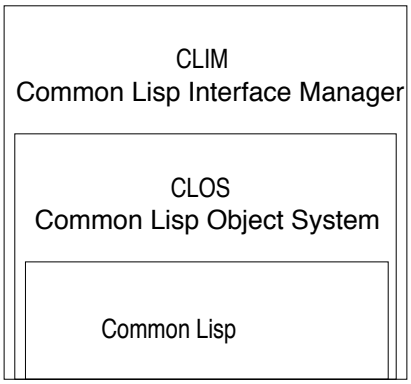

Bild 5.3 Common Lisp - eine Sprachfamilie für portierbare Programmentwicklungen

5.4.1. Programmautorenschaft

In diesem Kapitel wird die Geschichte und der Stand der Arbeiten an der CAST-FSM Umgebung beschrieben. Hier soll nicht der falsche Eindruck erweckt werden, daß der Autor dieser Arbeit auch der Autor der besprochenen Software wäre: Deshalb soll hier kurz aufgelistet werden, wer an welchen Teilen von CAST-FSM mitgearbeitet hat.

Thomas Müller-W.	Basics (Mengen, Funktionen, Cube-Darstellung von Boolschen Teilmengen), Partitionen mit schneller Addition und Multiplikation, diverse C-Funktionen, endliche Automaten, (fsm und fsm-large, geeignet für große Beispiele) effiziente Verbandsberechnung, Dekomposition, seriell und parallel CLIMT-GUI (Verbands-Browser), Mitarbeit bei File-Interface für Automaten und Schaltnetze, Integration der Schieberegisterrealisierung in CAST-FSM, Verbindung zu anderen VLSI-Tools, Anwendung auf MCNC Benchmark Maschinen.
Rudolf Mittelmann	Mitarbeit bei Addition und Multiplikation von Partitionen, einige Algorithmen für Cubes, File-Interface für Automaten und Schaltnetze, Übersetzung von Flavors in CLOS, Integration der Schieberegisterrealisierung in CAST-FSM, vollständiges GUI für Common Graphics (Procyon CL),

	"macCASTfsm" Macintosh-Applikation, CLIM-GUI Prototypen.
Josef Scharinger	einige Besonderheiten für Basics, gesamte (aber nicht CAST-kompatible) Schieberegisterrealisierung.
Franz Auernig	Erweiterungen in "fsm-extensions" (Inversion, Mealy-Moore-Konvertierung, FMM, Testsequenzen).

5.4.2. Der Aufbau von CAST-FSM

Der prinzipielle Aufbau von CAST-FSM kann als Schichtenmodell beschrieben werden. Die Grundlage ist eine Common Lisp Laufzeitumgebung, die ihrerseits auf Unix oder einem anderen Betriebssystem laufen kann. Ist (wie in CAST2) die Benutzerschnittstelle nicht vollständig ausgebaut, wird die volle Lisp-Entwicklungsumgebung als Ersatz benötigt, andernfalls genügt eine Art reines Laufzeitsystem.

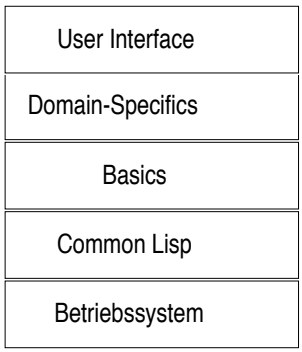

Bild 5.4 Schichtenmodell von CAST-FSM - allgemeines Prinzip

Bild 5.5 Schichtenmodell von CAST-FSM - Implementierungsbeispiele

Auf diese Grundlage ist die unterste Schicht von CAST-FSM gesetzt: CAST-Basics. In CAST-Basics sind die grundlegenden mathematischen Objekte definiert, aus denen die komplexeren systemtheoretischen Konstruktionen zusammengesetzt werden, wie zum Beispiel Mengen, Mengensysteme, Partitionen, Funktionen. Außerdem sind gewisse programmiertechnisch sehr

allgemeine Mechanismen, die in allen Teilen von CAST-FSM verwendet werden können, hier zusammengefaßt. Schließlich gibt es noch eine eigene File, die die wenigen Unterschiede zwischen verschieden Plattformen (Unix, Macintosh, ...) enthält. Ein Beispiel für einen solchen Unterschied ist etwa die Anfangsgröße für eine Hash-Tabelle: In einem Unix-basiertem System mit virtueller Speicherverwaltung wird man einen höheren Wert wählen, um Rechenzeit für rehashing einzusparen. Auf einer PC-Version dagegen ist Speicherplatz kostbar und man wird einen geringeren Anfangswert wählen. Ebenfalls Plattform-abhängig ist die Einbindung von Routinen, die in anderen Programmiersprachen geschrieben sind. In der derzeitigen Version von CAST-FSM wird dabei so vorgegangen: In Lucid Lisp (Apollo, Sun) werden die übersetzten C-Routinen mit einem eigenen Makro definiert und geladen. In anderen Lisp-Umgebungen (Procyon Common Lisp, Macintosh Common Lisp) werden dagegen die entsprechenden Lisp-Versionen dieser Routinen verwendet.

Die nächste Schicht von CAST enthält die Domain-spezifischen Objektklassen und Methoden. Für CAST-FSM sind das die Klassen für Automaten, Schieberegister, etc., und von den Methoden seien hier nur ganz wenige herausgegriffen: Verbandsberechnung, V-Graph-Berechnung nach Böhling, Parallelzerlegung, Serienzerlegung, Parallelzerlegung mit Serienergänzung, Quotientenbildung, Zustandsreduktion, Schieberegisterzerlegung, Abspeichern und Einlesen von Objekten im Berkeley Standard PLA Format und viele andere. In dieser Schicht gibt es ausschließlich portablen Sourcecode.

Beide Schichten zusammen machen den CAST-Kernel aus. In einer vollen Common Lisp Entwicklungsumgebung kann mit dem Kernel in der Lisp-typischen Weise gearbeitet werden, die Benutzerschnittstelle ist dann der Lisp-Listener (interaktiver Interpreter). Allerdings ist es nur schwer möglich, etwa den Verband eines Automaten zu beurteilen und vernünftige Kongruenzen daraus als Argumente für eine Dekompositionsmethode auszuwählen, wenn man keine graphischen Hilfsmittel zur Verfügung hat. Zwar hat praktisch jede Lisp-Umgebung einen sogenannten Inspector, d.h. ein Hilfsmittel zum Inspizieren und Verändern beliebiger Lisp-Datenstrukturen. Gewisse Zusammenhänge werden aber in einer graphischen Darstellung einfach besser sichtbar als in jeder textuellen Darstellung.

Dies motiviert die nächste Schicht von CAST, das graphische User Interface (GUI). Da der kommende Standard CLIM für CAST-FSM zu spät gekommen war, mußte auf andere Lösungen zurückgegriffen werden. Im Rahmen der VKS-Arbeiten [Burger91] entstand das Fenstersystem CLIMT für Lucid Common Lisp auf Apollo Workstations unter DomainOS, ein Unix-Derivat. Für CLIMT wurde schon bald ein Werkzeug zur Darstellung von Automaten-Verbänden geschaffen, das auch zur Beschreibung und Selektion von bestimmten Kongruenzen diente. In Procyon Common Lisp auf dem Macintosh entstand eine vollständige Oberfläche für CAST-FSM, die in Common Graphics geschrieben ist und deshalb mit geringstem Aufwand auf IBM-kompatible PCs portierbar ist, wenn die entsprechende Lisp-Umgebung (mit Common Graphics) vorhanden ist (für Windows benötigt man: Allegro CL/PC von Franz Inc., für OS/2 benötigt man: Procyon Common Lisp for OS/2 von Scientia Ltd.).

Derzeit wird an einer völlig neuen GUI-Version für CAST-FSM gearbeitet, die voll portabel sein wird durch die Verwendung von CLIM (Common Lisp Interface Manager).

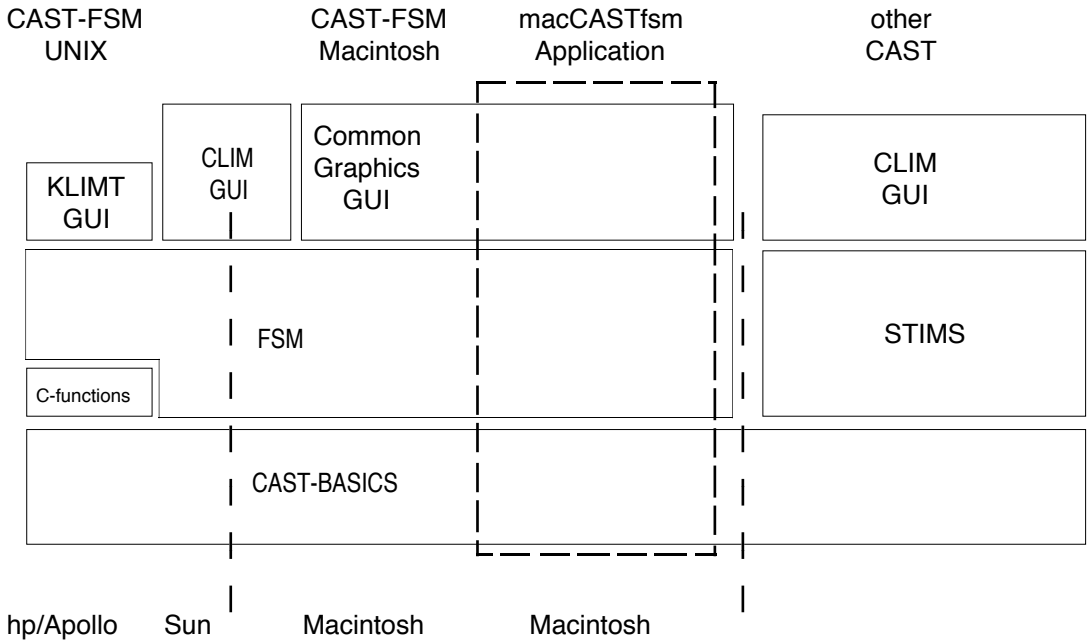

Bild 5.6 Diverse CAST Architekturvarianten

5.4.3. CAST Kernel: Objekte und Methoden der Systemtheorie

Die eigentliche Funktionalität von CAST-FSM, das heißt die objektorientierte Implementierung der Mathematischen Systemtheorie, ihrer Systemtypen und Systemtransformationen, ist im CAST-Kernel enthalten.

Um eine neugestartete Lisp-Umgebung auf das Laden von CAST-FSM vorzubereiten, muß zuallererst die File "init-cast" geladen werden, die vor allem für konsistente Package-Namen sorgt.

Der CAST-FSM Source ist in einem Verzeichnis "source" enthalten und in einige Unterverzeichnisse strukturiert. "source" enthält neben diesen Unterverzeichnissen noch einen eigenen Mechanismus zum Übersetzen und Laden des gesamten Systems oder von Teilen daraus. Die File "make-system" definiert diese allgemeine Ladefunktion und die File "make" enthält die CAST-FSM-spezifischen Funktionen make-CAST resp. make-CAST-* zum Erstellen und Laden eines kleinen resp. des vollen CAST-FSM Systems. Außerdem sind in "make" noch die Pfadnamen für die benötigten Unterverzeichnisse benannt. - Beide Files werden von "init-cast" automatisch geladen.

5.4.3.1. Unterverzeichnisse von "source":

basics	enthält CAST-BASICS: Mengen und Funktionen
partitions	Partitionen, geordnete Partitionen
fsm	Automaten, Verband, Maschinenkongruenzen, Verbandsberechnung, Quotientenbildung, Dekomposition, Outputkonsistenz
fsm-extensions	Mealy-Moore-Konversion, Finite-Memory-Eigenschaft, Inversion von Automaten, Homing- , Diagnose-, Synchronisierfolgen
files	Einlesen und Abspeichern von Automaten und Schaltnetzen im Berkeley Standard PLA Format
shiftreg	Schieberegister-Realisierung
lin-alg	Zusatz-Package für Lineare Algebra (muß bei Bedarf extra geladen werden)
browser	nur für CLIMT: Darstellung des Automatenverbandes in einem CLIMT-Fenster
additions	Sammelsurium von Ergänzungen zu den genannten Teilen von CAST-FSM, wie sie für macCASTfsm gebraucht werden, sowie die Hilfefunktion "?" und cprint.
crypto	"Kryptologische Methodenbank": Methoden zur statistischen und visuellen Beurteilung der kryptologischen Qualität von (Pseudo-) Zufallsfolgen
procyon-GUI	vollständige graphische Oberfläche zur Verwendung in Procyon Common Lisp und macCASTfsm
CLIM-GUI-west-coast	experimentelles portables GUI mit vielen Fenstern (west-coast-style) (derzeit noch eingeschränkte Funktionlität, entspricht etwa der CLIMT-Version, plus Laden von Systemen und Anzeige des V-Graphen)
CLIM-GUI-east-coast	experimentelles portables GUI mit nur einem unterteilten Fenster (east-coast-style) (derzeit noch ohne Funktionlität)
stx	(siehe Kapitel 8)

5.4.3.2. Grundlage für die mathematische Modellierung: CAST BASICS

Die CAST-BASICS Definitionen sind in einem eigenen Directory "basics" enthalten.

"defvar", "macros"

Die File defvar enthält nur einige globale Variablen und Konstanten, macros enthält Makros und Makro-ähnliche Hilfsfunktionen.

"machine-dependencies"

In der File machine-dependencies wird die plattformabhängige Anfangsgröße für Hashtabellen festgelegt.

"climt-replace"

In CLIMT [Burger91] ist die Klasse `viewable` definiert, die als Oberklasse von anzeigbaren Objekten zu verwenden ist. Diese Klasse stellt den Zugriff auf das Fenster zur Verfügung, in dem das Objekt gerade angezeigt wird. Nach CLIMT-Philosophie können das auch mehrere Fenster mit ev. verschiedenen Parametern sein, wie verschiedene Zoomfaktoren oder verschiedene Ausschnitte. Außerdem ist in CLIMT die Lispfunktion `close` (anwendbar ausschließlich auf Lisp-Streams) überschrieben von einer generischen Funktion gleichen Namens, die auch auf CLOS-Objekte anwendbar ist und beispielsweise Methoden für CLIMT-Fenster enthält.

Um bei den gesamten Verbandskonstruktionen eine einheitliche Grundlage zu haben, stellt die File climt-replace eine auf das erforderliche Minimum abgemagerte Version der Klasse `viewable` zur Verfügung und definiert die Funktion `close` entsprechend um. Sie wird als Ersatz dann geladen, wenn CLIMT selbst nicht vorhanden ist, also in jeder anderen Umgebung.

"memoization"

Im funktionalen Programmieren läßt sich mitunter auf sehr einfache Weise eine Effizienzsteigerung erreichen, indem vor dem Aufruf einer Funktion geprüft wird, ob sie mit den gleichen Argumenten bereits vorher einmal aufgerufen wurde. In diesem Fall berechnet man das Ergebnis nicht neu, sondern schaut es nur in einer mitgeführten Tabelle nach. - Dieses "Memoization" gennnte Konzept kann mit Erfolg eingesetzt werden, wenn nicht allzuviel verschiedene Fälle auftreten und wenn der Mehraufwand für das Testen des Arguments einen deutlichen Gewinn gegenüber einer Neuberechnung läßt.

In dem Buch von Peter Norvig [Norv92] wird ein allgemeines, und sehr elegantes, Schema für die Implementierung der Memoization angegeben. Dieses Programm wurde für CAST-FSM noch so erweitert, daß auch Funktionen, die mehrere Werte zurückgeben, richtig behandelt werden. Um aus einer normalen Funktion, die mit `defun` definiert wurde, eine memoizierte Funktion zu machen, braucht man nur `defun` durch `defun-memo` zu ersetzen. Es wird nur das erste Argument verglichen. Sollen ein oder mehrere andere Argumente für den Vergleich verwendet werden, muß statt dem Makro `defun-memo` die Funktion `memoize` mit den entsprechenden Argumenten verwendet werden.

"basics"

Einige kleine Hilfsprogramme.

"dl"

Definition, Konstruktor und Zugriffsfunktionen für die Delta-Lambda-Einträge der Funktionen vom Typ "`two-into-two-fun-large`", die in Automaten vom Typ "`fsm-large`" verwendet werden.

"cubes", "cube2"

Die Algebra der Cube Sets (Cubes sind Unterräume eines n-fachen Produktraumes eines diskreten Raumes wie etwa der Boolschen Menge {0, 1}). Wichtige Operationen sind:

`get-disjunct-binary-cube-list` berechnet aus einer Menge von nicht disjunkten Cubes eine äquivalente Menge von disjunkten Cubes.

`bin-cube-difference` berechnet die Differenz zweier Cubes. Das Ergebnis wird als Liste von disjunkten Cubes zurückgegeben.

`make-binary-tupels-exc` berechnet zu einem binären Vektor eine minimale Liste von disjunkten Cubes, die den Vektor nicht enthalten.

`disjoint-cube-union` berechnet zu einer Liste von disjunkten Cubes eine äquivalente minimale Liste disjunkter Cubes.

`simplify-cubes` berechnet zu zwei Cubes den kleinsten, der beide überdeckt (hätte covercubes oder sup heißen sollen).

"classes"

Alle CLOS-Klassen der Basics sind in dieser File zusammengefaßt.

"sets", "js-sets", "predicate-set"

Methoden für Mengen, Produktmengen, durch Prädikate definierte Mengen, etc.

`make-enum-set` legt ein Mengenobjekt an, welches die als Liste gegebenen Elemente enthält.

`make-prodset-of-enumsets` legt eine Produktmenge an, die die als Liste von enum-sets gegebenen Mengen als Faktoren enthält.

Die Umkehrfunktion zu diesen beiden Konstruktoren ist `elems-descr`. Sie liefert also die Liste der Namen der Elemente bzw. die Liste der enum-sets.

value-list ist das gleiche wie elems-descr.

`make-binary-set` erzeugt die Boolsche Menge der angegebenen Dimension.

`memberp` testet, ob das angegebene Element in der Menge liegt.

`equal-sets` testet zwei Mengen auf Gleichheit der Elemente.

`get-defining-list` liefert die Menge in Listenschreibweise.

`copy-set` kopiert eine Menge (neues Objekt mit gleichem Inhalt).

`binary-p`: Ist die Menge binär?

`no-of-elements` gibt die Anzahl der Elemente einer Menge, für Produktmengen das Produkt der Kardinalität der einzelnen Mengen.

Die Elemente einer Menge werden in CAST-FSM auf verschiedene Weise dargestellt. Die "externe" Darstellung oder der "Name" eines Elementes ist das Lispobjekt, welches beim Anlegen der Menge genannt wurde. Die "interne" Darstellung oder die "Nummer" eines Elementes ist der intern verwendete Zahlencode des Elementes. Dieser wird automatisch verwaltet und ist für den Benutzer von CAST-FSM normalerweise nicht sichtbar. Für den Entwickler von CAST-FSM ist dagegen in vielen Fällen der Umgang mit dem internen Code einfacher und effizienter. Die Konvertierung von Namen zu Nummern besorgt die Funktion `num`, die umgekehrte Konvertierung von Nummern zu Namen wird von der Funktion `nam` erledigt. Statt von "interner Darstellung" wird daher oft auch einfach von "num(s)" gesprochen, statt von "Elementnamen" wird auch von "nam(s)" gesprochen.

Außer symbolischen Elementnamen werden auch Vektordarstellungen von Elementen von Produktmengen in CAST-FSM ausgiebig verwendet, häufigstes Beispiel sind Bitvektoren und Cubes. Ein Cube kann als Teilmenge aufgefaßt werden, die alle Vektorelemente enthält, die man durch Auffüllen der don't-care Codes mit Mengenelementen erhalten kann.

Beispiel: Der binäre Cube `1-00-` steht für die Menge {`10000`, `10001`, `11000`, `11001`}.

Die externe Darstellung in CAST-FSM für einen Cube ist eine Lisp Sequence mit "-" als don't-care Code. Die interne Darstellung eines Cubes ist ein Vektor mit 0 als don't-care Code und der normalen num-Kodierung für die Elemente der Produktmenge.

Beispiel: Der Cube `(0 1 -)` hat die interne Darstellung `#(1 2 0)` (unter der Annahme, daß die `nam-to-num-map` 0 auf 1 und 1 auf 2 abbildet).

Die Übersetzung von der externen Darstellung auf die interne Darstellung erfolgt für Cubes mit der Funktion "`cube`". In der umgekehrten Richtung verwendet man wieder `nam`.

Ein (wichtiger) Sonderfall sind noch echt binäre Vektoren (also ohne don't-care Codes), oder anders gesagt, die Elemente (statt Teilmengen) der Produktmengen. Für diese liefert num aus der externen Darstellung einen numerischen Code (eine natürliche Zahl). Dies ermöglicht etwa eine Array-Indizierung mit Elementen einer binären Produktmenge.

Zusammengefaßt ergibt sich die folgende Tabelle für die Darstellung von Mengenelementen sowie von Teilmengen von Produktmengen in CAST-FSM:

von \ nach mit der Funktion:	"nam" (Element einer Aufzählmenge "enum-set")	"vector" Element einer Produktmen-ge ((binäre) Sequence ohne DCs und mit "nam"s an jeder Stelle)	"nam-cube" Teilmenge einer Produktmenge (Sequence mit "-" als DC und "nam"s an jeder Stelle)	"num" (natürliche Zahl 0 reserviert für DC)	"cube" Teilmenge einer Produktmenge (kodierter Zahlen-Vektor mit 0 als DC und "num"s an jeder Stelle)
"nam" (Element einer Aufzählmenge "enum-set")	identity			num (num set nam)	
"vector" Element einer Produktmenge ((binäre) Sequence ohne DCs und mit "nam"s an jeder Stelle)		identity		num (num set vector)	cube (cube set vector)
"nam-cube" Teilmenge einer Produktmenge (Sequence mit "-" als DC und "nam"s an jeder Stelle)			identity		cube (cube set nam-cube)
"num" (natürliche Zahl 0 reserviert für DC)	nam (nam set num) für enum-sets	nam (nam set num) für prodset-of-enumsets		identity	
"cube" Teilmenge einer Produktmenge (kodierter Zahlen-Vektor mit 0 als DC und "num"s an jeder Stelle)		nam (nam set cube) falls kein DC in cube vorkommt	nam (nam set cube) falls DCs in cube vorkommen		identity

Tabelle 5.1 Datentypen für Mengen und binäre Tupel

"functions", "js-functions"

Methoden für Funktionsobjekte (CAST-Funktionen, nicht zu verwechseln mit Lisp-Funktionen). Das folgende (CLOS-)Protokoll gilt im wesentlichen für alle Funktionen:

`def-range` resp. `val-range`	geben Definitionsbereich resp. Bildbereich zurück.
`print-fn`	dient zum Ausdrucken einer lesbaren Beschreibung.
`get-defining-lists`	liefert drei Werte: Die Elemente des Definitionsbereiches, die des Bildbereiches und die sämtlichen Argument-Wert-Tupeln.
`insert-product`	nimmt ein Input-Tupel und ein Output-Tupel und trägt diese in das Funktionsobjekt ein. (Input-Tupel = Arg.-Tupel

	= Element des Definitionsbereiches, Output-Tupel = Wert-Tupel = Element des Bildbereiches)
`input-ports` resp. `output-ports`	liefert die respektiven Ports als Liste.
`concat-port-values`	hängt einzelne Port-korrespondierende Werte entsprechend einer Portbeschreibung zusammen (= flacht sie aus).
`extract-nth-port-value`	extrahiert den zum gegebenen Index gehörenden Portwert aus einem Arg.- oder Werttupel (ausgeflacht) entsprechend einer Portbeschreibung.
`extract-port-values`	extrahiert alle Portwerte aus einem Arg.- oder Werttupel (ausgeflacht) entsprechend einer Portbeschreibung.
`apply-fn`	wendet eine Funktion auf eine Liste von Portwerten (nicht ausgeflacht) an. Ergebnis ist eine Liste von Portwerten. (Bug: außer bei projections)
`apply-fn-to-arg`	wendet eine Funktion auf ein Arg.- oder Werttupel (ausgeflacht) an. Ergebnis ist ausgeflacht.

`expl-fn, hash-fn`

Es gibt hier für allgemeine Zwecke die Funktionen vom Typ "`hash-fn`", die eine beliebige Abbildung von einer Definitionsmenge in eine Bildmenge durch explizites Abspeichern der Argument-Wert-Tupel modellieren. Die Mengen können dabei noch zusätzlich eine Struktur tragen, was durch Ports ermöglicht wird. Eigene Umwandlungsfunktionen konvertieren Argumente oder Werte von Funktionen von Port-Struktur in ausgeflachte Form oder umgekehrt.

Spezielle Funktionstypen wurden für Automatenalgorithmenzwecke eingeführt:

`two-into-one-fun`

Optimal für die Verwendung als Delta oder Lambda (Zustandsüberführung bzw. Ausgabefunktion) in kleineren Automaten sind Funktionen vom Typ "`two-into-one-fun`". Wie der Name andeutet, bilden Funktionen dieses Typs von zwei Ports auf ein Port ab. Die interne Darstellung ist einfach ein Array der Länge der Anzahl der Elemente des einen Ports, welches an jeder Stelle ein Array der Länge der Anzahl der Elemente des zweiten Ports enthält. Der Eintrag an jeder Stelle dieser Arrays ist der jeweilige Funktionswert. Algorithmen, die in inneren Schleifen für ein Argument des einen Ports alle Funktionswerte für alle Argumente des zweiten Ports benötigen, können so sehr effizient durch Abarbeiten eines Vektors erledigt werden. (Es gibt Makros, wie `do-all-fsm-inputs`, die die Syntax solcher Algorithmen drastisch vereinfachen.) Bei der Anwendung als Delta resp. Lambda eines FSM von der Klasse "`fsm`" ist das erste Port die Menge der Zustände, das (äußere) Array enthält also für jeden Zustand einen Vektor, der wiederum für jeden Input (zweites Port) einen Eintrag enthält, nämlich den Folgezustand resp. den Output. Der wirkliche Inhalt dieser Vektoren ist ein "num" (kodierter symbolischer Wert) oder ein kodierter Cube (für einen Binärvektor).

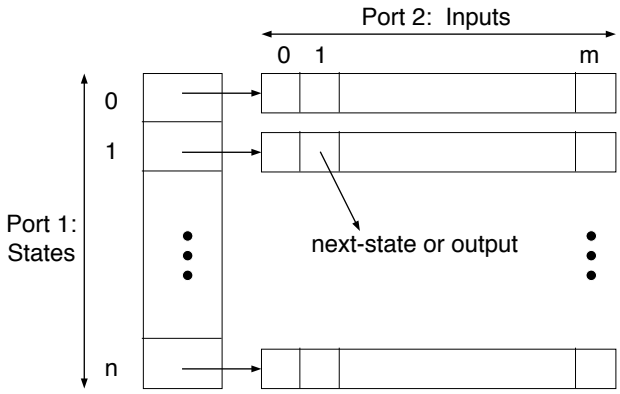

Bild 5.7 Datenstrukur der Klasse two-into-one-fun

Anmerkung: Eine Eigentümlichkeit der internen Datenstrukturen von CAST-FSM ist die, daß die verwendeten Arrays (1) eindimensional sind, zweidimensionale Matrizen werden als Vektor von Vektoren dargestellt, und (2) sind diese Vektoren um eins länger als notwendig; die Stelle 0 der Vektoren wird nicht verwendet (jedenfalls nicht zur Repräsentation von Daten).

`two-into-two-fun-large`

Für größere Automaten, insbesondere solche, die zwar ein großes Produkt Anzahl-der-Zustände mal Anzahl-der-Inputworte haben, aber andererseits doch durch relativ wenige Produktionen (etwa im Berkeley Standard PLA Format) beschrieben werden können, wurde eine eigene Datenstruktur geschaffen. In den BASICS gibt es dazu die Funktionen vom Typ `two-into-two-fun-large.` Eine solche Funktion dient der gleichzeitigen Darstellung von Delta und Lamba eines Automaten. Sie enthält ein Array mit einem Eintrag für jeden Zustand (genauer: Zustands-Code, also ein num zwischen 1 und `no-of-states`; wieder bleibt der zusätzliche Eintrag für den Index 0 unbenutzt). Der Eintrag ist eine Liste von "`dl`"-Datenstrukturen mit jeweils drei Komponenten: Einem (kodierten) Input-Cube, dem (kodierten) Folgezustand, also ein "num" für symbolische Zustandsmengen, und dem (kodierten) Output-Cube. Auch diese Klasse hält sich an das allgemeine Funktionenprotokoll, enthält aber noch einige zusätzliche Funktionen, die für allgemeinere Zwecke sinnlos wären.

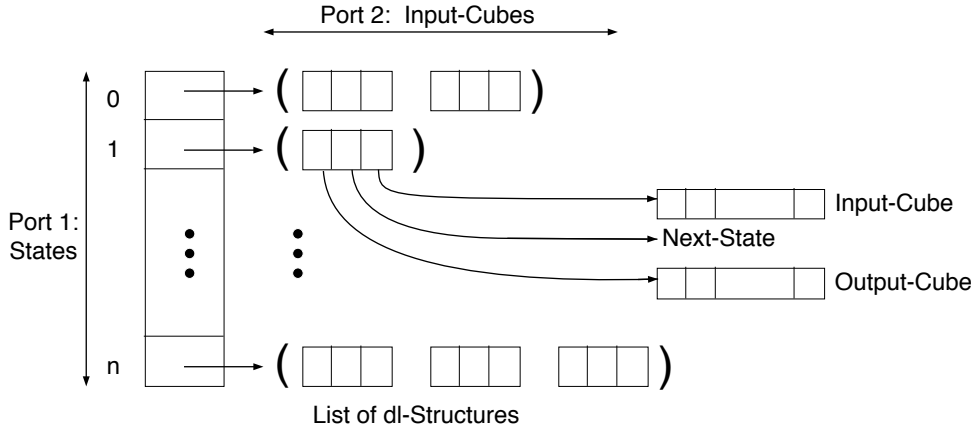

Bild 5.8 Datenstruktur einer Funktion der Klasse two-into-one-fun-large

`fun-projection`

Diese Funktionen bilden die Projektion einer Funktion auf ein bestimmtes Ausgangsport. Die erste Anwendung in CAST-FSM ist für Delta und Lambda eines Automaten vom Typ `fsm-large`. Dieser enthält ja die eigentliche Information für Zustandsüberführung und Ausgabe in einer einzigen Funktion `delta-lambda` mit zwei Ausgabeports. Für viele Zwecke benötigt man aber genau die Delta-Funktion oder nur die Lambda-Funktion des Automaten. Dazu enthält der Automat in den Slots `delta` resp. `fsm-lambda` ein Objekt vom Typ `fun-projection`, welches das allgemeine Protokoll für Funktionen einhält und dem Benutzer des slots `delta` resp. `fsm-lambda` eine normale Funktion vom Typ `two-into-one-fun` vorspiegelt, mithilfe seines Verweises auf die `delta-lambda` Originalfunktion und ohne redundante Informationsverdopplung.

"cast-net"

Initialisierungsroutinen für Systeme vom Typ Netzwerk.

5.4.3.3. Domain-spezifische Schicht für Endliche Automaten

Der domain-spezifische Teil des CAST-Kernel für die Behandlung von FSMs, Schaltwerken, Schieberegistern etc. gliedert sich in die folgenden Teile (Unterverzeichnisse):

Directory "partitions"

Dieser Teil von CAST-FSM sollte vielleicht noch zu den BASICS gerechnet werden. Partitionen werden hier nämlich auf allgemeingültige Art behandelt - die Spezialisierungen für die Zwecke der Verbandsberechnungen sind im "fsm"-Teil enthalten.

"classes" Klassendefinitionen für Partitionen und geordnete Partitionen.

"macros" Die Implementierung der in der File "partitions" definierten Operationen mithilfe von Makros und einfachen Lispfunktionen sowie je nach Plattform mit externen C-Funktionen findet sich in dieser (unglücklich benannten) File. Außerdem enthält sie einige globale Variablen und Konstanten.

"partitions" Hier sind die wesentlichen Rechen- und Vergleichsopertionen für Partitionen zu finden: `add` resp. `add+` für die Addition von zwei resp. mehreren Partitionen, `mult` resp. `mult+` für die Multiplikation von zwei resp. mehreren Partitionen, `compare` und `ip-equal` zum Vergleichen. Es gibt außerdem noch eine Art Division von Partitionen, d.h. eine Umkehroperation zur Multiplikation: Zu zwei gegebenen Partitionen p1, p2 wird diejenige Partition p3 gesucht, für die gilt: mult(p2, p3) = p1.

Die Konstruktoren zum Anlegen neuer Partitionen sind `make-part-of-enumsets`, der eine Liste von disjunkten Teilmengen und die Grundmenge benötigt, und der zum Testen praktische `make-integer-partition`, der eine Liste von Listen mit natürlichen Zahlen zu einem Partitionsobjekt macht.

Innerhalb komplexer Algorithmen werden neue Partitionen eher direkt durch Aufruf von `make-instance` erzeugt. Dann müssen die Instance-Variablen (slots) nachträglich gesetzt werden. Mindestens der Slot `element-array` muß einen gültigen Wert haben. Für Hash-Zugriff im Verband muß auch `element-str` gesetzt sein (dieser Slot wird als Schlüssel in Hash-Tabellen verwendet). Die Datenstruktur von `element-array` ist nicht einfach, aber sehr günstig für die Verbandsberechnung. Die Stelle 0 des Arrays enthält die Anzahl der Blöcke der Partition. Die Stelle i (i = 1 ... n) (die Grundmenge enthalte n Elemente) enthält die Nummer des Blocks, zu dem das Element i gehört. Die Blocknummer ist dabei die Nummer desjenigen Repräsentanten des Blockes, der die kleinste Nummer hat. Beispiel: Die Partition ((1 7) (8) (2 5 6) (3 4)) hat das `element-array`: #(4 1 2 3 3 2 2 1 8) , d.h. 4 Blöcke mit den Repräsentanten (Blocknummern) 1, 2, 3, 8. Die Umkodierung von Elementnamen zu Nummern (in diesem Beispiel die Identität) wird wie überall in CAST-FSM durch die zugrundeliegende Menge (`set-of-elements`) und den Funktionen `num` und `nam` ermöglicht.

Weitere Hilfsfunktionen sind `card-of-max-block` (wieviele Elemente liegen im größten Block?), `no-of-blocks` (wieviele Blöcke hat die Partition?), `get-block` (liefert zu einem gegebenen Elementnamen den Block, zu dem das Element gehört) und `name-element-list` (liefert die Listenschreibweise einer Partition mit Elementnamen) und einige andere.

"ordered-partitions" Für die Schieberegisterrealisierung werden noch speziellere Partitionen benötigt, die nämlich zusätzlich eine Ordnung ihrer Blöcke einhalten. Der Konstruktor ist `make-ordered-part-of-enumsets`. Bei der Addition von geordneten Partitionen bleibt die Ordnung der Blöcke erhalten, bei der Konstruktion mit `make-ordered-part-of-enumsets` wird die Ordnung der Argumentliste übernommen. Die Multiplikation kann die Ordnung natürlich nicht erhalten, liefert aber wieder ein Objekt der Klasse `ordered-partition`.

Directory "fsm"

"defvar" Globale Variable und Konstanten für "fsm".

"macros" Makros `do-fsm-int-inputs` und `with-all-fsm-inputs`, die die Syntax der Iteration über alle Inputs (resp. Input-Cubes) in Automaten-Algorithmen von der Datenstruktur des Automaten (`fsm` resp. `fsm-large`) unabhängig machen.

"classes" Alle Klassendefinitionen dieses Teils von CAST-FSM.

"fsm-basics" Die Funktionen `q-next` und `output` liefern für die drei Argumente FSM, Zustand, Input den nächsten Zustand (Delta) und den Output (Lambda). (Zustand, Input und Funktionswerte sind `nams`.)

Die Funktionen `get-q-next` und `get-output` liefern für die drei Argumente FSM, Zustandscode, Inputcode den nächsten Zustand (Delta) und den Output (Lambda). (Zustandscode, Inputcode und Werte sind `nums` resp. `cubes`.)

Weitere Funktionen dienen der Initialisierung und der schrittweisen Definition von Delta und Lambda (z.B. `insert-in-delta-lambda` etc.) und dem Test auf verschiedene einfache Eigenschaften.

"new", "primitives" Weitere einfache Funktionen für Automaten.

"fsm" Weitere Prädikate, und die Hauptfunktionen zur Verbandsberechnung: `make-lattice` erzeugt die Datenstruktur mit `create-lattice`, und führt dann die Berechnung mit `calculate-lattice` aus. Die Berechnung kann vom Benutzer mit der Return-Taste vorzeitig beendet werden, da eine Berechnung des gesamten Verbandes sehr rechenaufwendig sein kann, bzw. sogar unmöglich (Beispiel "scf.kiss2"). Andererseits läßt sich mit einem teilweise berechneten Verband eventuell bereits etwas anfangen. Auch wenn die Berechnung des Verbandes mit GUI-Befehlen gestartet wurde, ist sie mit der Returntaste abbrechbar. Eine Fortsetzung der Berechnung kann (derzeit) allerdings nur durch Aufruf von `continue-make-lattice` im Lisp-Listener erreicht werden.

Auch die alten Funktionen `batch` und `batch-large`, die einen Automaten erst einlesen (als `fsm` bzw. `fsm-large`) und dann seinen Verband berechnen, sind hier enthalten.

"fsm-large" enthält einige Unterschiede (Spezialisierungen) der Klasse `fsm-large` zu `fsm`.

"fsm-basic-kongs" Der Hauptteil der Verbandsberechnung ist die Methode `basic-kongs-complete` in dieser File.

"fsm-lattice" Hilfsfunktionen für `output-consistency`

"lattice" Initialisierungsmethoden für Verbände. Die `initialize-instance :after` Methode kann als Beispiel für die Garbage-freie Konstruktion neuer Partitionen dienen. Die darin verwendeten Funktionen (`make-item`, `make-element-array`) sind ebenfalls in dieser File enthalten, sowie allgemeine Zugriffsfunktionen udgl.

Wichtige Hilfsfunktionen für die Verbandsberechnung in dieser File sind `add-etp-do-addition`, `add-to-lattice`, `convert-etp`, `add-etp`. Eine etp ist eine einfache Datenstruktur (ein Vektor), die bei der Basiskongruenzberechnung erzeugt wird und mit `convert-etp` in ein partition-Objekt umgewandelt wird.

"set-system-lattice-mixin" und "set-system-fsm-mixin" Methoden für Verbands- bzw. Automaten-spezifische Operationen auf Partitionen.

"fsm-partition" Tests und Print-Funktionen für Kongruenzrelationen von Automaten.

"browser-viewable" Methoden für die CLIMT-Darstellung von Verbänden, insbesondere `display-in`.

"fsm-output-consistency" Hauptfunktionen für `output-consistency`

"fsm-div" Die Funktionen "`/-r`" resp. "`/-oc`" für die Bildung eines r-Quotienten resp. eines reduzierten Automaten aus einem gegebenen Automaten und einer Kongruenzrelation resp. einer Output-konsistenten Kongruenzrelation. Diese sind durch zwei Methoden von `divide` implementiert. Etliche Hilfsfunktionen stellen die erforderlichen Abbildungen von Zuständen des Originalautomaten zu Zuständen der Quotientenmaschine (die wiederum Blöcken der Kongruenzrelation entsprechen) her. Die Funktion `make-/-simulator` kann als Hauptfunktion zum Dividieren von Automaten verwendet werden, sie erzeugt als Ausgabe ein Objekt vom Typ Simulator und trägt den Quotienten und die zugehörigen Abbildungen entsprechend ein.

"fsm-decomp" Hauptfunktionen der "Algebraischen Strukturtheorie der Endlichen Automaten": Parallelzerlegung mit zwei oder mehr Kongruenzrelationen, Serienzerlegung mit einer Kongruenzrelation und einer (optional anzugebenden oder automatisch berechneten) Partition, Parallelzerlegung mit (wahlweise automatischer) Berechnung einer Serienergänzung, falls das Produkt der angegebenen Kongruenzen nicht Output-konsistent ist.

"fsm-q-splitting" Methode zum heuristischen Aufspalten eines Zustandes in zwei neue Zustände. Es wird der Zustand aufgespalten, in den die meisten Überführungen zielen. Die Heuristik kann durch optionale Argumente abgeschaltet werden.

"foreigns" Enthält eine Foreign-Function-Definition für Lucid CL der Hilfsfunktion `conv`, und eine Lisp-Definition von `conv` für Procyon CL. Diese Funktion erzeugt aus einem Array einen String. Die Anwendung ist das schnelle Auffinden von neu berechneten Partitionen im halbfertigen Verband durch Hash-Technik. (In MCL und allen anderen neuen Lisp-Umgebungen nach [CtLII] bzw. ANSI-Standard wird `conv` nicht benötigt, da man hier Arrays als hash-key verwenden kann, mit "`:key #'equalp`" bei `make-hash-table`.) - Weiterhin sind Lisp-Definitionen ehemaliger Foreign-Functions für die Berechnung kompatibler Zustände enthalten.

"null+count-tests" Erzeugung von Testautomaten (überflüssig, seit die internationalen Benchmarkautomaten "mcnc" zur Verfügung stehen).

"intermed" Provisorium als Ersatz für richtige Ports (benannte Eingänge und Ausgänge für Schaltwerke).

Directory "fsm-extensions"

Die Funktionen dieses Teils von CAST-FSM sind ausführlich in dem Bericht "FSM-Extension: Implementation of System Transformations for CAST-FSM*" von F. Auernig dokumentiert [Auernig93].

"fsm-finite-memory" Berechnung der Länge des Gedächtnisses eines Automaten, der eine Instance der Klasse `fsm` sein muß (nicht `fsm-large`), und keine don't-cares haben darf.

"fsm-invert" Berechnung der Verlustlosigkeit von Automaten. Berechnung von invertierten Automaten aus gegebenen Automaten endlicher Verlustlosigkeit. Anwendung auch für kryptologische Zwecke.

"fsm-misc" Enthält u.a. die Funktion `output*`, die der Funktion λ* der Automatentheorie entspricht (Erweiterung der Ausgabefunktion auf Worte).

"fsm-moore" Berechnung eines Moore-Automaten aus einem gegebenen Mealy-Automaten, und das Prädikat `moorep`, welches überprüft, ob ein gegebener Automat vom Typ Moore ist, d.h. ob der Output nur vom Zustand und nicht vom Input abhängt.

Anmerkung: In CAST:FSM wurden Moore-Automaten durch eine eigene Klasse dargestellt. Da der Unterschied zwischen Moore- und Mealy-Typ in der Systemtheorie keine so große Bedeutung hat, wurde in CAST-FSM keine eigene Datenstruktur mehr für den (spezielleren) Moore-Fall verwendet. In den Fällen, wo der Unterschied eine Rolle spielt, kann mit dem Prädikat `moorep` unterschieden werden.

"fsm-test" Nur für Automaten von der Klasse `fsm` (nicht `fsm-large`) können die Synchronisierfolgen, die Homingexperimente und die Diagnoseexperimente berechnet werden. Die Variable `*verbose*` bestimmt, ob der Fortgang der Rechnung im interaktiven Fenster (Listener) angezeigt wird. Das Ergebnis besteht aus zwei Listen, eine enthält die Zustände, für die Experimente gefunden wurden, die zweite enthält für jeden Zustand eine (wiederum listenförmige) Beschreibung des Experiments. - Die Funktion `test-experiments` liefert als ersten Wert die Anzahl der testbaren Transitionen, als zweiten Wert diese Transitionen. Wenn `*verbose*` auf true gesetzt ist, wird der Prozentsatz der testbaren Transitionen ausgegeben, der ein Maß für die funktionale Testbarkeit des Automaten darstellt.

Directory "file-interface"

Dieser Teil von CAST-FSM ermöglicht das Einlesen von Objekten der wichtigsten Systemtypen von externen Textfiles in die interne Darstellung, und umgekehrt das Abspeichern von Systemen in der internen Darstellung auf externe Textfiles. Um eine Weiterbearbeitung mit anderen Werkzeugen zu erleichtern, wird als Format der textuellen Darstellung das "Berkeley Standard File Format" für PLAs verwendet. Dieses wird etwa von den an der Abteilung für Systemtheorie vorhandenen VLSI-Werkzeugen "Octtools" ebenfalls verwendet.

"read-jedi+kiss" enthält den Reader für textuell dargestellte Systeme: die Funktion "`load-system`". `load-system` liest zunächst den Header der File und bestimmt (teilweise unter Verwendung von Heuristiken) den Systemtyp, d.h. die Klasse, des anzulegenden Objektes. Das

optionale Argument `:class-name` erlaubt aber auch das Einlesen einer File in ein Objekt einer bestimmten Klasse.

Wenn das optionale Argument `:remove-reset` (nur für FSMs) angegeben ist, werden alle Produktionen in den im Header mit der .r-Syntax deklarierten Reset-Zustand überlesen, d.h. in der internen Repräsentation des Automaten weggelassen. Wenn das optionale Argument `:extract-reset` (nur für FSMs) angegeben ist, werden nur die Produktionen in den im Header mit der .r-Syntax deklarierten Reset-Zustand eingelesen, d.h. in die interne Repräsentation des Automaten aufgenommen. Diese Optionen sind für die Anwendung gewisser Systemalgorithmen interessant - ein Beispiel ist die Schieberegisterzerlegung, die für Automaten mit einem typischen Reset-Zustand (d.h. für einen bestimmten Input, eben den Reset, werden die meisten Zustände in ein und denselben Zustand überführt) kaum gute Resultate liefern kann. Behandelt man dagegen das Reset-Verhalten in einer separaten Logik, kann der eigentliche FSM oft in eine günstige Schieberegisterstruktur gebracht werden.

"write-cast-objects" Funktion `write-to-file` (ruft auf: `write-to-kiss2-file`)

"write-files" Die Funktionen `write-to-kiss2-file` und `write-to-yedi-file` haben Methoden für die wesentlichen Systemtypen in CAST-FSM. Sie öffnen zunächst eine neue Textfile und schreiben dann die drei Komponenten einer Berkeley-PLA-Beschreibung: Den Header, die Produktionen, und das Ende-Zeichen. Bei einer Erweiterung von CAST-FSM müßten für die neuen Systemtypen entsprechende Methoden geschrieben werden.

Directory "shiftreg"

Der CAST-FSM-Teil "shiftreg" enthält die Algorithmen für die Schieberegisterzerlegung nach Böhling. Die Algorithmen wurden noch für spezielle Fälle erweitert, um auch da noch brauchbare Ergebnisse zu liefern, wo die einfache Böhlingmethode aufgibt. Insbesondere für nicht 1-3-vollständige Automaten ist noch eine Schieberegisterzerlegung möglich.

Die gesamte Schieberegisterzerlegung besteht aus vielen einzelnen Schritten, die jeweils über diverse Argumente zu steuern sind. Insbesondere verwenden einige Schritte Heuristiken, die für die meisten Fälle die optimale Lösung finden, aber in Sonderfällen möglicherweise ein suboptimales Resultat haben könnten. Durch explizites Aufrufen dieser Schritte mit Argumenten, die die eingebauten Heuristiken ersetzen, ließen sich dann bessere Ergbnisse erzielen. Für die Auswahl der sr-ind-Strukturen steht außer der heuristischen/exhaustiven Auswahlfunktion "`auto-select`" noch die interaktive Variante "`user-select`" bereit. Hier wird dem Benutzer die schrittweise Auswahl der sr-inds in einem Dialog ermöglicht.

Für einfache Fälle steht auch eine Funktion "`do-sr-real`" zur Verfügung, die den ganzen Systemalgorithmus zur Schieberegisterzerlegung enthält.

Für eine ausführliche Dokumentation dieses wichtigen Teils von CAST-FSM kann auf die Arbeit [Scharinger92] verwiesen werden. Auch der Quellcode dieses Verzeichnisses ist ausführlich kommentiert (allerdings in Deutsch).

Directory "additions"

Hier sind Zusatzfunktionen zu finden, die aus Kompatibilitätsgründen nicht oder noch nicht in die entsprechenden Teile der Basics oder des Kernels einsortiert wurden, aber für die Verwendung mit bestimmten Plattformen gebraucht werden. Insbesondere Ergänzungen der Standardmethoden für die macCASTfsm Applikation sind hier gespeichert. Als Beispiel sei die Namensgebung herausgegriffen. Jedes CAST-Objekt *kann* einen Namen haben. macCASTfsm erwartet aber, daß jedes CAST-Objekt wirklich einen Namen hat. Entsprechende `:after-` Methoden, die evtl. unbesetzte Namensfelder mit Default-Namen belegen, sind in "additions" enthalten.

5.4.3.4. Domain-spezifische Schicht für kryptologische Untersuchungen an Zufallszahlengeneratoren

Directory "crypto"

Für die Simulation von Pseudozufallsgeneratoren und die Evaluierung ihrer kryptologischen Qualität gibt es ein anderes, noch sehr rudimentäres CAST-System, genannt CAST-CRYPTO. Es handelt sich dabei im wesentlichen um einen in CLOS geschriebenen Simulator für bestimmte Schieberegisterstrukturen, sowie um eine Sammlung von statistischen und visuellen Analysewerkzeugen, die teilweise auf die "Kryptologische Methodenbank" (eine FORTRAN Programmsammlung aus den Siebzigern der Abteilung für Systemtheorie und Informationstechnik) zurückgeht.

Der Analyseteil von CAST-CRYPTO ist auch als Zusatz in die macCASTfsm Umgebung nachladbar. Dieser Zusatz heißt CAST-CRYPTO.fasl und ist eine kompilierte Procyon-CL-File. Ebenso können die Lisp-Files des "source"-Verzeichnisses "crypto" auch in die portablen Versionen von CAST-FSM geladen werden, wobei dann aber nur die statistischen Tools genutzt werden können (die Grafik ist derzeit noch in Common Graphics geschrieben). Ausgabefolgen von I/O-Experimenten von CAST-Objekten wie Automaten und Schieberegistern können dann mit den statistischen und visuellen Analysewerkzeugen untersucht werden.

5.4.4. CAST User Interface

5.4.4.1. User Interface in Standard Common Lisp

Wird der CAST-Kernel (Source-Files oder compilierte Files) in eine Common Lisp Umgebung geladen, wird der Lisp Listener als Command-Line-Interface verwendet. Die elementarsten Lisp-Funktionen und die gesamten Methoden des Kernel stellen den Befehlsvorrat dar. Die etwas ungewöhnliche, aber im Grunde sehr einfache, weil mathematisch orientierte, Lisp-Syntax wird zur Eingabe einfacher Kommandos verwendet. Datenstrukturen von beliebigen Objekten in dieser Umgebung können mit den typischen Hilfsmitteln der Lisp-Umgebung (Inspector, Editor, Browser, ...) angeschaut und verändert werden. Als Beispiel könnte die Schieberegisterrealisierung eines Automaten namens "planet" mit der folgenden Befehlsfolge interaktiv durchgeführt werden:

(Die Benutzereingabe folgt auf den Prompt ">", die Rückgabewerte stehen in *kursiv*.)

```
> (setq val 2)
2
> (setq depth 100)
100
> (setq fsm (load-system "planet"))
#<FSM-LARGE #x580419>
> (setq transition-desc (analyze-fsm fsm))
#<TRANSITION-DESCRIPTOR #x5D2639>
> (multiple-value-setq (bas-sr len-limit disc-limit) (basic-sr-inds transition-
desc))
(#<SR-IND #x607D19> #<SR-IND #x6014C9> ...)
> (print-bas-sr-characteristics bas-sr)
...
> (setq bas-sr (find-relevant-sr-inds bas-sr fsm val))
(#<SR-IND #x5EF419> #<SR-IND #x5EF3B1>)
> (print-bas-sr-characteristics bas-sr)
...
> (setq comb-sr-inds (auto-comb bas-sr val depth transition-desc))
(#<SR-IND #x5B1F69> #<SR-IND #x5AF679> ...)
> (improve-sr-inds comb-sr-inds transition-desc)
(#<SR-IND #x5B1F69> #<SR-IND #x5AF679> ...)
> (setq sr-ind-fam (auto-select comb-sr-inds :mode nil))
(#<SR-IND #x5A7C51> #<SR-IND #x5B1F69> ...)
> (descr-fam sr-ind-fam)
...
> (setq sr-realization (compute-srm-simulator fsm sr-ind-fam))
#<SIMULATOR #x5DBCB9>
> (setq function-block (make-fun-block (sim-obj sr-realization)))
#<HASH-FN #x5ED4C9>
> (write-to-file function-block
         (concatenate 'string (name fsm) "-sr-fun-block") :directory
         *dump-file-directory* :format :kiss))
#P"host:volume:cast:dump-directory:planet-sr-fun-block"
```

Bild 5.9 Lisp Listener als Kommandozeileninterpreter

5.4.4.2. User Interface in CLIMT

Wird das Programmsystem CAST-Kernel in eine CLIMT-Umgebung geladen, steht zusätzlich zu den in Common Lisp vorhandenen Möglichkeiten ein Browser zur Verfügung, der besonders für die Darstellung von Automatenverbänden geeignet ist. Auch sehr große Verbände können genau untersucht werden, da Ausschnittvergrößerungen (Zoomen) und Verschieben leicht möglich sind. Bestimmte Wege im Verband können bequem verfolgt werden, indem alle Kanten, außer den zu und von ausgewählten Knoten führenden, ausgeblendet werden. Produkte und Summen von Knotenmengen können angezeigt werden, ebenso die outputkonsistenten Partitionen. Schließlich kann die Verbandsberechnung eines Automaten "animiert" ausgeführt werden, d.h. es wird der jeweils bereits berechnete Teil des Verbandes in einem Fenster gezeigt und immer wieder aktualisiert. Dadurch hat der Benutzer die Möglichkeit, die Berechnung in dem Moment zu unterbrechen, in dem genügend viele "interessante" Kongruenzen gefunden wurden, um bestimmte weitere Schritte mit diesen durchzuführen.

Falls diese nicht zu den gewünschten Resultaten führen, kann die Berechnung des Verbandes dann an der Stelle der Unterbrechung fortgesetzt werden. Durch den enormen Freiheitsgrad der vollen Lisp-Umgebung kann der "eingeweihte" Benutzer sogar während einer solchen Unterbrechung die Knotenbeschriftung im Browser ändern, um die neu gefundenen Kongruenzen auf ein anderes Kriterium absuchen zu können, oder ein automatisches Abbruchkriterium einführen. Auf diese Weise lassen sich auch sehr große Automatenverbände in CAST-FSM behandeln.

Zu Dokumentationszwecken können die Graphiken von CLIMT-Browsern in Postscript-Dateien übersetzt und dann auf einem geeigneten Drucker ausgegeben werden.

5.4.4.3. User Interface in Common Graphics (Procyon Common Lisp)

Die bisher einzige vollständige graphische Oberfläche für den CAST-Kernel ist in Common Graphics geschrieben und deshalb an Procyon Common Lisp (auf Macintosh oder OS/2) oder Allegro CL/PC (auf Windows PCs) gebunden.

Die systemtheoretischen Objekte (Mengen, Funktionen, Automaten, Schieberegister, ...) werden durch Icons (kleine graphische Elemente) angezeigt. Diese graphisch repräsentierten Systeme können durch Anklicken mit dem Zeigegerät (Maus) selektiert werden. Auf selektierte Systeme können dann Menukommandos angewendet werden. Es gibt auch ein paar globale Kommandos, die sich nicht auf ein selektiertes System beziehen. Dazu zählt der Befehl "Load System", der das Laden eines Systems von der externen Repräsentation (Textfile) in die interne Darstellung bewirkt. Das sichtbare Ergebnis der Operation ist dann das neue Icon auf dem Bildschirm.

Normalerweise arbeitet man mit Realisierungsgraphen (oder Lösungsgraphen, siehe 4.2.). Das sind Graphen, die die Ergebnisse von Syntheseoperationen als Knoten enthalten, durch Kanten mit dem ursprünglichen System verbunden. Implementiert sind Realisierungsgraphen als eine spezielle Art Browser, sie dienen als wichtigstes Element der Befehlseingabe und Ergebnisanzeige der CAST-Oberfläche [Mitt90-3] [Präh90-1]. Auch hier werden Systeme durch Icons dargestellt, und können mit der Maus selektiert werden. Zur bequemen Auswahl von Kommandos werden systemspezifische Popup-Menus verwendet. Ein vom Benutzer gesteuerter Systemalgorithmus wird einen im wesentlichen baumartigen Graphen ergeben, dessen Blätter die Lösung des bearbeiteten Problems darstellen (vgl. Kap. 4).

Eine andere Art von Browser dient zur Darstellung des Verbandes der Kongruenzrelationen eines Automaten. Hier kann ein Ausschnitt ausgewählt werden, und die Verbandsoperationen können auf einer ausgewählten Teilmenge von Kongruenzen ausgeführt werden. Auch die outputkonsistenten Partitionen können angezeigt werden. Für große Automatenverbände steht die Scroll-Funktion zur Verfügung. Außerdem können auf bequeme Weise Unterverbände in eigenen Fenstern geöffnet werden.

Noch spezieller ist der Browser für die Darstellung des V-Graphen eines Automaten, der die für die Schieberegisterzerlegung nach Böhling interessanten Schieberegisterpartitionen enthält. Da der V-Graph nur ein (sup-) Halbverband ist, aber gerade das i.a. nicht im Verband liegende

Produkt der in einer Zeile nebeneinanderliegenden Partitionen zur Qualitätsbeurteilung wichtig ist, wird dieses Zeilenprodukt hier immer neben der Zeile angezeigt.

Schließlich gibt es noch einen Browser, der die CLOS Klassenhierarchie anzeigt und mit Hilfe des Procyon Inspectors auch den Zugriff auf die inneren Strukturen einer selektierten Klasse sowie der auf die Klasse spezialisierten Methoden erlaubt. Dieser Klassenbrowser ist für Anwender sinnvoll, die den CAST-Kernel in einer vollen Lisp-Entwicklungsumgebung laufen lassen und verändern oder erweitern wollen.

Die Bemerkungen zur dynamischen Änderbarkeit und Erweiterbarkeit aus 5.4.4.2. gelten sinngemäß auch hier. Allerdings erfolgt ein Programmieren von zusätzlichen Funktionen dann in der normalen Lisp-Programmierumgebung, nicht in der graphischen Oberfläche. In Procyon Common Lisp kann man beides problemlos nebeneinander benutzen.

5.4.4.4. User Interface in CLIM

5.4.4.4.1. CLIM-GUIs

Im zukünftigen Standard für Lisp GUIs "CLIM" gibt es derzeit nur eine sehr rudimentäre Oberfläche. Der Source-Code ist dabei weitgehend plattformunabhängig, d.h. echt portabel. Gewisse Restabhängigkeiten sind auf Fehler oder Unzulänglichkeiten der jeweiligen CLIM-Implementierung zurückzuführen.

In CLIM erscheint nach dem Laden der Programme und dem Aufruf `(init-cast)` das CAST Hauptfenster. Von diesem aus kann ein System geladen werden. Das geladene System, i.a. ein Automat, wird in einem eigenen Fenster (oder einem eigenen Teil des Hauptfensters) angezeigt. Mit CLIM Kommando-Tasten kann dann der Verband angezeigt werden, oder der V-Graph. In beiden Verbandsarten kann gerechnet werden, und es können Partitionen selektiert werden, die dann einer globalen Variablen zugewiesen werden. Alles andere muß im Lisp Listener auf die übliche Art erledigt werden (siehe Punkt 4.1). Immerhin ist mit diesem simplen ersten Schritt in Richtung einer portierbaren Oberfläche gerade die wichtigste GUI-Funktionalität von CAST-FSM bereits vorhanden: die Examinierung von Verbänden, die im Kommandozeilen-Modus so schwierig ist.

5.4.4.4.2. West-Coast und East-Coast

Im Bereich der GUI-Entwicklung werden zwei grundsätzlich verschiedene Stile der Fenstertechnik unterschieden:

Im ersteren, der auch "west-coast-style" genannt wird, gibt es meist viele, überlappende Fenster auf dem Bildschirm. Innerhalb eines Fensters wird nur direkt zusammenhängende Information dargestellt, für jeden neuen Kontext wird ein neues Fenster geöffnet.

Der Gegensatz, der "east-coast-style", bevorzugt dagegen wenige große Fenster, normalerweise ein Fenster für eine Applikation, im Extremfall (Oberon-Workstation "Ceres") sogar überhaupt keine überlappenden Fenster. In diesem GUI-Stil werden verschiedene Kontexte statt in neuen Fenstern in Unterteilungen der vorhanden Fenster angezeigt. So könnte ein typisches Fenster etwa in die Bereiche ("panes") Kommando-Tasten/Menus, Texteingabe, Textausgabe, Grafikeditor, online-Hilfe und diverse Statusanzeigen eingeteilt sein.

CLIM ist im Prinzip auf keine bestimmte Art von GUI-Stil (man spricht hier auch vom "look-and-feel" einer Oberfläche) festgelegt, aber man darf doch anmerken, daß die "east-coast" Variante eindeutig bevorzugt wird, d.h. leichter zu implementieren ist. Die in Procyon Common Graphics geschriebene CAST-FSM Oberfläche ist dagegen eindeutig in der Macintosh-typischen "west-coast" Variante gehalten.

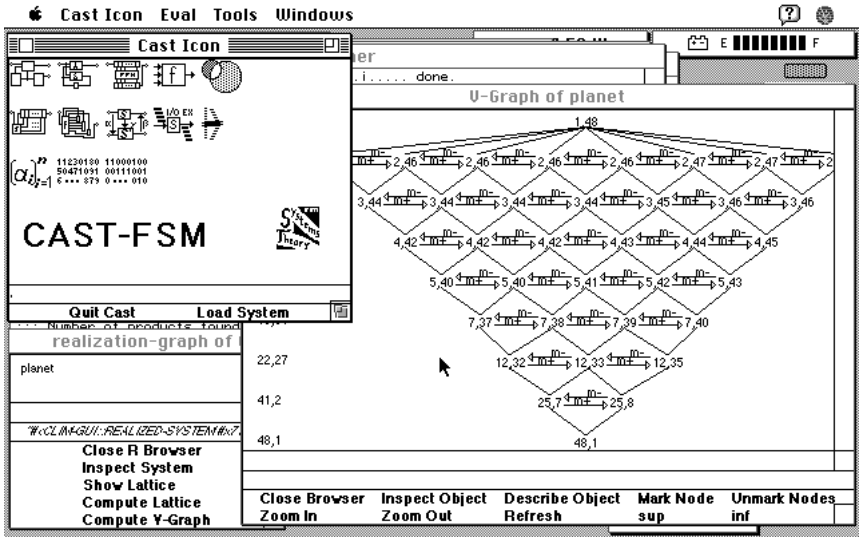

Bild 5.10 Experimentelles CLIM User Interface für CAST-FSM

5.4.5. CAST-FSM als Anwendungsprogramm

5.4.5.1. Macintosh Anwendung macCASTfsm

5.4.5.1.1. Anwendungsprogrammentwicklung mit Lisp

CAST-FSM ist, wie bereits dargestellt, in Lisp geschrieben. Im Gegensatz zu vielen anderen Programmiersprachen erzeugt der Compiler eines Lisp-Systems keine "stand-alone-application", d.h. keine unabhängigen Programme, die separat abgespeichert und dann vom Benutzer

direkt im Betriebssystem eines Rechners aufgerufen und verwendet werden können. Vielmehr sind die vom Programmierer geschriebenen Lisp-Programme, ganz gleich ob compiliert oder im Source interpretiert, nur in einer komplexen Laufzeitumgebung lauffähig. Normalerweise ist die Laufzeitumgebung eben genau die Lisp-Entwicklungsumgebung, in der die Programme auch entwickelt wurden. Sie enthält neben der eigentlichen Laufzeitumgebung noch die je nach Hersteller mehr oder weniger zahlreichen Werkzeuge für die Erstellung der Programme, das Testen, die Fehlersuche und -Behebung, die Examinierung komplexer Datenstrukturen, diverse Dokumentationswerkzeuge und Hilfesysteme, Leistungsmessungs- und -Optimierungswerkzeuge, etc., und natürlich den Compiler.

Soll aus einer Menge von Lisp-Programmen eine "stand-alone-application" erstellt werden, reicht es demnach nicht aus, die Programme selber zu schreiben. Es muß eine möglichst kleine Laufzeitumgebung erzeugt werden und in diese müssen die Programme geladen werden. Dann wird dieser Zustand des Systems evtl. noch optimiert und dann eingefroren. Dieses abgespeicherte "Bild" des Programmsystems kann dann als stand-alone-application verwendet werden. Dieser ganze Vorgang wird je nach verwendetem Lisp verschieden stark unterstützt. Je nach Lisp muß für jedes weitergegebene Exemplar einer so hergestellten Applikation eine Gebühr an den entsprechenden Lisp-Hersteller abgeführt werden, oder nur einmalig für eine Art von Applikation. Noch eine andere Variante wird von Procyon/Scientia verfolgt, hier zahlt man nur für die Möglichkeit, Applikationen zu erzeugen, aber keine weiteren Gebühren mehr.

Viele Teile der Entwicklungsumgebung wird man aus der Applikation gerne herauslassen, damit diese nicht zu groß wird. (Wirklich kleine Programme lassen sich mit Lisp ohnehin nicht realisieren.) Andere Teile *muß* man aus der Applikation herauslassen, weil der Hersteller sie nicht zur Weitergabe freigibt. Zu letzteren gehören meist der Compiler, und die meisten Tools für Fehlersuche/Beseitigung (Debugging).

5.4.5.1.2. Lisp-Interpreter

In macCASTfsm sollte die Möglichkeit zur Entwicklung eigener Systemalgorithmen bestehen, deshalb wurde ein Interpreter geschrieben, der den größten Teil der Sprache Common Lisp unterstützt. Ausgelassen wurden vor allem die Makro-Möglichkeiten und alle CLOS-Konstrukte (ersteres um den Aufwand zu begrenzen, zweiteres weil hierzu der Compiler bzw. Information über das Format compilierter Funktionen notwendig gewesen wäre, beides aus rechtlichen Gründen nicht möglich). Jedenfalls steht dem macCASTfsm-Benutzer eine sehr leistungsfähige Sprache zur Formulierung eigener Systemalgorithmen zur Verfügung. In der macCASTfsm-Distribution sind Beispiele für eigene Programme enthalten.

5.4.5.1.3. Inspector

Auch der Inspector von Procyon Common Lisp mußte aus der Applikation gestrichen werden. Ein einfacher Ersatz, der die Inspektion der meisten in CAST vorkommenden Datentypen erlaubt, wurde deshalb eingeführt. Mit diesem Werkzeug ist der Benutzer in der Lage, die "Innereien" der CAST-Objekte zu untersuchen, und im Extremfall sogar zu ändern. Natürlich sollte bei "normaler" Arbeit mit macCASTfsm ein solcher Einblick oder gar Eingriff in die inter-

nen Mechanismen der Implementierung nicht notwendig sein. Aber gerade im Zusammenhang mit der Entwicklung eigener Algorithmen ist derlei evtl. doch sinnvoll.

5.4.5.1.4. Initialisierung

Um aus einer Menge von Lisp-Programmen eine eigene Applikation zu machen, ist es schließlich noch notwendig, eine Intitialisierungsroutine zu schreiben, die beim Start des Programms den Menubalken einrichtet und die richtigen Fenster öffnet, sowie gewisse Datenstrukturen einrichtet (in macCASTfsm sind das vor allem die diversen Popup-Menus und Dialogfenster), und schließlich die sogenannte "main-event-loop" (also die ereignisverarbeitende Hauptschleife) startet.

5.4.5.1.5. Macintosh Look&Feel

Die Applikation macCASTfsm verwendet die unter 4.3 beschriebene Oberfläche mit den hier beschriebenen Erweiterungen, und ein eigenes Protokollfenster "CAST session protocol". Dieses Fenster, welches in Stichworten die vom Benutzer ausgewählten Menu-Operationen protokolliert, so daß man bei überraschenden Resultaten nachvollziehen kann, wie diese zustande kamen, dient vor allem als Ein/Ausgabefenster für den Lisp-Interpreter 5.1.2, das heißt als Medium für den Kommandozeilenverkehr mit dem Benutzer. Mit den Macintosh-typischen Operationen "Copy" und "Cut" ins Clipboard gestellte Objekte können im Protokollfenster mithilfe der Variablen "=" angesprochen werden. Die Resultate der letzten Evaluierungen können mit den globalen Common-Lisp-Variablen *, **, *** bzw. /, // , /// weiterverwendet werden, auch etwa im Inspector.

Insgesamt ergibt sich ein GUI-Stil, der dem Macintosh-Standard weitgehend entspricht [Mitt92]. Eine wesentliche, vom Autor aber für sinnvoll erachtete, Ausnahme ist die Verwendung von objekt-spezifischen Pop-Up-Menus im Realisierungsgraphen. Außerdem ist es in der derzeitigen Version noch nicht möglich, Grafiken (z.B. aus Browsern) über das Clipboard zu exportieren oder zu importieren. (Das globale Macintosh-Clipboard wird nur mit Text unterstützt.)

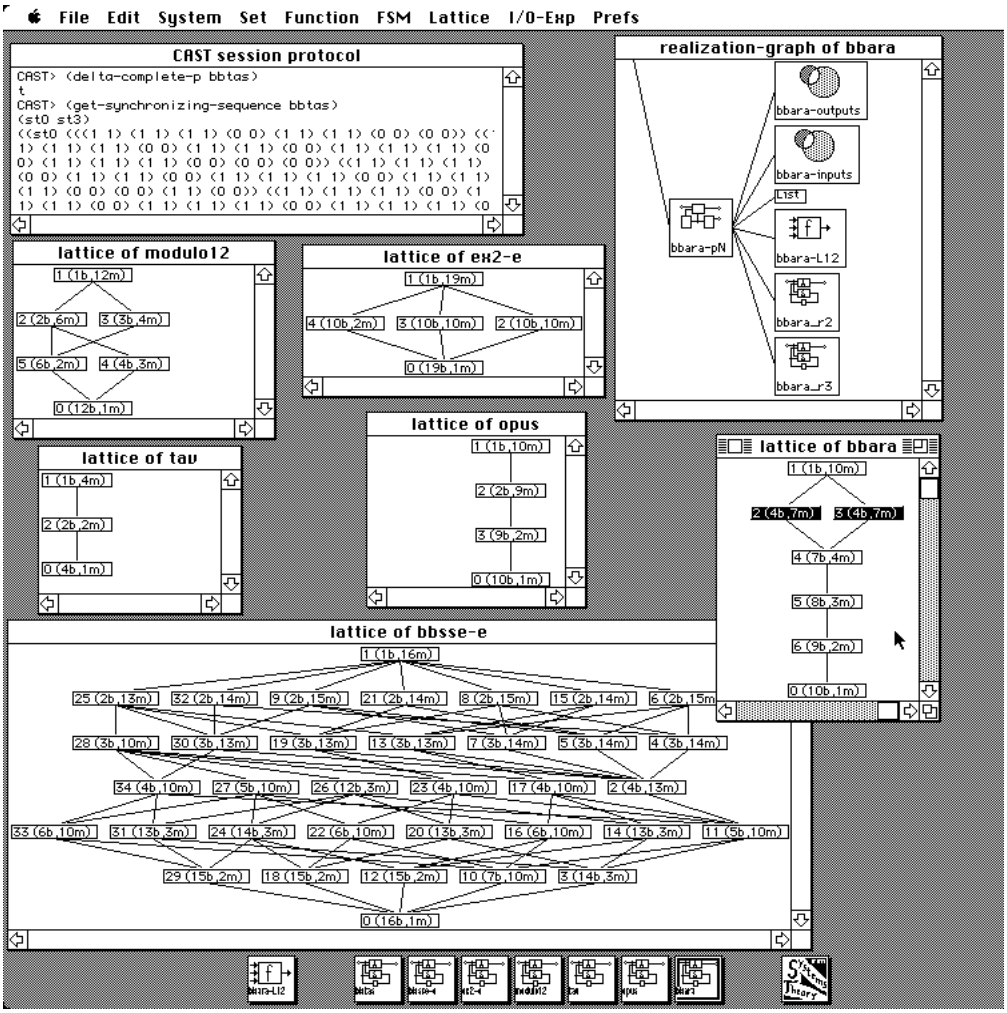

Bild 5.11 macCASTfsm - Algebraische Strukturtheorie, Verbände, Parallelzerlegung

Dieser Bildschirm-Schnappschuß von macCASTfsm zeigt mehrere Automatenverbände, und einen Ausschnitt eines Realisierungsgraphen nach einer durchgeführten Parallelzerlegung. Die für die Parallelzerlegung als Argumente verwendeten Kongruenzrelationen sind zuvor im Verband markiert worden (inverse Darstellung). - Die geladenen Automaten sind in der Icon-Leiste am unteren Rand repräsentiert. Im Protokoll-Fenster wurden zuletzt die Synchronisierfolgen eines Automaten berechnet.

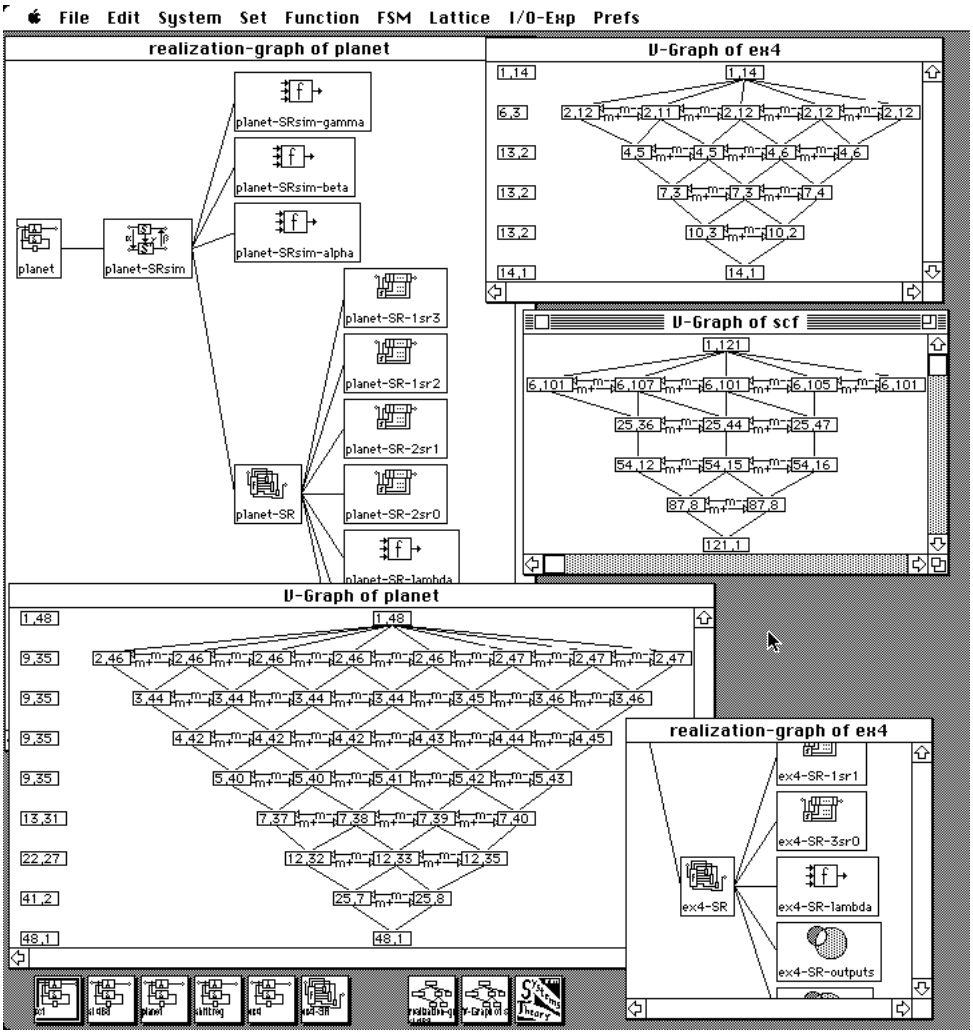

Bild 5.12 macCASTfsm - V-Graphen, Schieberegisterzerlegung

In diesem Schirmbild werden mehrere V-Graphen gezeigt. Zwei durchgeführte Schieberegisterrealisierungen sind in Realisierungsgraphen dargestellt.

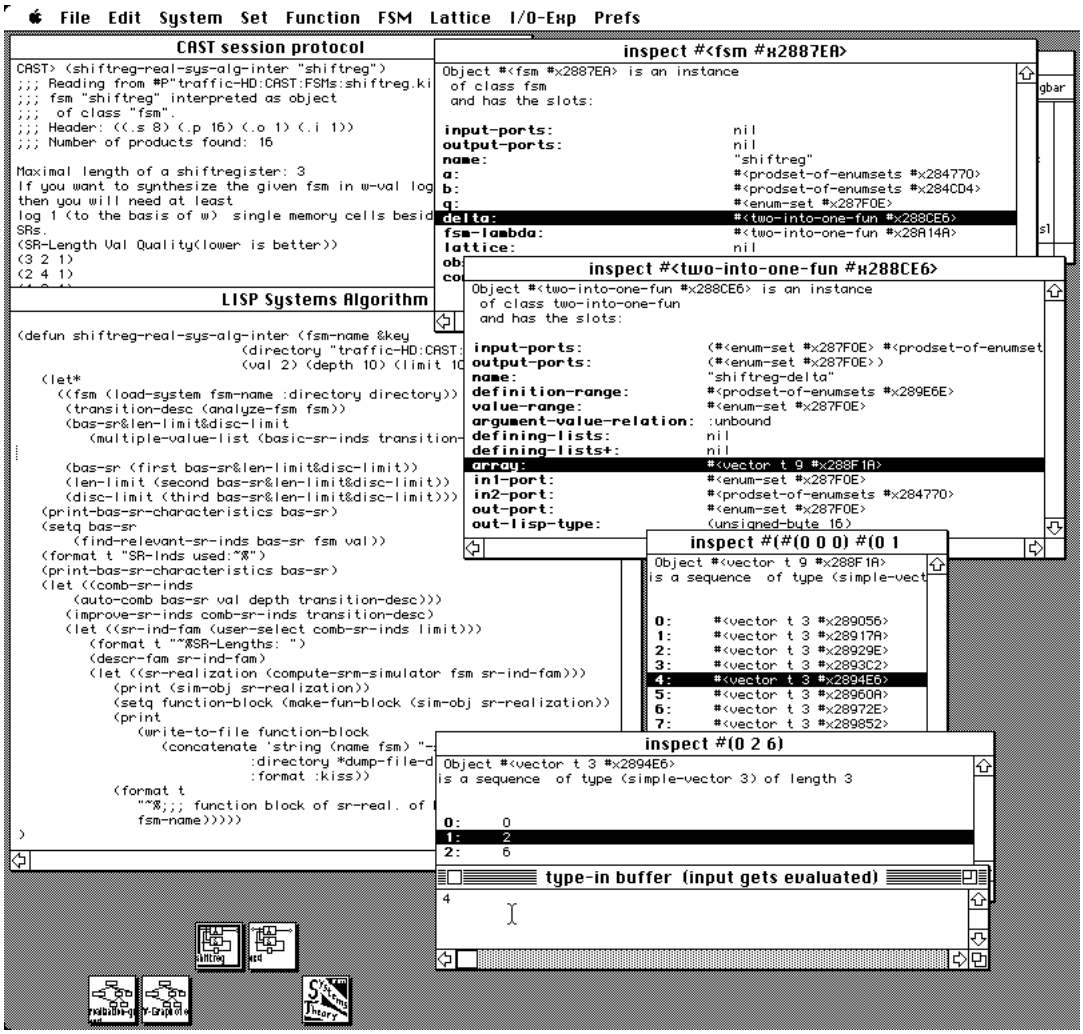

Bild 5.13 macCASTfsm - Systemalgorithmus-Entwicklung

In der Text-Datei "LISP Systems Algorithm" wurde ein Systemalgorithmus zur Schieberegisterrealisierung unter Verwendung der interaktiven Auswahl von sr-inds definiert. Im Protokollfenster sieht man den Aufruf dieser Benutzer-geschriebenen Erweiterung von macCASTfsm. Auf der rechten Bildhälfte sind mehrere Inspector-Fenster offen. Im Input-Buffer wird gerade ein Wert einer internen Datenstruktur überschrieben (=geändert - Vorsicht!).

5.4.5.2. CLIM-basierte Anwendung

Sobald die auf CLIM basierte Oberfläche für CAST-FSM einigermaßen funktionell und vollständig ist, kann eine eigenständige Applikation mit diesem neuen User Interface erstellt werden, und zwar auf mehreren Plattformen. Dies wird zunächst wieder der Apple Macintosh

sein, diesmal mit Macintosh Common Lisp, dann aber auch die Sun Sparc Workstation Familie mit Lucid Common Lisp oder Harlequin LispWorks.

6. Kapitel - Automatisierung des Problemlösungs - prozesses mit Systemalgorithmen

Ein erster Schritt zur Automatisierung von Problemlösungsvorgängen mit der STIPS-Maschine wird durch die Speicherung kompletter Systemalgorithmen erreicht.

6.1. Algorithmen zur Lösung von Systemproblemen

Jede Operation einer STIPS-Maschine entspricht der Anwendung einer systemtheoretischen Methode oder Systemtransformation. Die unter Umständen zahlreichen Schritte zur Lösung von bestimmten Systemproblemen müssen der STIPS-Maschine immer wieder als Input-Stimulus eingegeben werden. Eine naheliegende Idee ist daher die Zusammenfassung von Inputs zu Inputfolgen. Eine lineare Abfolge von Methodenanwendungen (wobei die Argumente implizit als die jeweils zuletzt erzeugten Systeme gewählt werden können) hilft bereits in einfachen Fällen. Input einer solchen Anordnung ist dann die Menge der Ausgangssysteme, sowie die Angabe einer Sequenz von Methoden.

Lassen sich diese Inputsequenzen abspeichern, so können wiederholte Problemstellungen (etwa Anwendung einer mehrschrittigen Synthese auf verschiedene Daten, d.h. Ausgangssysteme) einfach durch Auswahl der passenden Sequenz erledigt werden.

Im allgemeinen wird eine lineare Sequenz von Elementaroperationen aber nicht ausreichen, um eine komplexe Synthese oder eine komplexe Analyse von Systemen durchzuführen. Vielmehr ist die Verwendung von Kontrollstrukturen unumgänglich. Aus den Sequenzen werden dann Algorithmen. Da diese Algorithmen mit Hilfe von Systemtransformationen auf Systemen arbeiten, werden sie Systemalgorithmen genannt.

Eine Maschine STA, die eine Menge von Systemalgorithmen als Komponente ihres Inputs enthält, wird im folgenden beschrieben.

6.2. Die Maschine STA

Sei STIPSM(S_F, T_F) eine STIPS-Maschine für das Themengebiet F. Die Systemalgorithmen-Maschine STA-F ist gegeben durch ein n-Tupel (STIPSM(S_F, T_F), Alg_{FL}), wobei Alg_{FL} eine bel. Menge von Systemalgorithmen bezüglich F (in der formalen Sprache L) ist.

Die Maschine STA hat als Input einen Algorithmus a aus Alg_{FL} und eine Folge von Anfangs-Systemspezifikationen $s \in seq(ink^*(S_F))$. Zu diesem Input errechnet die Maschine STA dann den Output $a(s)$, der die gesuchte Lösung enthält.

Diese Berechnung der Lösung $a(s)$ geschieht in der folgenden Weise:

Der Algorithmus a wird Schritt für Schritt interpretiert, indem in jedem ausführbaren Schritt i = (m, s_i) an die in STA enthaltene STIPS-Maschine der Input (m, s_i) angelegt wird, wobei m die Methode des Schrittes ist, und s_i seine Argumentsystemfolge. Die Kontrollanweisungen wenden ihre Prädikate auf die errechneten Systeme an und verzweigen oder beenden die Abarbeitung.

Die Lösung $a(s)$ liegt dann in Form eines Realisierungsgraphen vor, dessen Systeme die gesuchten Eigenschaften haben.

6.3. Erstellung von Systemalgorithmen für STA

Einfache Systemalgorithmen lassen sich aus fertigen Realisierungsgraphen ableiten.

Zu einem konkreten Systemproblem P=(I,C,F) wird mit einer STIPS-Maschine die Lösung R(P) errechnet. Dieser Realisierungsgraph R(P) wird nun als Berechnungsvorschrift umgedeutet, indem jede Kante in einen ausführbaren Schritt mit einer eindeutigen Nummer und der Kantenbeschriftung als Methodenname transformiert wird.

Im allgemeinen sind die Elemente von Alg_{FL} als Algorithmen zu definieren, d.h. für die praktische Anwendung, in einer geeigneten Programmiersprache zu schreiben. Das Erstellen dieser Programme für STA setzt vor allem sehr gute Kenntnisse des von STA zu bearbeitenden Gebietes voraus. Es könnte hier schon der Begriff "Knowledge Engineering" verwendet werden.

Eine komfortablere Möglichkeit zur Erstellung von Systemalgorithmen, die dem Anspruch der CAST-Philosophie der leichten Benutzbarkeit besser entspräche, läge in der Verwendung von graphischen Programmierwerkzeugen. Durch die direkte Manipulierbarkeit von Ablaufdiagrammen am Bildschirm würde die Notwendigkeit, eine Programmiersprache zu erlernen, entfallen, und die Konstruktion und Verbesserung von Systemalgorithmen könnte dem Anwender überlassen bleiben.

Dieser Weg, die Anwendbarkeit der STA zu verbessern, wird hier nicht weiter verfolgt, da die wissensbasierte Steuerung einer STIPS-Maschine, wie sie in den nächsten Kapiteln dargelegt wird, erfolgversprechender erscheint.

6.4. Beispiele für fertige Systemalgorithmen

Drei einfache Systemalgorithmen wurden als Ergänzung zu der STIPS-Maschine "CAST-FSM" definiert. Alle drei sind kaum mehr als die jeweils entsprechende Systemtransformation, aber im Gegensatz zu dieser werden die zusätzlich benötigten Argumente hier mitberechnet, so daß beim Aufruf nur mehr das Ausgangssystem angegeben werden muß.

6.4.1. pdecomp-realizations

Der erste Systemalgorithmus berechnet Parallelzerlegungen eines Automaten.

Neben dem erforderlichen Argument (dem zu zerlegenden Automaten) können noch weitere Argumente optional mitgegeben werden:

```
(defmethod pdecomp-realizations ((fsm fsm) &key max-results (verbose *verbose*)
                                 (collect-results nil)
                                 (selection-fn #'random-select)
                                 (filter-fn #'identity))
        &body)
```

`:max-results` ist die maximale Anzahl der berechneten Zerlegungen. `:collect-results` ist ein Flag, welches angibt, ob die berechneten Ergebnisse gesammelt oder nur angezeigt werden sollen. `:verbose` gibt dem Anwender Einfluß auf die Ausführlichkeit der Protokollierung.

Die beiden wichtigen Argumente sind `:selection-fn` und `:filter-fn`. Auf die Menge der unvergleichbaren Kongruenzrelationen des Automaten wird zunächst die Filterfunktion angewendet, die unter dem Key `:filter-fn` mitgegeben wird. (Der Default "`identity`" läßt diese Menge unverändert.) Diese Funktion kann einerseits bestimmte, als von vornherein ungeeignet erkannte, Kongruenzenpaare ausfiltern (d.h. weglassen), und andererseits die verbliebenen Paare nach vermutlicher Qualität sortieren.

Eine bereits vorbereitete Filterfunktion ist `pdecomp-filter-fn`, die jedoch außer der Liste der unvergleichbaren Paare noch weitere Argumente benötigt, weshalb man sie bei der Verwendung als `:filter-fn` in einen `lambda`-Ausdruck einbetten sollte:

```
... :filter-fn #'(lambda (uncs)
                   (pdecomp-filter-fn uncs fsm no-of-states no-of-ffs verbose))
```

Dabei müssen die hier beispielhaft gezeigten Variablen so belegt sein: `fsm` ist der zu bearbeitende Automat, `no-of-states` die Anzahl seiner Zustände, `no-of-ffs` die Zahl der zur Realisierung mindestens notwendigen binären Speicherelemente (FlipFlops), und `verbose` wie oben angegeben.

Für die nach der Filterung verbleibenden Kongruenzenpaare werden die entsprechenden Parallelzerlegungen berechnet, dabei wird in dem Fall, daß das Produkt der beiden Relationen nicht outputkonsistent ist, automatisch eine Serienergänzung hinzugefügt. Die Berechnung weiterer Zerlegungen stoppt, wenn (1) alle Relationenpaare verbraucht sind oder (2) die maximale Anzahl gewünschter Ergebnisse (`:max-results`) erreicht ist. Dabei bestimmt die Funktion, die für das Keyword-Argument `:selection-fn` mitgegeben wurde, welches Relationenpaar als nächstes aus der Menge der noch nicht verwendeten Paare hergenommen wird. Zwei sinnvolle Funktionen stehen bereits zur Verfügung, nämlich "`random-select`", die ein zufälliges Paar aussucht, und "`first`", die das jeweils erste Paar der (geordneten) Restmenge aussucht. Natürlich steht es dem Anwender immer frei, eigene Funktionen zu übergeben.

6.4.2. sdecomp-realizations

Der zweite Systemalgorithmus berechnet Serienzerlegungen eines Automaten.

Wieder können neben dem erforderlichen Argument (dem zu zerlegenden Automaten) die oben beschriebenen optionalen Argumente mitgegeben werden:

```
(defmethod sdecomp-realizations ((fsm fsm) &key max-results (verbose *verbose*)
                                 (collect-results nil)
                                 (selection-fn #'random-select)
                                 (filter-fn #'identity))
    &body)
```

Hier ist zu beachten, daß die Filterfunktion `:filter-fn` mit der Menge der nichttrivialen Kongruenzrelationen aufgerufen wird, statt mit den unvergleichbaren Paaren.

Eine mögliche Filterfunktion für sdecom-realizations ist:

```
:filter-fn
#'(lambda (parts)
    (let ((result
           (loop for part in parts
                 when (not (oc-p part))
                 collect part)))
      (sort result #'<
            :key #'(lambda (part)
                     (+ (* (no-of-blocks part)
                           (no-of-blocks part))
                        (* (card-of-max-block part)
                           (card-of-max-block part)
                           2))))
      ))
```

Diese Filterfunktion eliminiert zunächst alle outputkonsistenten Relationen aus der Menge der Kandidaten, und sortiert diese dann nach (z.B. quadratisch gewichteten) erforderlichen Zustandszahlen in Kopf - und Schwanzautomat.

6.4.3. shiftreg-realizations

Der dritte Systemalgorithmus berechnet Schieberegisterzerlegungen bzw. Zustandskodierungen eines Automaten.

Neben dem erforderlichen Argument (dem zu zerlegenden Automaten) können noch weitere Argumente optional mitgegeben werden:

```
(defmethod shiftreg-realizations ((fsm fsm) &key max-results (verbose *verbose*)
                                  (collect-results nil)
                                  (selection-fn #'first)
                                  (filter-fn #'identity)
                                  (depth 100)
                                  (mode :prefer-few-cells)
                                  (val 2))
  (declare (ignore selection-fn filter-fn))
  &body)
```

Dabei gibt `:depth` die Suchtiefe innerhalb des Schieberegisteralgorithmus an, und `:mode` steuert die Heuristik der Suche: mögliche Werte sind `:prefer-few-cells` und `:prefer-few-regs`. Im ersteren Fall wird die Zahl der insgesamt benötigten Registerzellen minimiert, im zweiten Fall die Anzahl der Schieberegister innerhalb der gesamten Schieberegisterstruktur. Das Argument `:val` gibt die Wertigkeit der Logik, d.h. die Anzahl der Zustände pro Zelle an, für normale zweiwertige Flipflops ist zwei anzugeben (der Default).

(Die weiteren Argumente `:selection-fn`, `:filter-fn` und `:max-results`, die die Reihenfolge und Anzahl erzeugter verschiedener Realisierungen steuern, sind in dieser Version ohne Bedeutung, da die Methode hier ohnehin nur *eine* Realisierung berechnet.)

6.5. Eine einfache STA

Der Zustand einer STIPS-Maschine ist als Realisierungsgraph beschreibbar. Wendet man die Systemalgorithmen einer STA auf ein Ausgangssystem an, wird die Änderung des Zustandes der zugrundeliegenden STIPS-Maschine in einer Erweiterung ihres Realisierungsgraphen sichtbar.

Eine einfache STA ohne besondere Benutzerschnittstelle, die direkt auf eine Lisp-Realisierung der STIPS-Maschine CAST-FSM zurückgreift, soll noch kurz vorgestellt werden.

Zu den drei auf Automaten anwendbaren Systemalgorithmen aus 6.4. werden "Hüllen" zur Verfügung gestellt, die direkt auf Knoten (nodes) des Realisierungsgraphen arbeiten. Es sind dies die drei Funktionen:

```
(defgeneric pdecomp-transformation (real-node &key selection-fn)
  (:documentation "parallel decomposition in realization graph"))

(defgeneric sdecomp-transformation (real-node &key selection-fn)
  (:documentation "serial decomposition in realization graph"))

(defgeneric sr-real-transformation (real-node)
  (:documentation "shiftregister realization in realization graph"))
```

Beispiel:

Der Zustand der STA sei gegeben durch folgendes Diagramm:

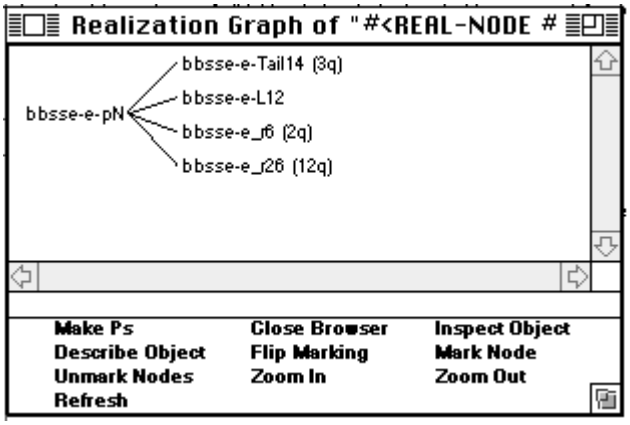

Bild 6.1 STA vor Anwendung des Systemalgorithmus Serienzerlegung

Dann wird der Systemalgorithmus Serienzerlegung aufgerufen:

```
(mapcar #'sdecomp-transformation (get-all-nodes-of 'fsm *g*))
```

Der neue Zustand der STA ist dann:

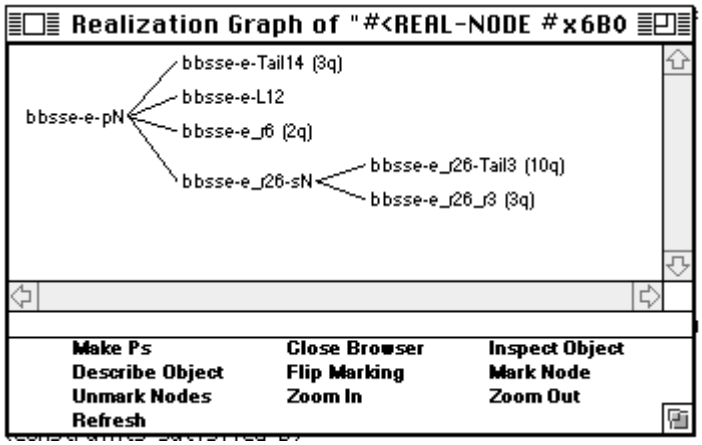

Bild 6.2 Zustand der STA nach Ausführung des Algorithmus

Vorhandene Algorithmen können zu neuen, komplexeren kombiniert werden. Die einfachste Art der Kombination ist die Hintereinanderausführung, in Lisp durch die `progn`-Form ausgedrückt. Im nächsten Beispiel werden zunächst alle Knoten serienzerlegt, danach in Schieberegisterstrukturen transformiert:

```
(progn
  (mapcar #'sdecomp-transformation (get-all-nodes-of 'fsm *g*))
  (mapcar #'sr-real-transformation (get-all-nodes-of 'fsm *g*))
)
```

Der Zustand der STA wird dann durch folgenden Realisierungsgraphen angegeben:

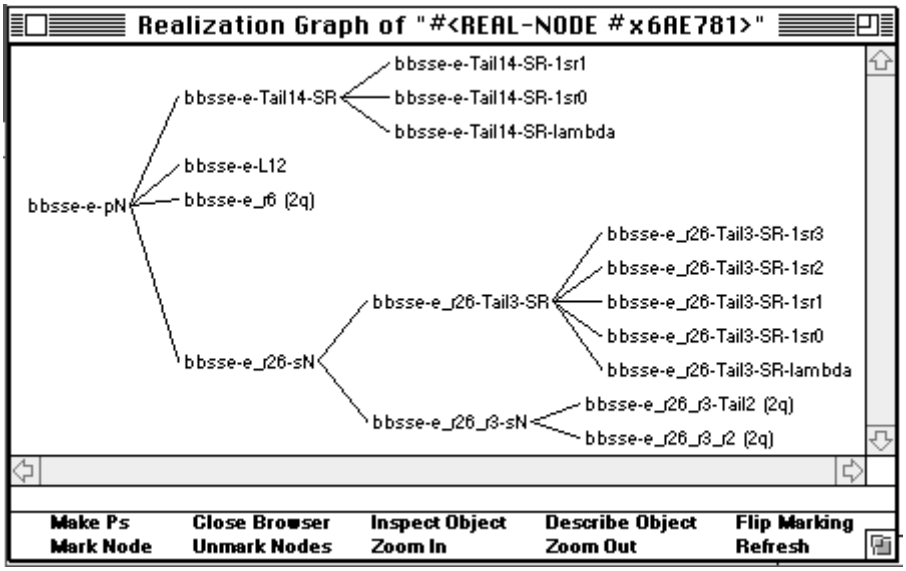

Bild 6.3 Endzustand der STA

7. Kapitel - Wissensbasierte Systeme als Programmierparadigma

7.1. Wissensbasierte Methoden

In den Siebzigern wurden in den amerikanischen Forschungslaboratorien für Künstliche Intelligenz sogenannte Expertensysteme entwickelt, die die Aufgabe hatten, einen menschlichen Experten in einem bestimmten Fachgebiet zu ersetzen oder zumindest zu unterstützen. Die Beschränkung auf ein eng begrenztes Fachgebiet kann als Reaktion auf die enttäuschenden Resultate mit allgemeinen Problemlösungsmaschinen wie GPS [Newe72] der Sechziger verstanden werden. Berühmt gewordene frühe Beispiele für Expertensysteme sind MYCIN, XCON, DENDRAL [Barr81], [McCo79]. In den Achtzigern fanden Expertensysteme Eingang in die Praxis. Obwohl die ursprünglich hohen Erwartungen an die Fähigkeiten solcher Systeme im Lauf der Jahre immer weiter zurückgesteckt wurden, gibt es heute unzählige erfolgreiche Anwendungen z.B. in der Industrie [Cara93], in denen Expertensysteme verwendet werden, oft ohne daß der Anwender des Systems überhaupt davon weiß.

Die Architektur typischer Expertensysteme unterscheidet sich wesentlich von der Struktur konventionellerer EDV-Programme.

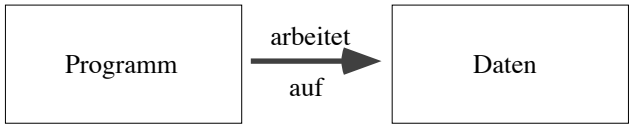

Bild 7.1 "Herkömmliche" EDV-Programmstrukur

Während in letzteren zwischen Daten und Programmen unterschieden wird, oder anders ausgedrückt zwischen Datenstrukturen und Algorithmen, trennt man bei Expertensystemen drei Hauptkomponenten: Daten, Regeln, und Inferenzmechanismus. Letzterer ist ein allgemeines Ableitungsschema, welches vorgelegte Fragestellungen mit Hilfe der vorhandenen Regeln und Daten zu beantworten versucht. Die Daten sind wie bei konventioneller Software Informationen über das konkrete Problem und den Zustand der Abarbeitung. Die Regeln sind die eigentliche Besonderheit dieser Konstruktionen, in ihnen ist das Wissen über ein bestimm-

tes Fachgebiet (domain) in deklarativer Weise enthalten. Aus programmiertechnischer Sicht sind die Regeln also auch nichts anderes als (spezielle) Daten, also keine Programme.

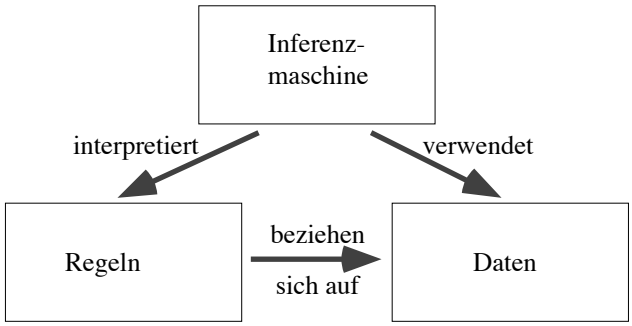

Bild 7.2 Programmstruktur eines Expertensystems

Soll das Wissen eines Expertensystems erweitert, geändert oder korrigiert werden, müssen nur Regeln zur Regelbasis hinzugefügt oder verändert werden. Der Algorithmus, der das in den Regeln gespeicherte Wissen auf konkrete Fälle anwendet, nämlich die Inferenzmaschine, ist von solchen Änderungen nicht betroffen. Andererseits kann der Inferenzmechanismus verbessert werden, ohne daß die das Fachwissen enthaltene Regelbasis verändert werden müßte. Will man ein erfolgreiches Expertensystem für ein neues Anwendungsgebiet verwenden, muß das Expertenwissen in Regeln ausgedrückt ("knowledge engineering") zu einer neuen Regelbasis zusammengefaßt werden. Dies kann zur "Laufzeit" des Systems passieren, ein Compiler/Entwicklungssystem zum Neuübersetzen der Algorithmen der Inferenzmaschine ist hierfür nicht notwendig (Vorteil deklarativer Programmiertechnik).

Obwohl Expertensysteme als "angewandte Künstliche Intelligenz" zunächst als etwas völlig anderes als übliche Software vermarktet wurden, und die Erstellung eines bestimmten Expertensystems für eine bestimmte Anwendung als etwas völlig anderes als Programmieren hingestellt wurde, zeigte sich doch bald, daß hier im Grunde nur ein neuer Programmierstil vorlag. Diesen Programmierstil bezeichnet man heute als "regelbasiert" oder "wissensbasiert". Das Wort Expertensystem ist inzwischen wieder in den Hintergrund getreten.

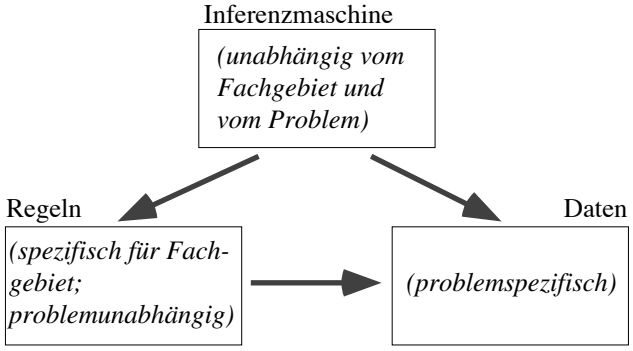

Bild 7.3 Abhängigkeit von Anwendungsgebiet und Problem

Diese Dreiteilung der prinzipiellen Architektur von Expertensystemen ist demnach der wesentliche Unterschied zu herkömmlicheren Systemen. Es ergeben sich einige Konsequenzen, die in Vor- und Nachteile gegliedert werden können:

Vorteile regelbasierten Programmierstils:

- ▲ Verbesserungen der Inferenzmechanismen betreffen das in den Regeln gespeicherte Fachwissen im Idealfall nicht.
- ▲ Verbesserungen, Erweiterungen und Anpassungen der Repräsentation des Fachwissens hat im Idealfall keine Auswirkungen auf Daten und Inferenzmaschine.
- ▲ Neue Elemente des Fachwissens werden als zusätzliche Regeln in die Regelbasis aufgenommen, ohne den Bestand verändern zu müssen. Im Gegensatz dazu müssen in konventionellen Programmen schon bei kleinen, trivialen Änderungen der Spezifikation oft ganze Module neu geschrieben werden. Hier klingt auch der Unterschied zwischen softwaretechnischem Top-Down-Programmieren einerseits und explorativem Programmieren oder "Rapid Prototyping" andererseits an.
- ▲ Durch Mitverfolgung (Tracing) der Aufruffolge der Regeln zur Laufzeit kann sehr leicht eine Erklärungskomponente angeboten werden, die die "Entscheidungen" des Expertensystems gegenüber dem Anwender plausibel machen und so die Akzeptanz der Verwendung des Systems erhöhen kann.
- ▲ Aufgrund von verschiedenen Mechanismen sind regelbasierte Systeme erfolgreicher als konventionelle bei der Verarbeitung "unsicheren" Wissens. Viele Expertensysteme enthalten in der Regelsprache Notationen für Unsicherheitsfaktoren, Fuzzy-Mengenoperatoren, o.ä. Obwohl die Funktionsweise solcher Systeme sich der theoretischen Fundierung weitgehend entzieht, sind sie doch in einigen Anwendungsfällen äußerst erfolgreich. Besonders Systeme zur Steuerung und Regelung komplexerer mechanischer Geräte, deren Behandlung mit exakten Methoden einen untragbar hohen Aufwand erfordern würden, sind hier als Beispiel zu nennen.

Nachteile regelbasierten Programmierstils:

- ▼ Die meisten Ansätze für Inferenzmechanismen bieten nur wenig oder keine Unterstützung für komplexere Abläufe. Programmierer behelfen sich durch Einführung von Hilfsdatenstrukturen zur Zustandsspeicherung in der Datenbasis und durch Einführung von Hilfsregeln oder Hilfsbedingungen in normalen Regeln. Dadurch wird die strikte Trennung von Inferenz, Wissensbasis und Daten unterlaufen und deren Vorteile gehen teilweise verloren.
- ▼ Die Wartbarkeit größerer Regelbasen ist, entgegen den ursprünglichen Erwartungen, sehr aufwendig und fehleranfällig. Dies hängt vor allem damit zusammen, daß die Korrektheit bzw. Fehlerfreiheit bei solchen Systemen noch viel schwieriger zu ermitteln ist als bei konventionelleren Systemen.

Die ersten wissensbasierten Systeme waren durchwegs in Lisp geschrieben. Die in Europa entwickelte und in Japan favorisierte Programmiersprache Prolog wurde dann ebenfalls für solche Anwendungen eingesetzt. Im Grunde ist jedes Prolog-Programm bereits regelbasiert, da die

elementare prädikatenlogische Ableitungsmaschine bereits in Prolog selbst enthalten ist und ein Prolog-Programm nur mehr aus (logischen, deklarativen) Regeln besteht. Allerdings ist diese Sicht für die Praxis ohne große Bedeutung, da die Prädikatenlogik als Regelsprache für echte Anwendungen zu einfach ist. Expertensysteme in Prolog verwenden die Logik-Regeln daher zur Konstruktion eines komplexeren Regelinterpreters, der dann erst die eigentlichen, domain-spezifischen Regeln abarbeitet.

Heute wird das wissensbasierte Programmierparadigma vielfach nur noch für bestimmte Komponenten komplexer Softwaresysteme eingesetzt, während andere Teile konventionell in imperativen Sprachen wie C oder C++ programmiert werden. Zur Vereinfachung der softwaretechnischen Methodik wird daher auch oft schon der regelbasierte Teil komplexer Anwendungen in solchen prozeduralen, imperativen Sprachen geschrieben [Giar94], entweder direkt oder mithilfe entsprechender Bibliotheken (z.B. in CLIPS).

7.2. Navigation bei der Lösungssuche im Zustandsraum

Die bekannten Programmierparadigmen der Künstlichen Intelligenzforschung, die sich auf die Lösungssuche symbolisch gestellter Probleme beziehen, werden vorgestellt und auf ihre Eignung für die Automatisierung der STIPS-Methode untersucht.

7.2.1. Symbolische Datenverarbeitung

Es gibt einige Ansätze zur symbolischen Problemlösung in der Artificial Intelligence, die unter verschiedenen Namen eng zusammenhängende Konzepte verfolgen. Dazu gehören allgemeines Problemlösen, Suchalgorithmen, logisches Programmieren, regelbasiertes Programmieren, Wissensverarbeitung, Produktionsregelsysteme, etc. [Norv92] [Nils80]

> Andere Gebiete der Künstlichen Intelligenz, wie neuronale Netze und Mustererkennung, spielen da u.U. auch eine Rolle; die Zusammenhänge werden von verschiedenen Forschungsgruppen untersucht (z.B. [McCl88], [Rume86]), sollen aber hier unberücksichtigt bleiben.

Beim "symbolischen Rechnen" werden die Objekte, die zu behandeln sind, durch Symbole dargestellt. Das Wort Symbol meint hier ein Wort einer bestimmten formalen Sprache, in der Praxis einfach ein zulässiger Name der verwendeten Programmiersprache. Das Symbol steht für ein bestimmtes reales Objekt, es gibt also eine semantische Deutung, die aber dem symbolisch rechnenden System, hier also der Inferenzmaschine, nicht bekannt ist [Born86]. Für dieses System unterscheiden sich die Symbole ausschließlich durch die Zeichenfolge, durch die sie dargestellt sind, die Bedeutung als Verweis auf irgendetwas ist ihm völlig unbekannt. In jeder anderen Hinsicht sind also die Symbole untereinander austauschbar.

Der berühmte Mathematiker Hilbert hat diese Austauschbarkeit auch für die in der Mathematik verwendeten Namen gefordert, und damit auf den völlig formalen und somit gewissermaßen "inhaltsleeren" Charakter dieser reinen Wissenschaft hingewiesen [Herm63]. Der Mathematiker und Philosoph Russel hat dies so ausgedrückt: "... mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true" [Herm63].

> Wenn in einem Expertensystem für die Diagnose von Automotordefekten überall das Symbol "Motor" gegen das Symbol "Fahrersitz" ausgetauscht würde, würde sich das Verhalten des Systems in keiner Weise ändern - unter der Voraussetzung, daß bei allen Interaktionen mit dem Benutzer diese Vertauschung rückgängig gemacht würde, denn er benötigt diesen semantischen Bezug natürlich auf jeden Fall. Das Expertensystem dagegen braucht den Bezug nicht und hat ihn auch nicht.

Die verschiedenen Arten von regelbasierten Systemen (ein anderer gängiger Begriff ist Produktionsregelsystem) lassen sich u.a. in folgende Arten einteilen: Vorwärtsverkettete Systeme, rückwärtsverkettete Systeme und bidirektionale Systeme [Nils80][Tour86][Kana88].

(Manche Autoren reservieren den Begriff Produktionsregel für vorwärtsverkettete Regeln. In diesem Text soll die Unterscheidung lieber explizit angegeben werden [Norv92].)

Die drei wesentlichen Teile eines Produktionsregelsystems werden im folgenden mit Inferenzmaschine, Regelbasis und Datenbasis bezeichnet.

Bei der folgenden Aufgliederung muß beachtet werden, daß hier nur die Enden des Spektrums der Möglichkeiten herausgestellt werden können. In der Praxis gibt es viele Mischformen, die mehrerer Eigenschaften der drei prinzipiellen Varianten miteinander kombinieren.

7.2.1.1. Vorwärtsverkettete Regelsysteme

Vorwärtssysteme gehen von den Anfangsbedingungen aus und versuchen durch Anwendung von Regeln zur Lösung des Problems zu gelangen.

Bei vorwärtsverketteten Systemen enthält die Datenbasis Beschreibungen des Anfangszustandes und des aktuellen Zustandes des Problemlösungsprozesses. Im Sinne der Definitionen in Kapitels 3 wäre das die Menge der Ausgangssysteme I, sowie die Resultate der Anwendung von Systemtransformationen in den Schritten der Regelabarbeitung.

Die Regeln eines vorwärtsverketteten Systems sind von der typischen IF-THEN-Gestalt, d.h. sie bestehen aus einem Bedingungsteil und einem Ausführungsteil. Der Bedingungsteil (IF-Teil) enthält ein Prädikat, welches auf dem Inhalt der Datenbasis definiert ist. Ist das Prädikat für den aktuellen Inhalt der Datenbasis erfüllt, ist die Regel anwendbar, andernfalls ist sie im aktuellen Zustand nicht anwendbar. Für den Fall, daß in einem Zustand mehrere Regeln anwendbar sind, halten die verschiedenen Inferenzmaschinen unterschiedliche Strategien zur Konfliktlösung bereit. Hier gibt es viele Möglichkeiten: Beispielsweise kann einfach die erste Regel der (sortierten) Regelbasis ausgeführt werden. Das bekannte Regelsystem OPS5 verwen-

det statt dessen ein Prioritätsschema. Außerdem wird in diesem System eine Regel nach ihrer Anwendung zurückgestellt, so daß sie nicht wieder angewendet werden kann, um ein Steckenbleiben in Zyklen zu vermeiden. In einer anderen Strategie werden alle, oder möglichst viele, der anwendbaren Regeln parallel angewendet. Dabei muß der Fall berücksichtigt werden, daß durch die Anwendung einer Regel die Vorraussetzung für die Anwendung einer anderen Regel aus dem Konfliktsatz ungültig wird, was wieder auf verschiedene Arten behandelt werden kann.

Der THEN-Teil einer vorwärtsverketteten Regel enthält eine Aktionsvorschrift, die bei der Anwendung der Regel dann durchgeführt wird. Diese Aktion bewirkt meist per Seiteneffekt eine Änderung der Datenbasis. Dabei können zwei Arten von Aktionen unterschieden werden: Die eigentlichen Nutzeffekte bewirken eine Berechnung o.ä., in einer STIPS-Anwendung etwa die Berechnung einer Systemtransformation. Die zweite Art von Aktionen dagegen manipuliert nur irgendwelche Variablen, die in den Prädikaten der Regeln vorkommen und keinen anderen Sinn haben, als den Anwendbarkeitsstatus der Regeln der Regelbasis zu ändern, und somit den Interferenzprozeß zu steuern. Regeln, die hauptsächlich die Interferenz beeinflussen, werden auch Metaregeln genannt.

Durch übertriebenen Gebrauch solcher "Meta"-Regeln werden die Vorteile des regelbasierten Ansatzes zunichtegemacht (siehe die Nachteilliste in 7.1.). Andererseits sind die bekannten Interferenzmechanismen nicht leistungsfähig genug, ganz ohne Metaregeln auskommen zu können.

Man könnte eventuell noch eine dritte Art von Aktionen definieren, nämlich solche, die weder mit der Objektmanipulation noch mit der Ablaufsteuerung direkt zu tun haben, sondern die der Kommunikation mit dem Benutzer dienen, oder die den History-Mechanismus oder die Erklärungskomponente unterstützen. Diese Regeln werden hier aber außer acht gelassen, weil sie nicht typisch sind für Vorwärtssysteme. Sie werden in 7.2.2. behandelt.

7.2.1.2. Rückwärtsverkettete Regelsysteme

Rückwärtsverkettete Systeme gehen von der Zieldefinition aus. Sie brechen das Ziel in kleinere Unterziele ("subgoals"), die ihrerseits solange weiter zerlegt werden, bis ein solcher Pfad die Ausgangssituation trifft, in welchem Fall das Problem als positiv gelöst betrachtet wird, oder bis bewiesen ist, daß eine Lösung nicht möglich ist.

Rückwärtsverkettete Systeme haben daher eher einen existentiellen als einen konstruktiven Charakter. Trotzdem sind solche Systeme, die auch "automatische Beweiser" genannt werden, auch für konstruktive Aufgaben zu verwenden. Während die typische Frage an einen Beweiser etwa wäre: "Gibt es eine Schaltwerk-Realisierung des vorgegebenen Verhaltens mit 4 Speicherzellen?", und die Antwort ein lapidares "Ja.", kann durch entsprechende Formulierung der Regeln ein Existenzbeweis erzwungen werden, der dann die gesuchte Realisierung liefert.

Die Datenbasis eines rückwärtsverketteten Systems enthält die jeweils neu gefundenen, noch zu lösenden Unterprobleme. In vielen Rückwärtssystemen werden diese Unterziele aber nicht explizit in einer Datenbasis gespeichert, sondern erscheinen nur auf dem Stack (Kellerspeicher)

der Interferenzmaschine. Dies ist etwa auch bei dem bekanntesten rückwärtsverketteten System der Fall, nämlich der Programmiersprache PROLOG.

Die Regeln eines Rückwärtssystems haben die Form: Wenn das aktuelle Teilproblem von der Form xyz ist, dann ist es bewiesen (gelöst), wenn die Unterprobleme abc, def, ... bewiesen (gelöst) werden können. Dabei ist xyz ein formaler Ausdruck, der von dem Beweiser (Inferenzmechanismus) mit den schon abgeleiteten, aber noch zu lösenden Teilproblemen der Datenbasis verglichen wird. Die Leistungsfähigkeit solcher Systeme hängt daher stark von den Eigenschaften und Möglichkeiten der Vergleichsalgorithmen ab. Einfache Mustererkennung reicht hier nicht, Verallgemeinerungen wie "Unifikation" sind notwendig. Typischerweise sind die beiden Teile der Rückwärtsregeln in derselben Syntax anzugeben, im Gegensatz zu den Vorwärtsregeln des vorigen Abschnitts.

7.2.1.2.1. Beispiel: Testmusterberechnung für einen Automaten mit einem rückwärtsgekoppeltem Regelsystem

Als ein Beispiel für die Anwendung regelbasierten Programmierens auf dem Gebiet der Automatentheorie soll hier die Errechnung eines Testmusters für einen eingebetteten Automaten gezeigt werden [Mitt88-2][Zein87][Zaji88].

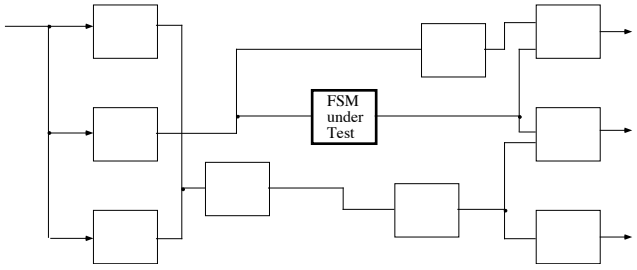

Bild 7.4 Test eines eingebetteten Automaten

Die Problemstellung ist folgende: Ein Automat s ist Teil eines größeren Schaltwerkes. Dieses wird hier als nicht-zyklisch angenommen. Daher kann die Struktur der Gesamtschaltung ohne wesentliche Beschränkung der Allgemeinheit als Serienschaltung von drei Automaten aufgeteilt werden:

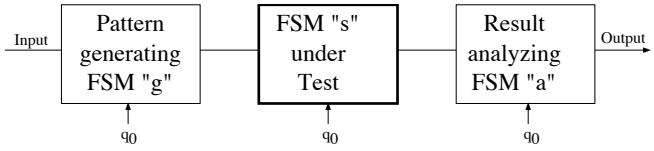

Bild 7.5 Vereinfachte Testumgebung mit FSMs g, s, a

Der Input der Gesamtschaltung geht in den Eingangsautomaten g, dessen Ausgang wirkt auf den Eingang des zu untersuchenden Automaten s, dessen Ausgang ist auf den Eingang des Ausgangsautomaten a geschaltet. Der Automat a gibt dann die Ausgabe der Gesamtschaltung ab.

Die hier verwendete Syntax der Regeln ist auf geklammerte Listen aufgebaut, da dies für die hier verwendete, in Lisp geschriebene Interferenzmaschine am einfachsten ist. Die Syntax ist sehr einfach:

(rulesyn 1) Jede Regel wird durch eine Liste (siehe rulesyn 4-3) angegeben.

(rulesyn 1-1) Diese Liste besteht aus der Folgerung als erstem Element, und allen (implizit UND-verknüpften) Vorraussetzungen als weiteren Elementen.

(rulesyn 2) Eine Regel ohne Vorraussetzungen beschreibt einen Fakt, d.h. ist immer wahr.

(rulesyn 3) Folgerungen und Vorraussetzungen haben die gleiche Syntax wie folgt:
Sie bestehen aus einer Liste, deren erstes Element ein Prädikatsname (Regelname) ist, und deren weitere Teile die Argumente des Prädikats sind.

(rulesyn 4) Argumente eines Prädikats können sein: Konstanten, Variablen und Listen

(rulesyn 4-1) Konstanten sind Namen aus alphanumerischen Zeichenkombinationen wie in Programmiersprachen üblich. (Das erste Zeichen darf kein "`?`" sein.) Auch Zahlen werden hier als Konstanten behandelt.

(rulesyn 4-2) Namen, die mit einem Fragezeichen "`?`" beginnen, werden als Variablen behandelt.

(rulesyn 4-3) Listen beginnen mit "(" und enden mit ")". Listenelemente werden durch Leerräume getrennt. Ein einzelner, durch Leerzeichen vorne und hinten abgegrenzter Punkt in einer Liste hat eine besondere Bedeutung. Der hier nur interessante Fall ist von der Form `(?x . ?y)`, wobei dann die Variable `?x` das erste Element der Liste darstellt, und `?y` den Rest.

Wenn die Liste `(a b c d)` gegen die Liste `(?x . ?y)` verglichen wird, wird `?x` an `a` gebunden, dagegen wird `?y` an die Restliste `(b c d)` gebunden.

Die Automaten werden in Regelform angegeben, in der Form:
`(dl Automat Input State NextState Output).`

Jede Transition eines Automaten wird also als eine Regel geschrieben, insgesamt machen alle diese Regeln für alle hier definierten Automaten das globale Prädikat "`dl`" (für delta-lambda) aus.

Hier folgen die Regeln für die drei Automaten `g`, `a`, und `s`:

```
((dl g 0 1 2 0))
((dl g 1 1 3 1))
((dl g 0 2 1 0))
((dl g 1 2 4 1))
((dl g 0 3 2 1))
((dl g 1 3 3 0))
((dl g 0 4 1 1))
((dl g 1 4 4 0))

((dl a 0 1 2 0))
((dl a 1 1 3 1))
((dl a 0 2 1 0))
((dl a 1 2 4 1))
((dl a 0 3 2 1))
((dl a 1 3 3 0))
((dl a 0 4 1 1))
((dl a 1 4 4 0))

((dl s 0 1 6 0))
((dl s 1 1 3 1))
((dl s 0 2 5 0))
((dl s 1 2 4 1))
((dl s 0 3 6 1))
((dl s 1 3 3 0))
((dl s 0 4 5 1))
((dl s 1 4 4 0))
((dl s 0 5 2 0))
((dl s 1 5 7 1))
((dl s 0 6 1 0))
((dl s 1 6 8 1))
((dl s 0 7 2 1))
((dl s 1 7 3 0))
((dl s 0 8 1 1))
((dl s 1 8 8 0))
```

Die Automaten `g` und `a` haben also 4 Zustände, der Automat `s` hat 8. Die folgenden Regeln der Regelbasis bilden die wichtigsten Prädikate.

Zuerst die Zustandsüberführungs- und Ausgabefunktionen für Buchstaben und Worte (Worte werden als Listen von Buchstaben dargestellt):

```
((step ?m ?in ?oldstate ?newstate ?out)
   (dl ?m ?in ?oldstate ?newstate ?out))
((step* ?m nil nil ?initstate))
((step* ?m (?in . ?inrest) (?out . ?outrest) ?initstate)
   (dl ?m ?in ?initstate ?newstate ?out)
   (step* ?m ?inrest ?outrest ?newstate))
((step+ ?m nil nil ?initstate ?initstate))
((step+ ?m (?in . ?inrest)(?out . ?outrest) ?initstate ?endstate)
   (dl ?m ?in ?initstate ?newstate ?out)
   (step+ ?m ?inrest ?outrest ?newstate ?endstate))
```

Dabei ist `step` die Einschrittüberführung eines Automaten, eine einfache Zurückführung auf die `dl`-Fakten. `step*` ist die Mehrschrittüberführung, entsprechend der Wortausgabefunktion λ* der Automatentheorie. Außerdem ist noch die Variante `step+` aufgeführt, die zusätzlich zur Wort-Ein- und Ausgabe noch den Endzustand berechnet. Zu beachten ist hier, daß alle Regeln in "jeder Richtung" ausgewertet werden können. Dies ist ein großer Vorteil insbeson-

dere des rückwärtsverketteten regelbasierten Paradigmas, bzw. des logik-orientierten Programmierens.

Die einfachste Problemstellung ist die Frage nach der Wahrheit einer Aussage: (das erste "?" ist der Eingabeprompt des interaktiven Regelinterpreters)

```
? (backchaining-inference '((step s 0 1 6 0)) *rule-base*)
```

liefert true, das heißt im Zustand 1 geht der Automat `s` mit dem Input 0 in den Zustand 6 und gibt eine 0 aus. Im nächsten Aufruf wird nach dem neuen Zustand und der Ausgabe gleichzeitig gefragt, wobei Automat, Input und Anfangszustand vorgegeben sind:

```
? (backchaining-inference '((step s 0 1 ?qnext ?output)) *rule-base*)

(QNEXT = 6)
(OUTPUT = 0)
More? yes
NIL
```

Das System liefert also erstens die richtigen Werte für `qnext` und `output`, und bestätigt zweitens (durch die Angabe von `NIL` anstelle einer Bindung), daß es keine weiteren Lösungen gibt. Natürlich ist dies noch keine besondere Leistung, da das Prädikat `step` nichts weiter als eine Umbenennung der Fakten des `dl`-Typs ist.

Als nächstes soll zu einem vorgegebenen Folgezustand und Outputbuchstaben ein passender Ausgangszustand und Input gefunden werden:

```
? (backchaining-inference '((step s ?input ?q 5 1)) *rule-base*)

(INPUT = 0)
(Q = 4)
More? yes
NIL
```

Um die Leistungsfähigkeit des wissensbasierten Programmierens anzudeuten, fragen wir jetzt nach demjenigen Automaten, der nach dem Input 1 in den Zustand 3 geht:

```
? (backchaining-inference '((step ?fsm 1 ?q 3 ?output)) *rule-base*)

(FSM = S)
(Q = 1)
(OUTPUT = 1)
More? yes

(FSM = S)
(Q = 3)
(OUTPUT = 0)
More? yes

(FSM = S)
(Q = 7)
(OUTPUT = 0)
More? yes

(FSM = G)
(Q = 1)
(OUTPUT = 1)
More? yes

(FSM = G)
(Q = 3)
(OUTPUT = 0)
More? yes

(FSM = A)
(Q = 1)
(OUTPUT = 1)
More? yes

(FSM = A)
(Q = 3)
(OUTPUT = 0)
More? yes
NIL
```

Wir bekommen sieben Antworten. Jeder der drei hier definierten Automaten kommt in Frage, mit jeweils mehreren möglichen Kombinationen von Ausgangszustand und Output.

Es ist klar, daß bei konvetioneller prozeduraler Programmierung für jede dieser Fragestellungen ein eigenes Programm, bzw. ein eigener Algorithmus als Teil eines größeren Programms, geschrieben werden müßte. Vom Systemdesigner nicht vorgesehene Abfragearten könnten nicht bearbeitet werden. Hier dagegen läßt sich das gesamte in den Regeln enthaltene Wissen vom Benutzer wieder herausziehen.

Nun soll das rekursiv definierte Prädikat `step*` angewendet werden. Dieses Prädikat ist durch zwei Regeln definiert. Die erste legt den Rekursionsanfang fest:

```
((step* ?m nil nil ?initstate))
```

In Worten: "Wenn Inputwort und Outputwort von der Länge Null sind, spielt der Anfangszustand keine Rolle". Aber auch: "Wenn das Eingabewort der Länge Null ist, ist das Ausgabewort für jeden Anfangszustand auch der Länge Null", und: "Wenn das Ausgabewort der Länge Null ist, ist das Eingabewort für jeden Anfangszustand auch der Länge Null".

Die zweite, rekursive Regel dieses Prädikates lautet:

```
((step* ?m (?in . ?inrest) (?out . ?outrest) ?initstate)
   (dl ?m ?in ?initstate ?newstate ?out)
   (step* ?m ?inrest ?outrest ?newstate))
```

Die Inputfolge und die Outputfolge im Folgerungsteil der Regel werden hier als Liste explizit dargestellt, mit jeweils eigenen Variablen für den Anfang und den Rest der Liste. Dies ermöglicht auf elegante Weise den rekursiven Abstieg im erneuten Aufruf von `step*` auf den Restwörtern. Die Regel besagt also, daß die Konkatenation von `?in` und `?inrest` als Inputwort einerseits und `?out` und `?outrest` als Outputwort andererseits bei dem Anfangszustand `?initstate`
dann die Mehrschrittüberführung des Automaten `?m` darstellt,

- ***wenn*** (1) `?in` den Automaten aus dem Zustand `?initstate` in den Zustand `?newstate` überführt, mit Ausgabe `?out`,
- ***und wenn*** (2) `?inrest`, `?outrest` und `?newstate` als Mehrschrittüberführung für `?m` gültig sind.

Wir fragen zunächst in herkömmlicher Weise nach der Outputtrajktorie für den Automaten `s` bei vorgegebenem Input und Anfangszustand:

```
? (backchaining-inference '((step* s (0 1 1 0 1) ?outword 1)) *rule-base*)

(OUTWORD = (0 1 0 1 1))
More? yes
NIL
```

Natürlich gibt es nur eine Lösung für das Ausgabewort. Jetzt fragen wir, wie in der *Kryptoanalyse* üblich, nach den möglichen Kombinationen für Anfangszustand und Inputwort bei beobachteter Ausgabefolge:

```
? (backchaining-inference '((step* s ?inword (1 0 1 0 0 1) ?initq)) *rule-base*)

(INWORD = (1 1 0 0 0 1))
(INITQ = 1)
More? y

(INWORD = (1 1 0 0 0 1))
(INITQ = 2)
More? y

(INWORD = (0 0 1 1 1 0))
(INITQ = 3)
More? y

(INWORD = (0 0 1 1 1 0))
(INITQ = 4)
More? y

(INWORD = (1 1 0 0 0 1))
(INITQ = 5)
More? y

(INWORD = (1 1 0 0 0 1))
(INITQ = 6)
More? y

(INWORD = (0 0 1 1 1 0))
(INITQ = 7)
More? y

(INWORD = (0 0 1 1 1 0))
(INITQ = 8)
More? y
NIL
```

Für jeden möglichen Anfangszustand des Automaten ergibt sich ein passendes Inputwort.

Wenn nur ein Teil des Ausgabewortes bekannt ist, andererseits auch ein Teil des Inputwortes und der Anfangszustand bekannt sind, kann so gefragt werden:

```
? (backchaining-inference '((step* s (1 1 0 ?y1 ?y2 ?y3 ?y4 ?y5 ?y6)
                                     (1 0 1 0 0 1 ?x1 ?x2 0) 6)) *rule-base*)

(Y1 = 0)
(Y2 = 0)
(Y3 = 1)
(Y4 = 0)
(Y5 = 0)
(Y6 = 0)
(X1 = 1)
(X2 = 0)
More? yes

(Y1 = 0)
(Y2 = 0)
(Y3 = 1)
(Y4 = 0)
(Y5 = 1)
(Y6 = 1)
(X1 = 1)
(X2 = 1)
More? yes

(Y1 = 0)
(Y2 = 0)
(Y3 = 1)
(Y4 = 1)
(Y5 = 0)
(Y6 = 0)
(X1 = 0)
(X2 = 1)
More? yes

(Y1 = 0)
(Y2 = 0)
(Y3 = 1)
(Y4 = 1)
(Y5 = 1)
(Y6 = 1)
(X1 = 0)
(X2 = 0)
More? yes
NIL
```

Nun die eigentliche Testmusterberechnung durch Zusammenschaltung der drei Automaten:

```
((tpl-1 ?gen-m ?test-m ?ana-m ?test-pattern ?test-result
      ?inw-gen ?outw-ana ?init-gen ?init-test ?init-ana)
   (step* ?gen-m ?inw-gen ?test-pattern ?init-gen)
   (step* ?test-m ?test-pattern ?test-result ?init-test)
   (step* ?ana-m ?test-result ?outw-ana ?init-ana))

((tpl-2 ?gen-m ?test-m ?ana-m ?tps ?trs ?inw-gen ?outw-ana ?init-gen
      ?init-test ?init-ana ?sync-ana ?diag-ana)
   (append ?trs ?diag-ana ?trsd)
   (append ?sync-ana ?trsd ?strsd)
   (step* ?gen-m ?inw-gen ?stpsd ?init-gen)
   (step* ?test-m ?stpsd ?strsd ?init-test)
   (step* ?ana-m ?strsd ?outw-ana ?init-ana)
   (append ?tps ?di-invers-t ?tpsd)
   (append ?sy-invers-t ?tpsd ?stpsd))

((tpl-3 ?gen-m ?test-m ?ana-m ?tps ?trs ?inw-gen ?outw-ana ?init-gen
      ?init-test ?init-ana ?sync-ana ?diag-ana)
   (tpl-1 ?gen-m ?test-m ?ana-m ?stpsd ?strsd ?inw-gen ?outw-ana ?init-gen
      ?init-test ?init-ana)
   (append ?trs ?diag-ana ?trsd)
   (append ?sync-ana ?trsd ?strsd)
   (sublist ?tps ?stpsd))
```

Das hier genauer betrachtete Prädikat ist `tpl-1`. Seine Semantik kann durch das folgende Diagramm visualisiert werden:

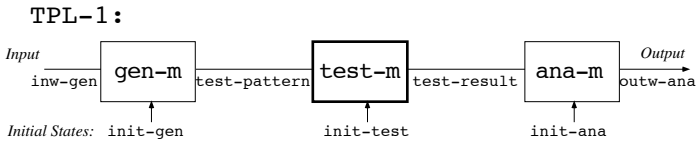

Bild 7.6 Die Regel TPL-1

Das Prädikat `tpl-1` ist durch eine einzige Regel definiert, die ganz einfach die Hintereinanderschaltung dadurch definiert, daß in der Mehrschrittüberführung der drei beteiligten Automaten das Ausgabewort des ersten Automaten und das Eingabewort des zweiten Automaten identisch sind, und entsprechend, daß das Ausgabewort des zweiten Automaten und das Eingabewort des dritten Automaten identisch sind. Die Identität wird durch Verwendung des gleichen jeweiligen Variablennamens erzwungen.

Eine Anwendung zeigt die folgende Abfrage: Welches Inputwort muß an den ersten Automaten angelegt werden, damit das angegebene Testmuster an den zu testenden Automaten (der sich im Anfangszustand 6 befindet) gelangt, und welche Ausgabe muß dabei an dem Ausgangsautomaten beobachtet werden? Es ergeben sich mehrere Antworten, je nach den Anfangszuständen des ersten und dritten Automaten:

```
? (backchaining-inference '((tpl-1 g s a (0 1 1 1 0 1 0 1 0 0 0 0 1 0 1 1)
         ?test-result ?inw-gen ?outw-ana ?init-gen 6 ?init-ana)) *rule-base*)

(TEST-RESULT = (0 1 0 0 1 1 1 1 1 0 0 0 1 1 1 0))
(INW-GEN = (0 1 0 1 1 0 0 1 1 1 1 1 0 0 1 0))
(OUTW-ANA = (0 1 1 0 1 0 0 0 0 1 0 0 1 0 0 1))
(INIT-GEN = 1)
(INIT-ANA = 1)
More? yes

...

(TEST-RESULT = (0 1 0 0 1 1 1 1 1 0 0 0 1 1 1 0))
(INW-GEN = (0 1 0 1 1 0 0 1 1 1 1 1 0 0 1 0))
(OUTW-ANA = (1 1 1 0 1 0 0 0 0 1 0 0 1 0 0 1))
(INIT-GEN = 2)
(INIT-ANA = 3)
More? nil        (abgebrochen)
T
```

Für bestimmte Anfangszustände aller drei Automaten ergibt sich nur noch eine Lösung:

```
? (backchaining-inference '((tpl-1 g s a (0 1 1 1 0 1 0 1 0 0 0 0 1 0 1 1)
         ?test-result ?inw-gen ?outw-ana 4 6 4)) *rule-base*)

(TEST-RESULT = (0 1 0 0 1 1 1 1 1 0 0 0 1 1 1 0))
(INW-GEN = (1 0 1 0 0 1 1 0 0 0 0 0 1 1 0 1))
(OUTW-ANA = (1 1 1 0 1 0 0 0 0 1 0 0 1 0 0 1))
More? yes
NIL
```

Die für den interessierten Leser aufgeführten Regeln für die Prädikate `tpl-2` und `tpl-3` berücksichtigen zusätzlich das Problem, daß die Automaten zunächst synchronisiert werden müssen, und die Testmusterausgabe mit einem Diagnosewort kontrolliert werden muß.

Die Regel `tpl-2` verwendet wieder die Mehrschrittüberführung `step*`, sowie das Grundprädikat `append`. Die Regel `tpl-3` kann durch die Verwendung von `tpl-1` kürzer formuliert werden, außerdem wird hier das Unterlistenprädikat `sublist` benötigt. Die Semantik und Funktionalität beider Regeln ist aber dieselbe.

Hier noch die Definition der verwendeten Primitivprädikate für `tpl-2` und `tpl-3`:

```
((append nil ?x ?x))
((append (?x . ?y) ?z (?x . ?u)) (append ?y ?z ?u))

((start nil ?x))
((start (?xh . ?xt) (?xh . ?yt)) (start ?xt ?yt))
((sublist ?x ?y)(start ?x ?y))
((sublist ?x (?yh . ?yt)) (sublist ?x ?yt))
```

Für eine weiterführende Betrachtung wird auf die angegebene Literatur verwiesen.

7.2.1.3. Bidirektional verkettete Regelsysteme

Eine anschauliche Darstellung der Arbeitsweise vorwärts- und rückwärtsverketteter Produktionssysteme zeigt die folgende Abbildung.

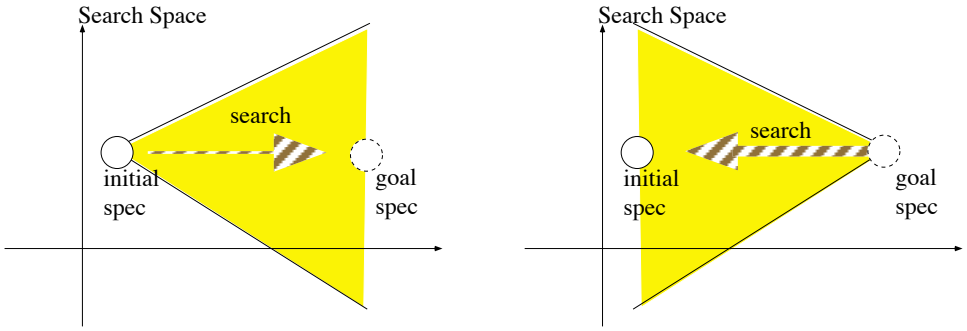

Bild 7.7 Lösungssuche bei vorwärtsverketteten und rückwärtsverketteten regelbasierten Systemen

Eine naheliegende Idee ist der Versuch, die Vorteile beider Varianten zu kombinieren. Bei bidirektionalen Systemen wird also einerseits vorwärts, ausgehend von den Ausgangsbeschreibungen, in Richtung Lösung gearbeitet, andererseits gleichzeitig auch rückwärts das Ziel in Unterziele aufgebrochen. Die Inferenzmaschine muß in der Lage sein, diese beiden Vorgänge zu koordinieren und das Zusammentreffen zu erkennen.

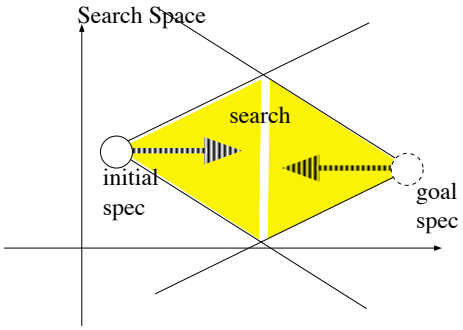

Bild 7.8 Bidirektionale Suche im Lösungsraum

Die sogenannte Means-End-Analysis ist ein Beispiel für diese Art von Inferenz-Strategie. Sie wurde bereits bei dem berühmten, aber wenig erfolgreichen GPS (General Problem Solver) von Simon und Newell angewendet [Newe72].

7.2.2. Zusätzliche Elemente wissensbasierter Systeme

Während die bisher besprochenen Komponenten Inferenzmaschine, Regelbasis und Datenbasis die eigentliche Funktionalität wissensbasierter Systeme bereitstellen, müssen praktische Implementierungen noch weitere Elemente aufweisen, um dem Anwender in effizienter Weise zu nützen, um auf einfache Weise die Untersuchung von Alternativen zu ermöglichen, und um die Plausibilität einer gefundenen Lösung aufzuzeigen.

7.2.2.1. Erklärungskomponente, History-Mechanismus

Bei Systemen, die zur Beratung menschlicher Benutzer herangezogen werden, war ein Problem früher Beispiele für konkrete Expertensysteme die Akzeptanz der Ergebnisse. Eine gewisse Skepsis war (und ist) ja auch durchaus angebracht, was ganz allgemein für die Anwendung neuer Technologien gilt. Insbesondere waren Fachleute natürlich mißtrauisch, inwieweit eine Maschine bessere Diagnosen, Designs etc. erstellen könne als sie selbst. Gerade bei solchen Expertensystemen [Geva82], die mit "unscharfem" Wissen arbeiten, ist diese Skepsis auch von den Wissenschaftlern geteilt worden, die sich mit den Grundlagen der Verarbeitung von Wahrscheinlichkeitsfaktoren befassen.

Um einerseits die Akzeptanz der von Expertensystemen erarbeiteten Resultate zu erhöhen, und andererseits die inneren Mechanismen der Inferenzmaschinen transparenter zu machen, wurde daher schon in frühen Systemen eine sog. Erklärungskomponenete eingeführt. Dies ist ein im Grunde sehr einfacher Zusatz zur Inferenzmaschine, der den jeweils aktuellen Pfad von Regelanwendungen (Regeln mit zugehörigen aktuellen Parametern/Variablen) speichert und auf Anforderung ausgibt. Inzwischen verworfene Irrwege werden dabei nicht eingeschlossen. In aufwendigeren Systemen könnte der Benutzer allerdings nachfragen, warum eine bestimmte Regel angewendet oder nicht angewendet wurde, und das System gibt dann auch für verworfene Pfade die entsprechenden Auskünfte.

Oft werden die Texte oder Textfragmente, die die Erklärungskomponente benötigt, um Auskünfte in natürlicher Sprache geben zu können, schon in der Syntax für die Regeln der Regelbasis verlangt [Norv92]. Die Erklärungen sind dann gut lesbar.

Für eingebettete Expertensysteme, die selbst nur Komponenten komplexerer Systeme sind, ist eine Erklärungskomponente nicht notwendig. Dagegen kann ein sog. History-Mechanismus, der den Verlauf einer Inferenz speichert und wiederholbar macht und als Mittel zur Erstellung eines Erklärungssystems dienen kann, auch bei eingebetteten Systemen sinnvoll sein, um etwa auf abgebrochene Vorgänge wiederaufsetzen zu können.

7.2.2.2. Interaktive Unterstützung

Wenn dem wissensbasierten System für ein bestimmtes Problem zuwenig Wissen zur Verfügung steht, um es zu lösen, kann es den menschlichen Benutzer wiederum um Rat fragen. Dies erscheint zunächst als wenig sinnvoll, da der Mensch ja durch die Maschine unterstützt werden soll, und nicht umgekehrt. Jedoch ist die hier angedeutete Strategie durchaus nützlich für solche Anwendungen, bei denen ein Dialog mit dem menschlichen Anwender möglich ist. Es kann nämlich sein, daß dem Programm nur ein Detail fehlt, welches dem Benutzer durchaus bekannt ist, so daß das Expertensystem dann mit dieser zusätzlichen Information weiterrechnen könnte. Man führt daher eine sog. Dialogkomponente ein, die dem Expertensystem erlaubt, dem Benutzer Fragen zu stellen und die Antworten für den weiteren Inferenzprozeß zu verwenden.

Weiterhin könnte diese zusätzliche Information auch in die Regelbasis aufgenommen werden, um so das Wissen des Systems zu erweitern und für künftige ähnliche Probleme nutzbar zu machen. Dialogkomponenten sind daher Voraussetzung für solche lernfähigen Expertensysteme.

Im Zusammenhang mit CAST zeigt sich noch eine andere Anwendung für die Dialogkomponente. Hier stellt sich in bestimmten Fällen nicht die Frage nach Information an sich, sondern nach dem vertretbaren Aufwand, der in eine Berechnung gesteckt werden soll. Es kann etwa sein, daß zur Lösung eines Optimierungsproblems ein Objekt berechnet werden müßte, welches eine enorme Rechenzeit benötigen würde. Die Dialogkomponente könnte hier nach Rechenzeitvorgaben fragen, die dann den Aufwand (und die Qualität des Ergebnisses) begrenzen.

7.2.2.3. Graphisches Benutzer-Interface

Viele Expertensysteme haben eine einfache Benutzerschnittstelle, die zeichenorientiert in einem normalen Textmodus am Bildschirm ablaufen kann. Diese Tatsache resultiert teilweise einfach aus der meist zugrundeliegenden Programmiersprache Lisp. In Lisp als interaktiver Programmierumgebung ist es sehr einfach, eine interaktives, zeichenorientiertes Programm zu erstellen. Nach dem geringen Aufwand kommt als zweiter Vorteil noch die ganze Mächtigkeit der Sprache Lisp zum Tragen, die sich so am leichtesten durchreichen läßt, so daß der Anwender die vielen Möglichkeiten dieser dynamischen Sprache ausnutzen kann. Dies ist auch für in Prolog geschriebene Systeme weitgehend der Fall. Natürlich muß der Anwender dann in der verwendeten Programmiersprache (Lisp oder Prolog) über entsprechende Kenntnisse verfügen, um diesen Vorteil nutzen zu können. (Auch in CAST-FSM wird die zeichenorientierte Schnittstelle zu Lisp daher angeboten.)

Für viele Zwecke und Anwendungen ist eine zeichenorientierte Ein- und Ausgabe aber nicht gut geeignet. Graphische Benutzeroberflächen bieten für derartige Anwendungen mehr Komfort, Effizienz und Klarheit für den menschlichen Anwender. Es gibt zahlreiche Expertensysteme mit integriertem graphischen Interface. Dies gilt für Systeme der Diagnose-Art (medizinische Diagnosesysteme, Automotorendiagnose, ...) und ebenso für Systeme aus dem Design-Gebiet. Nahezu notwendig werden gut gestaltete graphische Oberflächen, wenn

der Kreis der Anwender des Systems wenig oder keine Vorbildung in Bezug auf Computersysteme hat. Von Laien wird das Erlernen einer Befehlssprache und das Memorieren des jeweiligen Systemzustandes als Hemmschwelle zur Benutzung zeichenorientierter Systeme empfunden. Doch auch für erfahrene Fachleute ist es sicher leichter, z.B. das als defekt vermutete Teil einer Motorsteuerung vom Expertensystem in einer Graphik auf dem Bildschirm gezeigt zu bekommen, anstatt nur einer textuellen Bezeichnung mit (günstigstenfalls) einem Bezug in einen Teilekatalog.

Natürlich ist diese Diskussion der Vorteile zeichenorientierter und graphischer Benutzeroberflächen nicht spezifisch für Expertensysteme, sondern vielmehr auf jedes Softwaresystem anwendbar.

7.3. Interaktive und heuristische Entscheidungsfindung: STX

Für die Anwendung des wissensbasierten Paradigmas auf ein Expertensystem für die bequeme und (teilweise) automatische Anwendung systemtheoretischer Methoden ergeben sich einige besondere Aspekte.

7.3.1. Prinzipielle Architektur von STX

Definition STX

> Ein wissensbasiertes System, welches eine STIPS-Maschine als ausführende Einheit enthält, wird im folgenden STX genannt.
>
> STX_F := ($STIPS_F$, $R(T_F)$, D_F, E, P(F)) ist eine STX zum Thema F. Dabei ist wieder
> $STIPS_F$ = STIPSn(S_F, T_F) eine STIPS-Maschine zum Thema F (s. Abschn. 4.2.2.),
> $R(T_F)$ ist eine Regelbasis zum Thema F und
> D_F ist eine Datenbasis zum Thema F.
> E ist ein Inferenzmechanismus (Algorithmus).
> P(F) ist eine Systemproblemklasse vom Thema F.

Die Regelbasis $R(T_F)$ enthält für jede Systemtransformation $t \in T_F$ die Regeln, die angeben, unter welchen Vorraussetzungen t aufgerufen werden kann, welche Elemente aus der Datenbasis genommen werden und welche Elemente durch die Anwendung der Regel bzw. ihrer Transformation t neu zur Datenbasis hinzukommen.

Die Datenbasis D_F enthält die vorgegebenen und errechneten Systemspezifikationen in der Form von Lösungsgraphen $LG(S_F,T_F)$, sowie die Symbolmengen für die Lösungsplanung.

Von der Inferenzmaschine E wird verlangt, daß sie nach der Art der Planer (engl. planner) zunächst versucht, einen Plan zur Lösung eines gegebenen Problems aufzustellen, der dann in einer zweiten Phase von STX ausgeführt wird. Mitunter wird es notwendig sein, diese beiden Phasen feinkörniger in Unterphasen aufzuteilen, da verschiedene Entscheidungen erst nach Durchführung gewisser Vorbereitungen getroffen werden können. E wird in 7.3.3. verfeinert.

In jedem Fall liefert E ein Programm, ein Stück eines Systemalgorithmus. Dieser wird von STX interpretiert, indem die einzelnen ausführbaren Anweisungen an die STIPS-Maschine gegeben werden. Die ausführbaren Anweisungen bestehen wieder aus je einer Systemtransformation und einer Folge von Argumenten (Systemen). Die bedingten Anweisungen verwenden Analyseoperationen auf der Menge sys(lg(LGM(S_F,T_F))) der bisher errechneten Systeme (siehe Kapitel 4), um den Ablauf des Algorithmus zu steuern.

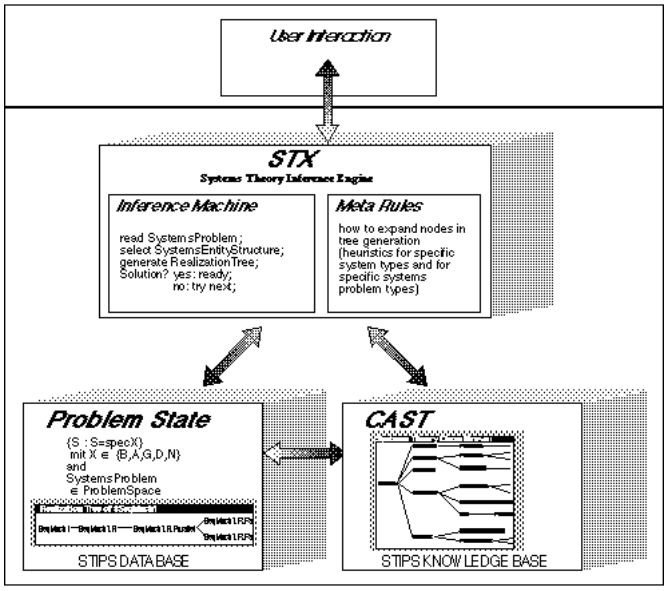

Bild 7.9 STX Architektur (schematisch)

Bild 7.9 zeigt die ursprüngliche Idee einer STX-Architektur, zuerst gezeigt in [Mitt87-1].

7.3.2. Implementierungsaspekte: Heterogene Programmierung

Der Inferenzmechanismus E der STX ist als ein Algorithmus definiert. In einer praktischen Realisierung wird E als regelbasiertes Programm geschrieben. Es gelten die am Anfang des Kapitels gemachten Bemerkungen. Ein wesentlicher Unterschied zwischen reinen regelbasierten Programmen und STX ist darin zu sehen, daß die verwendete STIPS-Maschine i.a. nicht regelbasiert ist und das Gesamtsystem daher von heterogener Architektur sein wird. Die Anbindung der Methoden in der STIPS-Maschine an die Inferenzmaschine E kann auf verschiedene Weise erfolgen. Leben beide Teilsysteme in einer Lisp-Umgebung, ist die Verknüpfung unproblematisch möglich durch gegenseitige Aufrufe exportierter Funktionen.

Die Datenbasis D wäre hier eine von beiden Teilen genutzte Datenstruktur innerhalb der Lisp-Umgebung. - Bei dieser Lösung könnte ein evtl. vorhandener Multiprozeß-Mechanismus ausgenutzt werden, um eine gewisse Parallelisierung zu erreichen, vor allem im Zusammenhang mit der Benutzerschnittstelle.

Heutige Betriebssysteme erlauben auch weitgehende Kommunikation zwischen Anwendungen. Daher könnte ein STX-System auch durch zwei getrennte Applikationen realisiert werden, eine kommunikationsfähige Implementierung der STIPS-Maschine und als separater Ableitungsmechanismus der Teil E. Die Datenbasis D würde dann als Datenbank als eigene Datei(en) realisiert.

In Betriebssystemen, die für Multiprocessing besonders geeignet sind, wie Unix, bietet es sich an, die STIPS-Maschine überhaupt aufzuteilen in viele kleine Programme, die auf Dateien arbeiten. Jede Methode (Systemtransformation oder Analysemethode) wird als eigenes ausführbares Programm erstellt und läuft bei Aufruf als eigener Prozeß (verwaltet vom Betriebssystem). Die übergeordnete Ablaufsteuerung E kann diese Prozesse dann auch auf anderen Rechnern aktivieren, so daß eine verteilte Verarbeitung möglich wird. Wenn die Benutzerschnittstelle ebenfalls als eigener Prozeß realisiert wird, kommen die allgemeinen Vorteile von Client-Server Strukturen zum Tragen.

7.3.3. Planung von Systemalgorithmen

Die Inferenzmaschine E hat die Aufgabe, ein gestelltes Systemproblem durch Anwendung von Regeln zu lösen. Dieser Vorgang, die Lösung zu finden, kann als Suchvorgang im Sinne der KI verstanden werden. Die in 7.2. gebrachte Übersicht verdeutlicht verschiedene Varianten regelbasierter Programmiermethoden zur Lösungssuche in einem Suchraum (state-space-search).

Eine sehr einfache Möglichkeit, eine STIPS-Maschine zu einem automatisch arbeitenden Systemtheorie-Experten zu erweitern, ist demnach so realisierbar, indem die Inferenzmaschine E in jedem Zustand des Problemlösungsprozesses eine der möglichen Transformationen anwendet und den vermutlichen Abstand zur Lösung testet. Die in der KI-Forschung üblichen Suchmethoden könnten hier angewendet werden. Eine solche Erweiterung wurde bereits 1987 vom Autor für das Interlisp/LOOPS-basierte System CAST:FSM erprobt (und verworfen). Ohne eine sehr gute "Meta-Wissensbasis", d.h. ohne eine Menge von Regeln zur Fachgebiets- und Problem-abhängigen Steuerung, ist ein allgemeiner Suchalgorithmus nicht in der Lage, selbst für fast trivial kleine Beispiele brauchbare Lösungen zu finden.

Ein wesentlicher Unterschied zwischen vielen typischen Problemklassen, die in der KI behandelt werden, wie Schachprogrammen, Puzzles etc., und den hier behandelten Systemproblemen, liegt in dem Rechenaufwand für die Ausführung *eines* Elementarschrittes. In den KI-Beispielen ist der Aufwand für die Durchführung eines Schrittes (d.h. einer Zustandsüberführung im Suchraum) sehr gering, die Anzahl der in einem bestimmten Zustand möglichen Schritte (d.h. der Verzweigungsfaktor) ist aber groß. Die daraus resultierende exponentielle Explosion der Anzahl der möglichen Wege (Trajektorien) wird von den üblichen Suchalgorithmen wie A* durch Anwendung von sogenannten heuristischen Funktionen be-

kämpft. Diese Heuristiken steuern die Auswahl der Operatoren nach dem (geschätzten) Abstand zur gesuchten Lösung.

Für die Lösungssuche von Systemproblemen ergibt sich ein etwas anderer Schwerpunkt. Hier ist die Ausführung einer Zustandsüberführung in vielen Fällen ein sehr rechenaufwendiger Schritt, während der Verzweigungsfaktor eher klein ist. Daher sollte im Idealfall schon *vor* dem Start einer Berechnung einer Systemtransformation feststehen, daß das Ergebnis den Abstand von der gesuchten Lösung verringert und somit lohnend ist. Um die Auswahl des nächsten Schrittes zu bestimmen, muß die Inferenzmaschine also die vermutlichen Eigenschaften der Ergebnisse der möglichen Überführungen kennen.

Es gibt in der KI einen Zweig, der sich mit einer ähnlichen Problematik befaßt: Planning. Die Anwendung von Planning-Methoden liegt vor allem in der Robotersteuerung u.ä. [Stee92]. Abstrahierte Robotik-Probleme werden meist mit dem Begriff "Blocks World" assoziiert. Bekommt ein Roboter die Aufgabe, bestimmte Gegenstände zu holen, neu anzuordnen u.ä., so können die einzelnen Schritte zur Lösung ebenfalls nicht einfach durchprobiert werden. Erstens könnten dadurch Gegenstände oder die Maschine selbst zerstört werden, zweitens würde viel Energie und Zeit verschwendet. Die prinzipielle Lösungsmethode ist hier natürlich, die Folge der Aktionen im voraus zu planen. Diese Strategie, Planning, kommt demnach für E von STX in Frage.

Ein Planning-Algorithmus berechnet einen Plan, dessen *Anwendung* oder Durchführung erst die gewünschte Lösung des Eingangsproblems erzeugt. Dazu wird die Änderung des Zustandes durch die Anwendung eines Schrittes in einem Modell der Realität nachgebildet. Statt der Ausführung eines Schrittes wird nur der Befehl zur Ausführung in der Ausgabe abgelegt. Das Ergebnis des Planners ist demnach wieder ein Algorithmus, gewöhnlich ein sehr einfacher Algorithmus bestehend aus einer linearen Folge von ausführbaren Operationen. Da die Elementaroperationen dieses Algorithmus im Falle der STX-Maschine Systemtransformationen sind, kann festgehalten werden: Die Inferenzmaschine E der Maschine STX soll einen Systemalgorithmus berechnen, der das vorgelegte Systemproblem löst.

Offensichtlich muß zur erfolgreichen Lösungsfindung eines Systemproblems Wissen über die vorliegenden Systeme vorhanden sein oder berechnet werden können. Dem Anwender einer STIPS-Maschine stehen dazu die in 2.6. definierten Analyseoperationen zur Verfügung. Auch E von STX muß auf die Analyseergebnisse zugreifen können. Es können zwei Vorgehensweisen unterschieden werden:

(1) Planung eines allgemeingehaltenen Systemalgorithmus mit bedingten Strukturen, der die Analyseoperationen zur Laufzeit (Anwendung oder Durchführung des Systemalgorithmus) auf die als Zwischenresultat erzeugten Systeme anwendet, um die weitere Abarbeitung zu steuern.

    Zur Lösung eines Systemproblems berechnet E in der Planungsphase den Systemalgorithmus, in der anschließenden Ausführungsphase wird dieser durchgeführt (siehe auch 6.2., STA).

(2) Planung und Abarbeitung gehen Schritt für Schritt ineinander verzahnt vonstatten. E plant den nächsten Schritt aufgrund der Ergebnisse der Analyse der bisher errechneten Systeme. Dann wird der Schritt ausgeführt.

Zwischen diesen beiden Extremen kann man sich auch ein gemischtes Vorgehen vorstellen, bei dem E soweit vorausplant, als mit begrenztem Aufwand weitere Schritte vorbereitet werden

können. Wird der Aufwand für die Komplexität überschritten, wird der bisher geplante Teil des Systemalgorithmus abgearbeitet und das Ergebnis analysiert. Dann kann die Planung fortgesetzt werden.

Definition der Inferenzmaschine E von STX

> Eine Inferenzmaschine E ist ein Tripel (PLAN, $Prob_F$, Alg_{FL}). Dabei bezeichnet $Prob_F$ die Eingabemenge, die Menge der bearbeitbaren (System-)probleme; Alg_{FL} , die Ausgabemenge, die Menge der Systemalgorithmen zu diesem Thema; PLAN einen sog. Planner, ein System vom Typ Algorithmus.

Die Maschine E bearbeitet ein als Input vorgelegtes Problem p, indem sie die Ist-Zustandsbeschreibung und die Zielzustandsbeschreibung aus p extrahiert und dem Planner PLAN vorlegt. Dieser liefert eine Programm, welches als Systemalgorithmus formuliert als Output ausgegeben wird.

Aus der Literatur bekannte Planning-Algorithmen sind für die Verwendung in der STX-Inferenzmaschine E nicht in jedem Fall geeignet. Tatsächlich scheinen sich zwei prinzipiell verschiedene Arten von Plannern durchzusetzen. Die erstere Art ist die bei praktischen Anwendungen eingesetzte Variante, bei der auf große Flexibilität und Ausdruckskraft Wert gelegt wird. Solche Systeme sind meist Teile von umfasssenden kommerziellen AI-Produkten, über deren genaue Funktionsweise nicht viel bekannt wird. Die andere, wissenschaftlich orientierte Richtung schränkt die Ausdruckskraft der zugelassenen Terme stärker ein, um zu eleganten Aussagen über Vollständigkeit etc. zu kommen, d.h. um eine Theorie der Planung aufstellen zu können. Ein moderner Vertreter dieser mathematisch orientierten Richtung ist das System UCPOP. [Penb92][Barr94]

Definition der Komponente PLAN

> Der Planner PLAN(E) ist ein Tripel (A_{plan}, Inits, Goals)
>
> mit: Inits ist die symbolische Beschreibung des Ausgangszustandes, Goals ist die symbolische Beschreibung des Zielzustandes, A_{plan} ist der in formaler Sprache angegebene Algorithmus zur Aufstellung des Planes.

Der Einsatz eines mit mathematischer Theorie ausgestatteten Planers in STX ist ein noch nicht gelöstes Problem, da diese Planer sehr starke Einschränkungen bei den verwendeten Datenstrukturen enthalten. So ist bei UCPOP etwa die Problembeschreibung und die Regelsprache im Wesentlichen auf logische Verknüpfungen von Mengen von Symbolen (in speziellen Fällen auch Zahlen) beschränkt, was für systemtheoretisch orientierte Anwendungen völlig unzureichend ist. Auch die Darstellung des Zustandes im Lösungsraum ist als einfache Symbolmenge für STIPS-Anforderungen zu primitiv. Leider sind die Datenstrukturen nicht leicht zu ändern, an einer solchen Anbindung eines Planers wie UCPOP wird jedoch weiter gearbeitet.

7.3.4. Ausführung von Systemalgorithmen

Wenn das Ergebnis der Planung in Form eines Systemalgorithmus oder eines Stückes eines Systemalgorithmus vorliegt, kann die Ausführung desselben auf der zugrundeliegenden STIPS-Maschine erfolgen. Durch die Ausführung wird der Realisierungsgraph erweitert, und es kann von der Inferenzmaschine auf Erfolg (oder verbleibenden Abstand vom im Systemproblem angegebenen Ziel) geprüft werden.

7.4. Bewertung wissensbasierter Systeme für die Implementierung von STX

Nachdem in diesem Kapitel eine Fülle verschiedener wissensbasierter Paradigmen vorgestellt wurde, die alle spezifische Vor- und Nachteile haben, muß hier festgestellt werden, daß die Unterschiede für die praktische Anwendung nicht so gravierend sind. Für den Einsatz in einer konkreten STX-Implementierung kommen mehrere Inferenzmethoden in Frage. Es lassen sich folgende Listen von wünschenswerten Eigenschaften und weniger wichtigen Eigenschaften aufstellen:

Für STX sinnvolle Eigenschaften:

- ▲ Leichte Integrierbarkeit mit Lisp
  (da/solange die zugrundeliegende STIPS-Maschine in Lisp geschrieben ist)
- ▲ Heterogene Programmiermöglichkeit
  (die verwendete STIPS-Maschine stellt die Implementierung der Regeln der Wissensbasis dar, ist aber i.a. selbst nicht regelbasiert geschrieben)
- ▲ Dialogfähigkeit
  (um den interaktiven Charakter von CAST zu erhalten)
- ▲ Erklärungskomponente
  (um die getroffenen Entscheidungen für den Benutzer nachvollziehbar darstellen zu können)
- ▲ Fähigkeit zum Planen
  (viele Systemalgorithmen müssen (aus Gründen beschränkter Ressourcen) vor der Durchführung geplant werden, im Gegensatz zu Problemen, die einfach ausprobiert werden können)
- ▲ Dynamisches Verhalten, dynamische Umgebung
  (Pläne, Unterziele etc. sollen aufgrund von Rechenproblemen (Rechenzeit, Speicherplatz) vom Benutzer jederzeit geändert werden können)

Für STX nicht notwendige Merkmale:

- Wahrscheinlichkeitsfaktoren
  (das zugrundeliegende Wissen stammt aus Ingenieurwissenschaften)
- Fuzzy-Mengen
- Neuronale-Netz-Komponenten
- Echtzeitverhalten
  (Interaktivität soll gewährleistet sein, harte Echtzeitanwendungen sind jedoch nicht vorgesehen)

Weiterhin ist eine graphische Oberfläche in einer ersten Ausbaustufe von STX nicht notwendig, jedoch sollte die Möglichkeit einer Anbindung eines GUI vorgesehen sein.

8. Kapitel - Beispiel einer STX für Automaten und Schaltwerke

In diesem Kapitel wird eine Beispielimplementierung einer STX beschrieben, die als zugrundeliegende STIPS-Maschine das in Kapitel 5 vorgestellte CAST-FSM verwendet. Zunächst wird der prinzipielle Aufbau vorgestellt, und in 8.2. werden wichtige Methoden objektorientierten Programmierens vorgestellt, die für den Entwurf komplexer Systeme im Kontext dieser Arbeit relevant sind. Dann folgen in 8.3. die Beschreibung der Implementierung der STX sowie einige Beispiele für Systemproblemlösungen in 8.4.

8.1. Trennung von Daten, Wissensbasis, und Steuerung

Das Blockdiagramm zeigt den prinzipiellen Aufbau der STX_{FSM}.

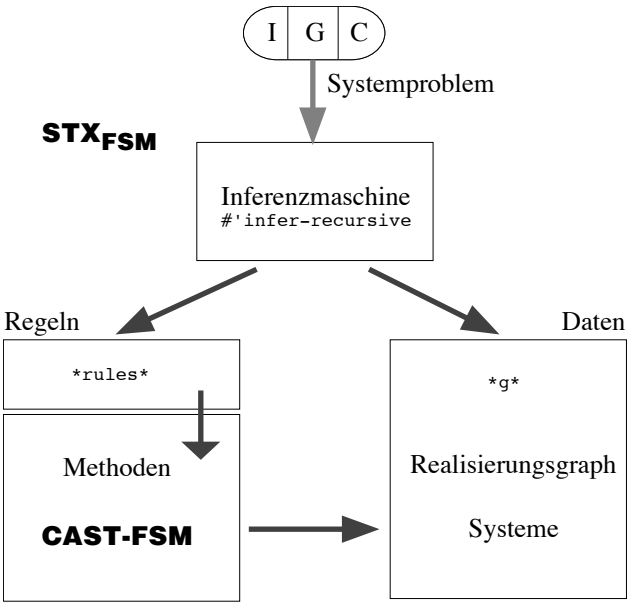

Bild 8.1. Aufbau von STX_{FSM}

Die Trennung von Wissensbasis und Steuerung ist durch die regelbasierte Definition des domainabhängigen Wissens einerseits und die davon ganz unabhängige Implementierung des Inferenzmechanismus als normale Lispfunktion besonders deutlich. Allerdings ist die Inferenzmaschine von STX eine austauschbare Komponente: es sind mehrere Varianten vorhanden, die leistungsfähigste ist derzeit die vorwärtsverkettende Ableitungsmaschine `infer-recursive.` Natürlich kann man auch eine ihrerseits regelbasiert geschriebene Inferenzkomponente einsetzen, deren Regeln dann als Metaregeln bezeichnet würden und die die Anpassung des Inferenzverhaltens auf regelbasierte Weise ermöglichen würde.

8.2. Objektorientierte Programmierungstechniken

Die wesentliche Idee des objektorientierten Programmierparadigmas ist die kanonische Art der Modellierung: Jedem "echten" Objekt in dem für die vorliegende Programmieraufgabe interessierenden Realitätsauschnitt entspricht in der objektorientierten Darstellung direkt ein "Objekt" im Programm. Der Entwurf neuer Programmsysteme soll so deutlich vereinfacht werden, denn der Designer definiert die auftretenden Datenstrukuren und Programme als Objektklassen und Methoden analog zum realen Vorbild.

In den vielen Jahren seit der Einführung der Idee und des Begriffs "objektorientiert" haben sich viele divergierende Richtungen herausgebildet, so daß dieser Begriff ohne Kontext ziemlich inhaltsleer geworden ist. Tatsächlich fällt es schwer, auch nur ein einziges Element, einen einzigen Bestandteil anzugeben, den alle objektorientierten Methoden gemeinsam hätten. Die nach allgemeiner Meinung erste objektorientierte Programmiersprache war Simula, den Durchbruch der Idee "Objektorientiertheit" (OO) brachte aber Smalltalk Ende der Siebziger. Die meisten Konzepte von Smalltalk wurden bald in die diversen Lispdialekte übernommen. Die beiden wichtigsten Varianten [Stef86], Flavors und LOOPS, wurden dann zu dem 1990 erschienenen CLOS weiterentwickelt, das heute in der Lispwelt Standard ist. - Das heute allgemein dominierende C++ hat nur wenige der 1980 für OO wesentlich gehaltenen Eigenschaften, viele sprachliche Nachteile von C++ sind auf Kompromisse wegen der verlangten weitgehenden Kompatibilität zu C zurückzuführen. Aber auch die von N. Wirth vertretene Richtung objektorientierten Programmierens, die ihren Ausdruck in der Sprache Oberon gefunden hat, ist von gänzlich anderer Art als die der dynamischen Sprachen wie Smalltalk oder CLOS oder als die des verbreiteten C++.

8.2.1. Wichtige objektorientierte Programmiersprachen

Smalltalk
: Es gibt Objekte und Methoden.
Jedes Objekt ist Instanz einer Klasse. Jede Klasse ist ein Objekt. Eine Klasse, deren Inkarnationen Klassen sind, heißt Metaklasse. Klassen stehen in einer Vererbungsrelation zueinander. Eine Klasse, die einer anderen Klasse vererbt, heißt ihre Superklasse. Eine Klasse kann standardmäßig nur eine Superklasse haben (single inheritance), aber: siehe unten.

Jede Klasse hat eine Menge von Methoden.
Ein Smalltalk-Programm besteht aus einer Folge von Nachrichten-Sende-Aufrufen (message passing). Eine Nachricht ist an ein bestimmtes Objekt gerichtet und enthält den Selektor (Name der auszuführenden Methode) und evtl. zusätzliche Argumente.
Neue Objekte vom Typ c werden durch Senden einer "New"-message an die Klasse c erzeugt. Durch Spezialisierung der "New"-Methode in der entsprechenden Metaklasse (= Klasse der Klasse c) kann eine besondere Initialisierungsprozedur eingerichtet werden.
Durch Programmierung in der Metaklassenebene kann vieles vom Standardverhalten von Smalltalk geändert werden. Insbesondere kann auf diese Weise auch Mehrfachvererbung eingeführt werden.
Besonders wichtig ist in Smalltalk der Gedanke der *dynamischen Systemumgebung*. Smalltalk ist nicht nur eine reine Sprache, in der man von einem Compiler zu übersetzende Programme schreibt und so zu in irgendeinem Betriebssystem ablauffähigen Maschinenprogrammen (executables) kommt. Vielmehr ist Smalltalk eine komplette Systemumgebung, innerhalb derer die Programme erstellt, getestet und angewendet werden, ohne daß eine Aufteilung in Editier-, Complier- und Ablaufphase sichtbar wäre; also eine einheitliche Umgebung für Programmentwicklung und -nutzung, in der der Programmierer und der Nutzer des Systems arbeiten. Außerdem sind viele Konzepte graphischer Oberflächen (GUI) erstmals in Smalltalk entwickelt worden, was jahrelang zu einer engen Assoziation von OOP und GUI führte [Gold83].

CLOS

CLOS ist eine echte Erweiterung von Common Lisp, d.h. jedes Common Lisp Programm ist auch ein gültiges CLOS-Programm. Genau wie Smalltalk ist Lisp (und CLOS) eine dynamische Umgebung, nicht nur eine Sprache. Praktisch alle sprachlichen Konzepte von Smalltalk sind in CLOS übernommen worden, mit vielen Erweiterungen und Verallgemeinerungen. Dabei steht es dem Programmierer frei, bei jedem einzelnen Statement zu entscheiden, ob er auf reines Common Lisp zurückgreifen oder CLOS-Konstruktionen verwenden will.
Die wichtigsten Verallgemeinerungen von CLOS sind:
Multimethods: Die sonst im OOP übliche Notation des Message-passing kann in Lisp geschrieben werden als ein Aufruf: `(send obj selector arg1 arg2 ...)`, wobei `obj` das Objekt ist, an das die Nachricht geschickt wird, `selector` ist der Name der Methode, die `obj` ausführen soll, und `argi` sind die zusätzlichen Argumente. `send` ist nur ein Schlüsselwort und eigentlich überflüssig. In CLOS würde der Aufruf geschrieben als: `(selector obj arg1 arg2 ...)` oder besser `(selector arg0 arg1 arg2 ...)`, wobei `selector` jetzt der Name einer generischen Funktion ist. Zur Laufzeit wird von den für diese Argumente in der generischen Funktion vorhandenen Methoden die speziellste ausgesucht, bzw. mehrere solche werden kombiniert (method combination). Dabei kann jede Methode der generischen Funktion nicht nur auf das erste Objekt (`obj` oder `arg0`) qualifiziert sein, sondern auf jedes beliebige oder mehrere aus {`arg0`, `arg1`, `arg2`, ...}. Daher ist die message-passing Sprechweise nicht mehr angebracht: Die Nachricht wird ja quasi an alle Argumente geschickt, aber natürlich nur einmal kombiniert und ausgeführt.
Method Combination: Man kann Zusatz-Methoden definieren, die nur vor (oder nur nach) der Hauptmethode ausgeführt werden sollen, oder die um die Hauptmethode herumgelegt werden sollen. Dazu gibt es noch eine Fülle weiterer Kombinationsmechanismen, insbesondere auch vom Programmierer selbst defi-

nierbare.
Initialisierungsalgorithmus: Es gibt einen genau festgelegten, aber sehr flexibel erweiter- und anpassbaren Mechanismus der Initialisierung neu angelegter Objekte mit Defaultwerten und Initialisierungsargumenten. Insbesondere können die Initialisierungsmethoden den vollen Umfang des Vererbungsmechanismus ausnutzen [Keen89].

Eiffel

Die Sprache Eiffel wurde von B. Meyer ohne Kompromisse in Bezug auf Kompatibilität als reine objektorientierte Sprache entworfen. Die wichtigsten Ziele von Eiffel sind einfache Modellierung, Wiederverwendbarkeit, Robustheit und Wartbarkeit, d.h. gute Unterstützung software-technischer Forderungen. Die Sprache ist streng getypt, Klassen sind das wesentliche Definitionsmittel (aber selbst keine Objekte). Es gibt Mehrfachvererbung, gute Schutzmechanismen und automatische Speicherverwaltung. Letztere kann vom Programmierer so beeinflußt werden, daß Echtzeitanwendungen möglich werden. Das Programmiermodell enthält Definitionen für Vorbedingungen, Nachbedingungen, Invarianten und Ausnahmen, die vom Compiler, aber auch zur Laufzeit, überwacht werden und die hohen Robustheitsanforderungen sicherstellen [Meye88].

Oberon

N. Wirth wollte mit seinem Projekt Oberon [Wirt92] nachweisen, daß die heutige Tendenz zu immer größeren Programmen und Betriebssystemen nicht notwendig ist, sondern daß sich ein kleines, hocheffizientes Betriebssystem mit Compiler für eine Workstation mit sehr geringen Speicherplatzanforderungen realisieren läßt [Wirt94]. Entstanden ist, neben einer Workstation "Ceres" und einem speziellen Betriebssystem, eine neue Programmiersprache namens Oberon, die als Weiterentwicklung von Modula-2 angesehen werden kann.
Diese Sprache wird auch als objektorientiert bezeichnet, obwohl die Unterstützung objektorientierter Prinzipien eher spartanisch ist. Wie ihre Vorgänger ist Oberon ebenfalls streng getypt, was dynamische Konzepte weitgehend ausschließt. Als einziges wesentliches Element objektorientierten Ansatzes findet sich ein Vererbungskonzept (Typerweiterung, single inheritance) in dieser Sprache, die ansonsten eher für klassischen Systementwurf im Sinne der Wirthschen Schule zu verstehen ist.

C++

Die heute verbreitetste objektorientierte Sprache ist C++, das ursprünglich als Erweiterung von C geschrieben wurde und als Präprozessor implementiert wurde. Heute übersetzen C++ Compiler aber direkt (native).
C++ ist eine getypte, hybride Sprache, inzwischen mit Mehrfachvererbung. Objektorientierte Konstruktionen können mit konventionellen prozeduralen C-Programmen frei kombiniert werden. Nur explizit als solche angelegte Daten sind Objekte.
Klassen sind ähnlich wie Typdefinitionen nur zur Übersetzungszeit bekannt. Die einzige Metainformation, die im ablauffähigen Programm (evtl.) noch vorhanden ist, ist die Methoden-Zeiger-Struktur, die zur Laufzeit zur Auflösung dynamisch gebundener Methoden dient. Dieses dynamische Verhalten ist jedoch nicht die Regel, sondern muß vom Programmierer extra für jede Methode angegeben werden. Solche Methoden, für die die dynamische Auflösung (dispatch) nicht angegeben wurde, können in Erweiterungen daher nicht spezialisiert werden, was die allgemeine Wiederverwendbarkeit von Bibliotheksprogrammen erschwert.
Die Speicherverwaltung muß vom Programmierer ausprogrammiert werden.
Trotz vieler konzeptioneller Nachteile hat sich C++ in vielen Bereichen durchgesetzt, was auf verschiedene Gründe zurückgeführt werden kann: Weitgehende

> Kompatibilität zu C; die Möglichkeit, effiziente und kleine ausführbare Programme zu erzeugen; die Verfügbarkeit freier oder preiswerter Compiler für praktisch jede Hardware-Betriebssystem-Kombination. [ARM-90]

Während alle hier aufgeführten Sprachen das Konzept der Vererbung zur Bildung von abgeleiteten Datentypen (Klassen) gemeinsam haben, gibt es auch andere Varianten, die die Unterscheidung von Klassen und Instanzen fallen lassen. In solchen Sprachen kann von jedem Objekt ein anderes Objekt abgeleitet werden, also wie eine Klasse verwendet werden. Man spricht hier statt von Vererbung von Prototyping (nicht zu verwechseln mit der Entwurfsmethode Prototyping). Da sich diese Systeme weniger durchgesetzt haben, soll auf eine genauere Diskussion hier verzichtet werden.

8.2.2. Vererbungsstrategien

Viele als objektorientiert bezeichnete Entwurfsmethoden empfehlen das Erstellen der Klassenhierarchie als einen frühen Schritt im Entwurfsprozeß. Ein wesentlicher Vorteil des objektorientierten Paradigmas soll ja die analoge Modellierung realer Objekte durch Objekte der Programmiersprache sein. Es ist daher naheliegend, auch die Beziehungen zwischen diesen realen Objekten und ihren Abstraktionen schon frühzeitig in Klassen und Vererbungshierarchien abzubilden.

Die Erfahrung, die bei der Implementierung der CAST Software gewonnen wurde, läßt eine zu frühe Festlegung von Vererbungsstrukturen nicht ratsam erscheinen. Vor allem die direkte Übersetzung von sogenannten "is a"- Relationen ("...ist ein...", oder einfach ISA-Relation) in Superklassen-Relationen kann zu erheblichen Konsistenzproblemen bei der Software-Entwicklung führen, die in der Folge auch die Wartbarkeit und die Wiederverwertbarkeit negativ beeinflussen.

In der Artificial Intelligence wurde im Forschungsgebiet Wissensdarstellung (knowledge representation) die Konstruktion "semantisches Netz" [Brac83] eingeführt. Die Knoten eines semantischen Netzes stellen irgendwelche Konzepte dar, die (beschrifteten) gerichteten Kanten entsprechen gewissen Relationen zwischen diesen. Eine besonders einfache Relation zur Modellierung ist die "...ist ein..."-Relation, z.B. "Tiger" "...ist ein..." "Säugetier", "Maus" "...ist ein..." "Säugetier", "Wal" "...ist ein..." "Säugetier", "Säugetier" "...ist ein..." "Tier", "Vogel" "...ist ein..." "Tier". Diese Beziehung scheint sich ohne weiteres auf eine isomorphe Klassenhierarchie übertragen zu lassen. Tatsächlich ist dieser Ansatz zur objektorientierten Modellierung weit verbreitet und anscheinend in vielen Bereichen erfolgreich. [Rumb91]

> Eine Besonderheit von CAST Systemen ist darin zu sehen, daß die Objekte mathematische Systemtypen sind. Die Exaktheit ihrer mathematisch-präzisen Definition scheint zunächst eine Vereinfachung zu sein gegenüber anderen Systemen, die sich mit den nur unscharf definierbaren Objekten der realen Welt auseinandersetzen müssen. Es zeigt sich jedoch, daß sich selbst die einfachsten mathematischen Grundobjekte, wie Mengen und Funktionen, nur mit gewissen Einschränkungen, aber nie in voller Allgemeinheit, als Datenstrukturen darstellen lassen. Selbst Zahlen, auf die ja in Digitalrechnern letztlich alle anderen Objekte zurückgeführt werden, sind in ihrer vollen Allgemeinheit nur

schwer zu fassen. In keiner der üblichen Programmiersprachen werden etwa reelle Zahlen zur Verfügung gestellt, eine Ausnahme bildet hier die in das mathematische Werkzeug eingebettete Sprache von Stephen Wolframs "Mathematica". Die Sprache Lisp enthält wenigstens beliebig große ganze Zahlen und die rationalen Zahlen. Die meisten Programmiersprachen haben nur spezielle Zahlklassen wie "integer" und "real", die dann oft nicht einmal bzgl. der Addition eine Algebra sind, geschweige denn Gruppeneigenschaft haben, und vom mathematischen Standpunkt sehr unbefriedigend sind.

Daher muß ein System, das auf mathematische Konstruktionen angewiesen ist, schon bei diesen einfachsten mathematischen Objekten beginnen und eine für das Thema adäquate Darstellung finden. Komplizertere Objekte (Schaltfunktionen, Automaten, diskrete Ereignissysteme etc.) können dann aus den einfacheren Objekten zusammengesetzt werden. Dabei zeigt sich, daß die naive Anwendung der ISA-Relation als Klassenhierarchie zu Inkonsistenzen führt. Ein Beispiel soll hier genannt werden.

Die Klassen endlicher Automat, deterministischer Automat, linearer Automat, Schaltkreis und linearer Schaltkreis werden in folgendem Diagramm als ISA-Graph dargestellt.

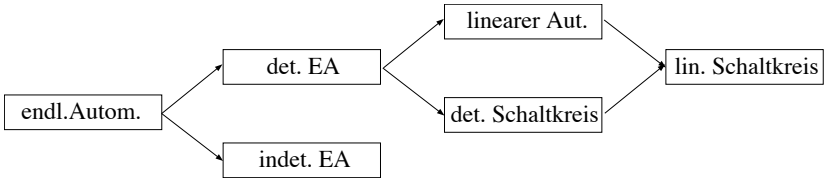

Bild 8.2 Systeme vom Typ Generator als semantisches Netz mit ISA-Kanten

Nimmt man dieselbe Hierarchie als Vererbungsgraph, würde die Klasse "linearer Automat" die Algorithmen und Implementierungsdetails der Klasse "endlicher Automat" erben, obwohl die meisten dieser Klassenelemente für diese Unterklasse ohne Bedeutung wären, oder völlig überschrieben werden müßten. Im Sinne einer guten Erweiterbarkeit und Wartbarkeit von komplexen Programmsystemen ist die Vererbung von nicht relevanten Elementen sehr ungünstig, da sie oft zu Inkonsistenzen und verdeckten Fehlern führt.

Das Beispiel zeigt, ohne auf Implementierungsdetails einzugehen, daß die ISA-Hierarchie nicht unbedingt die beste Vererbungshierarchie ist. Im nächsten Abschnitt soll eine Vorgangsweise zur Erstellung einer in gewissem Sinne "optimalen" Vererbungsstruktur vorgestellt werden.

Wie weiter unten deutlich werden wird, ist die Möglichkeit zur Mehrfachvererbung eine wichtige Eigenschaft objektorientierter Systeme, die verschiedene Designstrategien erst ermöglicht. Allerdings muß man bei der Verwendung der Mehrfachvererbung darauf achten, daß die Vererbungshierarchie weitgehend konfliktfrei bleibt. Zwar werden in modernen Programmiersprachen wie CLOS Vererbungskonflikte in eindeutiger Weise aufgelöst, so daß die Semantik eines Programmes klar definiert bleibt. Die Konsequenzen geringer Änderungen in der Klassenhierarchie komplexer Systeme werden aber schwer durchschaubar, und die Wartbarkeit von größeren Programmsystemen ist sehr gefährdet, wenn viele Konflikte berücksichtigt werden müssen. Es ist in jedem Fall besser, die Klassenhierarchie so umzustellen, daß man ohne Konflikte auskommt.

8.2.3. Algorithmus für Vererbungsstrukturen

Im folgenden soll ein Algorithmus vorgestellt werden, der vorgegebene Klassen in eine optimale Vererbungshierarchie einordnet. Zur Vereinfachung werden klassenspezifische, vererbbare Konstruktionen wie Methoden und Variablen (instance variables, class variables, default-values, data members, attributes, slots) hier nicht unterschieden, sondern generisch als Merkmal der Klasse bezeichnet.

Notation: Als "abstrakte" Klassen werden solche Klassen bzeichnet, von denen niemals Instances gebildet werden sollen, die also nur der Organisation der Programme dienen. Alle anderen Klassen werden als "echte" Klassen bezeichnet. Die Variablen und Methoden einer Klasse werden hier unter dem Oberbegriff Eigenschaft zusammengefaßt. Die Menge C bezeichne die Menge aller Klassen. Das Prädikat *feature* definiert die Zugehörigkeit einer Eigenschaft zu einer Klasse:

feature: Fx$C \rightarrow$ {wahr, falsch}:

feature(f, c) := wahr :$\Leftrightarrow$ Klasse c hat Eigenschaft f

Algorithmus Inheritance

(inherit-1) Erhebung der Menge rC aller echten Klassen und für jede Klasse $ci \in rC$ die Menge ihrer Eigenschaften. Diese Daten werden aus der Spezifikation des Programmes abgeleitet.

(inherit-2) Bildung der Menge F aller Eigenschaften aller Klassen. Dabei müssen die folgenden Regeln eingehalten werden: Eigenschaften verschiedener Klassen, die verschiedene Semantik haben und daher verschiedene Implementierung haben sollen, müssen als verschiedene Elemente in F aufgenommen werden (evtl. umbenennen). Dagegen müssen Eigenschaften, die die gleiche Semantik haben, als eine einzige Eigenschaft in F geführt werden.

(inherit-3) Bildung der Menge K als Teilmenge der Potenzmenge von rC, d.h. K ist eine Menge von Teilmengen von rC, in der folgenden Weise:

K ist die Vereinigung der Menge aller einelementigen Teilmengen von rC mit der Menge der Äquivalenzklassen von echten Klassen modulo einer jeden Eigenschaft in F:

$$K := \left\{ X \subset rC : \left(card(X) = 1 \right) \vee \left[\exists f \in F : \left(\left(\forall x \in X : feature(f, x) \right) \wedge \left(\forall x \notin X : \neg feature(f, x) \right) \right) \right] \right\}$$

Es wird also für jede Eigenschaft f in F die Menge der Klassen gebildet, die f enthält. Diese Mengen, zusammen mit den echten Klassen, bilden die Elemente der Menge K.

(inherit-4) Bildung der Menge der benötigten Klassen Kf als Teilmenge von Kx $\wp$ (F), wobei $\wp$ die Potenzmenge bezeichnet, als:

$$Kf := \left\{ (X,Y) : X \in K \wedge Y \subset F : \forall f \in Y : \left[\left(\forall x \in X : feature(f,x) \right) \wedge \left(\forall x \notin X : \neg feature(f,x) \right) \right] \right\}$$

Kf enthält für jede Menge *X* aus *K* ein Paar, bestehend aus der Menge *X* und der Menge der Eigenschaften, die genau in *X* enthalten ist, aber nirgends sonst.

(inherit-5) Jedes Element (X,Y) von *Kf* muß durch eine Klasse implementiert werden. Die erste Komponente *X* bestimmt die Stellung der Klasse in der Vererbungshierarchie, die zweite Komponente *Y* bestimmt die Menge der Eigenschaften, die in dieser Klasse zu implementieren ist.

Die Vererbung wird auf die Teilmengenrelation zurückgeführt: Seien die beiden Paare $c_1=(X_1,Y_1)$ und $c_2=(X_2,Y_2)$ Elemente von *Kf*, d.h. Klassen. Dann erbt c_2 von c_1, wenn X_2 eine Teilmenge von X_1 ist.
c_1 ist eine direkte Oberklasse von c_2 , wenn es keine andere Klasse $c_3=(X_3,Y_3)$ in *Kf* gibt, mit X_2 ist Teilmenge von X_3 und X_3 ist Teilmenge von X_1.

Beispiel: Gegeben seien die Klassen:
A mit Eigenschaften x, y, z
B mit Eigenschaften x, y
C mit Eigenschaften x, t
D mit Eigenschaften s, t

Die Menge *K* ist kann nach inherit-3 bestimmt werden als:

K={ {A}, {B}, {C}, {D}, {A, B, C}, {A, B}, {C, D}}

Daraus ergibt sich mit inherit-4 *Kf* als:

Kf={ ({A}, {z}), ({B}, {}), ({C}, {}), ({D}, {s}), ({A, B, C}, {x}), ({A, B}, {y}), ({C, D}, {t})}

Die graphisch dargestellte Vererbungshierarchie nach inherit-5 ist:

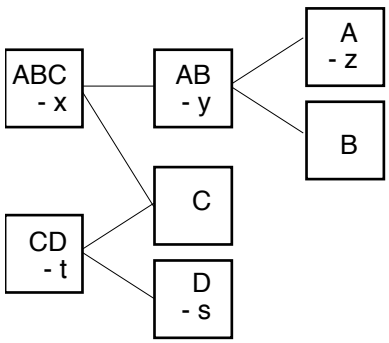

Bild 8.3 Berechnete Vererbungsstruktur

Wie in dieser Arbeit üblich, ist die Vererbungsrelation nach rechts gezeichnet, d.h. C ist Unterklasse von CD und von ABC, D ist Unterklasse von CD, etc.

Durch die Anwendung dieses Algorithmus ist sichergestellt, daß gilt:

- jede Eigenschaft $f \in F$ ist nur einmal implementiert (keine Redundanz)

- keine Klasse erbt eine für sie nicht relevante Eigenschaft

- kein Überschreiben von Eigenschaften

Ein gewisser Nachteil in der Anwendung des Algorithmus muß auch genannt werden. Jede kleine Änderung an der Programmspezifikation, die sich auf die Eigenschaften der Klassen auswirkt, kann zu einer beliebig großen Änderung der Vererbungshierarchie führen. Daher muß auch nach vergleichsweise unbedeutenden Änderungen jeweils die gesamte Vererbungshierarchie neu berechnet werden. Dies ist in strenger Konsequenz für alle herkömmlichen Programmiersprachen ein unvertretbar hoher Aufwand, der die Vorteile der Optimalität der Klassenhierarchie wieder aufhebt. Man wird daher in der Praxis den Algorithmus nicht in mathematischer Präzision anwenden, sondern mehr als Richtlinie für den ersten Entwurf und für wesentliche Redesigns verwenden.

Andererseits sind auch ganz andere Ansätze für die Erstellung und Verwaltung von Programmsystemen denkbar und zum Einsatz gekommen, die eine automatische Veränderung des Programm-Sourcecodes erlauben. So wurde der erste CAST-Prototyp CAST:FSM etwa in der integrierten Umgebung LOOPS entwickelt, die eine weitgehende systemprogrammgesteuerte Verwaltung von Programmen enthielt. In solchen Umgebungen wäre ein automatisches Mitziehen der Änderungen in der Vererbungsstruktur durch eine Implementierung des oben angegebenen Algorithmus denkbar, so daß jederzeit eine redundanzfreie, saubere Hierarchie vorliegt, ohne daß der Programmierer sich um die Organisation der abstrakten Klassen überhaupt kümmern müßte.

8.2.4. Information Hiding

Während in vielen objektorientierten Sprachen die Klasse die kleinste Einheit für Datenkapselung (information hiding) ist, ist in CLOS die Einheit für Zugriffsbeschränkungen das "package", das meist mehrere Klassen enthält. Andererseits könnte man sogar Bestandteile einer Klasse auf mehrere Packages verteilen, was wohl selten sinnvoll ist.

Außerdem ist in CLOS der Zugriff nur bedingt eingeschränkt: Durch eine spezielle Syntax kann immer jede Funktion im System auf jedes Detail eines jeden Objektes zugreifen, wenn der Programmierer dies wünscht. Dies entspricht der allgemeinen Philosophie der zugrundeliegenden Sprache Lisp, in der die größtmögliche Freiheit von jeher als unverzichtbares Ziel der Sprachentwicklung zur Unterstützung explorativen Programmierens angesehen wurde. Andererseits wird ein diszipliniertes Programm von dieser Möglichkeit der Umgehung des Zugriffsschutzes keinen Gebrauch machen.

Die Anwendung des Package-Konzeptes zur Kapselung (in Lisp spricht man eher von der Trennung der Namensräume) ist seit langem erfolgreich angewendete Praxis, und führt dazu, daß man ein zur Verfügung gestelltes Programm (Library) zu dem eigenen Sourcecode einfach

dazuladen kann, ohne daß sich die Teile gegenseitig stören/überschreiben. Die Frage ist, ob durch die Verwendung von objektorientierten Konstruktionen (CLOS) die Vorteile der Kapselung erhalten bleiben. Leider ist die Verwendung von Packages mit CLOS nicht so leicht zu realisieren, wenn man die Vorteile der Wiederverwendbarkeit erhalten will.

Der Programmierer eines Packages legt fest, welche Teile des Programms sichbar sind, indem er diese explizit "exportiert". Alle anderen Teile können von außen, also von anderen Teilen eines komplexen Programmsystems, nicht angesprochen werden. Normalerweise ist es einfach, festzulegen, welche Teile ein Benutzer eines Packages sehen soll und welche nicht. Dies ergibt sich zumeist direkt aus der Anforderung (Spezifikation) dieses Programmstückes. Als Beispiel kann die graphische Darstellung einer Klasse von Objekten betrachtet werden. Die Operationen zum Zeichnen, Aktualisieren etc. müsen dem Anwender des Pakets zur Verfügung gestellt werden. Interne Algorithmen zum Layout von Graphen, zum Zeichnen von Bildsymbolen usw. bleiben dem Benutzer dagegen verborgen; er hätte für diese internen Funktionen auch keine Verwendung.

Anders sieht es für einen Programmierer eines Programmstückes aus, das eine (objektorientierte) Erweiterung des ersteren Packages darstellen soll. Um Methoden, Klassen und deren Elemente erweitern und spezialisieren zu können, muß er eine Vielzahl von internen Definitionen des Package kennen und darauf zugreifen können. (Dagegen ist der Source-Code für die objektorientierte Erweiterung bestehender Programme nicht notwendig; die compilierte, ablauffähige Form ist ausreichend.)

Es können daher zwei Anwendergruppen von Programmpaketen mit grundsätzlich verschiedenen Sichtbarkeitsbedürfnissen unterschieden werden.

1. Der Programmierer, der weitgehenden Zugriff auf interne Konstruktionen vorgegebener Programmteile benötigt, um flexibel und feinkörnig spezialisieren und erweitern zu können.
2. Der Endanwender, der nur die Funktionalität des Pakets nutzen will, ohne sie zu verändern.

Für den Endanwender ist das in Lisp vorhandene Package-Konzept genau passend. Für den Programmierer, der die Vorteile objektorientierten Arbeitens ausnutzen will, ist das Lisp Package dagegen wenig geeignet, es müßten die meisten Symbole exportiert werden, was den Sinn des Package aber in Frage stellt. Die andere Lösung erscheint hier sinnvoller, nämlich Erweiterungen zu einem bestehenden Package in diesem Package zu schreiben. Dies ist in Lisp deshalb leicht möglich, weil ein Package auf mehrere Source-Files verteilt sein kann, und auch zu nur compiliert vorliegenden Packages neue Source-Files hinzugefügt werden können, ohne die bestehenden Teile neu compilieren zu müssen.

Zusammengefaßt kann festgehalten werden, daß man bei der Einführung neuer Packages in CLOS vorsichtig sein sollte: Nur wirklich eigenständige Teile eines Ganzen sollten in einem eigenen Package enthalten sein. Bei zu vielen, kleinen Packages wird die objektorientierte Wiederverwertbarkeit eingeschränkt.

8.2.5. Prototyping

Ein wichtiger Grund für die Verwendung einer dynamischen Programmiersprache wie Lisp (oder Smalltalk) liegt in deren guten Unterstützung für das sogenannte Prototyping, auch "rapid prototyping" oder "explorative programming" genannt. Gemeint ist hier, daß ein komplexes Softwaresystem nicht am Schreibtisch in Topdown-Manier komplett entworfen und erst dann als fertiges Programm eingegeben und getestet wird, sondern daß bereits sehr frühzeitig im Entwicklungsprozeß, bereits während der Spezifikation, mit der Implementierung begonnen wird. Dabei wird meist erst mit dem für den Anwender direkt sichtbaren Teil des Ganzen angefangen, d.h. mit der Benutzeroberfläche. Nach und nach werden dann weitere Teile hinzugefügt, getestet, geändert und ausgebessert. Die vorläufigen Varianten des Softwaresystems werden dabei nicht als zu vermeidende Ressourcenverschwendung angesehen, sondern als ein wichtiges Werkzeug zur Gewinnung der endgültigen Spezifikation und der finalen Implementierung angesehen.

Beim Prototyping ist es immer wieder notwendig, noch nicht vorhandene Programmteile durch provisorische Konstruktionen zu ersetzen. So kann etwa eine komplizierte Funktion durch eine vorläufige ersetzt werden, die zwar die gleiche Interface-Definition hat, aber deren Rumpf nichts anderes tut, als einen konstanten Wert zurückzugeben, der als Beispiel für den eigentlich zu berechnenden Wert genommen werden kann. In dynamischen Sprachen, in denen kleine Änderungen, wie die Definition eines Funktionrumpfes oder einer Parameterliste, sehr schnell und einfach durchzuführen sind, ist Prototyping daher ein natürlicher, naheliegender Programmierstil. Bei der Implementierung der verschiedenen CAST-Systeme hat sich auch aus diesem Grund der Einsatz der Sprache Lisp bewährt.

8.2.6. Zusammengesetzte Objekte

Ein in vielen Programmiersprachen nicht explizit unterstütztes Konzept ist das der zusammengesetzten Objekte. Es geht hier um Objekte, die andere Objekte als Bestandteile haben. Natürlich ist dies ein elementares Problem, das zumeist auf die einfache Art gelöst wird, daß die Variablen des Objektes (Instance-Variables, slots, data members, attributes) einen expliziten oder impliziten Verweis auf die Komponenten des Objektes enthalten. Ein Beispiel ist das Objekt Automat, das in seiner mathematisch-systemtheoretischen Definition als 5-Tupel aus drei Mengen und zwei Funktionen besteht (siehe 2.1). Es ist daher naheliegend, in der objektorientierten Implementierung den Systemtyp Automat als Klasse mit fünf Variablen zu definieren, nämlich State-Set, Input-Set, Output-Set, State-Transition und Output-Function. Jede Variable enthält das entsprechende Element als Unterobjekt, im Beispiel also Mengen und Funktionen (oder Zeiger auf diese, je nach Programmiersprache). Eine solche Implementierung des mathematisch als 5-Tupel definierten Systems als ein Objekt mit fünf Einträgen könnte als "reine" Implementierung bezeichnet werden.

Nun ergibt sich aber häufig die Notwendigkeit, weitere Objekte mit diesen "rein" implementierten zu assoziieren. Beispielsweise ist die Berechnung des Verbandes eines Automaten eine zeitaufwendige Rechenoperation, daher sollte der fertigberechnete Verband für zukünftige Verwendung abgespeichert werden können.

Weiterhin sind in vielen praktischen Systemen Benutzeroberflächen vorzusehen. Die graphische Darstellung von Objekten beispielsweise durch Graphen, Diagramme oder Bildsymbole (Icons) muß ebenfalls dem repräsentierten Objekt zugeordnet werden.

Ein weiteres Beispiel für die Notwendigkeit, zusammengesetzte Objekte behandeln zu können, ergibt sich aus dem Problem, Objekte dauerhaft zu speichern und zwischen verschiedenen Systemen austauschen zu können. Auch diese externen Repräsentationen von Objekten sollten mit der internen, reinen Darstellung verknüpft sein (andernfalls müßten externe Darstellungen immer wieder neu erzeugt und gelesen werden, wie dies in den bisherigen Inkarnationen von CAST-FSM der Fall war.)

Es gibt nun einige verschiedene Möglichkeiten, zusammengesetzte Objekte in einer objektorientierten Programmiersprache darzustellen.

1. **Quick-and dirty:**
Die Klassendefinition der reinen Klasse, die zusätzliche Objektverweise erhalten soll, wird einfach um die neuen Variablen erweitert.

- \+ Einfach und schnell zu programmieren, falls Source-Code zugänglich. Zugriff zur Laufzeit effizient.
- \- Keine "reine" Implementierungsschicht mehr vorhanden. Wartbarkeit wird verschlechtert. Nicht modular. Programme der mathematischen Ebene sehen Elemente der GUI-Ebene etc. Änderungen einer Schicht können sich auf das ganze Programm auswirken.

2. **Geschachtelte Objekte (durch Delegierung):**
Die reine Klasse bleibt unverändert. Für jede weitere Schicht werden Hüllobjektklassen angelegt, die in einer Variablen einen Verweis auf das reine(re) Objekt enthalten, sowie die spezifischen neuen Elemente in weiteren Variablen.

- \+ Die jeweiligen Schichten bleiben unverändert, beispielsweise erfordert das Hinzufügen einer graphischen Benutzeroberfläche keinen Eingriff in den Code der zugrundeliegenden Programme. Beispielsweise kann einer mathematisch orientierten Kernimplementierung eine graphische Benutzeroberfläche hinzugefügt werden, ohne den Kern zu verändern.
- \+ Der Benutzer kann zur Laufzeit entscheiden, welche Art von Objekten er anlegt, der Kern funktioniert ohne oder mit zusätzlichen Schichten gleich. Reine Systeme bleiben reine Systeme und können ohne Einschränkung so verwendet werden.
- \- Die Reihung der Schichten muß sorgfältig geplant sein, damit externe Aufrufe immer an das äußerste Hüllenobjekt gerichtet und an den unveränderten Kern weitergereicht (delegiert) werden können.
- \- Die Delegation kann in vielen Programmiersprachen nicht automatisch vorgenommen werden, sondern muß ausprogrammiert werden.
- \- Die Delegation kostet einen gewissen Laufzeit-Overhead.

3. **Hierarchische Objekte (durch Vererbung):**
Die Definition der Kernklassen bleibt auch hier wieder unverändert. Die Klassen der Hüllobjekte werden jedoch als Unterklassen der Kernklassen definiert. Die spezifischen Variablen solcher Schichten werden demnach durch Vererbung in die

zur Laufzeit verwendeten Objekte integriert. Die Kernklassen werden zu abstrakten Klassen.

+ Die Definition der Kernschicht bleibt unverändert.

+ Kein Laufzeit-Overhead. Die Methoden der äußeren Schichten stehen durch Vererbung direkt zur Verfügung, von extern kommende Aufrufe brauchen nicht delegiert zu werden.

- Zur Laufzeit sind die Objekte ausgeflacht, Variablen des Kerns und der Hüllen stehen gleichberechtigt nebeneinander wie bei der ersten Methode (1.). Dadurch wird Debugging und Wiederverwendung erschwert. Die Größe der Objekte ist durch die Anzahl der geladenen Schichten bestimmt. Die Erzeugung von reinen Objekten ist nicht möglich, nicht benutzte Variablen haben keine Bedeutung, obwohl sie vorhanden sind und gelesen werden können.

In der Praxis haben sich die beiden Varianten 2. bzw. 3. in einigen Fällen bewährt, in anderen Fällen dagegen als weniger günstig herausgestellt, so daß hier keine allgemeingültige Empfehlung gegeben werden kann. Jedenfalls wäre es wünschenswert, daß die Modellierung komplexer Objekte schon von der Programmiersprache selbst besser unterstützt würde.

8.2.7. Design von User Interface Systemen

Das Gebiet der Gestaltung und Implementierung von graphischen Benutzeroberflächen ist sehr komplex und kann hier nur gestreift werden. Für eine STX-Realisierung ist eine graphische Oberfläche nicht unbedingt notwendig, viele Expertensysteme arbeiten mit einer eher textorientierten Ein-/Ausgabeschnittstelle. Wie schon im Kapitel 5 bei der Vorstellung von CAST-FSM gezeigt wurde, sind jedoch einige Zusammenhänge ohne Graphik nur schwer verständlich, so daß die Darstellung von Browsern ohnehin vorgesehen sein sollte. Der bei der Arbeit einer STIPS-Maschine wachsende Realisierungsgraph kann vorteilhaft als Steuerzentrum für die gesamte Bedienung verwendet werden.

Die STX kann im Prinzip ebenfalls den Realisierungsgraphen als Oberflächenelement verwenden. Es sind noch einige zusätzliche Funktionen vorzusehen. Dazu gehören die Darstellung und Editiermöglichkeit für die Regelbasis, der Aufruf der Erklärungskomponente und die Problemspezifikationsmöglichkeit. Letztere spielt hier die Hauptrolle, denn die Definition und, im Laufe des Experimentierens, die Änderung der Problemspezifikation ist die wichtigste Interaktion zwischen Anwender und STX.

Ein wichtiger Punkt, der bei dem Entwurf einer Bedienoberfläche zu beachten ist, soll hier noch näher erläutert werden. Wie bereits in 8.3.6. erwähnt, sollen die Objekte eines Systems nicht wahllos mit zusätzlichen (Instance-) Variablen angereichert werden, um eine klare Schichtung bzw. Modularisierung zu erreichen.

Für Bedienoberflächen ergeben sich folgende Anforderungen:

- Das zugrundeliegende System soll auch ohne graphische Oberfläche, zum Beispiel nur unter Verwendung eines zeichenorientierten Dialogs (Lisp-Listener), funktionsfähig sein.
- Die Oberflächenschicht soll austauschbar sein, etwa für verschiedenes "look and feel" auf verschiedenen Rechnerumgebungen ("Plattformen").

Eine mögliche Lösung, wie sie schon in macCASTfsm erfolgreich erprobt wurde, wird hier skizziert.

Die zugrundeliegende (innere) Schicht des Systems bleibt unverändert. Alle zur Oberfläche gehörenden Programmteile werden in eigenen Modulen untergebracht. Objekte, die zur (graphischen) Repräsentation der inneren Objekte dienen, werden durch eigene Klassen definiert, die in keinem Vereerbungsverhältnis zu Klassen der inneren Schicht stehen. Statt dessen erhalten diese repräsentierenden Objekte Zeiger (in Instance-Variablen) auf die repräsentierten Objekte.

Durch diese Anordnung ist die saubere Trennung der Ebenen gewährleistet. Es ergibt sich dann das Problem, die Programme so zu strukturieren, daß jeder erforderliche Zugriff auf die Elemente bzw. Objekte der Ebenen möglich ist.

In einem System wie macCASTfsm ist dies verhältnismäßig leicht zu erreichen. In CAST selektiert der Benutzer Objekte und Operationen, die dann vom System durchgeführt, und schließlich in der Anzeige dargestellt werden. Mit anderen Worten geht die Aktivität immer vom Anwender des Systems aus, während das System diese nur beantwortet. Jede Aktivität des Benutzers wird vom Betriebssystem als "Event" (Ereignis) kodiert und an CAST gesendet. Die Objekte der Oberfläche reagieren auf diese Events, indem sie etwa eine Berechnung in der inneren Schicht auslösen. Dabei verwendet das repräsentierende Objekt seine Verbindung zu dem (den) entsprechenden inneren Objekt(en). Das bei der Berechnung in der inneren Schicht entstehende Ergebnis hat keine Referenz auf die Graphik, soll aber angezeigt werden. Diese Aufgabe muß daher ebenfalls noch die Methode des Oberflächenobjektes erledigen.

In einer Realisierung eines weitgehend automatisch ablaufenden Systems, wie es STX darstellt, kommt man mit dem eben vorgestellten Konzept weniger gut zurecht. Der Ablauf eines Inferenzprozesses müßte als eine Einheit, eine einzige Reaktion auf eine entsprechende Benutzeraktion aufgefaßt werden. Solange dieses Modell eingehalten wird, gibt es kein neues Problem.

Wenn die interaktive Komponente eine Entscheidungshilfe vom Benutzer abfragen will, kommt die Aktivität jedoch vom System, nicht vom Benutzer. Soll die Frage graphisch präsentiert werden, muß demnach die Möglichkeit vorhanden sein, von der inneren Schicht zu den entsprechenden Objekten der repräsentierenden Schicht zu gelangen. Um das Einbringen von direkten Zeigern zu den äußeren Objekten in die inneren Objekte zu vermeiden, weil dies der in den vorangegangenen Abschnitten vorgestellten Prinzipien guter objektorientierter Schichtung widerspräche, müssen andere Mechanismen verwendet werden, um diesen Zugriff zu ermöglichen.

Eine Möglichkeit ist die Verwendung einer globalen Zuordnungstabelle, die für innere Objekte die Abbildung auf die repräsentierenden Objekte enthält und von diesen verwaltet und aktualisiert wird. Allerdings wird so nur der Datenteil (die Objekte) des Systems saubergehalten. Die Programme (Methoden) müssen dennoch einen Aufruf einer Aktualisierung der äußeren

Schicht erhalten, der bei einem Betrieb der eigentlichen Schicht allein nicht notwendig wäre. Eine gute Lösung für dieses Problem ist in CLOS mit der Methodenkombination gegeben. Sogenannte :after-Methoden können im Modul der äußeren Schicht mit Kernmethoden gleichen Namens kombiniert werden, ohne daß im Kernmodul irgendetwas geändert werden müßte (es muß nicht einmal neu übersetzt werden).

8.2.8. Einbindung von Systemen in eine Umgebung (Steuerung, Regelung und Interface)

Eine Verallgemeinerung der in 8.2.7. eben gezeigten Problematik liegt vor, wenn ein komplexes System in eine Umgebung eingebettet werden soll. Beispiele sind Echtzeitsysteme, die einerseits die bisher genannten Eigenschaften haben (Expertensystem, interaktives Beratungssystem, CAST) und andererseits gleichzeitig Roboter, Produktionsprozesse, etc. steuern sollen.

Hier muß das Konzept dahingehend verallgemeinert werden, daß die STX nun mit mehreren Einheiten kommunizieren muß, und jederzeit auf Anfragen (requests) von diesen Einheiten reagieren können muß. Dies betrifft einerseits die Unterbrechbarkeit langwieriger Berechnungen, und die Möglichkeit, die Zeit zur Lösungssuche zur Einhaltung von Echtzeitbedingungen von vorneherein zu beschränken, andererseits aber auch die objektorientierte Strukturierung von Oberfläche und anderen Schichten (persistente Dateirepräsentation, etc.), da nun nicht mehr von einer zentralen Quelle aller Ereignisse ausgegangen werden kann.

Die hier vorgestellte, einfach gehaltene STX ist für die Anwendung in derartigen Umgebungen nicht geeignet. Für die Bearbeitung von Designproblemen im Bereich von endlichen Automaten und Schaltwerken spielt diese Einschränkung aber keine Rolle.

8.3. Implementierung in Common Lisp

Im folgenden Teil wird eine einfache STX-Implementierung geschrieben, die als Prototyp für eine leistungsfähige automatisierte Systemproblem-Lösungsmaschine dienen kann. Es handelt sich um ein regelbasiertes System mit (derzeit) vorwärtsverkettetem Inferenzmechanismus. Der Plan zur Lösung der Probleme ist in dieser Version durch die Regeln vorgegeben, im Gegensatz zu einem flexibleren echten Planer.

Das System STX wird hier in Bottom-Up-Weise dargestellt. Zugrundegelegt wird die STIPS-Maschine CAST-FSM in ihrer in Kapitel 5 beschriebenen, auf Common Lisp und CLOS aufbauenden Variante. Als provisorisches User Interface dienen der Lisp Listener und der in CLIM realisierte, graphische Browser.

Bemerkung zur verwendeten Syntax:

In diesem Abschnitt werden die wichtigsten Funktionen, Makros, Klassendefinitionen, etc. dieser STX-Implementierung vorgestellt. Als Syntaxbeschreibung wird dazu bei Makros, Funktionen und Methoden nur der Kopf der Definition in Lisp angegeben. Der Rumpf dieser Konstrukte wird meist durch das Symbol "`&body`" ersetzt, d.h. abgekürzt. Nach Meinung des Autors ist dies die unmißverständlichste, eindeutigste Art, Lisp-Programme zu dokumentieren. (Dem interessierten Leser kann der vollständige Source zur Verfügung gestellt werden.) Für das Verständnis dieser Arbeit und für die Anwendung von STX ist die hier gewählte Darstellungsweise jedenfalls übersichtlich und ausreichend.

Ein Beispiel: Die Lisp-Funktion `fibonacci`, die die Fibonacci-Zahlen (naiv-rekursiv) berechnet, wird definiert z.B. durch:

```
(defun fibonacci (n &optional (runtimecheckp t) (errorp nil))
  "returns the nth Fibonacci-number"
  ;; n should be a positive integer
  ;; runtime arg check if runtimecheckp is true
  ;; runtime arg error risen only if errorp is true
  ;; naive recursive variant
  ;; could be made efficient easily by using memoization
  (when runtimecheckp
        (unless (and (numberp n) (integerp n) (plusp n))
            (if errorp
                (error "~s not a valid integer argument." n)
                (warn "Illegal argument ~s: not integer." n))
            (return)))
  (cond ((<= n 2) 1)
        ;; fib(n):= fib(n-1) + fib(n-2)
        (t (+ (fibonacci (- n 1))
              (fibonacci (- n 2))))))
```

Als Syntaxdokumentation reicht die Angabe:

```
(defun fibonacci (n &optional (runtimecheckp t) (errorp nil))
  "returns the nth Fibonacci-number"
  ;; n should be a positive integer
  ;; runtime arg check if runtimecheckp is true
  ;; runtime arg error risen only if errorp is true
  &body)
```

Die genaue Lambda-Liste mit Argumenttypen und Defaultwerten ist ersichtlich, die Details der Implementierung können verborgen bleiben. Sofern der Rückgabewert von Funktionen wichtig ist, wird dieser im Kommentar angegeben.

8.3.1. Realisierungsgraphen

Für die Zwecke der vorliegenden Arbeit wurden Realisierungsgraphen in einer von der in macCASTfsm verwendeten abweichenden Form neu implementiert. Insbesondere werden in dieser neuen Form die Simulationsobjekte, die das eigentliche Ergebnis einer jeden

Systemtransformation darstellen, nicht explizit in den Graphen aufgenommen, um einerseits eine gewisse Vereinfachung von Regelsprache und Inferenzmechanik zu erreichen, ohne andererseits die Allgemeinheit der Verfahren zu beeinträchtigen.

Natürlich könnte man jederzeit zu den allgemeinen Realisierungsgraphen mit Simulationsobjekten zurückkehren, wie es für eine echte Anwendung auch notwendig wäre. Die Simulationsobjekte enthalten ja vor allem die Abbildungen von Komponenten der realisierten Objekte zu den entsprechenden Komponenten der realisierenden Objekte, wie sie zum Aufbau von Gesamtsystemsimulatoren und echten Realisierungen (in der realen Welt) notwendig sind. Es muß noch darauf hingewiesen werden, daß die STIPS-Maschine die Simulationsobjekte durchaus berechnet, unabhängig davon, ob diese von STX in den Realisierungsgraphen eingehängt werden oder nicht.

Für die Demonstration der Möglichkeiten von STX wird hier aber das einfachere Modell verwendet. Sowohl die Regelsprache als auch die graphisch gezeigten Ergebnisse werden dadurch einfacher und übersichtlicher.

Definition der Realisierungsgraphen und ihrer Knoten:

```
(defclass real-graph ()
  ((root :accessor root :initarg :root))
  (:default-initargs :root nil))

(defclass real-node ()
  ((system :accessor system :initarg :system)
   (successors :accessor successors :initarg :successors)
   (last-systems :accessor last-systems :initarg :last-systems))
  (:default-initargs :successors nil :last-systems nil))
```

Eine "`initialize-instance :after`"-Methode stellt sicher, daß jedem Knoten tatsächlich ein System zugeordnet ist. Wenn ein Knoten expandiert wird, d.h. ein System transformiert wird, wird der Slot "`system`" des Knotens auf das neue System, das die Realisierung des ersteren darstellt, umgesetzt, und der alte Wert dieses Slot wird in "`last-systems`" gesichert. Der Slot "`successors`" enthält die Verweise auf Komponenten der neuen Realisierung, wenn diese kein Blatt des Graphen ist. Für Blätter ist dieser Slot leer.

Neue Graphen werden angelegt durch: `(setup-real-graph init)` , wobei "`init`" das durch die Wurzel des Graphen repräsentierte System ist, also ein Objekt vom Typ "`cast-object`".

Wichtige Methoden für Realisierungsgraphen sind:

```
(defmethod get-all-nodes ((graph real-graph))
  (get-all-nodes-of t graph))
```

Liefert die Menge aller Knoten des Graphen als Liste.

```
(defmethod get-all-nodes-of ((type symbol) (graph real-graph))
  &body)
(defmethod get-all-nodes-of ((type function) (graph real-graph))
  &body)
```

Liefert alle Knoten des Graphen, deren Systeme vom Typ "`type`" sind. Dabei kann "`type`" ein Lisp Typ, der Name einer CLOS Klasse, oder ein beliebiges (als Funktion definiertes) Prädikat sein.

```
(defmethod get-all-nodes-from ((type function) (graph real-graph))
  &body)
```

Liefert alle Knoten des Graphen, für die die Prädikatsfunktion "`type`" erfüllt ist. Hier wird das Prädikat also auf die Knotenobjekte selbst angewendet, während in "`get-all-nodes-of`" auf das repräsentierte System abgefragt wird.

```
(defmethod show-real-graph ((graph real-graph) &rest args)
  &body)
```

Zeigt den Graphen unter CLIM 1, CLIM 1.1 oder CLIM 2.0 in einem eigenen Fenster auf dem Bildschirm an. Das Fenster enthält Kommandofelder, die u.a. das inspizieren einzelner Objekte erlauben, sowie den Befehl zur Erstellung einer Postskript-Version zum Ausdrucken auf Postskript-Druckern.

Ein wichtiges Prädikat für Knoten ist der Test, ob es sich um einen Endknoten (ein Blatt) des Graphen handelt:

```
(defun leafp (node)
  &body)
```

Der Ausdruck `(leafp node1)` liefert `t` (true), wenn `node1` ein Endknoten ist, `nil` sonst.

8.3.2. Quantifizierte Ausdrücke als Prädikate

Für die Verwendung als Bedingung in Problemspezifikationen, sowie allgemein in Regeln, wird immer wieder eine Ausdrucksart benötigt, die in etwa mit "Für alle Systeme, die ... erfüllen, soll ... gelten" oder "Es gibt Systeme der Art ..., für die gilt: ..." wiedergegeben werden kann. Diese Quantorausdrücke werden in STX in verschiedenen Varianten zur Verfügung gestellt und sie werden auf verschiedene Weise angewendet. Die erste Art der Verwendung ist als ein Bedingungsteil, als ein (komplexes) Prädikat. Die in STX angebotenen Varianten sind:

M0. there-is-goal

Testet, ob einige der als Argument angegebenen zielbeschreibenden Symbole im bearbeiteten Problem relevant sind.

```
(defmacro there-is-goal (&rest list-of-goals)
  "existential quantor over goals in current problem"
  &body)
```

M1. there-are-systems

Testet, ob es wenigstens ein System im aktuellen Realisierungsgraphen gibt, das die angegebenen Eigenschaften hat.

```
(defmacro there-are-systems ((system-symbol) &rest clauses)
  "existential quantor over systems in current realization graph"
  ;; is true iff there is a system for which all clauses hold
  &body)
```

Eine typische Anwendung wäre

```
(there-are-systems (s)
      (typep s 'fsm)
      (> (no-of-states s) 2))
;    -> returns t or nil
```

Der Ausdruck gibt `t` (true) zurück, wenn es im aktuellen Realisierungsgraphen mindestens ein System `s` vom Typ FSM gibt, welches mehr als zwei Zustände hat.

Wenn die Menge der Systeme, die die Bedingungen "`clauses`" erfüllen, leer ist, ist der Wert des ganzen Ausdruckes auf jeden Fall `nil`.

M2. there-are-nodes

Testet, ob es wenigstens einen Knoten im aktuellen Realisierungsgraphen gibt, der die angegebenen Eigenschaften hat.

```
(defmacro there-are-nodes ((node-symbol) &rest clauses)
  "existential quantor over nodes in current realization graph"
  ;; is true iff there is a node for which all clauses hold
  &body)
```

Hier werden die Bedingungen nicht auf die repräsentierten Systeme, sondern direkt auf die Knotenobjekte des Graphen angewendet. Dieselbe Beispielanwendung wie oben müßte bei Verwendung dieses Makros geschrieben werden als:

```
(there-are-nodes (n)
      (typep (system n) 'fsm)
      (> (no-of-states (system n)) 2))
;    -> returns t or nil
```

Dieses Makro sollte aber nur benutzt werden, wenn auf Eigenschaften des Knotenobjekts selbst eingegangen wird, beispielsweise mit dem Prädikat "`leafp`".

M3. for-all-systems

Testet, ob die Systeme des angegebenen Typs alle die nachfolgende Eigenschaft erfüllen.

```
(defmacro for-all-systems ((system-symbol) clause &rest actions)
  "universal quantor over systems in current realization graph"
  ;; returns a list of successp flags or t
  &body)
```

Liefert `t` (true), wenn für alle diejenigen Systeme, die die Bedingung "`clause`" erfüllen, der Ausdruck "`actions`" erfüllt ist. Andernfalls wird eine Liste von Flags zurückgegeben, die für

jedes System `t` oder `nil` enthält, je nachdem, ob "`actions`" für dieses System erfüllt war oder nicht. (Die Liste muß also mindestens ein `nil` enthalten.)

Der "`actions`"-Teil sollte mit der Form `(satisfies ...)` umschlossen werden. (Diese Regel kann immer eingehalten werden; notwendig ist der Gebrauch von "`satisfies`" dann, wenn in "`actions`" mehrere Tests geprüft werden sollen.)

Wenn die Menge der Systeme, die die Bedingung "`clause`" erfüllen, leer ist, ist der Wert des ganzen Ausdruckes auf jeden Fall `t`.

Die Beispielanwendung

```
(for-all-systems (s)
      (typep s 'fsm)
   (satisfies (> (no-of-states s) 2)))
```

liefert demnach `t` genau dann, wenn alle Automaten im Realisierungsgraphen mehr als zwei Zustände haben.

M4. for-all-nodes

Wieder gibt es eine Version, die direkt auf Knoten arbeitet, statt auf repräsentierten Systemen:

```
(defmacro for-all-nodes ((node-symbol) clause &rest actions)
  "universal quantor over nodes in current realization graph"
  ;; returns a list of successp flags or t
  &body)
```

Eine sinnvolle Anwendung wäre:

```
(for-all-nodes (n)
      (leafp n)
   (satisfies (or (typep (system n) 'fsm)
                  (typep (system n) 'fn))))
```

Liefert `t` wenn alle Endknoten des Graphen Automaten oder Funktionen sind.

8.3.3. Quantifizierte Ausdrücke als Programmstücke

Im vorangegangen Abschnitt wurden die Quantor-Ausdrücke als (komplexes) Prädikat verwendet, d.h. als eine Funktion, die wahr oder falsch liefert.

Es zeigt sich, daß auch Programmstücke, z.B. die Aktionsteile von Regeln, günstig mit einer solchen Syntax ausgedrückt werden können. Obwohl es sich um teilweise dieselben Makros handelt, wird deren Definition hier wiederholt, denn die Bedeutung der Argumente und die Semantik der Anwendung ist in diesem Kontext leicht geändert.

M5. for-all-systems

Wendet ein Programm (Lisp-Ausdruck) auf alle Systeme des angegebenen Typs an.

```
(defmacro for-all-systems ((system-symbol) clause &rest actions)
  "universal quantor over systems in current realization graph"
  ;; returns a list of successp flags or t
  &body)
```

Für alle Systeme, die die Bedingung "`clause`" erfüllen, werden die Ausdrücke in "`actions`" ausgeführt. Zur Erhaltung einer klaren Semantik sollte "`actions`" nur einen Ausdruck enthalten, der aber selbstverständlich beliebig komplex sein darf.

Für die Anwendung in THEN-Teilen von Regeln ist zu beachten, daß der Ausdruck in "`actions`" genau dann `nil` zurückgeben muß, wenn die Durchführung des gesamten Ausdruckes als erfolglos betrachtet werden soll.

Die Beispielanwendung

```
(for-all-systems (s)
      (typep s 'fsm)
   (print (no-of-states s)))
```

druckt die Anzahl der Zustände aller Automaten im aktuellen Realisierungsgraphen.

Vergleiche M3.

M6. for-all-nodes

Wieder gibt es eine Version, die direkt auf Knoten arbeitet, statt auf repräsentierten Systemen:

```
(defmacro for-all-nodes ((node-symbol) clause &rest actions)
  "universal quantor over nodes in current realization graph"
  ;; returns a list of successp flags or t
  &body)
```

Eine sinnvolle Anwendung wäre:

```
(for-all-nodes (node)
      (and (leafp node)
           (typep (system node) 'fsm)
           (delta-complete-p (system node))
           (>= (no-of-states (system node)) 8))
   (pdecomp-transformation node))
```

Wendet die Systemtransformation Parallelzerlegung auf alle diejenigen Endknoten des aktuellen Realisierungsgraphen an, die alle angegebenen Bedingungen erfüllen: Es muß sich um Automaten mit mindestens acht Zuständen und vollständiger Zustandsüberführung handeln.

Vergleiche M4.

M7. for-a-node

Bei diesem Makro handelt es sich zwar nicht um einen Quantor-Ausdruck, wegen der Ähnlichkeit in Syntax und Anwendung zu den vorangegangenen Konstruktionen (vergleiche M6) soll "`for-a-node`" aber hier behandelt werden.

Die Syntax ist der von `for-all-nodes` sehr ähnlich. Allerdings wird der "`actions`"-Teil hier nur auf eines der qualifizierenden Knoten angewendet.

```
(defmacro for-a-node ((node-symbol &key (selection-fn #'first))
                      clause &rest actions)
  "do an action for a node in current realization graph"
  ;; returns a list of successp flags or t
  &body)
```

Mit dem zusätzlichen Keyword-Argument `:selection-fn` kann die Auswahl des einzigen in den "`actions`" verwendeten Knotens beeinflußt werden. Defaultmäßig wird der erste qualifizierende Knoten genommen (via "`first`"). Eine andere sinnvolle Auswahlfunktion ist "`random-select`", das einen Knoten aus der Menge der qualifizierenden zufällig auswählt. Natürlich kann der Anwender auch eigene Funktionen oder anonyme Lambda-Ausdrücke verwenden.

Eine Anwendung ist:

```
(for-a-node (node :selection-fn #'random-select)
      (and (typep (system node) 'fsm)
           (> (no-of-states (system node)) 2))
   (sr-real-transformation node))
```

Von den Automaten im aktuellen Realisierungsgraphen mit mehr als zwei Zuständen wird einer zufällig ausgewählt und in eine Schieberegister-Struktur verwandelt.

8.3.4. Systemprobleme

Systemprobleme werden als CLOS-Klasse definiert:

```
(defclass problem ()
  ((initial :reader initial :initarg :initial
            :documentation
            "initial state specified as a system or a real-graph")
   (goals :reader goals :initarg :goals
          :documentation
          "list of goals - a goal is a symbol or a conditional expression")
   (constraints :reader constraints :initarg :constraints
                :documentation
                "list of constraints - a constraint is a conditional expression")
   (name :reader name :initarg :name
         :documentation "description of the problem as a string"))
  (:default-initargs :initial nil :goals nil :constraints nil
                     :name "unnamed systems problem"))
```

und können mit dem Konstruktor "`make-problem`" bequem angelegt werden:

```
(defun make-problem (&rest args)
  &body)
```

Zum Beispiel:

```
(setq myprob (make-problem :name "sinnloser Test" :initial (find-system "fsm7")
                           :goals '(fewstates (for-all-systems (s) ...))
                           :constraints '((there-are-systems (s) ...)) ))
```

Der Slot "`name`" eines Systemproblems soll eine kurze, aber prägnante Beschreibung des Problems enthalten, die dann in Protokollausdrücken verwendet wird.

Die anderen Slots dieser Klasse werden im folgenden erklärt:

8.3.4.1. Ausgangssystem (initial system)

Der Slot "`initial`" erhält den Ausgangszustand eines Systemproblems. Da der Zustand der zugrundeliegenden STIPS-Maschine durch ihren Lösungsgraphen gegeben ist, ist es praktisch, diesen auch als Datenstruktur für ein Systemproblem zu verwenden. Im Vergleich zu Abschnitt 3.3. dieser Arbeit fällt auf, daß dort die "Ausgangssysteme I" eines Systemproblems als eine Menge von Systemspezifikationen definiert wurde. Läßt man die allgemeinen Systemspezifikationen zunächst außer acht, so ist die hier gegebene Konstruktion die allgemeinere, da offensichtlich ein Graph mehr Struktur enthält als eine Menge, und die Menge als Graph dargestellt werden kann. (Allgemeine Systemspezifikationen sind in dieser ersten STX noch nicht vorgesehen.)

Für neue Systemprobleme ist es oft praktischer, mit nur einem Ausgangssystem zu starten. Die STX erlaubt daher auch Systemprobleme, deren "`initial`" nur ein einziges System enthält. Der zugehörige Realisierungsgraph mit nur einem Knoten wird dann automatisch von STX angelegt.

Eine praktische Funktion für die Angabe eines Systems ist "`find-system`". Diese Funktion wird mit dem Namen (als Symbol oder String) des gewünschten Systems aufgerufen und liefert entweder das schon geladene System, oder versucht es vom Filesystem zu laden.

Eine Folge der Verwendung von Realisierungsgraphen als "`initial`" des Systemproblems ist die Möglichkeit, auf abgebrochene Lösungsvorgänge leicht wiederaufsetzen zu können, und sogar auf den Lösungszustand eines Problems ein anderes anzusetzen, um so verschiedene Zielstellungen in einer Art Serienschaltung kombinieren zu können. Um derartige Vorgänge zu ermöglichen, ist der aktuelle Realisierungsgraph immer in der globalen Variablen "`*g*`" zugreifbar.

8.3.4.2. Zielbeschreibungen (goals)

Eine Zielbeschreibung eines Systemproblems für STX ist eine Liste, die aus zwei verschiedenen Arten von Elementen besteht. Sie enthält einerseits bestimmte Symbole, die zur Vorauswahl geeigneter Regeln zur Bearbeitung des Problems bestehen. Die Menge der hierfür sinnvoll verwendbaren Symbole wird durch die verwendete Regelbasis vorgegeben. In den Beispiel-Regelbasen kommen Symbole wie "`fewstates`" und "`binary-coded`" vor, die in intuitiver Weise die Art des Problems charakterisieren.

Die Liste "`goals`" kann außerdem bedingte Ausdrücke enthalten. Diese werden zur Laufzeit und im Kontext des Problemlösens ausgewertet. Wenn alle diese Ausdrücke erfüllt sind, wird das Problem als gelöst angesehen.

Die Bedingungsausdrücke für die Formulierung der "`goals`" können vorteilhaft mit den oben beschriebenen Makros M1-M4, sowie den logischen Verknüpfungen `AND`, `OR` und `NOT`, gebildet werden.

8.3.4.3. Einschränkungen (constraints)

Der Slot "`constraints`" kann eine Liste von Bedingungsausdrücken enthalten (oder leer bleiben).

Diese Bedingungen (Constraints) werden zur Laufzeit nach jeder Regelanwendung ausgewertet. Ist auch nur eine dieser Constraints nicht erfüllt, wird die letzte Regelanwendung rückgängig gemacht und ein anderer Lösungsweg versucht. Die Constraints müssen also nicht nur am Ende, sonder auch während des Lösungsvorganges erfüllt sein.

Auch die bedingten Ausdrücke für die Formulierung der "`constraints`" können mit den oben beschriebenen Makros M1-M4, sowie den logischen Verknüpfungen `AND`, `OR` und `NOT`, gebildet werden, oder aus beliebigen anderen Lisp-Konstrukten.

8.3.5. Regeln

Die Regeln für STX werden als Liste definiert. (Eine Liste ist bequemer zu ändern und als File zu speichern als eine große Zahl von CLOS-Objekten, daher wurde hier der elementare Datentyp Liste verwendet. Selbstverständlich ist die gesamte STX von dieser Entscheidung unabhängig, und eine Implementierung der Regeln als CLOS-Objekte ist jederzeit nachträglich möglich, ohne an den Programmen etwas zu ändern.)

Jede Regel besteht aus vier Elementen:

- Nummer und Name (Liste, bestehend aus Nummer (Zahl) und Name (String))
  Die Nummer einer Regel sollte eine Ganzzahl sein, und sie sollte innerhalb einer Regelbasis eindeutig sein. Ob die Nummer einer Regel als Priorität bei der Navigation im Lösungsraum verwendet wird, hängt von der verwendeten Inferenzmaschine ab.
  Der Name einer Regel sollte diese kurz, aber eindeutig charakterisieren, um vernünftige, lesbare Protokollausdrucke zu ermöglichen.

- IF-Teil (Bedingungsausdruck)
  (siehe 8.3.5.1.)

- THEN-Teil (Aktionsausdruck)
  (siehe 8.3.5.2.)

- Typ der Regel (Symbol, eines von {`deterministic`, `random`})
  Der Typ einer Regel kann in dem Inferenzprozeß verwendet werden. Dabei bedeutet "`deterministic`", daß eine zweite Anwendung der Regel auf denselben Ausgangszustand notwendigerweise zum selben Ergebnis (oder Mißerfolg) führt, wie die erste Anwendung. "`random`" bedeutet, daß jede weitere Anwendung einer Regel möglicherweise zu einem neuen Ergebnis führt.

8.3.5.1. Der "Wenn"-Teil einer Regel

Der IF-Teil einer Regel ist ein Bedingungsausdruck, der mit den Lisp-Formen `AND`, `OR`, `NOT` und den Makros M0, M1-M4, sowie allen anderen Lisp-Formen gebildet werden kann. Die Verwendung der Makros vereinfacht dabei den Zugriff auf die Knoten und Systeme im Realisierungsgraphen erheblich.

Ein Beispiel für einen IF-Teil einer Regel ist:

```
(and (there-is-goal fewstates)
     (there-are-systems (m)
           (typep m 'fsm)
           (delta-complete-p m)
           (> (no-of-states m) 7)))
```

8.3.5.2. Der "Dann"-Teil einer Regel

Der THEN-Teil einer Regel ist ein Ausdruck, der mit der Form "`action`" eingeleitet wird. Im Rumpf dieser Form können eine oder mehrere Aktionen stehen. Eine Aktion ist hier ein Programmstück, das per Seiteneffekt wirkt, normalerweise durch Transformation eines Systems und Eingliederung des Resultats in den Realisierungsgraphen.

Insbesondere können hier auch bedingte Ausdrücke vorkommen, die mit den Makros M5-M7 gebildet werden. Ein Beispiel für einen THEN-Teil einer Regel könnte so aussehen:

```
(action (for-all-nodes (node)
                (and (typep (system node) 'fsm)
                     (delta-complete-p (system node))
                     (>= (no-of-states (system node)) 8))
           (pdecomp-transformation node)))
```

Da die hier verwendbaren Systemtransformationen als Argument den Knoten benötigen (nicht nur das repräsentierte System), muß man für Transformationen das Makro M6 "`for-all-nodes`" (statt M5 "`for-all-systems`") oder M7 "`for-a-node`" verwenden. Für andere Arten von Aktionen, wie zum Beispiel Testausdrucke u.ä., kann auch M5 eingesetzt werden.

8.3.5.3. Regelbasen

Eine Regelbasis ist eine geordnete Menge von Regeln. Als Implementierung bietet sich die Liste an. Die für die STX verwendeten Regelbasen sind also ganz einfach Listen von Regeln, geordnet nach Priorität. Ob diese Prioritätsordnung im Rahmen der Poblemlösung auch ausgenutzt wird, hängt von der verwendeten Inferenzmaschine ab.

8.3.6. Inferenzmaschinen

Für die Anwendung in STX wurden mehrere Inferenzmechanismen erprobt, die wahlweise im aktuellen Aufruf von STX eingesetzt werden können. Allen bisher erstellten Inferenzmaschinen gemeinsam ist die Vorwärtsverkettung (vgl. 7.2., 7.3.).

Die im allgemeinen beste Variante stellt die Funktion "`infer-recursive`" dar. Diese geht wie folgt vor: Die Regeln werden der Reihe nach durchgegangen. Ist eine Regel für das Problem passend und im aktuellen Zustand anwendbar, wird sie angewendet und die Einhaltung der Constraints wird überprüft. Sind diese nicht erfüllt, wird die Anwendung der Regel rückgängig gemacht und ein anderer Lösungsweg gesucht. Andernfalls wird dieselbe Regel wieder angewendet, solange bis die Anwendung erfolglos bleibt oder Constraints verletzt werden. War die Anwendung wenigstens einmal erfolgreich, wird danach wieder mit der ersten Regel begonnen, andernfalls bei der nächsten Regel fortgesetzt.

Beendet wird die Inferenz, wenn alle Ziele (`goals`) erfüllt sind (Erfolg), oder wenn keine anwendbare Regel mehr gefunden werden kann (Mißerfolg).

Subtile Unterschiede ergeben sich noch in Abhängigkeit vom Regeltyp (`deterministic` oder `random`).

Eine der Übersichtlichkeit wegen leicht vereinfachte Version des Herzstückes der Inferenzmaschine ist das folgende Code-Stück:

```
(defun infer-recursive-int (rules graph goals
                            &optional all-rules (succp nil) (cnt *maxloop*))
  "recursive forward-chaining inference engine (internal)"
  (cond ((zerop cnt) (values graph))
        ((endp rules) (values graph))
        ((goals-satisfied-p goals) (values graph))
        (t (let ((resultp (apply-rule (first rules) graph)))
             ;; resultp is the old realization graph (can be used as UNDO)
             (if (not resultp)
               (if succp
                 (infer-recursive-int all-rules graph goals all-rules nil)
                 (infer-recursive-int (rest rules) graph goals all-rules succp))
               (if (constraints-satisfied-p)
                 (infer-recursive-int rules graph goals all-rules t)
                 (progn
                   (setq *g* resultp)
                   (infer-recursive-int
                    (cond ((and succp (rule-typep (first rules) 'random)) all-rules)
                          ((rule-typep (first rules) 'random) rules)
                          (t (rest rules)))
                    resultp ;; resultp is the old graph used as UNDO here
                    goals all-rules nil
                    (1- cnt))
                   )))))))
```

Simplere (und schnellere) Inferenzmaschinen sind: `infer-iterative-1`, `infer-iterative` und `infer-recursive-1`.

8.3.7. STX

Die Hauptfunktion der STX, nämlich "`stx`", ist sehr einfach. Außer einer Vorbereitung der Datenstrukturen und Argumente hat sie nichts weiter zu tun, als die angegebene Inferenzmaschine aufzurufen:

```
(defun stx (&key (problem *problem*)
                 (rules *rules*)
                 (inference-engine #'infer-recursive)
                 (verbose t))
  (setq *log* nil)
  (clear-all-memoization)
  (setq *problem* problem)
  (when verbose
    (format t "~%STX tries to solve: ~s on ~s~%"
            (name problem) (name (initial problem))))
  (setup-problem problem)
  (funcall inference-engine rules *g* (goals problem)))
```

Besetzt man vor dem Aufruf von "`stx`" die globalen Variablen `*problem*` mit dem zu lösenden Problem und `*rules*` mit der zu verwendenden Regelbasis, kann man die STX einfach durch:

```
(stx)
```

aufrufen, oder etwa mit:

```
(show-real-graph (stx :inference-engine #'infer-iterative))
```

wenn man den resultierenden Realisierungsgraphen gleich angezeigt haben und eine andere Inferenzfunktion verwenden will.

Als rudimentäre Erklärungskomponente steht die Funktion "`print-report`" zur Verfügung. Aufruf:

```
(print-report)
```

Beispiele folgen im nächsten Abschnitt.

8.4. Beispielanwendungen

In diesem Abschnitt werden einige Beispiele durchgerechnet. Es soll gleich zu Anfang betont werden, daß hier keine bedeutenden neuen systemtheoretischen Erkenntnisse erwartet werden dürfen, sondern daß es vielmehr um die Demonstration der Möglichkeiten des automatisierten Problemlösens geht.

8.4.1. Die erste Regelbasis

Die erste Regelbasis enthält nur drei Regeln, die die Anwendung der drei in diesem Minimalsystem vorhandenen Systemtransformationen zur Parallelzerlegung, Serienzerlegung und Schieberegisterdarstellung kontrollieren.

```
(defvar *standard-rules* (sort '(
((1 "parallel decomposition")
 (and (there-is-goal fewstates)
      (there-are-systems (m)
            (typep m 'fsm)
         (delta-complete-p m)
         (> (no-of-states m) 3)))
 (action (for-a-node (node :selection-fn #'random-select)
               (and (typep (system node) 'fsm)
                    (delta-complete-p (system node))
                    (>= (no-of-states (system node)) 4))
            (pdecomp-transformation node :selection-fn #'random-select)))
 random)
((2 "serial decomposition")
 (and (there-is-goal fewstates)
      (there-are-systems (m)
            (typep m 'fsm)
         (delta-complete-p m)
         (> (no-of-states m) 2)))
 (action (for-a-node (node :selection-fn #'random-select)
               (and (typep (system node) 'fsm)
                    (delta-complete-p (system node))
                    (>= (no-of-states (system node)) 3))
            (sdecomp-transformation node :selection-fn #'random-select)))
 random)
((3 "shiftregister realization")
 (and (or (there-is-goal binary-coded)
          (there-is-goal shiftregister-structure))
      (there-are-systems (m)
            (typep m 'fsm)
         (> (no-of-states m) 2)))
 (action (for-a-node (node :selection-fn #'random-select)
               (and (typep (system node) 'fsm)
                    (> (no-of-states (system node)) 2))
            (sr-real-transformation node)))
 random)
) #'< :key #'rule-number
))
```

Zunächst soll das folgende Problem versucht werden: Der Automat "`bbsse-e`" soll in ein Netzwerk kleiner binärkodierter Komponenten transformiert werden. Die Problemdefinition in STX erfolgt durch Anlegen einer Instance der Systemproblemklasse "`problem`" mit Hilfe des Konstruktors "`make-problem`":

```
(setq *problem*
      (make-problem
             :name "transform into decomposition network of small binary components"
             :initial (find-system "bbsse-e")
             :goals '(fewstates
                      binary-coded)
             :constraints ()))

(setq *rules* *standard-rules*)
```

Durch die Zuweisungen des Problems auf `*problem*` und der Regelbasis auf `*rules*` kann STX nun so aufgerufen werden:

```
(show-real-graph (stx))
```

Nachdem die Berechnung abgeschlossen ist, wird der errechnete Realisierungsgraph angezeigt:

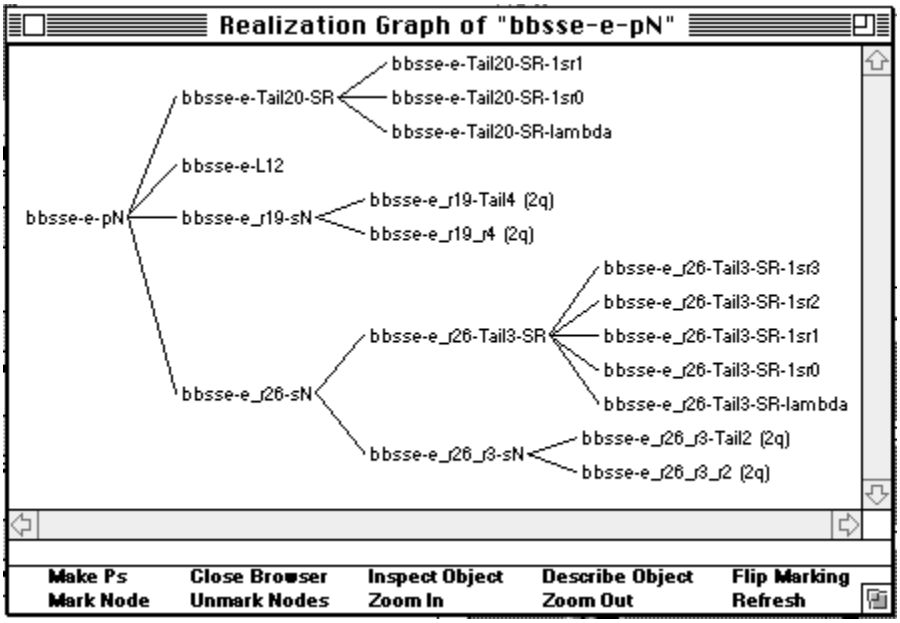

Bild 8.4 Ergebnis der Berechnung "kleine binäre Komponenten"

Genauere Auskunft über den Verlauf der Berechnung gibt "`print-report`":

```
(print-report)

Problem: "transform into decomposition network of small binary components"
- Initial: "bbsse-e" of class FSM-LARGE
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 successfully)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 not applicable)
(trying rule 1 "parallel decomposition")
(rule 1 not applicable)
(trying rule 2 "serial decomposition")
(rule 2 not applicable)
(trying rule 3 "shiftregister realization")
(rule 3 is applicable)
(executed rule 3 successfully)
(trying rule 3 "shiftregister realization")
(rule 3 is applicable)
(executed rule 3 successfully)
(trying rule 3 "shiftregister realization")
(rule 3 not applicable)
(trying rule 1 "parallel decomposition")
(rule 1 not applicable)
(trying rule 2 "serial decomposition")
(rule 2 not applicable)
(trying rule 3 "shiftregister realization")
(rule 3 not applicable)
(stop unsuccessful no-more-rules-found)
(15 INFERENCES)
```

Mit anderen Worten, STX findet hier keine Lösung, die das vorgegebene Ziel vollständig erfüllt. Gleichwohl könnte das erreichte Resultat in einer praktischen Anwendung als ausreichend angesehen werden und die Aufgabe somit als (suboptimal) gelöst.

Der Grund, daß STX das Problem als nicht gelöst betrachtet, liegt darin, daß in der Problemspezifikation zwar die qualitativen Goals "wenig Zustände" und "Binärkodierung" angegeben waren, aber keine quantitativen Goals, die zum Test auf Erfolg verwendet werden hätten können. Das Beispiel zeigt, daß man durch die vorwärtsverkettete Arbeitsweise der hier eingesetzten Inferenzmaschine auch bei unvollständig bzw. sehr grob spezifizierten Systemproblemen u.U. zu Ergebnissen kommen kann.

Das nächste Beispiel ist das gleiche Problem, aber mit einem zusätzlichen Goal und einer weiteren Bedingung als Constraint.

```
(setq *problem*
      (make-problem
       :name "transform into decomposition network of small bin components"
       :initial (find-system "bbsse-e")
       :goals '(fewstates
                binary-coded
                (for-all-systems (system)
                      (typep system 'fsm)
                   (satisfies (<= (no-of-states system) 5)))
                )
       :constraints
       '((there-are-systems (system)
              (typep system 'fsm)
           (> (no-of-states system) 2)))))
```

Nach dem Aufruf von STX gibt die "Erklärungskomponente" Auskunft über den Verlauf der Lösungssuche:

```
Problem: "transform into decomposition network of small bin components"
- Initial: "bbsse-e" of class FSM-LARGE
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 successfully)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 not applicable)
(trying rule 1 "parallel decomposition")
(rule 1 not applicable)
(trying rule 2 "serial decomposition")
(rule 2 not applicable)
(trying rule 3 "shiftregister realization")
(rule 3 is applicable)
(executed rule 3 successfully)
(stop successful solved on "bbsse-e-pN")
(9 INFERENCES)
```

Das Problem wurde demnach erfolgreich gelöst. In der graphischen Darstellung:

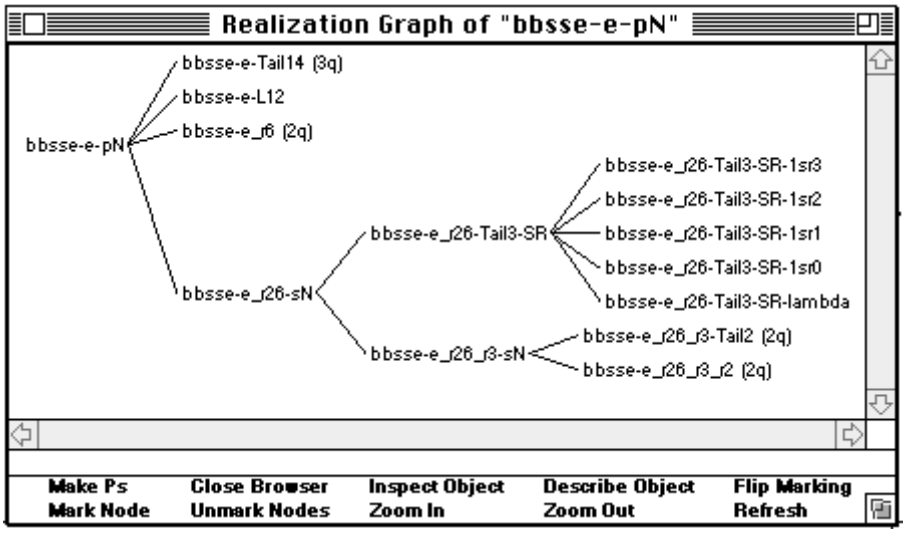

Bild 8.5 Ergebnis von "kleine binäre Komponenten" mit Goal und Constraint

Im nächsten Problem soll die Wirkung von Constraints verdeutlicht werden:

```
(setq *problem*
      (make-problem
       :name "transform into decomposition network of small components"
       :initial (find-system "fsm7")
       :goals '(fewstates
                (for-all-systems (system)
                      (typep system 'fsm)
                   (satisfies (<= (no-of-states system) 4)))
                )
       :constraints
       '((there-are-systems (system)
              (typep system 'fsm)
          (> (no-of-states system) 2)))))
```

Das Protokoll der Berechnung ist hier:

```
Problem: "transform into decomposition network of small components"
- Initial: "fsm7" of class FSM-LARGE
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 successfully)
(constraints force undo of rule 1)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 successfully)
(constraints force undo of rule 1)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 successfully)
(constraints force undo of rule 1)
(loop detected in rule 1)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(stop successful solved on "fsm7-sN")
(4 INFERENCES)
```

Obwohl die Regel 1 mehrfach erfolgreich angewendet wurde, mußte das Ergebnis jedesmal verworfen werden, weil es die Bedingung in "constraints" nicht erfüllte. Das Ergebnis ist:

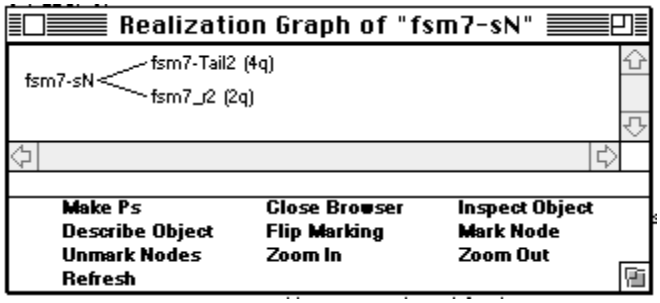

Bild 8.6 Ergebnis für "fsm7"

Schließlich noch ein größeres Beispiel:

```
(setq *problem*
      (make-problem
       :name "transform into network of smaller FSMs and SRs"
       :initial (find-system "planet")
       :goals '(fewstates
                shiftregister-structure
                (for-all-systems (system)
                      (typep system 'fsm)
                   (satisfies (<= (no-of-states system) 4)))
                (there-are-systems (system)
                      (typep system 'sr)
                   (>= (len system) 2))
                )
       :constraints
       '()))
```

Nach acht Ableitungsschritten ist das System "`planet`" in ein Netz kleiner Automaten und nicht-trivialer Schieberegister zerlegt.

```
Problem: "transform into network of smaller FSMs and SRs"
- Initial: "planet" of class FSM-LARGE
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 no result)
(trying rule 1 "parallel decomposition")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 2 "serial decomposition")
(rule 2 is applicable)
(executed rule 2 no result)
(trying rule 3 "shiftregister realization")
(rule 3 is applicable)
(executed rule 3 successfully)
(stop successful solved on "planet-sN")
(8 INFERENCES)
```

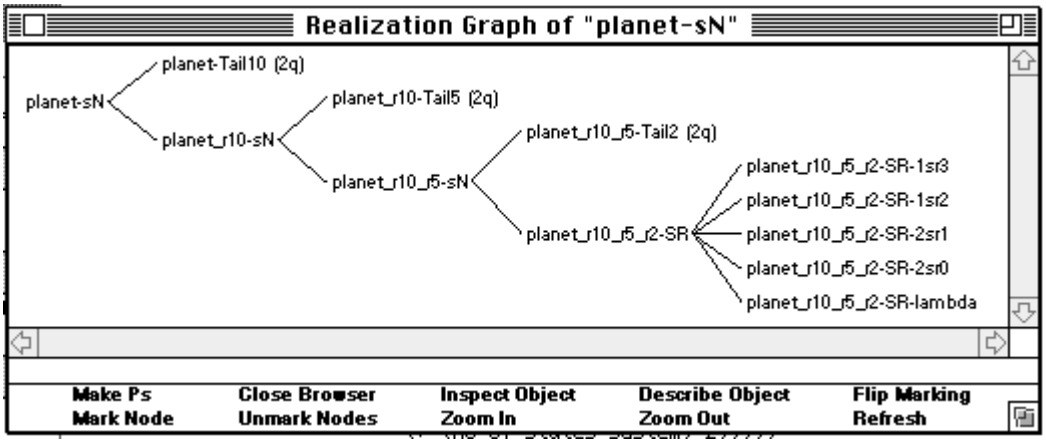

Bild 8.7 Schieberegisterdarstellung

8.4.2. Andere Regelbasen

Eine andere Regelbasis ist durch `*parallel-rules*` gegeben. Diese Regeln werden hier parallel genannt, weil sie eine Systemtransformation immer gleich auf alle Knoten anzuwenden versuchen. Die Lösungssuche wird dadurch schneller, aber unflexibler.

```
(defvar *parallel-rules* '(
((1 "parallel decompositions")
 (and (there-is-goal fewstates)
      (there-are-systems (m)
            (typep m 'fsm)
         (delta-complete-p m)
         (> (no-of-states m) 7)))
 (action (for-all-nodes (node)
               (and (typep (system node) 'fsm)
                    (delta-complete-p (system node))
                    (>= (no-of-states (system node)) 8))
            (pdecomp-transformation node :selection-fn #'first)))
 deterministic)
((2 "serial decompositions")
 (and (there-is-goal fewstates)
      (there-are-systems (m)
            (typep m 'fsm)
         (delta-complete-p m)
         (> (no-of-states m) 3)))
 (action (for-all-nodes (node)
               (and (typep (system node) 'fsm)
                    (delta-complete-p (system node))
                    (>= (no-of-states (system node)) 4))
            (sdecomp-transformation node :selection-fn #'first)))
 deterministic)
((3 "shiftregister realizations")
 (and (or (there-is-goal binary-coded)
          (there-is-goal shiftregister-structure))
      (there-are-systems (m)
            (typep m 'fsm)
         (> (no-of-states m) 2)))
 (action (for-all-nodes (node)
               (and (typep (system node) 'fsm)
                    (> (no-of-states (system node)) 2))
            (sr-real-transformation node)))
 deterministic)
))
```

Zunächst wird wieder dieses Problem getestet:

```
(setq *problem*
      (make-problem
       :name "transform into decomposition network of small components"
       :initial (find-system "fsm7")
       :goals '(fewstates
                (for-all-systems (system)
                      (typep system 'fsm)
                   (satisfies (<= (no-of-states system) 4)))
                )
       :constraints
       '((there-are-systems (system)
              (typep system 'fsm)
          (> (no-of-states system) 2)))))
```

Dies ist dasselbe Problem, wie oben bereits mit den Standardregeln behandelt.

```
(print-report)
Problem: "transform into decomposition network of small components"
- Initial: "fsm7" of class FSM-LARGE
(trying rule 1 "parallel decompositions")
(rule 1 not applicable)
(trying rule 2 "serial decompositions")
(rule 2 is applicable)
(executed rule 2 successfully)
(stop successful solved on "fsm7-sN")
(2 INFERENCES)
```

Die Lösung wird aber bedeutend schneller gefunden.

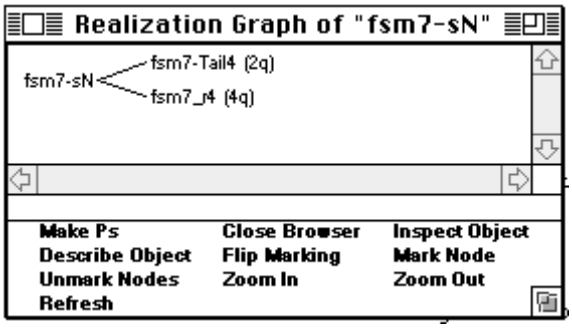

Bild 8.8 Lösung für "fsm7" mit parallelen Regeln

Nun wird noch einmal das Problem der binären kleinen Komponenten mit Constraints hergenommen:

```
(setq *problem*
      (make-problem
       :name "transform into decomposition network of small bin components"
       :initial (find-system "bbsse-e")
       :goals '(fewstates
                binary-coded
                (for-all-systems (system)
                      (typep system 'fsm)
                   (satisfies (<= (no-of-states system) 5)))
                )
       :constraints
       '((there-are-systems (system)
               (typep system 'fsm)
            (> (no-of-states system) 2)))))
```

Hier wird keine Lösung gefunden:

```
Problem: "transform into decomposition network of small bin components"
- Initial: "bbsse-e" of class FSM-LARGE
(trying rule 1 "parallel decompositions")
(rule 1 is applicable)
(executed rule 1 successfully)
(trying rule 1 "parallel decompositions")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 1 "parallel decompositions")
(rule 1 is applicable)
(executed rule 1 no result)
(trying rule 2 "serial decompositions")
(rule 2 is applicable)
(executed rule 2 successfully)
(trying rule 2 "serial decompositions")
(rule 2 not applicable)
(trying rule 1 "parallel decompositions")
(rule 1 not applicable)
(trying rule 2 "serial decompositions")
(rule 2 not applicable)
(trying rule 3 "shiftregister realizations")
(rule 3 is applicable)
(executed rule 3 successfully)
(constraints force undo of rule 3)
(stop unsuccessful no-more-rules-found)
(8 INFERENCES)
```

Aus diesen und vielen weiteren Beispielen kann man schließen, daß Regeln in der Art der Standardregelbasis im allgemeinen leistungsfähiger sind.

In den Source-Files sind weitere Beispiele für andere Regeln vorhanden (`*benchmark-rules*`, `*rm-rules*`). Aus Platzgründen wird hier auf eine Wiedergabe verzichtet, der interessierte Leser wird auf die Originalprogrammdateien verwiesen.

Durch kleine Änderungen in einzelnen Regeln läßt sich das Verhalten von STX in vielfältiger Weise beeinflussen. So sind die Mindestanzahlen von Zuständen, wie sie in den hier gezeigten Regeln vorkommen, ein weites Feld für weitere Experimente. Auch die Formulierung der Systemprobleme ist von großem Einfluß darauf, ob Ergebnisse gefunden werden können und von welcher Qualität diese sind. Der Benutzer von STX hat demnach trotz eines in gewissem

Sinne automatisierten Problemlösungsprozesses noch genug Möglichkeiten, eigene Vorstellungen einzubringen und Experimente zu gestalten.

Tiefergehende Veränderungen ergeben sich durch die Weiterentwicklung der Inferenzmaschine, durch Erweiterung der Ausdrucksfähigkeit der verwendeten Sprachmittel zur Problembeschreibung und zur Abfassung der Regeln, und durch Verbesserung der Werkzeuge, wie Erklärungskomponente, Interaktivität, Oberfläche, etc.

9. Kapitel - Verallgemeinerungen und Ausblick

9.1. Unabhängigkeit von einem bestimmten Anwendungsgebiet

Wie in Kapitel 7 dargestellt wurde, ist die strikte Trennung von Ablaufsteuerung und anwendungsgebietsabhängigem Wissen die herausragende Eigenschaft wissensbasierter Systeme. Um ein wissensbasiertes System in einem anderen Anwendungsgebiet einsetzen zu können, ist demnach im Idealfall nur die Wissensbasis, meist also der Regelsatz, auszutauschen. (In der Praxis können natürlich auch noch andere Anpassungen, wie etwa eine Änderung der Benutzerschnittstelle, erforderlich sein.)

In Kapitel 1 wurde der Anspruch des Wissenschaftsgebietes Systemtheorie dargestellt. Ein wichtiges Ziel der Systemtheorie ist die zur Verfügungstellung von praktisch anwendbaren Methoden in den diversen Ingenieurdisziplinen. Eine universelle STIPS-Maschine, die alles kann, erscheint aber wenig realistisch. Daher wurde schon bei der Notation der STIPS-Maschine die Beschränkung auf einen Themenschwerpunkt F eingeführt. Je nach Anwendungsschwerpunkt können so spezifische Mengen von Systemtypen S_F und betrachteten Systemtransformationen T_F gebildet werden. Das computer-unterstützte systemtheoretische Problemlösen wurde im Rahmen des CAST-Projektes im wesentlichen für die Anwendung beim Design von komplexen digitalen Systemen (CAST-FSM) erforscht [Müll94]. Prinzipiell sollte die Vorgangsweise aber in anderen verwandten Disziplinen, wie Nachrichtentechnik, Kryptographie, Steuer- und Regelungstechnik usw. auf gleiche Weise funktionieren, und auch die Anwendung in ganz anderen Gebieten, wie Maschinenbau, Ökologie, etc. erscheint möglich. Für die Zukunft besonders interessant ist das Zusammenspiel von Informatik, Digitaltechnik, Analogtechnik, Mikromechanik, Optik und evtl. weiterer Gebiete bei mikromechatronischen Entwicklungen.

Auch das automatisierte Problemlösen mit STX kann auf andere Anwendungsgebiete übertragen werden.

Die in dieser Arbeit vorgestellte $STX_F := (STIPS_F, R(T_F), D_F, E, P(F))$ (aus Kap. 7.3.1.) enthält domainspezifisches Wissen in der Regelbasis R, wie jedes Expertensystem. Aber auch die ausführende Einheit der STX, nämlich ihre STIPS-Maschine, ist für das Anwendungsgebiet F bestimmt. Tatsächlich muß ja in R für jede Transformation *t* in T wenigstens eine Regel vorhan-

den sein, die die Bedingungen für die Ausführung von *t* beschreibt. Die Regeln von STX ergeben erst im Zusammenspiel mit den Transformationen T einen Sinn.

Soll also das Konzept der mit STX automatisierten Problemlösung von einem Anwendungsgebiet F auf ein anderes Anwendungsgebiet F' übertragen werden, muß zunächst eine STIPS-Maschine für F' erstellt werden, und dann kann die neue Regelbasis $R(T_F)$ geschrieben werden.

Zusammen mit der für das neue Thema F' interessierenden Systemproblemklasse P(F') ergibt sich die neue STX zu: $STX_{F'} := (STIPS_{F'}, R(T_{F'}), D_{F'}, E, P(F'))$. Insbesondere der Inferenzmechanismus E ist vom Übergang von F zu F' nicht betroffen.

9.2. CAST-CRYPTO

Zur computerunterstützten Untersuchung von kryptologischen Fragestellungen wurde die STIPS-Maschine CAST-CRYPTO entwickelt. Dieses System [Mitt94-2] entstand als Erweiterung von CAST-FSM, bzw. macCASTfsm (siehe Kapitel 5).

Ein typisches Syntheseproblem aus der angewandten Kryptographie ist die Erzeugung von Pseudozufallsfolgen, die gewisse Qualitätseigenschaften haben müssen.

Ein weiteres Syntheseproblem ist die Erzeugung geeigneter Pseudozufallsgeneratoren, beispielsweise als digitales Schaltwerk, oder spezieller, als rückgekoppeltes Schieberegister.

Besonders wichtig ist das zugehörige Analyseproblem: Von welcher kryptographischen Qualität ist eine gegebene Folge? Welche Pseudozufallseigenschaften erfüllt sie? Je nach Problem kann eine statistische Auswertung oder eine graphische Anzeige der Analyseergebnisse sinnvoller sein.

9.2.1. Beispiele für graphische Analysemethoden

Eine für die Kryptoanalyse interessante Frage ist die nach der linearen Komplexität einer vorliegenden Folge. Wie lang ist das kürzeste linear rückgekoppelte Schieberegister, welches diese Folge produziert? Diese Frage kann mit der Methode nach Massey-Berlekamp sogar konstruktiv beantwortet werden. Ein Beispiel für die Anwendung dieser Analysemethode auf eine Folge von 1024 Bit zeigt Bild 9.1. Es zeigt sich, daß die Folge eine lineare Komplexität von nur 54 hat, obwohl sie in allen anderen Tests sehr gut abschneidet, und somit für Verschlüsselungszwecke eine viel zu geringe Sicherheit bietet.

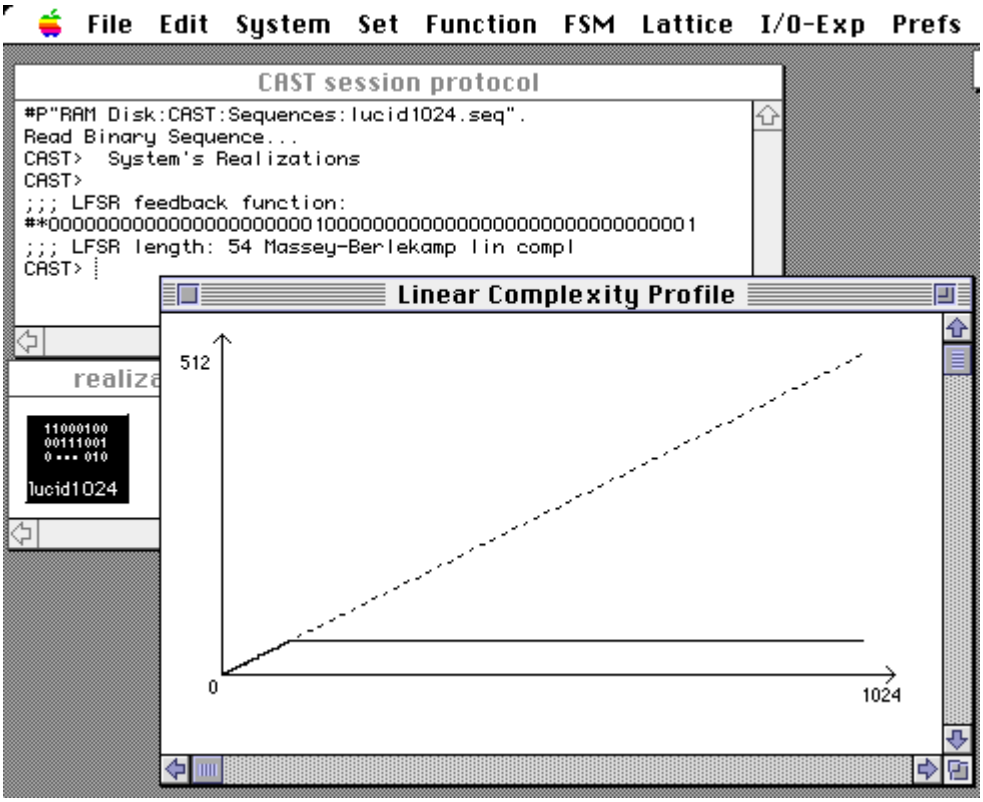

Bild 9.1 Ergebnis einer Analysemethode in CAST-CRYPTO

Bild 9.2 zeigt die graphische Ausgabe einer Auszählung bestimmter Bitmuster in einer Binärfolge.

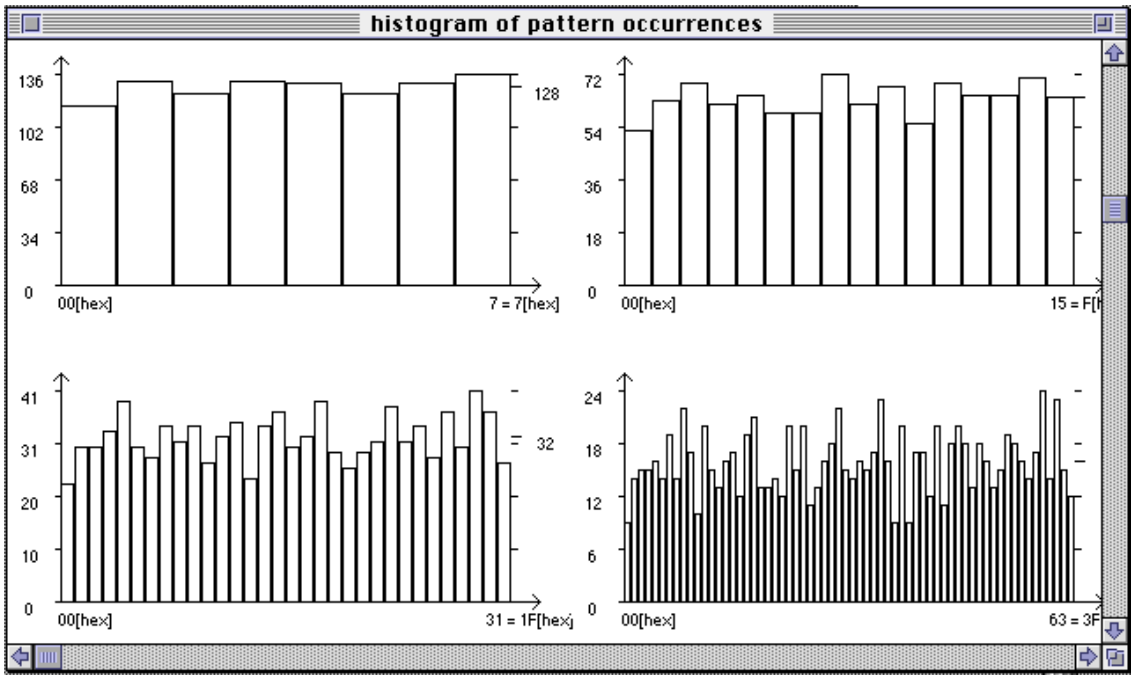

Bild 9.2 Ergebnis einer weiteren Analysemethode mit graphischer Ausgabe

Die hier aufgezählten Problemklassen können mit CAST-CRYPTO bearbeitet werden. Wieder (wie bei macCASTfsm) liegt der Zustand der STIPS-Maschine als Lösungsgraph vor und wird als Browser angezeigt. Die jeweils für ein ausgewähltes System anwendbaren Systemtransformationen werden interaktiv angeboten. Die Navigation bleibt, wie bei CAST-FSM, dem Benutzer überlassen. Allerdings enthält CAST-CRYPTO den vorgefertigten Systemalgorithmus ("Testbatterie") für das häufigste Analyseproblem, nämlich die statistische Auswertung einer Menge von Zufälligkeitstests über gleichmäßige Teilstücke einer gegebenen Folge [Knut81].

9.2.2. Eine STX für die Kryptographie

Die Implementierung einer STX_{CRYPTO} als automatisiertes Werkzeug für kryptographische Anwendungen ist in der im vorangegangenen Abschnitt dargestellten Weise möglich. Die Regeln würden die Anwendung der diversen Synthese- und Analysemethoden nach den in der Problemspezifikation angegebenen Zielen steuern. Auch die einzelnen Schritte und Parameter der Testbatterie könnten durch entsprechende Regeln an die Eigenschaften der zu testenden Folge angepaßt werden.

(Aufgrund der Tatsache, daß in bisherigen Projekten die Anwendung des Analyse-Systemalgorithmus test-battery als Problemlösungsmethode dominierend und erfolgreich war, wurde der Erstellung einer STX für diese Zwecke bisher keine Priorität eingeräumt.)

9.3. Vertiefungsthemen und offene Probleme

Neben den bereits in 8.4.2. angedeuteten Möglichkeiten zur Verbesserung der Lösungsfindung der STX sollen hier weitere Ansätze aufgezeigt werden.

9.3.1. Sprachen für Constraints und Bewertungsmetriken

Eine Schwäche der in Kapitel 8 vorgestellten STX liegt in der zu großen Mächtigkeit der verwendeten Sprachen für die Problembeschreibung, insbesondere für die Zielspezifikation und die Constraints. Eine Aufgabe für zukünftige Verbesserungen ist daher die Erarbeitung einer zwar ausdrucksstarken, aber vereinfachten Beschreibungssprache, die insbesondere auch die rückwärtsverketteten Inferenzmethoden besser unterstützt. Als Vorbild können publizierte Systeme, wie NONLIN oder UCPOP [Barr94], dienen.

Allerdings erscheint die wiederum zu geringe Leistungsfähigkeit der Sprachen dieser Systeme teilweise unvereinbar mit der Ankopplung an eine STIPS-Maschine. Diese Problematik ist noch offen und sollte in weiterführenden Arbeiten untersucht und gelöst werden.

9.3.2. Anwendungsnahe Heuristiken

In der derzeitigen Version der STX ist in der Regelbasis keine klare Trennung von definitivem Wissen und Heuristiken vorgesehen. Der Inferenzmechanismus könnte bei einer Trennung aber entsprechend optimiert werden. Es ist daher eine saubere Trennung in der Regelsprache anzustreben zwischen eher unscharfem heuristischem Wissen und klaren (mathematischen) Tatsachen.

9.3.3. Systemspezifikationen und Problemklassen

Eine weitere Richtung, in der die bestehende STX verbessert werden könnte, ist die Realisierung von Systemproblemobjekten. Unter Ausnutzung objektorientierter Programmierung könnten Verallgemeinerungen und Spezialisierungen von Systemproblemklassen formalisiert werden. Der in dieser Arbeit vorgestellte Mechanismus allgemeiner Systemspezifikationen wird dann zum zentralen Baustein für die Problemdefinition und kann von der Inferenzmaschine zur Steuerung des Problemlösungsprozesses verwendet werden.

A. Anwendung von CAST - FSM in der Lehre

In diesem Anhang werden die Möglichkeiten und Erfahrungen mit dem Einsatz von CAST-FSM in der Lehre der Abteilung Systemtheorie und Informationstechnik der Johannes Kepler Universität beschrieben. Diese Erfahrungen gründen sich zunächst auf die Durchführung verschiedener Programmierpraktika mit der "CAST:FSM" Version in den Jahren 1987 bis 1989, dann mit CAST-FSM ab 1990, und jetzt auch mit den Grundvorlesungen für den ersten Studienabschnitt "Diskrete Mathematik" und "Systemtheorie". Daneben wurden auch früher in diesen Grundvorlesungen und anderen Lehrveranstaltungen immer wieder Demonstrationen der jeweils aktuellen CAST Systeme abgehalten, um die Lebendigkeit der Lehrinhalte vorzuführen und Diplomanden anzuwerben.

A.1. CAST-Programmierpraktikum

In den Programmierpraktika wurden die Studenten zunächst mit den Grundprinzipien von CAST-FSM vertraut gemacht, sowie mit seiner Implementierung. Meist war es erforderlich, die Teilnehmer auch in die für CAST verwendete Programmiersprache LISP einzuführen. Als Projektaufgabe wurde dann eine kleine Erweiterung von CAST-FSM aufgegeben. Manche der so entstandenen Programmpakete war für eine dauerhafte Einbindung in das aktuelle CAST System nicht geeignet, aber andere Teile konnten dann (evtl. nach kleinen Nachbesserungen) in das Gesamtsystem übernommen werden.

A.2. Programmiertechniken der KI

Da in den Programmierprojekten regelmäßig eine Einführung der Teilnehmer in die Philosophie und die Praxis der für CAST fast ausschließlich verwendeten Programmiersprache Lisp gegeben werden mußte, bevor eigentliche CAST-Themen bearbeitet werden konnten, und diese Einführung selbst bei äußerster Knappheit die Hälfte des Semesters beanspruchte, lag es nahe, die Programmierung in Lisp in einer eigenen Lehrveranstaltung (Vorlesung und Übung) zu lehren (Prähofer/Mittelmann ab WS 92/93). Der Besuch dieser Veranstaltung wurde zwar nicht formal zur Voraussetzung für CAST-Programmierprojekte und Diplomarbeiten gemacht,

aber den Studenten doch dringend empfohlen. Ein weiterer Schwerpunkt der "Programmiertechniken" neben den schon im Titel genannten KI-Methoden (z.B. regelbasiertes Programmieren) war auch die objektorientierte Programmierung mit der in Lisp integrierten, extrem flexiblen objektorientierten Erweiterung CLOS, die für CAST von zentraler Bedeutung ist.

A.3. Übungen mit macCASTfsm

In **macCASTfsm** können wesentliche Teile der Automatentheorie geübt und vertieft werden, sowie einfache kryptologische Untersuchungen durchgeführt werden. Es ist damit für den Einsatz in den Übungen für die von der Abteilung für Systemtheorie betreuten Pflichtlehrveranstaltungen geeignet. Unter Verwendung des eingebauten Lisp-Interpreters kann der Benutzer die einfachsten Grundlagen der Programmiersprache Lisp erlernen, soweit ihre Kenntnis zur Konstruktion von eigenen Systemalgorithmen notwendig ist.

Natürlich ist macCASTfsm kein Lernprogramm für autodidaktisches Erlernen der Automatentheorie. Das Mitlernen in der Vorlesung und das Studium der Fachliteratur wird für den Studenten auf keinen Fall überflüssig. Aber die Ausführung der einzelnen Algorithmen für größere Beispiele mit Papier und Bleistift ist eine fehleranfällige, langwierige und oft auch stupide Arbeit, die man sich durch die Verwendung von CAST ersparen kann. Die Umsetzung von allgemeinen Aufgaben in konkrete Abfolgen von Systemmethodenanwendungen, also in anderen Worten die Konstruktion von Systemalgorithmen zur Lösung von Systemproblemen, bleibt dem Benutzer von macCASTfsm überlassen - die tatsächliche Ausführung der einzelnen Schritte (Systemtransformationen) dagegen werden vom Rechner erledigt. Als Entscheidungshilfe bei der Auswahl der Schritte stehen dabei Hilfsmittel zur Verfügung, deren wichtigste der Verbandsgraph der Maschinenkongruenzen und der V-Graph der Schieberegisterpartitionen sind, sowie einige kleinere Analysefunktionen. Die mit diesen Werkzeugen erhaltenen Analyseresultate kann der Benutzer aber wiederum nur mit dem entsprechenden theoretischen Wissen optimal interpretieren.

A.3.1. "Diskrete Strukturen"- und "Systemtheorie"-Übungen

Die Themen, die in macCASTfsm ganz oder teilweise abgedeckt werden, sind:

CAST-FSM

- Definition von Automaten und Schaltfunktionen (Schaltnetzen)
- Analyse: Berechnung von Homing, Synchronisation und Diagnosefolgen
- Berechnung des Verbandes der Maschinenkongruenzen

- Zustandsreduktion, Quotientenautomaten, r-Quotienten
- Dekompositionen (nach der algebraischen Strukturtheorie)
- Schieberegisterrealisierung (nach Böhling)
- Maschineninversion
- Abspeichern von berechneten Systemen in Standardformaten

CAST-CRYPTO

- Definition einer Zahlenfolge
- Häufigkeitsanalyse mit Histogrammen
- Visueller Test
- Messung der linearen Komplexität
- Statistische Tests
- Fourier- und Walsh-Transformationen
- Synthese von LFSR mit vorgegebener Periode
- Synthese von maximalperiodischen SR vorgegebener Länge
- Synthese von Pseudozufallsfolgen mit 4 verschiedenen Generatoren

Bei all diesen Themen werden die eigentlichen Algorithmen natürlich von der Maschine erledigt. Der Student befaßt sich z.B. nicht mit den Details der Berechnung des Verbandes eines Automaten. Vielmehr lernt er beim Umgang mit macCASTfsm bei der Bearbeitung größerer Beispiele, wie der automatisch errechnete Verband zu interpretieren ist. Allgemeiner gesagt, nicht die automatentheoretischen Algorithmen selbst sind hier das Lernziel, sondern ihre Interpretation und sinnvolle Anwendung (auf Beispielprobleme).

In **macCASTfsm** können auch eigene Algorithmen entwickelt werden, die vor allem im Sinne von Systemalgorithmen auf die vorhandenen Methoden zurückgreifen und diese zu komplexeren Abläufen kombinieren sollen. Die Kenntnis der Grundzüge der Programmiersprache Lisp ist hierfür Voraussetzung.

Eine "nahtlose" Erweiterung der in **macCASTfsm** eingebauten Klassen und Methoden ist jedoch nicht möglich, da hierfür die ganze (sehr kostspielige, lizenzpflichtige) Lisp-Entwicklungsumgebung benötigt würde. Für solche Lehrveranstaltungen, in denen eine echte Erweiterung von CAST-FSM behandelt werden soll, muß daher ein volles Lisp mit den gesamten CAST-Files zur Verfügung stehen (z.B. CAST-Programmierpraktikum).

Weitere Literatur

Für intensivere Arbeit mit CAST-FSM bzw. macCASTfsm müßten die Berichte

- "CAST-FSM" R. Mittelmann, interner Bericht am Institut für Systemwissenschaften, 1993 ,
- "FSM Extensions" F. Auernig und R. Mittelmann, interner Bericht am Institut für Systemwissenschaften, 1992 ,

die Diplomarbeit

- "Implementierung der Schieberegisterrealisierung für CAST-Systeme" J. Scharinger, Diplomarbeit am Institut für Systemwissenschaften, 1992

und das Buch

- "CAST - Methods in Modelling" F. Pichler, H. Schwärtzel (eds.), Springer Berlin, 1992

herangezogen werden; für Erweiterungsarbeiten an CAST-FSM selbst müßten einschlägige Lisp-Bücher sowie der CAST-FSM Source Code verwendet werden.

A.4. Übungsaufgabensammlung

Die im folgenden vorgestellte Aufgabensammlung kann einerseits als Grundlage für die Gestaltung eines mit macCASTfsm gestalteten Übungsbetriebes verwendet werden. Andererseits, und das ist der eigentliche Grund, sie in diese Arbeit aufzunehmen, zeigt die Aufgabensammlung in kompakter Form, welche Arten von Problemen mit CAST-FSM bzw. CAST-CRYPTO tatsächlich bearbeitet werden können: *Jede Aufgabe bzw. Aufgabengruppe zeigt ein konkretes Beispiel für die computerunterstützte Lösung eines Systemproblems*.

A.4.1. CAST-FSM

A.4.1.1. Erstellen und Editieren von Systemen

1. Machen Sie sich mit dem User's Guide und der **macCASTfsm** Umgebung vertraut.

2.1. Öffnen Sie eine neue Textfile (File: New) und schreiben Sie den folgenden Automaten im "kiss2"-Format:

δ, λ	0	1	0	1
s1	s3	s4	1	0
s2	s3	s4	1	0
s3	s4	s4	0	1
s4	s1	s2	0	1

Anleitung: Verwenden Sie das i-o-s-p Format (User's Guide Kap. 2; vergleichen Sie auch mit den Beispielen in der Mappe ":FSMs:").
Sichern Sie die Datei unter dem Namen "M2.kiss2" in der (lokalen) Mappe ":FSMs:". Laden Sie das System (System: Load System...) und öffnen Sie einen Realisierungsgraphen. Wenden Sie den Menübefehl "Describe System" auf das selektierte Icon des M2 an, um Ihre Definition zu überprüfen.

2.2. Analysieren Sie den Automaten. Schließen Sie aus den Resultaten, welche Synthesemethoden in Betracht kommen und welche sicher nicht angewendet werden können. Z.B.:
Kann M2 parallelzerlegt werden?
Kann M2 serienzerlegt werden?
Kann M2 als nichttriviale (d.h. Länge(SR) >1) Schieberegisterstruktur realisiert werden?
Probieren Sie einige Methoden aus.

2.3. Erzeugen Sie ein I/O-Experiment (der Länge 16) für M2 und sehen Sie sich dieses mit dem Inspector an. Führen Sie das I/O-Experiment durch, vergleichen Sie die Anzeige im Inspector (nach "Recompute" im Titelmenü des Inspectors). Speichern Sie das Experiment ab (mit "Save System..."). Ändern Sie den Automaten M2 in der Textrepräsentation an wenigen Stellen leicht ab (z.B. Lambda(s4, 0) = 1 und ähnliches). Speichern Sie diese Variation als "M2-1". Laden Sie den geänderten Automaten M2-1. Laden Sie das I/O-Experiment für diesen Automaten, dabei muß sein Icon selektiert sein. Führen Sie es erneut aus und beachten Sie die Ausgabe im Protokollfenster.

A.4.1.2. Algebraische Strukturtheorie

3.1 Laden Sie den Automaten "bbsse-e.kiss2". Analysieren Sie ihn und untersuchen Sie insbesondere seinen Verband. (Anmerkung: Manche Berechnungen können für größere Objekte nicht durchgeführt werden, brechen Sie dann ab mit Befehl-Punkt.)

3.2. Erklären Sie den Begriff "Output-konsistente Partition". Welche Partitionen des bbsse-e sind Output-konsistent?

3.3. Beachten Sie die Beschriftung der Knoten im Verbandsdiagramm. Welche Partition beschreibt den zustandsreduzierten Automaten? Diese Reduktionsäquivalenz wollen wir mit Q bezeichnen. Welche Informationen über den reduzierten Automaten lassen sich aus dem Graphen (und seiner Beschriftung), insbesondere aus Q, direkt ablesen? Welche Zustände werden in Q zu einem Block zusammengefaßt?
(*Anleitung*: Am einfachsten erhält man einen Pointer namens "=o" auf eine Partition durch einen Befehlsklick. Den daraufhin erscheinenden Inspector schließt man sofort wieder. Dann tippt man "`(cprint =o)`" im Listener.)

3.4. Bilden Sie den Automaten bbsse-e/Q mit Hilfe des Kommandos "Divide by selected partition". Dazu muß die Relation Q im Verband markiert sein. Bilden Sie die Projektionen und wenden Sie die Analysemethoden an, um das System bbsse-e/Q kennenzulernen. Vergleichen Sie mit ihren Ergebnissen aus 3.3.

3.5. Betrachten Sie die Partition R, die 12 Blöcke hat, mit max. 3 Zuständen in einem Block. Welche Zustände werden zu einem Block zusammengefaßt?
Verwenden Sie das Fenster-Menu des Verbandsbrowsers, um den Untergraphen bzgl. R anzuzeigen.

3.6. Bilden Sie den r-Quotienten M2/R. Untersuchen Sie dieses System wie in 3.4.. Um welche Art von System handelt es sich hier? Vergleichen Sie den Verband von M2/R mit dem vorher erzeugten Untergraphen bzgl. R. Wie unterscheiden sich die beiden?

3.7. Der Automat bbsse-e soll nun als Parallelschaltung realisiert werden. Welche Partitionen kommen hierfür in Frage? Beachten Sie die Beschriftung der Knoten. Wieviele Zustände werden die Teilautomaten haben (vgl. 3.6)? Welche Parallelzerlegung erscheint daher in Bezug auf die Gesamtzahl der Zustände optimal? Wieviele Speicherzellen (Flipflops) werden pro Teilautomat benötigt, um die Zustände kodieren zu können?

3.8 Selektieren Sie die von Ihnen ausgewählten Kongruenzrelationen und führen Sie dann die Parallelzerlegung durch. Studieren Sie das Ergebnis und vergleichen Sie mit Ihren Voraussagen aus 3.7. Verwenden Sie dazu die Projektionsoperation und die diversen Analysemethoden. Speichern Sie außerdem die Teilautomaten und die Ausgabekodierfunktion als Textfile ab. Kontrollieren Sie diese Darstellungen.

3.9. Der Automat soll nun noch als Serienschaltung realisiert werden. Welche Partitionen eignen sich für eine Serienschaltung? Dies läßt sich wiederum aus der Knotenbeschriftung im Verbandsgraphen ablesen. Die Serienschaltung wird durch eine einzige Kongruenzrelation bestimmt, die weiters benötigte Äquivalenzrelation wird von macCASTfsm passend ergänzt. Die Anzahl der Zustände des Kopfautomaten ist die Anzahl der Blöcke in der Kongruenz. Die

Anzahl der Zustände des Schwanzautomaten ist durch die Anzahl der Elemente im größten Block derselben Kongruenz bestimmt.

3.10. Selektieren Sie die Ihnen optimal erscheinende Relation und führen Sie die Serienzerlegung durch. Betrachten Sie die Realisierung mit den Analysewerkzeugen. Stimmen die Ergebnisse mit Ihren Voraussagen überein? Speichern Sie wieder die Teilautomaten als Textfile und kontrollieren Sie diese.

3.11. Wenn Sie zwei Kongruenzrelationen für eine Parallelschaltung auswählen, können verschiedene Fälle unterschieden werden:
(a) Das Produkt der beiden Relationen ist die Null.
(b) Das Produkt ist eine Output-konsistente Relation größer als Null.
(c) Das Produkt ist nicht Output-konsistent.
Spielen Sie jeden dieser drei Fälle durch, indem Sie die Partitionen jeweils entsprechend wählen. Welche Ergebnisse erhalten Sie?
Wie reagiert macCASTfsm im Fall (c)?

A.4.1.3. Schieberegisterrealisierung I

4.1. Laden Sie den Automaten "shiftreg.kiss2" aus der Mappe "FSMs". Machen Sie sich mit Hilfe der Analyseoperationen mit diesem Automaten vertraut.

4.2. Interpretieren Sie die Textdarstellung des Automaten: was tut diese Maschine?
Verifizieren Sie Ihre Vorstellung, indem Sie mit macCASTfsm I/O-Experimente simulieren.

4.3. Sehen Sie sich den Verband dieses Automaten an. Was läßt sich über die Anwendbarkeit der Dekompositionsmethoden sagen?

4.4. Lassen Sie den Verband der Schieberegisterpartitionen (V-Graphen) berechnen. Wie lang ist das längste mögliche Schieberegister? Andererseits, wieviele Speicherzellen benötigt man in jedem Fall für irgendeine Schaltwerksrealisierung dieses Automaten?
Welche Qualität wird das längstmögliche Schieberegister haben? Wieviele zusätzliche speichernde Elemente werden benötigt werden?
(*Hinweis*: Aus Knotenbeschriftung im V-Graphen und dem Protokoll der Analyse im "CAST-protocoll"-Fenster ablesen.)

4.5. Führen Sie die Schieberegistertransformation durch (mit den vorgegebenen Parametern) und analysieren Sie das Ergebnis (mehrfache Projektionsanwendung etc.). Vergleichen Sie mit 4.4.

4.6. Zeichnen Sie ein Blockschaltbild der Schaltwerkstruktur aus 4.5. (Netzwerk aus Schiebezellen und Schaltnetzen)

4.7. Wenden Sie die Funktion make-fun-block an, um den Schaltnetz-Teil des Schaltwerkes aus 4.5. zu einem Objekt zusammenzufassen und speichern Sie dieses als Textfile ab.
Hinweis: Für den Wechsel von graphischer Benutzerschnittstelle zur Kommandoschnittstelle siehe Kapitel 6.1.1 im User's Guide. Als Beispiel für die Verwendung von "make-fun-block" siehe die Definition der Funktion "shiftreg-real-sys-alg" in der Beispieldatei "LISP-examples.CAST" in der Mappe "CAST programs".
Untersuchen Sie diese Textdarstellung des Funktionsblockes. Welche der Ausgabefunktionen gehören zur Zustandsüberführung, welche zur Ausgabe des Schaltwerkes?
Zeichnen Sie das Blockschaltbild jetzt so, daß nur ein Schaltnetz vorkommt.

A.4.1.4. Schieberegisterrealisierung II

5.0. Schließen Sie alle Fenster, die zu den bisher behandelten Objekten gehören, und zerstören Sie die Objekte selbst, um Hauptspeicher freizumachen.

5.1. Laden Sie das System "planet.kiss2". Öffnen Sie den Realisierungsgraphen und lassen Sie den V-Graphen anzeigen. (Die Berechnung dauert einige Zeit.)

5.2 Wie lang ist hier das längste mögliche Schieberegister?
Welche Qualität wird das längstmögliche Schieberegister haben? Wieviele zusätzliche speichernde Elemente werden benötigt werden? (Hinweis bei 4.4.)

5.3. Führen Sie die Schieberegistertransformation durch und analysieren Sie das Ergebnis. Führen Sie diese Transformation noch einmal durch, verdoppeln Sie dabei die Suchtiefe (vorgegebener Wert 24) auf 48. Wie hat sich das Ergebnis verändert? Führen Sie die Transformation einige Male durch. Ist das Ergebnis immer das gleiche? (Hinweis: Der Algorithmus verwendet eine zufallsgesteuerte Suchmethode.)
Welches Ergebnis wird optimal sein? Erhöhen Sie auf 100, um (mit einiger Sicherheit) zum optimalen Ergebnis zu kommen.

5.4. Zeichnen Sie wieder das Blockschaltbild der Schaltwerkstruktur aus 5.3. (Netzwerk aus Schiebezellen und Schaltnetzen)

6. Führen Sie dieselben Aufgaben wie in 5.1.-5.4. mit dem Automaten "ex4.kiss2" aus.

A.4.1.5. Maschineninversion

7.1. Laden Sie das System "dk27". Analysieren Sie diesen Automaten wie gewohnt, um ihn kennenzulernen. Der Automat ist von endlicher Verlustlosigkeit - von welcher Ordnung? Was bedeutet das?

7.2. Erzeugen Sie für dk27 ein I/O-Experiment der Länge 64 und führen Sie es aus. Öffnen Sie einen Inspector für dieses Experiment.

7.3. Invertieren Sie den Automaten dk27. Erzeugen Sie für den resultierenden Automaten dk27-i ein I/O-Experiment der Länge 64, ohne es auszuführen. Öffnen Sie auch für dieses Experiment einen Inspector. Kopieren Sie nun die Ausgabefolge des ersten Experiments in die Eingabefolge des zweiten Experiments.
Der invertierte Automat dk27-i müßte nun bei der Ausführung seines Experiments als Ausgabefolge die Eingabefolge des ersten Experimentes (von dk27) errechnen. Dazu muß aber erst noch der richtige Anfangszustand gefunden werden. Außerdem wird die Ausgabe von dk27-i erst um (Ordnung - 1) Takte verzögert der Eingabefolge von dk27 entsprechen. Sie können versuchen, einen geeigneten Anfangszustand für das Experiment von dk27-i durch Raten zu finden. Führen Sie dann das Experiment durch, und vergleichen Sie die Folgen (nach "Recompute" des Inspectors).

7.4. Sehen Sie sich nun die CAST-Programmdatei "Inverse Machines.CAST" in der Mappe "CAST programs" an. Verwenden Sie die dort beschriebene Vorgangsweise mit der Funktion "compute-inverse-initial-states", um die Kandidaten für einen passenden Anfangszustand für Ihr dk27-i-Experiment zu finden.

8.1. Erfinden Sie einen einfachen invertierbaren Automaten mit Verlustlosigkeit der Ordnung 2. Geben Sie diesen Automaten als Textfile im gewohnten PLA-Format ein und laden Sie das System. Verifizieren Sie die Verlustlosigkeit Ihrer Konstruktion mit der entsprechenden Analysemethode.

8.2. Führen Sie ein I/O-Experiment der Länge 16 durch.

8.3. Invertieren Sie ihren Automaten. Verifizieren Sie das Verhalten des invertierten Automaten durch das entsprechende I/O-Experiment mit der Ausgabefolge des ersten Experimentes als Eingabefolge und dem passenden Anfangszustand (entsprechend 7.4.).

A.4.1.6. Mealy- contra Moore-Automaten

9.1. Konvertieren Sie einen Mealy-Automaten (die Ausgabe hängt vom Zustand und vom aktuellen Input ab) in einen Moore-Automaten (die Ausgabe hängt nur vom Zustand ab). Vergleichen Sie beide Automaten in der Textdarstellung.

9.2. Erzeugen Sie I/O-Experimente für beide Automaten. Geben Sie beiden Experimenten die gleiche Eingabefolge und setzen Sie im zweiten Experiment einen Anfangszustand, der dem des anderen Experimentes entspricht.

9.3. Führen Sie beide Experimente durch und vergleichen Sie ihre beiden Ausgabefolgen. Was können Sie feststellen? Sind beide Folgen gleich, oder was ist der Unterschied? Wie erklären Sie das?

A.4.1.7. Schieberegisterrealisierung III

10.1. Laden Sie den Automaten "johnson.kiss2". Lassen Sie den V-Graphen anzeigen. Welche Ergebnisse versprechen Sie sich von einer Schieberegisterrealisierung?

10.2. Führen Sie die Schieberegistertransformation mehrmals durch, mit einer Suchtiefe von 100. Notieren Sie sich die Längenliste der Ergebnisstruktur als Qualitätsmaß (z.B. (3 1 1) oder (4 1 1)).

10.3. Führen Sie noch einige Male die Schieberegistertransformation durch, mit einer Suchtiefe von 100 und mit dem Heuristikschalter auf "few shift registers". (Die bisher immer verwendete Voreinstellung ist "less register cells".)
Vergleichen und interpretieren Sie die jetzt erhaltenen Qualitätslisten mit denen aus 10.2.

11.1. Öffnen Sie die Programmdatei "LISP examples.CAST" und evaluieren Sie die (am Ende der File stehende) Funktion "shiftreg-real-sys-alg-inter".

Anleitung: Doppelklicken Sie auf die öffnende Klammer am Zeilenanfang (vor `defun`), dies selektiert die gesamte Funktionsdefinition. Kopieren Sie diese und "pasten" Sie sie in das Lisp Listener Fenster. Drücken Sie die <enter>-Taste, um den Ausdruck zu evaluieren. Die Funktion ist dann definiert und kann sofort verwendet werden.

Evaluieren Sie dann den Ausdruck:
`(shiftreg-real-sys-alg-inter "johnson" :depth 50)`.
Die in dieser Funktion verwendete Selektionsfunktion "user-select" gestattet dem Benutzer die interaktive Auswahl der Schieberegisterindikatoren. Versuchen Sie, zu einem besseren Ergebnis zu kommen als die Funktion "auto-select", die in den bisherigen Aufgaben verwendet wurde.

11.2. Versuchen Sie die wesentlichen Elemente der Funktion shiftreg-real-sys-alg-inter zu verstehen. Ändern Sie die Definition so, daß das Simulationsobjekt zurückgegeben wird und der Funktionsblock nicht auf die File geschrieben wird. Probieren Sie andere kleinere Änderungen aus.

A.4.1.8. Testexperimente

12.1. Laden Sie wieder den Automaten "shiftreg.kiss2". Öffnen Sie seinen Realisierungsgraphen. Wählen Sie aus dem Menu "Prefs" den Befehl "Lisp Settings" und kreuzen Sie in diesem Dialog den Schalter "verbose" an.
Führen Sie für shiftreg dann die Analysemethode "Test Experiments" aus. Die erfolgende Ausgabe besagt: Dieser Automat ist vollständig testbar, d.h. CAST-FSM war in der Lage, wenigstens ein Synchronisationswort zu finden, und von diesem ausgehend, für jede Zustandsüberführung eine Homingfolge zu dem Ausgangszustand dieser Überführung und eine Diagnosefolge von dem Endzustand der Überführung zu finden.

12.2. Suchen Sie nach anderen Automaten mit guter Testbarkeit, und konstruieren Sie selber welche.

12.3. Führen Sie die zugrundeliegenden Lisp-Funktionen im Listener aus und untersuchen Sie die Ergebnisse. Für Synchronisationsfolgen verwenden Sie `(get-synchronizing-sequence fsm)`, für Homing verwenden Sie `(get-homing-sequence fsm)` und für Diagnose `(get-diagnose-sequence fsm)`. Verwenden Sie z.B. `(? get-diagnose-sequence)` für Information über die zurückgegebene Listenstruktur.

12.4. Wählen Sie einen Automaten mit wenigstens einer Synchronisationsfolge der Länge 3. Konstruieren Sie ein I/O-Experiment mit der Synchronisierfolge als Input-Folge und verifizieren Sie, daß der Automat mit jedem Anfangszustand den gewünschten Endzustand erreicht.

12.5. Wählen Sie einen Automaten mit einem Homing-Experiment der Länge 3 oder länger, konstruieren Sie das I/O-Experiment-Objekt und weisen Sie die Homing-Eigenschaft nach.

12.6. Wählen Sie einen Automaten mit einem Diagnose-Experiment der Länge 3 oder länger, konstruieren Sie das I/O-Experiment-Objekt und weisen Sie die Diagnose-Eigenschaft nach.

A.4.2. CAST-CRYPTO

A.4.2.1. Analyse einer Pseudozufallsfolge

C 0. Starten Sie macCASTfsm und laden Sie die CAST-Erweiterung "CAST-CRYPTO.fasl" mit dem Befehl "LoadLisp...".

C 1.1. Verwenden Sie das "Import.../Read Binary Sequence" Kommando aus dem "System"-Menu, um die Folge "lucid1024.seq" aus der Mappe "Sequences" zu laden. Verwenden Sie den Inspector, um die ersten Bits dieser pseudozufälligen Binärfolge anzusehen. Können Sie eine Regelmäßigkeit entdecken?
Öffnen Sie den Realisierungsgraphen für lucid1024.

C 1.2. Sehen Sie sich die Folge mit dem Tool "Visual Test" an. Sehen Sie hier eine Regelmäßigkeit? Lassen Sie sich die Häufigkeitsanalyse in Histogrammform anzeigen. Die einzelnen Histogramme zeigen die Häufigkeiten der Bitmuster bestimmter Länge.

C 1.3. Führen Sie die drei statistischen Tests "Markov Test" (Übergangswahrscheinlichkeiten), "Run Test" (Verteilung von reinen Nullfolgen oder reinen Einserfolgen) und "Generalized Serial Test" (liefert für diese kurze Folge kein Ergebnis) aus. Die Zufälligkeit der Folge wird hier mit dem statistischen Chi-Quadrat-Test bewertet. Stellen Sie vorher den Schalter "verbose" in "Prefs/Lisp Settings" wieder auf t (ankreuzen), um ein ausführlicheres Protokoll zu erhalten.

C 1.4. Untersuchen Sie die Autokorrelationseigenschaften der Folge. Schließen Sie auf die Pseudozufälligkeit.

C 1.5. Untersuchen Sie die lineare Komplexität der Folge mit der Analyse-Methode "Massey-Berlekamp lin compl". Diese Methode berechnet das kleinste, linear rückgekoppelte

Schieberegister, welches genau die vorgegebene Folge als Periode produziert. Was wissen Sie nun über den Zufallszahlengenerator, der die Folge "lucid1024" erzeugt hat? Ist dieser Generator für kryptologische Zwecke brauchbar?

C 1.6. Ändern Sie das 600ste Bit der Folge. Sie können das mit dem Inspektor machen (sehr langsam), oder in der Textfile (mit einem guten Editor) und die Folge dann erneut laden, oder direkt im Lisp Listener durch Aufruf von:
`(setq lucid1024 (first (selected-systems)))` (bei selektiertem Icon der Folge),
dann:
`(setf (aref (sequence lucid1024) 599) (- 1 (aref (sequence lucid1024) 599)))`
d.h. das 600ste Bit wird gesetzt auf Eins minus dem alten Wert des 600sten Folgen-Bits.
Rufen Sie wieder die Analyse der linearen Komplexität auf.

A.4.2.2. Synthese eines LFSR mit vorgegebener Periode

C 2.1. Importieren Sie die Folge lucid1024 erneut (zerstören Sie das alte Objekt vorher) und rufen Sie im Realisierungsgraphen die Syntheseoperation "Massey-Berlekamp SR" auf, um das kürzeste Schieberegister zu erhalten, welches diese Folge erzeugt.
Erzeugen Sie ein I/O-Experiment der Länge 200 und führen Sie es aus. Die im Inspector ansehbare Ausgabefolge ist in Listenschreibweise dargestellt. Mit der I/O-Experiment-Operation "Output Projection" kann ein normales Zahlenfolgenobjekt gewonnen werden. Mit der Lisp-Funktion "sequence-projection" können Sie aus dieser wieder eine Bitfolge machen und weiter untersuchen.

A.4.2.3. Analyse weiterer Zufallsgenaratoren verschiedener Bauart

C 3.1. Erzeugen Sie eine neue Pseudozufallsfolge mit den folgenden CLI Aufrufen, die den in macCASTfsm eingebauten Zufallszahlengenerator verwenden, und untersuchen Sie diese; vergleichen Sie dann mit den Ergebnissen von lucid1024:
(`(random n)` liefert eine Pseudozufallszahl zwischen 0 und n-1)

```
(setq c3 (make-instance 'binary-sequence
        :name "C3-1024"
        :sequence (make-array 1024 :element-type 'bit)))
(dotimes (i 1024)
        (setf (aref (sequence c3) i) (random 2)))
(make-cast-object-icon c3)
```

C 3.2. Erzeugen Sie eine neue Pseudozufallsfolge mit dem folgenden CLI Aufruf, der einen auf zellularen Automaten beruhenden Zufallszahlengenerator verwendet:
(der Preferences/Lisp Settings Schalter "verbose" muß angekreuzt sein, damit die Zustandstrajektorie sichtbar wird; vergrößern Sie auch das Protokollfenster)
`(ca-gen-binary-seq 32 (expt 2 26) 51 #'rule-30)`
Verwenden Sie wieder `(? ca-gen-binary-seq)` für Info über diese Funktion.

C 3.3. Erzeugen Sie eine neue Pseudozufallsfolge mit den folgenden CLI Aufrufen, die einen auf zellularen Automaten beruhenden Zufallszahlengenerator verwenden, und untersuchen Sie diese; vergleichen Sie dann mit den bisherigen Ergebnissen:
(der Preferences/Lisp Settings Schalter "verbose" sollte hier *nicht* angekreuzt sein, um zuviel Ausgabe im Protokollfenster zu vermeiden.)

```
(setq ca (ca-gen-binary-sequence 1024 333 31))
(setf (name ca) "ca1024")
(make-cast-object-icon ca)
```

A.4.2.4. Synthese eines maximalperiodischen Schieberegisters

C 4.1. Erzeugen Sie ein nichtlineares maximalperiodisches Schieberegister der Länge 8:
```
(setq kst8 (first (kst-create-maxper-shiftregs 8)))
(make-cast-object-icon kst8)
```
Erzeugen Sie eine neue Pseudozufallsfolge als Ausgabe eines I/O-Experiments der Länge 1024 des Schieberegisters kst8 und anschließender Verwendung des Befehls "Output Projection". Wenn Sie einen Grauwert- oder Farbbildschirm zur Verfügung haben, können Sie sich die Folge mit dem Tool "Visual Test" ansehen. Was fällt sofort auf?

C 4.2. Bilden Sie die Projektion der eben errechneten Folge "KSTsr8-io-exp-out" auf ihr erstes Bit mit der Funktion "sequence-projection" und untersuchen Sie diese.

Anleitung: Selektieren Sie das Icon der Folge, evaluieren Sie dann:
```
(setq KSTsr8-out (first (selected-systems)))
(setq kst8-1024 (sequence-projection KSTsr8-out 0))
(make-cast-object-icon kst8-1024)
```
Vergleichen Sie dann mit den bisherigen Ergebnissen. Wie groß ist die lineare Komplexität hier? Was schließen Sie daraus für die Konstruktion kryptologisch "guter" Zufallsgeneratoren für Verschlüsselungsgeräte? (Nur 8 Speicherzellen verwendet!)

A.4.2.5. Walsh- und Fourieranalyse von Binärfolgen

C 5.1. Öffnen Sie den Realisierungsgraphen für eine Binärfolge der Länge 1024 (z.B. ca1024 aus C 3.3. oder eine andere der bisher behandelten Folgen). Bilden Sie die Walsh-Transformierte der Folge. Bilden Sie die Rücktransformation durch nochmalige Anwendung der Walsh-Transformation. Vergleichen Sie die ursprüngliche Folge und die Rücktransformierte mit dem Tool "Visual Test".
Wenn Sie einen Farb- oder Grauwert-Bildschirm zur Verfügung haben, können Sie auch die Walsh-Transformierte ansehen. Entdecken Sie Ähnlichkeiten?

C 5.2. Erzeugen Sie eine Folge, die offensichtliche Regelmäßigkeiten enthält, z.B..
```
(setq cr (make-instance 'binary-sequence
      :name "Cr-1024"
      :sequence (make-array 1024 :element-type 'bit)))
(dotimes (i 1024)
      (setf (aref (sequence cr) i)
         (if (= 3 (mod i 7)) 1
            (if (= 4 (mod i 7)) 0
               (if (= 6 (mod i 7)) 1
                  (if (= 3 (mod i 11)) 1
                     (if (= 5 (mod i 11)) 1
                        (if (= 7 (mod i 11)) 1
                           (if (= 9 (mod i 11)) 1
                              (if (= 8 (mod i 11)) 0
                                 (random 2)))))))))))
(make-cast-object-icon cr)
```
Bilden Sie die Fourier-Transformierte und sehen Sie sich diese mit "Visual Test" an.

C 5.3. Laden Sie die Folge lucid1024 erneut und führen Sie eine Regelmäßigkeit ein:
```
(dotimes (i 1024)
(if (= 3 (mod i 7)) (setf (aref (sequence (first (selected-systems))) i) 1 )))
```
Wie ändert sich die lineare Komplexität? Wie das Fourierspektrum?
Untersuchen Sie auch die Ergebnisse der anderen Analysewerkzeuge wie in 1.

B. Literaturangaben

Abel85	Harald Abelson, Gerald Jay Sussman, with Julie Sussman: Structure and Interpretation of Computer Programs MIT-Press, Cambridge, MA USA, 1985
ARM-90	Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual AT&T Bell Laboratories, Addison-Wesley, Reading MA, 1990
Ashb56	W. Ross Ashby: An Introduction to Cybernetics Chapman & Hall, London, 1956
Auer92	Franz Auernig, Rudolf Mittelmann: FSM-Extension - An Implementation of System Transformations for CAST-FSM* Technical Report TR-92-2, Institut für Systemtheorie und Informationstechnik, Universität Linz 1992
Barr81	A. Barr, E. Feigenbaum, P. R. Cohen: The Handbook of Artificial Intelligence Vol. 1 - 3 William Kaufmann Inc., Los Altos, California, USA, 1981-1982
Barr94	Anthony Barrett, Keith Golden, et al.: UCPOP User's Manual Technical Report 93-09-06, Feb 01, 1994, Dep. of Computer Science and Engineering, University of Washington, Seattle, WA
Bert68	Ludwig v. Bertalanffy: General System Theory Braziller, New York, 1968
Boot67	Taylor L. Booth: Sequential Machines and Automata Theory John Wiley and Sons, New York, fst corr. printing, 1968
Born86	Rainer Born: Angewandte Wissenschaftstheorie: Philosophie der Mathematik - Gödel und die Folgen Seminar an der Universität Linz, Sommer 1986
Boss69	Georg Bosse: Grundlagen der Elektrotechnik III - Wechselstromlehre, Vierpol- und Leitungstheorie BI Hochschultaschenbücher Band 184, 1969
Brau74	G. Braunss, H.-J. Zubrod: Einführung in die Boolschen Algebren Akademische Verlagsgesellschaft, Frankfurt am Main, 1974
Burg91	Wilhelm Burger: A Versatile Development Tool for Knowledge-Based Computer Vision Cybernetics and Systems, vol. 22 (3), pp. 313-329 (June 1991)

Cara93	Gregory Carayannis: Artificial Intelligence and Expert Systems in the Steel Industry JOM (Publication of the Minerals and Metals Society), October 1993, pp 43-51
Cava79	Roger E. Cavallo: The Role of Systems Methodology in Social Science Research Boston/The Hague/London, 1979
Char85	Eugene Charniak, Drew McDermott: Introduction to Artificial Intelligence Addison-Wesley Publishing Company, Reading MA, 1985
Clau71	V. Claus: Stochastische Automaten Teubner Verlag Stuttgart, 1971
Coyn90	R. D. Coyne, M. A. Rosenman, et al.: Knowledge-Based Design Systems Addison-Wesley Publishing Co. , Reading MA, 1990
Doyl83	J. Doyle: Admissible State Semantics for Representational Systems IEEE Computer, Oct 1983 (Vol. 16), p.119-123
Dwin71	Ph. Dwinger: Introduction to Boolean Algebras Physica-Verlag Würzburg, 1971
Elco83	E. W. Elcock: How Complete are Knowledge Representation Systems? IEEE Computer, Oct 1983 (Vol. 16), p.114-118
Endl74	Kurt Endl, Wolfgang Luh: Analysis III - Eine integrierte Darstellung Akademische Verlagsgesellschaft, Frankfurt am Main, 1974
Frie76	Dan Friedman, D. Wise: "CONS should not evaluate its arguments" in: Automata Languages and Programming Edinburgh University Press, 1976, pp.257-284
Fu68	K. S. Fu: Sequential Methods in Pattern Recognition and Machine Learning Academic Press, New York, 1968
Gabr89	Richard P. Gabriel: The Common Lisp Object System AI Expert, March 1989, p. 54-65
Gass89	Friedrich Gassner: Entwurf und Implementierung eines komplexen systolischen 16x8 Schaltwerkes mit dem VLSI-Entwurfssystem VENUS Diplomarbeit am Institut für Systemwissenschaften, Uni Linz, 1989
Geig91	Martin Geiger: Finite State Machine Methods Applied to VLSI Synthesis Dissertation am Institut für Systemwissenschaften, Universität Linz, 1991
Geig91	Martin Geiger, Thomas Müller-Wipperfürth: FSM Decomposition Revisited: Algebraic Structure Theory Applied to MCNC Benchmark

	FSMs Proc. of the 28th Design Automation Conference, 1991
Geva82	W. B. Gevarter: An Overview of Expert Systems US Department of Commerce, National Bureau of Standards, May 1982
Giar94	Joseph Giarratano, Gary Riley: Expert Systems: Principles and Programming PWS Publishing Co., Boston, 2nd ed., 1994
Gier80	Gerhard Gierz, K. H. Hofmann, Klaus Keimel et al.: A Compendium of Continuous Lattices Springer Verlag, Berlin, 1980
Gold83	Adele Goldberg, David Robson: Smalltalk-80: The Language and its Implementation Addison-Wesley 1983, Reading, MA(USA)
Hart66	J. Hartmanis, R. E. Stearns: Algebraic Structure Theory of Sequential Machines Prentice-Hall, N.J. USA, 1966
Hart75	William E. Hartnett, (ed.): Systems: Approaches, Theories, Applications D. Reidel Publ., Dordrecht-Holland, 1977
Henn61	Frederick C. HennieIII: Iterative Arrays of Logical Circuits MIT Press & John Wiley, New York, 1961
Herm61	Hans Hermes: Aufzählbarkeit - Entscheidbarkeit - Berechenbarkeit Springer, Heidelberg, 3.Aufl. 1978
Herm63	Hans Hermes: Einführung in die mathematische Logik Teubner, Stuttgart, 3.Aufl. 1972
Hofs85	Douglas R. Hofstadter: Gödel, Escher, Bach: Ein Endloses Geflochtenes Band Klett-Cotta, Stuttgart, 1985 (autorisierte und verbesserte Übersetzung aus dem Englischen)
Hols87	Clyde W. Holsapple, Andrew B. Whinston: Decision Support Systems: Theory and Application Springer Verlag, Berlin, 1987
Hopc79	John E. Hopcraft, Jeffrey D. Ullman: Introduction to Automata Theory, Languages, and Computation Addison-Wesley, Mass. USA, 1979
Kain89	Hermann Kaindl: Problemlösen durch heuristische Suche in der Artificial Intelligence Springer Verlag Wien, 1989 1.Aufl.
Kana88	Laveen Kanal, Vipin Kumar (eds.): Search in Artificial Intelligence Springer New York 1988

Keen89	Sonya E. Keene: Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS Addison-Wesley 1989
Klir69	G. J. Klir: An Approach to General Systems Theory Van Nostrand, New York, 1969
Knut81	D. E. Knuth: The Art of Computer Programming Addison Wesley, Reading, 1981
Koes67	Arthur Koestler: The Ghost in the Machine Random House, New York, 1967
Koes70	Arthur Koestler, J.R.Smythes (ed.): Das Neue Menschenbild (Beyond Reductionism) Molden, Wien, 1970
Koha70	Zvi Kohavi: Switching and Finite Automata Theory McGraw-Hill, New York, 1970
Korf85	Richard E. Korf: Learning to Solve Problems by Searching for Macro-Operators Pitman Publishing, Mass. USA, 1985
Lang83	P. Langley, Alan Mackworth: Representational Issues in Learning Systems IEEE Computer, Oct 1983 (Vol. 16), p.47-52
Lee90	C. C. Lee: Fuzzy Logic in Control Systems: Fuzzy Logic Controller - Parts I and II IEEE Trans. on Systems, Man and Cybernetics, Vol. 20, No.2, Mar/Apr1990, pp.404-436
Mach82	Ernst Mach: Die ökonomische Natur der physikalischen Forschung In: Sitzung kais. Akademie der Wissenschaften, Wien, 1882
McCl86	James L. McClelland, David E. Rumelhard: Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Vol. II: Psychological and Biological Models MIT Press, Cambridge MA, 1986
McCl88	James L. McClelland, David E. Rumelhard: Explorations in Parallel Distributed Processing MIT Press, Cambridge MA, 1988
McCo79	Pamela McCorduck: Machines Who Think Freeman, San Francisco, 1979
Mesa70	M.D. Mesarovic, D. Macko, Y. Takahara: Theory of Hierarchical, Multilevel Systems Academic Press, New York, 1970
Meye88	Bertrand Meyer: Object-Oriented Software Construction Prentice Hall, Hertfordshire, 1988

Mich80 Donald Michie: Expert Systems
The Computer Journal, Vol. 23 (1980) No. 4

Mitt85 Rudolf Mittelmann: Automaten in Interlisp-D
interner Bericht, 1985, Systemtheorie, Universität Linz

Mitt86-1 Rudolf Mittelmann: Einfaches Regelsystem in LOOPS / Interlisp-D
interner Bericht, 1986, Systemtheorie, Universität Linz

Mitt86-2 Rudolf Mittelmann: Petrinetze in LOOPS / Interlisp-D
interner Bericht, 1986, Systemtheorie, Universität Linz

Mitt87-1 Rudolf Mittelmann: Knowledge-Based Systems
interner Bericht, 1987, Systemtheorie, Universität Linz

Mitt87-3 Rudolf Mittelmann: STIPS, CAST, Knowledge Based Systems
interner Bericht, Juni 1987, Systemtheorie Uni Linz

Mitt88-1 Rudolf Mittelmann: Object Oriented Implementation of Petri Nets Concepts
Cybernetics and Systems Research Conference '88 (ed. R.Trappl)
Kluver Academic Publishing 1988

Mitt88-2 Rudolf Mittelmann: TML in PROLOG - Einfache Implementierung des Test-Muster-Lift Algorithmus zur Berechnung von Testmusterfolgen für Automatennetzwerke in PROLOG
interner Bericht, 1988, Systemwissenschaften, Universität Linz

Mitt89-1 Rudolf Mittelmann: CAST:FSM user's guide
technical report 89-3, July 1989, Systemtheorie Uni Linz

Mitt89-2 Rudolf Mittelmann: Einige Gedanken über einen Vortrag von J. Moses: "Cultural Biases in Computer Sciences and AI"
internes Memo 89-1, Jänner 1989, Systemtheorie Uni Linz

Mitt89-3 Rudolf Mittelmann: Graphical User Interfaces - Gedanken über graphische Benutzeroberflächen für CAST-Systeme
internes Memo 89-3, September 1989, Systemtheorie Uni Linz

Mitt89-4 Rudolf Mittelmann: Object Oriented Design of CAST Systems
Lecture Notes in Computer Science, Springer Verlag, Berlin 1989

Mitt89-5 Rudolf Mittelmann, Herbert Prähofer: Design of an Object Oriented Kernel System for Computer Aided Systems Theory and Systems Theory Instrumented Modelling and Simulation
Lecture Notes in Computer Science, Springer Verlag, Berlin 1989

Mitt89-6 Rudolf Mittelmann: Recursive Entity Structure - Erweiterung der SES Notation für STIPS/STIX/STX Implementierungen
internes Memo 89-4, Oktober 1989, Systemtheorie Uni Linz

Mitt90-1 Rudolf Mittelmann: Die Größe des Verbandes von Automaten - Berechnung der maximalen Anzahl von Kongruenzrelationen
internes Memo 90-2, Februar 1990, Systemtheorie Uni Linz

Mitt90-2 Rudolf Mittelmann: Die Verbände von Zählerautomaten - Direkte Bestimmung von Automatenverbänden für eine bestimmte Klasse von Automaten
internes Memo 90-5, Februar 1990, Systemtheorie Uni Linz

Mitt90-4 Rudolf Mittelmann: Tools for Advanced CAST Environments
Proceedings of the Temesvar Workshop "Modern Trends in Cybernetics and Systems", 6-1990, CSFR

Mitt90-3 Rudolf Mittelmann: Tools for Systems Problem Solving in CAST Implementations
Proceedings of the Tenth EMCSR 1990, Vienna

Mitt92-1 Rudolf Mittelmann: Implementation of Mathematical Objects and Systems in CAST Environments
internal report, Systemwissenschaften, Uni Linz, 1992

Mitt92-3 Rudolf Mittelmann: Systems Theory on the Apple Macintosh - macCASTfsm User's Guide
Technical Report, Institut für Systemwissenschaften, v1.2 1992

Mitt93 Rudolf Mittelmann: CAST-FSM
technical report, Systemwissenschaften, Uni Linz, 1993

Mitt94-1 Rudolf Mittelmann: Navigation in Mathematical Systems Problem Solving
Cybernetics and Systems Research Conference '94 (ed. R.Trappl)
Kluver Academic Publishing 1994

Mitt94-2 Rudolf Mittelmann: CAST-CRYPTO User's Guide
Technical Report, Institut für Systemwissenschaften, Universität Linz, 1994

Mösc71 Albrecht Möschwitzer: Halbleiterelektronik
VEB Verlag Technik, Berlin, 1971

Müll86 Thomas Müller-Wipperfürth: LISAS - Der Linzer Systolic Array Simulator
Diplomarbeit, Institut für Systemwissenschaften, Uni Linz, 1986

Müll90 Thomas Müller-Wipperfürth: On Knowledge Representation for Inferences in CAST method banks
Cybernetics and Systems, 21(2-3):231-240, 3-4 1990

Müll91-1 Thomas Müller-Wipperfürth: An Efficient Implementation of the FSM Lattice Computation and Decomposition using Common Lisp / Flavors / C
Technical Report, Intitut für Systemwissenschaften, Universität Linz, 1991

Müll91-2 Thomas Müller-Wipperfürth, Martin Geiger: Algebraic Decomposition of MCNC Benchmark FSMs for Logic Synthesis
Proc. of the EUROASIC '91, Paris, 1991

Müll93-1	Thomas Müller-Wipperfürth, Josef Scharinger, Franz Pichler: FSM Shift Register Realization for Improved Testability EUROCAST '93
Müll93-2	Thomas Müller-Wipperfürth, Josef Scharinger, et al.: FSM State Assignment for Improved Testability - CAST-FSM Finite State Machine Optimizing Tool 1993 International Workshop on Logic Synthesis,Granlibakken Resort, Lake Tahoe, CA, USA
Müll94	Thomas Müller-Wipperfürth: Finite State Machine Structuring with CAST.FSM* applied to VLSI Controller Design Dissertation am Institut für Systemwissenschaften, Universität Linz, 1994
Mura89	Johann Murauer: Die Interlisp-D/LOOPS-Implementierung der System Entity Structure interner Bericht, Institut für Systemwissenschaften, Uni Linz, 1989
Mylo83	J. Mylopoulos, T. Shibahara, J. Tsotsos: Building Knowledge-Based Systems: The PSN Experience IEEE Computer, Oct 1983 (Vol. 16), p.83-89
Nau83	Dana S. Nau: Expert Computer Systems IEEE Computer, Feb 1983, p. 63-85
Newe72	Alan Newell, Herbert A. Simon: Human Problem Solving Prentice-Hall, N.J. USA, 1972
Nils71	Nils J. Nilsson: Problem Solving Methods in Artificial Intelligence Palo Alto 1971
Nils80	Nils J. Nilsson: Principles of Artificial Intelligence Tioga Pub.Co., Palo Alto 1980
Nye88	Adrian Nye: Xlib Programming Manual - Vol.1 of The Definite Guides to the X Window System O'Reilly&Associates, Inc. 1988
Nye-ed88	ed. Adrian Nye: Xlib Reference Manual - Vol.2 of The Definite Guides to the X Window System O'Reilly&Associates, Inc. 1988
ORei88	Tim O'Reilly, Valerie Quercia, Linda Lamb: X Window System User's Guide - Vol.3 of The Definite Guides to the X Window System O'Reilly&Associates, Inc. 1988
Otte78	Wilhelm Ottendörfer: Zerlegungen und algebraische Eigenschaften Endlicher Automaten auf der Basis verbandstheoretischer Konzepte Diplomarbeit am Institut für Systemwissenschaften der Universität Linz, 1978
Pear84	Judea Pearl: Heuristics: Intelligent Search Strategies for Computer Problem Solving Addison-Wesley, Reading Mass. USA, 1984

Penb92 — J. S. Penberthy, D. Weld: UCPOP: A sound, complete, partial order planner for ADL
In Proc. 3rd Conf. on Principles of Knowledge Representation and Reasoning, p. 103-114, Oct 1992

Pich75 — Franz Pichler: Mathematische Systemtheorie - Dynamische Konstruktionen
Walter deGruyter, Berlin 1975

Pich81 — Franz Pichler: Dekomposition komplexer Systeme aus der Sicht der Allgemeinen Systemlehre
Arbeitstagung "Probleme und Ansätze umfassender Modellierung regionaler Systeme", ZiF, Bielefeld 26.-29.3.1981,(eds. Albertin, Müller) Verlag TÜV Rheinland

Pich82-2 — Franz Pichler: Algorithmic Manipulation of Systems Models
Simulation and Model Based Methodologies: An Integrative View, July 26 - August 6, 1982, Ottawa, Ontario, Canada (eds. Ören,Zeigler,Elzas) Springer 1984 pp. 217-234

Pich82-1 — Franz Pichler: Stratifizierte Konzepte für Dynamische Systeme
International Conference on Systems Methodology, Washington D.C. USA, January 5-9 1982

Pich82-3 — Franz Pichler: Studie zur Systemtheorie der Kryptologie, Teil 11: Nichtlineare Schieberegister mit Maximalperiode
interner Bericht, Systemwissenschaften, Universität Linz

Pich82-4 — Franz Pichler: Symbolic Manipulation of System Models (Draft version)
Simulation and Model Based Methodologies: An Integrative View, July 26 - August 6, 1982, Ottawa, Ontario, Canada

Pich83-1 — Franz Pichler: Dynamical Systems Theory
Cybernetics: Theory and Applications (ed. R. Trappl) Springer 1983

Pich84-1 — Franz Pichler: General Systems Algorithms for Mathematical Systems Theory
Cybernetics and Systems Research 2, R.Trappl(ed.) , Proceedings of EMCSR 84, Vienna, Austria, North Holland, Amsterdam 1984, pp. 161-164

Pich85-1 — Franz Pichler: Trends in der Allgemeinen Systemtheorie
2nd International Symposium on Systems Analysis and Simulation, Berlin, German Democratic Republic, August 26-31, 1985

Pich86-1 — Franz Pichler: Mathematische Systemtheorie und Design Automation
internes Memo Jänner 1986

Pich86-2 — Franz Pichler: Model Components for Symbolic Processing by Knowledge Based Systems: The STIPS Framework
Modelling and Simulation Methodology in the Artificial Intelligence Era (eds. Elzas, Ören, Zeigler) North Holland, Amsterdam 1986

Pich90-3	Franz Pichler: CAST: Computer Aided Systems Theory. Perspectives for Research, Development and Education Syst. Anal. Model. Simul. 7 (1990) 4, p.251-266 Akademie-Verlag Berlin
Pich90-2	(ed.) Franz Pichler, Roberto Moreno-Diaz : Computer Aided Systems Theory - EUROCAST '89 Lecture Notes in Computer Science 410, Berlin, Springer Verlag 1990
Pich90-1	Franz Pichler, Heinz G. Schwärtzel: CAST: Computerunterstützte Systemtheorie Springer Verlag Berlin 1990
Pich92	Franz Pichler, Heinz Schwärtzel, (eds.): CAST: Methods in Modelling - Computer Aided Systems Theory for the Design of Intelligent Machines Springer Verlag, Berlin Heidelberg, 1992
Präh86	Herbert Prähofer: LOOPS-Implementierung Automaten-theoretischer Methoden Diplomarbeit am Institut für Systemwissenschaften der Universität Linz, 1986
Präh88	Herbert Prähofer, Alfred Spalt: An Interactive Simulation Environment Using Systems Theory Concepts and Object Oriented Programming Proc. of the 1988 ESM, Nice, France
Präh89-1	Herbert Prähofer: A Kernel System for Modelling and Simulation Using System Entity Structure Concepts for Systems Theory Knowledge Representation SCS Eastern Multiconference, Tampa, Florida, March 1989
Präh89-2	Herbert Prähofer, Bernard P. Zeigler: Modelling and Simulation of Non-Homogeneous Models - a System Theoretical Approach to Mixed-Mode Simulation Lecture Notes in Computer Science, Springer Verlag, Berlin 1989
Präh90-1	Herbert Prähofer, Rudolf Mittelmann: The Relevance of Browsers for CAST Implementations Proceedings of the Tenth EMCSR 1990, Vienna
Prat76	Ronald E. Prather: Discrete Mathematical Structures for Computer Science Houghton Mifflin Company, Boston, 1976
Pühr88	O. M. Pühringer, F. Wallner: Computer-Aided Quality Control System of Voest Alpine Metallurgica International, 1 (3), August 1988, pp. 192-197
Reis86	Wolfgang Reisig: Petrinetze - Eine Einführung Berlin, Springer Verlag 1986
Rich83	Elaine Rich: Artificial Intelligence McGraw-Hill New York 1983

Roze89 — Jerzy W. Rozenblit, Herbert Prähofer: Computer Aided Systems Theory and Knowledge Based System Design and Simulation; Directions to Explore
Lecture Notes in Computer Science, Springer Verlag, Berlin 1989

Rumb91 — J. Rumbaugh, M. Blaha, et al.: Object-Oriented Modeling and Design
Prentice-Hall, Englewood Cliffs, 1991

Rume86 — David E. Rumelhard, James L. McClelland: Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Vol. I: Foundations
MIT Press, Cambridge MA, 1986

Rych81 — Michael D. Rychener: Knowledge Based Expert Systems - A Brief Bibliography
Carnegie Mellon University, 26. Juni 1981

Scha92 — Josef Scharinger: Implementierung der Schieberegisterrealisierung für CAST-Systeme
Diplomarbeit am Institut für Systemwissenschaften der Uni Linz, 1992

Schi — Schieferle: Erfahrungen beim Einsatz von Expertensystemen ...
Informatik-Fachberichte 112, p.333 ff

Schö91 — Robert Schöch: Realisierung eines Blockdiagramm-Browsers für modulare, hierarchische Modellbildung
Diplomarbeit am Institut für Systemwissenschaften, Uni Linz, 1991

Schw81 — Heinz G. Schwärtzel: Ein konzeptioneller Ansatz zur Verwendung Dynamischer Konstruktionen als Instrumente für die Modellierung von Systementwurfsprozessen
Dissertation am Institut für Systemwissenschaften der Universität Linz 1981

Siek81 — Siekmann , (ed.): Proceedings of GWAI- 81
German Workshop on Artificial Intelligence, Bad Honnef, Jan 1981, Informatik-Fachberichte 47

Spal89-1 — Alfred Spalt, Siegfried Grabner: Portierung der Berechnung der Verlustlosigkeit endlicher Automaten von Interlisp-D auf WS Siemens 5815 nach CommonLisp auf Apollo DN 3500
internes Memo 89-2, April 1989, Systemtheorie Uni Linz

Spal89-2 — Alfred Spalt: Implementation of the system type Finite Memory Machine within the scope of CAST.FSM
technical report 89-2, April 1989, Systemtheorie Uni Linz

Spal90-1 — Alfred Spalt: Walsh-Fourieranalyse und Korrelationsimmunität von Schaltfunktionen und Schaltwerken
internes Memo 90-3, Februar 1990, Systemtheorie Uni Linz

Stee90 — Guy L. Steele: Common Lisp: The Language - Second Edition
Digital Press, Bedford MA, 1990

Stee92	Sam Steel: Notes on current trends in AI Planning in: Advanced Topics in Artificial Intelligence, Lecture Notes in Artificial Intelligence 617, Springer Berlin, 1992
Stef86	Mark Stefik, Daniel G. Bobrow: Object-Oriented Programming: Themes and Variations The AI Magazine, Vol.6, Number 4, Winter 1986, pages 40-62
Stoh91	T. H. Stohl , et al.: Development of a Scheduling Expert System for a Steel Plant Expert Systems in Mineral and Metal Processing, Pergamon Press, Oxford, 1991, pp. 39-44
Stoy88	Herbert Stoyan: Programmiermethoden der Künstlichen Intelligenz, Band 1 Springer, Heidelberg, 1988
Sydo80	Achim Sydow: Anwendungsaspekte der Systemanalyse Akademie-Verlag, Berlin, 1980
Sydo81	Achim Sydow: Modellbildung und Simulationstechniken Akademie-Verlag, Berlin, 1981
Sydo86	Achim Sydow: Analyse komplexer dynamischer Systeme Vorlesung an der Johannes Kepler Universität Linz, Sommer 1986
Teit83	Warren Teitelman, et al.: Interlisp Reference Manual Xerox PARC October 1983 Xerox Corp.
Tour86	David S. Touretzky: The Mathematics of Inheritance Systems Pitman, London, 1986
Trel82	P. C. Treleaven, I. G. Lima: Japan's Fifth Generation Computer Systems IEEE Computer, August 1982, p.79-88
Wate78	D. A. Waterman, F. Hayes-Roth: Pattern Directed Inference Systems Academic Press, New York, 1978
Wede74	Hartmut Wedekind: Datenbanksysteme I BI Wissenschaftsverlag Reihe Informatik/16, 1974
Wede76	Hartmut Wedekind, Theo Härder: Datenbanksysteme II BI Wissenschaftsverlag Reihe Informatik/18, 1976
Wien48	Norbert Wiener: Cybernetics Wiley, New York, 13.Aufl.1957
Wirt92	Nikolaus Wirth, J. Gutknecht: Project Oberon - The Design of an Operating System and Compiler Addison-Wesley, Reading, 1992
Wirt94	Nikolaus Wirth: Gedanken zur Software-Explosion Informatik Spektrum, 1994, 17, p. 5-10

Wins89	Patrick Henry Winston, Berthold Klaus Paul Horn: Lisp - third edition Addison-Wesley, Reading, MA USA, 1989
Wymo67	A. Wayne Wymore: A Mathematical Theory of Systems Engineering: The Elements Wiley, New York, 1967
Wymo76	A. Wayne Wymore: Systems Engineering Methodology for Interdisciplinary Teams Wiley, New York, 1976
Zaji88	Walter Zajicek: Ein Programmsystem und Formalismen für das wissensbasierte Programmieren Dissertation am Institut für Systemwissenschaften, Universität Linz, 1988
Zeig84	Bernard P. Zeigler: Multifacetted Modelling and Discrete Event Simulation Academic Press, London, 1984
Zeig86	Bernard P. Zeigler: Knowledge Representation from Newton to Minsky and beyond Proceedings of the Eighth European Meeting on Cybernetics and Systems Research - EMCSR 1986, Vienna
Zeig89	Bernard P. Zeigler, Herbert Prähofer: Systems Theory Challenges in the Simulation of Variable Structure and Intelligent Systems Lecture Notes in Computer Science, Springer Verlag, Berlin 1989
Zein87	Annemarie Zeinhofer: Ein Ansatz zur Testmustererstellung für sequentielle Schaltungen Diplomarbeit am Institut für Systemwissenschaften, Universität Linz, 1987

Lebenslauf

Rudolf Mittelmann

19. November 1953	Geburt in Frankfurt am Main Eltern: Gertrud Mittelmann, geb. Herrenbrück, Lehrerin Eduard R. Mittelmann, kaufm. Direktor
April 1960 - April 1964	Volksschule in Frankfurt
April 1964 - Juni 1972	Gymnasium in Frankfurt, Abschluß: Abitur
Juli 1972 - September 1973	Grundwehrdienst
Oktober 1973 - Februar 1981	Studium der Mathematik an der Technischen Hochschule Darmstadt Abschluß: Dipl-Ing. der Mathematik mit Schwerpunkt Informatik
März 1981 - Dezember 1982	Entwicklungsingenieur bei der Siemens AG in München
seit Jänner 1981	Assistent am Institut für Systemwissenschaften der Johannes Kepler Universität Linz
seit 25. 9. 1982	verheiratet mit Dipl.-Ing. Angelika Mittelmann
seit 12. 1. 1983	Vater von Thomas Mittelmann
seit Juni 1988	österreichische Staatsbürgerschaft