Anwendung von Methoden der KI bei der simulativen und analytischen Untersuchung speziell strukturierter Systeme

Diplomarbeit

Technische Universität Chemnitz-Zwickau
Fachbereich Informatik

eingereicht von Sven Hader
geb. am 25. Februar 1969 in Mühlhausen

Betreuer: Prof. Dr. rer. nat. habil. P. Köchel

Chemnitz, den 20. April 1993

Bibliographische Beschreibung

Anwendung von Methoden der KI bei der simulativen und analytischen Untersuchung speziell strukturierter Systeme /
Hader, Sven. - 1993. - 130 Seiten. (davon 43 Seiten Quelltext) , 6 Abbildungen ,
59 Literaturangaben,
Chemnitz, Technische Universität, Fachbereich Informatik,
Diplomarbeit.

Kurzreferat

Die Aufgabe dieser Arbeit besteht in der Suche nach geeigneten wissensbasierten Methoden zur Automatisierung der Entwurfsoptimierung parametrisierter Modelle. Dabei wird ein Ansatz gewählt, der von einer 'Simulation' des Verhaltens des Entwurfsingenieurs ausgeht. Es wird die Vermutung aufgestellt, daß sich das entsprechende Wissen in Form von WENN-DANN-Regeln darstellen läßt. Für die Formulierung von Dimensionierungsaufgaben und die Darstellung von Regeln werden spezielle Sprachen entwickelt. Es wird die Entwicklung eines Programmsystems in PROLOG beschrieben, mit dem es möglich ist, Dimensionierungsaufgaben regelbasiert zu lösen. Das Kernstück dieses Programmes ist ein vorwärtsverkettender Regelinterpreter. Das Programm wird anhand mehrerer Beispiele auf seine Anwendbarkeit getestet.

Gliederung

1. Einleitung 1

2. Der Entwurf technischer Systeme 3

2.1 Der Entwurfsprozeß 3

2.2 Klassen von Entwurfsaufgaben 6

2.3 Dimensionierung und mathematische Optimierung 7

3. Wissensbasierte Systeme 9

3.1 Die Geschichte der Künstlichen Intelligenz 9

3.2 Die Architektur wissensbasierter Systeme 11

3.3 Wissensrepräsentation und Wissensverarbeitung 13

3.3.1 Regeln 14

3.3.2 Frames 17

3.3.3 Semantische Netze 19

3.4 Anwendungsgebiete wissensbasierter Systeme 20

3.5 Die Verbindung von numerischer und wissensbasierter Datenverarbeitung 21

4. Die automatische Lösung von Dimensionierungsaufgaben 24

4.1 Beschreibung von Dimensionierungsmodellen 24

4.1.1 Die Formulierung der Dimensionierungsaufgabe 24

4.1.1.1 Notwendige Sprachkonstrukte 26

4.1.1.2 Realisierung in DABS 30

4.1.2 Die Modellierung des Systemverhaltens 35

4.2 Dimensionierung 37

4.2.1 Mathematische Dimensionierung 38

4.2.2 Wissensbasierte Dimensionierung 41

4.2.2.1 Regeldarstellung 41

4.2.2.2 Regelverarbeitung 49

5. Das Programmsystem 52

5.1 Die Nutzer-Schnittstelle 52

5.2 Die Beschreibung der wichtigsten Kommandos 53

5.3 Eine typische Sitzung 59

6. Ein Beispiel 61

7. Zusammenfassung und Ausblick 67

Literaturverzeichnis 69

Verzeichnis der Abbildungen 73

Anhang 74

A. Syntax der Dimensionierungsaufgaben-Beschreibungssprache DABS 74

B. Syntax der Regeldarstellungs-Sprache RDS 77

C. Quelltexte des Programmsystems 78

D. Beispiel einer DABS-Datei 120

E. Beispiel einer RDS-Datei 122

Selbständigkeitserklärung 124

Thesen 125

1. Einleitung

In der Informatik deutet sich in den letzten Jahren ein Paradigmenwechsel[1] an. Dazu wird in [Thu89] folgendes bemerkt:

> *Der Fortschritt jeder Wissenschaft erfolgt durch Paradigmenwechsel. Dies sind tiefgreifende Änderungen in ihren Grundkonzepten und Inhalten. Sie werden nötig, weil das 'alte Paradigma' auftretende Schwierigkeiten im Rahmen seiner Methoden und Techniken nicht mehr lösen kann.*

Nachdem sich um 1970 das Paradigma der **strukturierten Programmierung** (vor allem durch das Auftreten der Programmiersprache PASCAL) durchsetzte, bildet sich heute ein neues Paradigma heraus, das der **wissensbasierten Programmierung**.

Das neue Paradigma zeichnet sich vor allem dadurch aus, daß das für die Lösung eines Problems verwendete Wissen nicht mehr implizit (im Algorithmus kodiert), sondern explizit (in Form einer Wissensdarstellung) vorliegt. Dieser auch als deklarative Programmierung bezeichnete Ansatz besitzt gegenüber dem der strukturierten (oder allgemeiner: imperativen) Programmierung eine Reihe von Vorteilen.

Durch die explizite Darstellung des Wissens kann der Abarbeitungsmechanismus leicht an sich verändernde Aufgabenstellungen angepaßt werden. Diese Anpassung kann durch den Menschen (Neueingabe bzw. Modifizierung des Wissens) oder durch den Rechner selbst (Lernen) erfolgen.

Das neue Paradigma kann in vielen Fachgebieten zu Fortschritten führen, sowohl bei solchen, die schon vorher extensiven Gebrauch von der Rechentechnik machten (z.B. Operations Research, Simulation), als auch bei solchen, bei denen ein Rechnereinsatz bisher nur schwer möglich schien (Aufgaben, deren Lösung im weitesten Sinne 'menschliche Intelligenz' erfordert).

Ein Gebiet, das sowohl rechenintensiv ist, als auch ein hohes Maß an menschlicher Intelligenz erfordert, ist die **Entwurfsoptimierung** von (komplexen) technischen Systemen. Die Entwurfsoptimierung erfolgt i.a. durch die iterative Wiederholung zweier Schritte:

1. Analyse des vorliegenden Entwurfes (analytische Verfahren, Simulation, Experiment)

2. Verbesserung des vorliegenden Entwurfes (Fachwissen, Erfahrung, Intuition)

Der erste Schritt umfaßt Methoden zur Analyse technischer Systeme, deren Schwerpunkt auf der numerischen Datenverarbeitung liegt und die deshalb durch strukturierte Programmierung sehr gut gelöst werden können. Hierfür existiert eine große Anzahl ausgereifter und effizienter Programmpakete (zumeist in FORTRAN oder C implementiert).

Der zweite Schritt umfaßt Methoden, die einen eher symbolischen Charakter tragen und die eine gewisse 'Intelligenz' erfordern. Sie werden in der Praxis von hochqualifizierten Fachleuten ausgeführt, die über eine oft langjährige Erfahrung verfügen. Hier liegt ein zukünftiges Anwendungsgebiet der wissensbasierten Programmierung (z.B. in Form von Expertensystemen). Über

[1] Ein **Paradigma** ist die Menge der anerkannten Gegenstände, Theorien, Lehrinhalte und Methoden einer Wissenschaft (Theorie des Paradigmas von Thomas Kuhn).

die Frage, inwieweit der Fachexperte ersetzt werden kann, herrschen jedoch verschiedene Meinungen.

Die vorliegende Arbeit beschäftigt sich mit der Frage, inwieweit sich die Entwurfsoptimierung parametrisierter Modelle durch das Einbringen wissensbasierter Methoden automatisieren läßt. Dabei liegt der Schwerpunkt der Arbeit auf der ausführlichen Beschreibung der Entwicklung eines in gewisser Weise prototypischen Programmsystems zur automatischen Entwurfsoptimierung. Mit diesem Programmsystem soll es möglich sein, das Erfahrungswissen des Entwurfsingenieurs bzgl. der optimalen Gestaltung (Wahl optimaler Parameter) technischer Systeme in Regelform darzustellen, zu testen und anzuwenden.

Das Programmsystem wurde auf einer SUN-Sparc-Workstation in SNI-Prolog v3.01A implementiert und getestet.

2. Der Entwurf technischer Systeme

Der Entwurf[1] technischer Systeme im weitesten Sinne gehört zu den intellektuell anspruchsvollsten Tätigkeiten, zu denen der Mensch fähig ist. Dieser Prozeß wird in [Leh89] folgendermaßen beurteilt:

> *Konstruieren ist weitgehend eine Tätigkeit, die auf Wissen und Erfahrung gegründet ist. Dies gilt insbesondere für die Vorgehensweise in den frühen Konstruktionsphasen, die von den Erfahrungen des Konstrukteurs abhängig und weitgehend heuristischer Art sind, da sie ihren Ursprung in Ideen, Intuitionen und im Erfindungsvermögen haben. Zur Problemlösung und Lösungsoptimierung muß hier neben spezialisiertem Fachwissen hauptsächlich die Erfahrung des Konstrukteurs in der Behandlung von Problemen berücksichtigt werden.*

Trotz der Entwicklung und des kommerziellen Einsatzes von Systemen zum rechnerunterstützten Konstruieren (Computer Aided Design) ist der Konstrukteur auch heute noch im hohen Maße ausschlaggebend für das Ergebnis des Entwurfsprozesses. Die CAD-Systeme beschränken sich zumeist auf die Unterstützung der Zeichnungs- und Stücklistenerstellung und bestimmte Berechnungsmethoden (z.B. Strukturanalyse) und lassen den Konstrukteur bei den Arbeitsschritten, die entscheidend die effektive Funktion des Entwurfsgegenstandes bestimmen, allein.

Um die Teilaufgaben des Entwurfsprozesses, bei denen eine wissensbasierte Unterstützung oder Ersetzung des Konstrukteurs sinnvoll und durchführbar erscheint, zu kennzeichnen, werden wir uns in den folgenden Abschnitten mit den Phasen des Entwurfsprozesses im allgemeinen und der Entwurfsoptimierung im speziellen beschäftigen.

2.1 Der Entwurfsprozeß

Als erstes wollen wir uns verdeutlichen, wie der Entwurfsprozeß in den umfassenderen Prozeß der Auftragsbearbeitung in einem Unternehmen eingeordnet werden kann. Das Bild 2.1 verdeutlicht die typischen Phasen der Auftragsabwicklung vom ersten Kontakt mit dem Kunden bis zur Auslieferung des fertigen Produktes (aus [Hei91], S.8).

Bedarfsanalyse

Diese Phase umfaßt alle Tätigkeiten, die zur Ermittlung des konkreten Kundenwunsches (in informeller Form) führen. Sie werden meist von einem Mitarbeiter des Vertriebes im direkten Kontakt mit dem Kunden ausgeführt. Als Ergebnis 'entsteht' ein Kundenwunsch.

In dieser Phase ist ein Einsatz wissensbasierter Methoden unnötig, da der Kunde i.allg. schon eine mehr oder weniger feste Vorstellung von dem gewünschten Produkt hat.

Auftragserstellung

Diese Phase umfaßt alle Tätigkeiten, bei denen der (informelle) Kundenwunsch in eine (formelle) Beschreibung des gewünschten Produktes umgesetzt wird. Diese Beschreibung

[1] In der Literatur sind die Begriffe **Konstruktion** und **Entwurf** meist nicht ausreichend voneinander abgegrenzt. Deshalb werden sie in dieser Arbeit synonym benutzt.

enthält sowohl die geforderte Funktionalität des Produktes als auch kundenspezifische Restriktionen (z.B. finanzieller Art). Als Ergebnis entsteht ein Auftrag.

In dieser Phase ist der Einsatz wissensbasierter Methoden nur schwer möglich, da das Verstehen und Umsetzen des informellen Kundenwunsches ein hohes Maß an Abstraktionsfähigkeit erfordert.

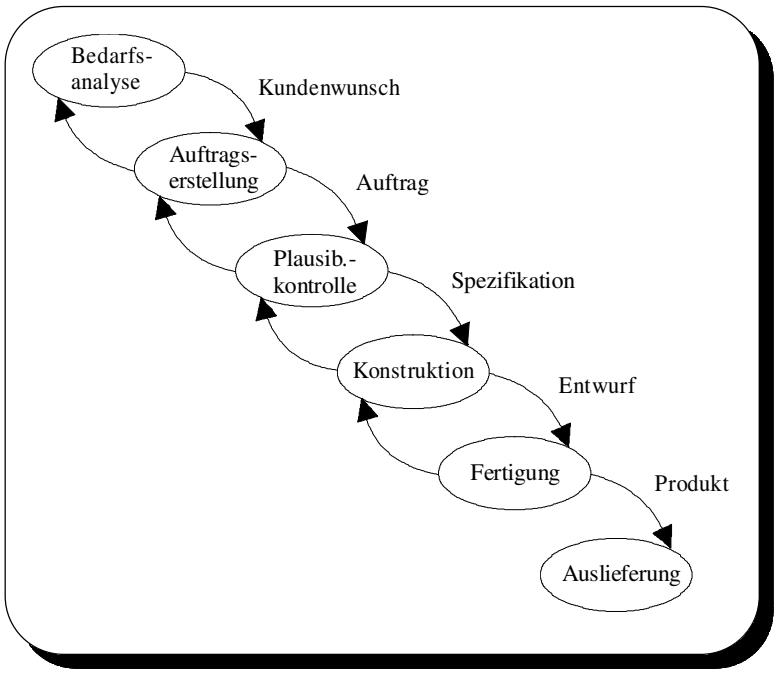

Abbildung 2.1: Phasen der Auftragsabwicklung (nach [Hei91])

Plausibilitätskontrolle

Der entstandene Auftrag wird auf der Ebene der Auftragspositionen auf Vollständigkeit und Korrektheit überprüft. Das kann z.T. noch durch den Vertriebsbeauftragten durchgeführt werden, aber oft ist wegen der schnellen Innovation der Produkte ein Konstruktions-Fachmann notwendig. Dieser übernimmt auch die danach notwendige Prüfung auf technische/finanzielle Realisierbarkeit. Als Ergebnis entsteht eine Spezifikation.

In dieser Phase können wissensbasierte Methoden zur Analyse der Plausibilität verwendet werden. Diese Methoden besitzen Wissen über die einzelnen Auftragspositionen. Sie weisen darauf hin, wenn für den Entwurf notwendige Angaben fehlen oder unrealisierbare Werte besitzen.

Konstruktion

Aus der Spezifikation wird ein Entwurf der technischen Realisierung erzeugt (wir gehen weiter unten genauer auf den eigentlichen Entwurf ein).

Mit dem Nutzen wissensbasierter Verfahren bei der Konstruktion werden wir uns später ausführlich beschäftigen.

Fertigung

Diese Phase umfaßt die Fertigungsplanung[1], die Produktion und die Qualitätskontrolle des Produktes. Als Ergebnis entsteht das fertige Produkt.

Sowohl die Fertigungsplanung als auch die eigentliche Produktion können durch wissensbasierte Methoden verbessert werden (CIM-Technologie).

Auslieferung/Installation

In der letzten Phase wird das Produkt ausgeliefert, montiert, aufgestellt und getestet.

Speziell für das Gebiet der Reparatur und der Wartung technischer Systeme wurden bereits eine Vielzahl von Expertensystemen entwickelt (SOPHIE, STEAMER u.a.).

Zwischen den oben geschilderten Phasen können mehrfache Rückkopplungen (verursacht durch ungenaue Kundenwünsche oder Probleme bei der technischen Realisierung) auftreten. Sie führen meist zur Verzögerung der Auftragsbearbeitung und zu erhöhten Gesamtkosten. Eine Faustregel besagt, daß nach der Plausibilitätskontrolle (Erstellung der Spezifikation) keine Rückkopplungen mehr auftreten sollten.

Im folgenden möchte ich genauer auf die oben kurz angesprochene Phase der Konstruktion eingehen. Der Entwurf technischer Systeme umfaßt im wesentlichen die beiden Schritte:

1. Finden eines Anfangsentwurfes

2. Analyse und, falls nötig, Verbesserung des Entwurfes (Entwurfsoptimierung)

Im ersten Schritt erstellt der Ingenieur einen Anfangsentwurf, der noch nicht allen Restriktionen genügen muß. Dabei stützt er sich oft auf früher entstandene Entwürfe für ähnliche Problemstellungen.

In diesem Schritt können wissensbasierte Entwurfsbibliotheken genutzt werden, die ein dem aktuellen Problem ähnliches Problem suchen, für das bereits ein Entwurf vorliegt. Durch dieses Vorgehen kann ohne große Mühe ein solider Ausgangspunkt für den Entwurfsprozeß geschaffen werden.

Im zweiten Schritt wird der aktuelle Entwurf bzgl. seiner Gültigkeit (Funktionalität + Restriktionen) analysiert und evtl. auftretende Defizite werden erkannt. Davon ausgehend wird der aktuelle Entwurf so geändert, daß die erkannten Defizite (möglichst) nicht mehr auftreten. Dieser Schritt wiederholt sich so lange, bis ein Entwurf gefunden wurde, der vollständig der Spezifikation entspricht.

In diesem Schritt können wissensbasierte Methoden zur Erkennung und Beseitigung von Spezifikationsverletzungen benutzt werden. Dafür ist es notwendig, das Wissen des Entwurfsingenieurs über Ursache-Wirkung-Beziehungen im zu entwickelnden System dem Rechner zugänglich zu machen. Der Entwurfsingenieur bräuchte in diesem Fall nur dann eingreifen, wenn der Rechner einer Entwurfssituation gegenübersteht, die er nicht beherrscht (für die er noch kein

[1] Ken G. Swift vertritt in [Swi90] die Ansicht, daß bei konsequenter Beachtung von Fertigungsproblemen bereits in der Phase der Konstruktion die späteren Fertigungskosten um 30-40% verringert werden könnten.

Wissen besitzt). Dieses Eingreifen könnte gleichzeitig dazu dienen, daß das wissensbasierte System das Vorgehen des Ingenieurs in dieser Situation 'lernt' und später selbst verwendet.

2.2 Klassen von Entwurfsaufgaben

Die in der Praxis vorkommenden Entwurfsaufgaben können aufgrund ihrer wesentlichen Merkmale in mehrere Klassen unterteilt werden. Die wichtigsten davon sind:

- Auswahlaufgaben
- Konfigurierungsaufgaben
- Anordnungsaufgaben
- Parametrisierungsaufgaben

Unter einer **Auswahlaufgabe** versteht man die Auswahl eines Objektes, das für eine bestimmte Aufgabe am besten geeignet ist, aus einer explizit gegebenen Menge von Objekten. Bedingt durch die Endlichkeit der Objektmenge ist auch der Problemraum endlich.

Beispiele für Auswahlaufgaben sind die Wahl eines Rechners bzw. einer Maschine, die für eine bestimmte Aufgabe optimal geeignet sind.

Unter einer **Konfigurierungsaufgabe** versteht man die Aufgabe des Zusammenstellens eines Objektes aus elementaren Objekten ('Atomen') bzw. Strukturen von Objekten ('Molekülen'), um eine bestimmte Funktionalität zu erreichen. Je nach Aufgabenstellung können sehr unterschiedliche Objekte als Elementarobjekte angesehen werden (z.B. Schrauben, Motoren, Maschinen). Durch die Begrenzung der Anzahl möglicher Elementarobjekte bedingt ist der Problemraum endlich, aber im allgemeinen sehr groß[1].

Beispiele für Konfigurierungsaufgaben sind Rechnerkonfigurierung (siehe [Hei91]) und Konfigurierung von Drehmaschinen (siehe [Leh89]).

Unter einer **Anordnungsaufgabe** versteht man die Aufgabe der Gestaltung des externen Aufbaus eines technischen Systems. Dabei geht es meist um die Positionierung und Verbindung einer Menge von Objekten (die bereits vorher zusammengestellt wurde). Durch die zumeist große Anzahl von Anordnungsmöglichkeiten kann der Problemraum sehr groß werden.

Beispiele für Anordnungsaufgaben sind die Anordnung von Rechnerkomponenten in einem Rechnergehäuse und die Aufstellung von Maschinen in einer Fabrikhalle.

Unter einer **Parametrisierungsaufgabe**[2] versteht man die Einstellung der Parameter von Objekten, um bestimmte Zielvorgaben zu erfüllen. Da die Parameter oft über einem unendlichen

[1] Hein erwähnt in [Hei91], daß Problemräume der Größenordnung 10^{15} oder größer in der Praxis keine Seltenheit sind.

[2] Der Begriff **Parametrisierung** bezeichnet i.allg. den Prozeß der Festlegung von Parametern für ein Modell bzw. ein Objekt und nicht den Prozeß der Festlegung konkreter Werte für diese Parameter. Um Mißverständnisse zu vermeiden, werde ich im folgenden den zweiten geschilderten Sachverhalt mit dem Begriff **Dimensionierung** bezeichnen.

Wertebereich definiert werden, kann der Problemraum unendlich sein. Hinzu kommt noch, daß die Werte der Parameter zumeist nicht unabhängig voneinander sind.

Beispiele für Parametrisierungsaufgaben sind der Entwurf von Flugzeugen und die Auslegung der technischen Parameter von Motoren.

In der Fachliteratur wird bemerkt, daß es sich bei Anordnungs- und Parametrisierungsaufgaben um zwei Extreme handelt, zwischen denen jedes beliebige Entwurfsproblem eingeordnet werden kann.

Bei der Erstellung des in dieser Arbeit vorgestellten Programmsystems habe ich mich auf die Behandlung von Parametrisierungsaufgaben (im folgenden **Dimensionierungsaufgaben** genannt) beschränkt. Das hat den Vorteil, daß die prinzipielle Struktur des Entwurfsgegenstandes bereits gegeben ist und die Anpassung an die geforderten Leistungskriterien allein über eine Dimensionierung (d.h. durch die Festlegung von Zahlenwerten) erfolgen kann. Durch die ausschließliche Behandlung von Zahlenwerten anstelle von konkreten Objekten kann ein großes Maß an Abstraktion vom augenblicklichen Entwurfsgegenstand erreicht werden. Dadurch besteht die Möglichkeit, den Entwurfsprozeß als mathematisches Optimierungsproblem aufzufassen und so zu formalisieren.

2.3 Dimensionierung und mathematische Optimierung

Bei der **mathematischen Optimierung** handelt es sich im weitesten Sinne um die Lösung einer Aufgabe mit bestimmten Restriktionen derart, daß eine gegebene Zielfunktion einen bestmöglichen Wert (Maximum oder Minimum) annimmt.

In diesem Abschnitt wollen wir uns mit einer speziellen mathematischen Optimierungsart beschäftigen, die bei Dimensionierungsaufgaben verwendet werden kann, der **statischen Parameter-Optimierung**. Der Begriff 'statisch' deutet darauf hin, daß sich mit dieser Optimierungsmethode nur zeitinvariante Aufgaben lösen lassen, also solche, bei denen die Zielfunktion und die Restriktionen nicht von der Zeit abhängig sind. Unter Parameter-Optimierung sind solche Optimierungsstrategien zusammengefaßt, bei denen für alle Variablen des Optimierungsproblemes nur Zahlen (oder allgemeiner Vektoren von Zahlen) als Werte zugelassen sind[1].

Aufgaben, die mit der statischen Parameter-Optimierung lösbar sind, kann man folgendermaßen darstellen:

$$\underline{\text{minimiere:}} \quad F(X) \quad \text{mit} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ : \\ x_{n-1} \\ x_n \end{bmatrix} \quad ; \quad x_i \in \mathbf{R} \;,\; n \in \mathbf{N}$$

$$\underline{\text{mit den Restriktionen:}} \quad g_j(X) \geq 0 \quad ; \quad j = 1..m \;,\; m \in \mathbf{N}$$

$$(m = 0 \;\rightarrow\; \text{keine Restriktionen})$$

[1] Demgegenüber werden bei der **Funktionen-Optimierung** auch Funktionen als Variablenwerte zugelassen.

Ungleichungs-Restriktionen, die nicht der oben gezeigten Form entsprechen, können ohne Probleme arithmetisch umgeformt werden. Gleichungs-Restriktionen führen dazu, daß jeweils eine der enthaltenen Variablen x_i als Abhängige der anderen enthaltenen Variablen dargestellt werden kann. In diesem Fall kann die Variable x_i in der Zielfunktion F(X) durch einen Ausdruck der anderen Variablen ersetzt werden und die Dimension des Problems nimmt um eins ab.

In Bezug auf Dimensionierungsaufgaben kann man die Zielfunktion als Maß für die Güte des zu entwerfenden Systems und die Restriktionen als Einschränkungen technischer, physikalischer, finanzieller oder ähnlicher Art auffassen.

3. Wissensbasierte Systeme

In diesem Kapitel wollen wir uns mit dem Aufbau und den Möglichkeiten wissensbasierter Systeme beschäftigen. Das ist erforderlich, um deutlich zu machen, welche Mittel uns zur Verfügung stehen, um das Wissen des Entwurfsingenieurs effektiv darstellen und verarbeiten zu können. Dieses Kapitel ist etwas umfangreicher gehalten, um auch Lesern, die keine oder nur geringe Kenntnisse der Künstlichen Intelligenz im allgemeinen und wissensbasierter Systeme im besonderen besitzen, das Wissen zu vermitteln, das für das Verständnis des vorzustellenden Programmsystems notwendig ist.

3.1 Die Geschichte der Künstlichen Intelligenz

Bei der Frage nach den Ursprüngen der **Künstlichen Intelligenz** (artificial intelligence) werden in der Fachliteratur meist die folgenden drei Disziplinen genannt, die die Entwicklung der Künstlichen Intelligenz (im weiteren kurz KI genannt) nachhaltig beeinflußten (siehe [Lun90]):

- die mathematische Logik
- die Algorithmentheorie
- die Rechentechnik

Die **mathematische Logik** beschäftigt sich mit der Formalisierung des Schlußfolgerungsprozesses. Ihr Ziel ist es, aus gegebenen Annahmen (Axiomen) wahre Aussagen herzuleiten. Leibniz[1] (1646-1716) beschreibt dieses Ziel so (aus [Dod90]):

> *Wenn Streitfragen auftreten würden, dann gäbe es zwischen zwei Philosophen nicht mehr Grund zum Streit als zwischen zwei Buchhaltern. Denn es würde genügen, ihre Stifte in die Hand zu nehmen, sich an ihre Schiefertafeln zu setzen und zu sich zu sagen (mit einem Freund als Zeugen, wenn sie wollten): Laß es uns berechnen.*

Dies Vorstellung von der Berechenbarkeit 'alles Wahren' aus einer Grundmenge von Axiomen hielt sich bis ins 20. Jahrhundert. Erst Kurt Gödel (1906-1978) bewies 1932 in seinem Unvollständigkeitstheorem, daß jedes widerspruchsfreie Axiomensystem unvollständig in dem Sinne ist, daß es Aussagen gibt, die zwar inhaltlich wahr sind, aus dem Axiomensystem jedoch nicht abgeleitet werden können.

Die mathematische Logik begegnet uns auf dem Gebiet der KI vor allem in Form von **Deduktionssystemen** und **logischer Programmierung**.

Die **Algorithmentheorie** beschäftigt sich mit der Frage, wie man komplexe Berechnungen auf eine Folge elementarer Umformungen zurückführen kann. Ihr zentraler Begriff, der auch für das gesamte Gebiet der Informatik von großer Bedeutung ist, ist der des Algorithmus. Grundlegend dafür waren die Forschungen von Turing (1912-1954) zur Frage, wie ein Rechner beschaffen sein muß, um einen (beliebigen) Algorithmus ausführen zu können (Turing-Maschine)

[1] Gottfried Wilhelm Leibniz: Dissertio de Arte Combinatoria. Leipzig, 1666.

und von Church mit seiner These, daß ein Algorithmus ein Verfahren ist, für das eine Turing-Maschine existiert.

Des weiteren wurde von Turing ein Kriterium für die 'Intelligenz' von Rechnern (der sogenannte **Turing-Test**) formuliert, der für die Beurteilung von Verfahren der KI von großer Bedeutung ist. Dabei kommuniziert eine Person A mit einer Person B und einem Rechner ('Person' C). Wenn Person A trotz geschickter Fragestellung nicht entscheiden kann, welche Antworten von Person B (Mensch) und welche von Person C (Rechner) kommen, dann kann der Rechner als 'intelligent' bezeichnet werden[1].

Die Erkenntnisse der Algorithmentheorie spiegeln sich vor allem in der in der KI weit verbreiteten **Symbolmanipulation** (nichtnumerische Algorithmen) wider.

Die Entwicklung der **Rechentechnik** geht in ihrer mechanischen Phase bis ins späte Mittelalter zurück. Die ersten Ideen zu rechentechnischen Konzepten, die den heutigen ähnlich sind, entwickelte Babbage (1792-1871). Aber erst im Jahre 1937 wurden diese Konzepte im Rechner Z1 von Konrad Zuse (geb. 1910) erfolgreich angewendet. Das von John von Neumann (1903-1957) 1947 geschaffene Prinzip einer sequentiellen Rechnerarchitektur bildet auch heute noch die Grundlage der meisten Rechner.

Die Bedeutung der Rechentechnik für die KI liegt natürlich in erster Linie in der Bereitstellung einer gerätetechnischen Grundlage für die Abarbeitung von wissensbasierten Programmen. Auf der anderen Seite beeinflußt die KI aber auch die Entwicklung neuer Rechnerarchitekturen, z.B. von Parallelrechnern und LISP-Maschinen.

Die Entwicklung der KI kann in vier, sich z.T. überlappende Epochen eingeteilt werden (nach [Wid89]):

Die 50er und 60er Jahre, die Epoche der allgemeinen Problemlöser[2]. Die KI verfolgte das Ziel, universelle Prinzipien herauszufinden, die intelligentem Verhalten allgemein zugrunde liegen. Dabei versuchte man Algorithmen zu finden, die beim Lösen von Problemen ohne großes Apriori-Wissen Intelligenz zeigen. Als Studienobjekte wurden dabei vor allem Spiele verwendet (z.B. Schach, Dame, Backgammon). Auf diesem Gebiet wurden einige beachtliche Erfolge erzielt, während bei der Bearbeitung von realen Problemen[3] nur wenige Fortschritte gemacht werden konnten.

1965 bis 1970, die frühe Phase der Expertenprogramme. Die Motivation für diese Epoche ergab sich aus der Erkenntnis, daß zur Realisierung von effektiven Problemlösern spezielles Wissen (Expertenwissen) über das betrachtete Fachgebiet notwendig ist. Eines der erfolgreichsten Projekte dieser Zeit war DENDRAL, ein kommerzielles Programm zur Unterstützung von Chemikern bei der Analyse der Struktur komplexer chemischer

[1] Dieser Test enthält auch implizit die Forderung, daß sich der Rechner an den Menschen anzupassen habe und nicht umgekehrt.

[2] Der auch heute noch umstrittene Begriff **artificial intelligence** setzte sich erst nach der berühmten Dartmouth-Konferenz im Jahre 1956 durch.

[3] Die Bewilligung von Forschungsmitteln hing damals wie heute von einer gewissen Praxisrelevanz der bearbeiteten Themen ab.

Verbindungen anhand von Massenspektrogrammen. Weitere erfolgversprechende Forschungsgebiete waren das Verstehen natürlicher Sprache[1] und das Computer-Sehen.

1970 bis 1980, die Wachstumsphase der Expertensysteme. Diese Expertensysteme, die vor allem medizinische Probleme behandelten, verfeinerten die Methoden zur Behandlung von realen Problemen mit ihren Besonderheiten wie Komplexität und Unsicherheit/Ungenauigkeit des verwendeten Wissens. Das wohl bekannteste Expertensystem dieser Epoche ist MYCIN[2], ein Programm, das Mediziner bei der Diagnose und Behandlung von Blutinfektionen unterstützt.

1975 bis zur Gegenwart, die Epoche der Expertensystem-Sprachen. Diese Sprachen beinhalten spezielle Unterstützung für die Wissensdarstellung und -verarbeitung (Inferenz), die es dem Nutzer erlaubt, selbst Expertensysteme für seine Aufgaben zu schreiben. Ein besonderes Augenmerk sollte auch auf das Projekt der 5. Rechnergeneration gerichtet werden, das in Japan im Jahre 1982 in Angriff genommen wurde (siehe [Bro86]). Dabei wird unter einem Rechner der 5. Generation ein auf Logik basierender, wissensverarbeitender Rechner verstanden, der mit seinem Nutzer natürlichsprachlich kommunizieren kann. Dadurch soll erreicht werden, daß der Dialog zwischen Nutzer und Computer weitestgehend 'menschliches' Niveau erreicht.

Die oben geschilderte Entwicklungsgeschichte zeigt, daß es sich bei der KI um ein sehr dynamisches Fachgebiet handelt, das in Zukunft immer mehr Praxisrelevanz erlangen wird.

3.2 Die Architektur wissensbasierter Systeme

Unter einem **wissensbasierten System** (knowledge based system) versteht man ein Programmsystem, das explizites Wissen über einen Problembereich besitzt und mit diesem Wissen arbeiten kann. Die Betonung liegt dabei auf 'explizitem' Wissen, d.h. das im Programm verwendete Wissen muß in irgendeiner Form deklarativ (in Form einer Wissensrepräsentation) und nicht prozedural (in Form eines Algorithmus) vorliegen. Dabei beinhaltet der Begriff **Wissen** Kenntnisse und Erfahrungen, nicht aber Erkenntnis und Einsicht.

Ein **Expertensystem** ist ein spezielles wissensbasiertes System, das Fachkompetenz in einem bestimmten, abgegrenzten Problemgebiet besitzt und bezüglich der Ergebnisse seiner Arbeit einem menschlichen Fachexperten ähnlich ist.

In der wissenschaftlichen Literatur zur KI erfolgt oft eine Gleichsetzung von wissensbasiertem System und Expertensystem. Das ist meiner Meinung nach nicht korrekt, da es sich bei einem Expertensystem um den Spezialfall eines wissensbasierten Systems handelt. An Expertensysteme wird die Forderung gestellt, daß sie prinzipiell in der Lage sind, einen Experten (zumindest auf einem eingeschränkten Fachgebiet) zu ersetzen. Die meisten wissensbasierten Systeme (und auch ein Teil von denen, die sich Expertensysteme nennen) sind dazu nicht in der Lage. Allen Newell sagt dazu (in [Buc84]):

[1] Die Programmiersprache PROLOG ist eines der Ergebnisse dieses Forschungsgebietes, da sie als Hilfsmittel zum Verstehen natürlichsprachlicher Texte entwickelt wurde.

[2] Eine sehr interessante Beschreibung des Systems und des Prozesses seiner Entwicklung aus der Sicht der Beteiligten ist in [Buc84] enthalten.

The population of so-called expert systems is rapidly becoming mongrelized to include any system that is applied, has some vague connection with AI systems and has pretentions of success.

Ein wissensbasiertes System besitzt i.allg. folgende Bestandteile (siehe auch Abbildung 3.1):

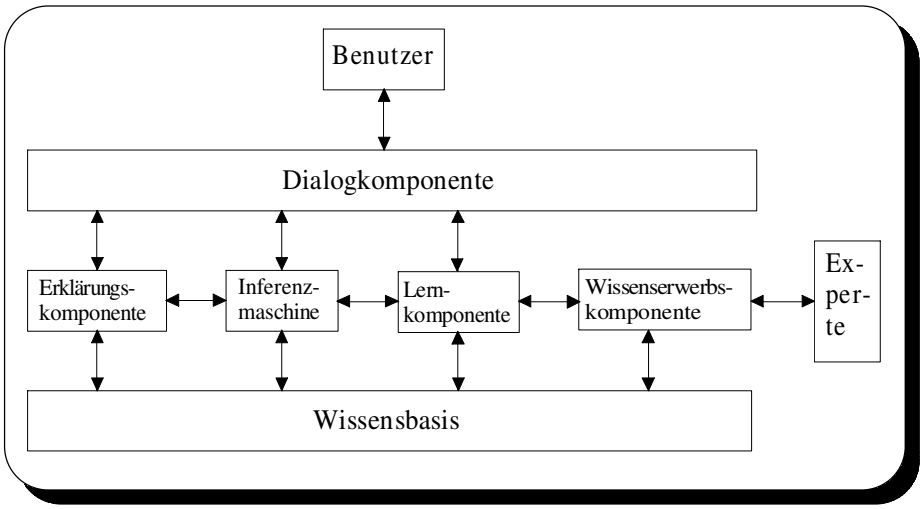

Abbildung 3.1: Aufbau eines wissensbasierten Systems

Eine **Wissensbasis**, die das bereichsspezifische Wissen (statische Wissensbasis) und das fallspezifische Wissen (dynamische Wissensbasis, Datenbasis) enthält.

Eine **Inferenzkomponente** ('Inferenzmaschine'), die das fall- und bereichsspezifische Wissen verarbeitet.

Eine **Erklärungskomponente**, die das Verhalten des Systems dem Benutzer verständlich machen soll.

Eine **Lernkomponente**, die aus gelösten Problemen neues Wissen ableitet und dieses in die Wissensbasis einfügt.

Eine **Wissenserwerbskomponente**, die es ermöglicht, neues bereichsspezifisches Wissen in das System zu integrieren.

Eine **Dialogkomponente**, die der Kommunikation mit dem Benutzer dient.

Viele wissensbasierte Systeme verfügen nicht über alle oben angeführten Systemkomponenten. Im allgemeinen spricht man von einem wissensbasierten System schon dann, wenn die drei Komponenten Wissensbasis, Inferenzmaschine und Dialogkomponente vorhanden sind. Dabei geht natürlich ein großer Teil der einem wissensbasierten System innewohnenden Flexibilität und Leistungsfähigkeit verloren. Das bedeutet im einzelnen:

Das Fehlen einer Erklärungskomponente führt dazu, daß der Nutzer die Lösung des Systems nicht hinterfragen kann. Das kann (besonders in solch sensiblen Bereichen wie der Medizin) dazu führen, daß der Nutzer das Programm vollständig ablehnt, da er ihm nicht traut.

Das Fehlen einer Lernkomponente führt dazu, daß das System während der Nutzung keine Erfahrungen sammeln kann. Es muß also auch bei einem Problem, das es schon ein dutzendmal gelöst hat, wieder vollständig von vorn anfangen. Andererseits ist die Entwicklung automatischer Lernkomponenten eine nichttriviale Aufgabe, die im Augenblick noch Thema der Forschung ist.

Das Fehlen einer Wissenserwerbskomponente führt dazu, daß das System auf einem festen Stand des Fachwissens verharrt. Wenn sich auf diesem Fachgebiet neue Erkenntnisse ergeben, dann kann sich das System nicht anpassen. Der einzige Ausweg ist dann eine (teilweise) Neuprogrammierung der Wissensbasis.

3.3 Wissensrepräsentation und Wissensverarbeitung

Der Begriff **Wissensrepräsentation** bezeichnet in der KI sowohl den Prozeß der Umformung informellen Wissens (Expertenwissen, Lehrbuchwissen usw.) in formalisiertes und damit dem Rechner zugängliches Wissen[1] als auch das Ergebnis dieses Prozesses, die konkrete Datenstruktur zur Speicherung des Wissens im Rechner. In diesem Abschnitt wollen wir uns nur mit der zweiten Bedeutung des Begriffes, der Art der Wissensspeicherung, beschäftigen.

An eine Wissensrepräsentation werden im wesentlichen drei Anforderungen gestellt:

1. Ausdrucksfähigkeit

2. Uniformität

3. Erhaltung von Strukturen

Unter **Ausdrucksfähigkeit** versteht man, daß sich alle wichtigen Sachverhalte des Problembereiches (z.B. Relationen, Vererbung) in der Wissensrepräsentation explizit darstellen lassen.

Unter **Uniformität** versteht man, daß ähnliche Sachverhalte des Problembereiches auch auf ähnliche Art und Weise dargestellt werden.

Unter **Erhaltung der Strukturen** versteht man, daß zusammengehörende Sachverhalte des Problembereiches bei der Formalisierung des Wissens zu Einheiten zusammengefaßt werden.

Eng mit dem Problem der Wissensrepräsentation ist das der **Wissensverarbeitung** verbunden. Bei der Wissensverarbeitung (**Inferenz**) geht es um die Schlußfolgerung von neuem Wissen aus schon bekanntem. Dieser Prozeß erfordert konkrete Informationen darüber, wie das Wissen strukturiert ist (d.h. über die Wissensrepräsentation). Nur so ist eine effektive und fehlerfreie Inferenz möglich.

Im Augenblick existiert leider noch keine geschlossene Theorie der Wissensrepräsentation, die es erlauben würde, für jeden Anwendungsbereich die geeignete Darstellungsform abzuleiten. In der Praxis hat sich im Laufe der Jahre eine Vielzahl von Wissensrepräsentationsschemata (ausgehend von konkreten Programmsystemen) herausgebildet. Trotz ihrer scheinbaren

[1] Dieser Prozeß (**knowledge engineering**) erfordert hochqualifizierte Fachkräfte, die oft als Wissens-Ingenieure bezeichnet werden.

Verschiedenartigkeit lassen sich jedoch allgemeine Darstellungsprinzipien erkennen. Die wichtigsten davon sind

- Regeln
- Frames
- semantische Netze

Diese drei Darstellungsformen sollen in den nächsten Abschnitten näher erläutert werden, wobei wir nicht aus den Augen verlieren wollen, daß wir eine passende Wissensrepräsentation zur Darstellung heuristischen Entwurfswissens suchen.

3.3.1 Regeln

Bei regelbasierten Systemen werden die Daten zumeist in Form von Objekt-Attribut-Wert-Tripeln dargestellt.

Beispiele: PINGUIN : kann_fliegen : nein

PINGUIN : kann_schwimmen : ja

PINGUIN : anzahl_junge : 1

Das erste Element (PINGUIN) bezeichnet ein Objekt, das zweite Element gibt ein Attribut (Name einer Eigenschaft) des Objektes an und das dritte Element quantifiziert dieses Attribut (je nach Art des Attributes numerischer Wert, boolescher Wert, Zeichenkette usw.).

In Regeln dargestelltes Wissen hat die allgemeine Form

WENN Bedingung **DANN** Schlußfolgerung .

Vom Standpunkt der mathematischen Logik betrachtet handelt es sich bei diesem Konstrukt um eine Implikation ($\rightarrow$).

Der Bedingungsteil der Regel dient zur Spezifikation einer Situation. Er besteht i.allg. aus der Konklusion (UND-Verknüpfung) von logischen Ausdrücken. Im einfachsten Falle handelt es sich dabei um Mustervergleiche (pattern matching), d.h. um Tests der Eigenschaftswerte von Objekten.

Beispiel: WENN X kann fliegen DANN X ist ein Vogel

Wenn alle Ausdrücke des Bedingungsteils wahr sind, dann kann der Schlußfolgerungsteil gefolgert werden.

Der Schlußfolgerungsteil der Regel dient zur Manipulation der Wissensbasis. Er besteht aus einer Menge von elementaren Operationen über Objekt-Attribut-Wert-Tripeln (Löschen, Einfügen, Ändern).

Wie bereits aus der obigen Beschreibung ersichtlich verwenden regelbasierte Systeme als Ableitungsmechanismus den **Modus Ponens**.

Modus Ponens:

$$\begin{array}{l} X \rightarrow Y \\ X \\ \hline Y \end{array}$$

Umgangssprachlich ausgedrückt bedeutet die Notation folgendes:

> Es existiert eine Implikation 'Wenn X wahr ist, dann ist auch Y wahr' und es existiert ein Fakt 'X ist wahr'. Daraus kann der Fakt 'Y ist wahr' geschlußfolgert werden.

Regelbasierte Systeme können die Regeln grundsätzlich auf zwei verschiedene Arten verwenden:

- mittels Vorwärtsverkettung
- mittels Rückwärtsverkettung

Vorwärtsverkettung

Bei der Vorwärtsverkettung werden aus der Menge aller Regeln diejenigen herausgesucht, deren Bedingungsteil erfüllt ist (die potentiell anwendbar sind). Aus den anwendbaren Regeln wird eine ausgewählt und deren Schlußfolgerungsteil ausgeführt. Dieser Prozeß wird solange fortgesetzt, bis ein vorher festgelegter Endzustand erreicht ist oder keine Regel mehr angewendet werden kann.

Die Regel wird bei dieser Methode 'von links nach rechts' angewendet, d.h. zuerst werden die Bedingungen ausgewertet und dann werden die Aktionen durchgeführt. Die Anwendung erfolgt in Richtung des (gedachten) Implikationspfeiles, also vorwärts.

Diese Art der Wissensverarbeitung entspricht dem Herleiten von Schlußfolgerungen aus bekannten Fakten.

Rückwärtsverkettung

Bei der Rückwärtsverkettung wird versucht, eine bestimmte Schlußfolgerung (ein 'Ziel') zu beweisen. Da eine Schlußfolgerung abgeleitet werden kann, wenn alle Bedingungen der entsprechenden Regel erfüllt sind, sind solche Schlußfolgerungen ('Teilziele') zu beweisen, so daß alle Bedingungen der Regel erfüllt werden. Dieser Prozeß setzt sich fort, bis für alle augenblicklichen Teilziele aktuell anwendbare Regeln gefunden wurden oder es für ein Teilziel keine zugehörige Regel mehr gibt.

Die Regel wird bei dieser Methode 'von rechts nach links' angewendet, d.h. zuerst wird eine zu beweisende Schlußfolgerung vorgegeben und dann wird versucht, alle benötigten Bedingungen herzuleiten. Die Anwendung erfolgt entgegen der Richtung des (gedachten) Implikationspfeiles, also rückwärts.

Diese Art der Wissensverarbeitung entspricht der Suche nach Fakten, die eine bestimmte Schlußfolgerung zulassen bzw. eine aufgestellte Hypothese bestätigen.

Vorteile der Wissensrepräsentation mit Regeln

Der wohl größte Vorteil dieser Repräsentationsart liegt in der Modularität der Regeln.

Unter **Modularität** verstehen wir dabei den Grad der Aufteilung der funktionellen Teile eines Programmes in isolierbare Stücke. Hoch modular sind solche Programme, deren funktionelle Teile geändert werden können (Hinzufügen, Löschen, Ersetzen), ohne daß das unvorhergesehene Auswirkungen auf die anderen funktionellen Teile hat. Daraus kann man die Faustregel ableiten, daß die Programmodularität umgekehrt proportional zur Stärke der Bindung zwischen den funktionellen Teilen ist.

Die Modularität von Regelsystemen ergibt sich daraus, daß die jeweils nächste zu verwendende Regel sich einzig und allein aus dem Inhalt der Wissensbasis ergibt und keine Regel je direkt gerufen wird (Interaktion der Regeln nur über die Wissensbasis).

Ein weiterer Vorteil ist, daß durch die Modularität bedingt eine einfache Modifizierung der Regeln möglich ist.

Die Erklärungsfähigkeit des wissensbasierten Systemes wird dadurch erleichtert, daß jede Regel für sich erklärt werden kann, da sie von den anderen Regeln unabhängig ist. Jede Regel stellt in diesem Sinne eine völlig autarke Wissenseinheit[1] dar.

Ein nicht zu unterschätzender Vorteil ist auch, daß nach heutigen Erkenntnissen Fachexperten oft ihre Vorgehensweise anhand von Regeln erklären (diese Annahme ist natürlich trotzdem mit Vorsicht zu behandeln, da über die Mechanismen des menschlichen Denkens im Augenblick nur wenig wirklich fundiertes Wissen vorliegt).

Nachteile der Wissensrepräsentation mit Regeln

Die Regeldarstellung hat durch ihren einfachen Aufbau (WENN-DANN-Struktur, einfache Objekt-Attribut-Wert-Tripel) eine nur beschränkte Ausdrucksfähigkeit[2].

Die Sichtbarkeit des Verhaltensflusses (Leichtigkeit, mit der das Gesamtverhalten eines Programmes verstanden werden kann) ist sehr gering. Selbst für konzeptuell einfache Aufgaben ist das schrittweise Verhalten eines Regelsystems eher undurchsichtig. Das hat vor allem zwei Ursachen:

1. Steuer- und Datenfluß bilden bei Regelsystemen eine Einheit

2. in jedem Programmzyklus kann sich die Wissensbasis ändern

[1] Im Englischen oft 'chunk of knowledge' (etwa: Wissensklumpen) genannt.

[2] Das bezieht sich natürlich auf ein vernünftiges Kosten-Nutzen-Verhältnis. Da man auch mit einem Regelsystem eine Turing-Maschine nachbilden kann, entspricht die Ausdruckskraft letztendlich der jeder beliebigen Programmiersprache (nach [Dav84]).

Eng damit verbunden ist die Frage der **Programmierbarkeit** (Wie leicht ist es, in diesem Formalismus zu programmieren?). Die Modularität wurde zwar oben als Vorteil von Regelsystemen genannt, bei manchen Aufgaben kann aber die nur minimale Interaktion zwischen den Regeln störend wirken. In diesem Fall muß man dann auf 'unsaubere' Programmierung mittels Seiteneffekten zurückgreifen, indem man z.B. benötigte Parameter für bestimmte Regeln in der Wissensbasis zwischenspeichert. Auch gibt es Aufgaben, die sich nicht in unabhängige Teilprobleme (Wissenseinheiten) zerlegen lassen.

Ein eher pragmatischer Nachteil ist, daß viele Regelsysteme sehr zeitintensiv sind, da in jedem Zyklus erneut alle Regeln ermittelt werden müssen, die anwendbar sind (auch wenn die Menge dieser Regel oft gleichbleibt oder sich nur geringfügig ändert). Deshalb wurden Strategien entwickelt, die die Zeit zur Bestimmung der Konfliktmenge verkürzen sollen[1].

3.3.2 Frames

Bei framebasierten Systemen geht man vom Prinzip der **Teilmengenhierarchie** aus (siehe [Fri90]).

Sei M eine Menge von Objekten eines Objektbereiches, die alle eine oder mehrere Eigenschaften e_i aus einer Eigenschaftsmenge E besitzen. M_i sei die Menge aller Objekte aus M, die die Eigenschaft e_i besitzen.

Man kann die Menge der Objekte M bezüglich einer ausgezeichneten Eigenschaft e_1 in zwei disjunkte Teilmengen aufteilen:

die Teilmenge M_1 aller Objekte aus M, die die Eigenschaft e_1 besitzen

die Teilmenge M_{-1} aller Objekte aus M, die die Eigenschaft e_1 nicht besitzen

Dieser Prozeß der Aufspaltung kann mit den erhaltenen Teilmengen und anderen Eigenschaften e_2, e_3, ... fortgesetzt werden, bis wir Teilmengen erhalten, die sich bezüglich ihrer Eigenschaften nicht weiter aufspalten lassen.

Das Ergebnis dieses Prozesses ist eine Teilmengenhierarchie, die in Form eines Hierarchiebaumes (Wurzel: Menge M, Blätter: nicht weiter aufspaltbare Teilmengen) dargestellt werden kann. Dabei gilt das Prinzip der **Vererbung** (inheritance), d.h. daß Teilmengen von ihren Vorgängermengen im Baum alle Eigenschaften erben. Hier zeigt sich auch die enge Verbindung dieses Ansatzes zur objektorientierten Programmierung.

Beispiel: M = {PINGUIN, STRAUSS, ADLER, ENTE}

E = {kann_fliegen, kann_schwimmen}

siehe Abbildung 3.2

In Analogie zur oben beschriebenen Teilmengenhierarchie wird bei Frames oft von einer **Framehierarchie** gesprochen.

[1] Ein Beispiel dafür ist der **RETE-Match-Algorithmus** von Forgy (siehe [For82]), der in der Produktionssystem-Sprache OPS5 zum Einsatz kommt.

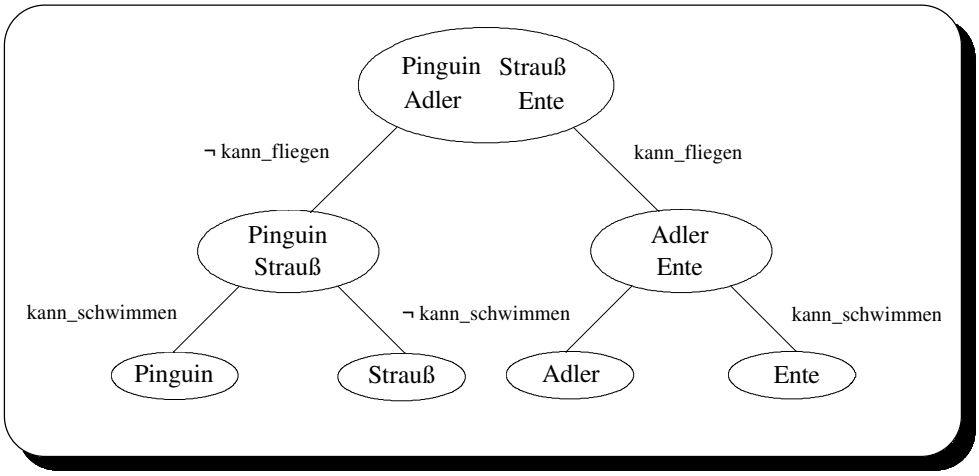

Abbildung 3.2: Beispiel für eine Teilmengenhierarchie

Ein **Frame** ('Rahmen') beschreibt eine konkrete Objektklasse; er ist in gewisser Weise der Prototyp der Objekte dieser Klasse. Er besteht aus einer Menge von **Slots** ('Einschübe'), die die Eigenschaften von Objekten beschreiben. Ein Slot ist ein Tupel (Slotname, Slotwert), wobei Slotname der Name einer Eigenschaft und Slotwert der (quantitative) Wert dieser Eigenschaft ist.

Die Realisierung der Framehierarchie erfolgt über zwei spezielle Slots. Der eine Slot (oft superslot, is_a oder instance_of genannt) enthält den Namen des unmittelbar übergeordneten Frames (der Obermenge), während der andere Slot (oft subslot, subclass oder instances genannt) die Namen aller unmittelbar untergeordneten Frames (der Teilmengen) enthält.

Ein Frame erbt alle Eigenschaften (Slots) der übergeordneten Frames, fügt evtl. neue Eigenschaften hinzu und vererbt diese weiter an die untergeordneten Frames.

Wie die beschriebene Struktur von Framesystemen deutlich macht, können mit Frames vor allem Probleme mit einer hierarchischen Struktur behandelt werden. Dabei gibt es grundsätzlich zwei Möglichkeiten der Inferenz:

1. Klassifizierung von Objekten

Gegeben ist ein Objekt mit einer oder mehreren Eigenschaften, gesucht wird nach dem Frame (der Klasse), der Objekte mit diesen Eigenschaften beschreibt. Dabei wird der Hierarchiebaum von der Wurzel aus durchlaufen und es wird je nach gerade betrachteter Eigenschaft verzweigt.

Beispiel: Objekt: Emu[1]

Eigenschaften: ¬ kann_fliegen ('¬' ist ein Negationsoperator)
¬ kann_schwimmen

→ Objekt Emu gehört zur Klasse Strauß

[1] etwa 1.5 m großer und 50 kg schwerer Laufvogel Australiens

2. Suche nach Eigenschaften

Gegeben ist ein Objekt, von dem bekannt ist, daß es zu einer bestimmten Klasse gehört, gesucht werden die Eigenschaften dieses Objektes. Dabei wird der Hierarchiebaum von dem Frame aus, der das Objekt repräsentiert, nach oben bis zur Wurzel durchlaufen. Die Eigenschaften der durchlaufenen Frames sind auch Eigenschaften des gegebenen Objektes.

Beispiel: Objekt Emu gehört zur Klasse Strauß

$\rightarrow$ ¬ kann_fliegen
¬ kann_schwimmen

3.3.3 Semantische Netze

Das Prinzip der semantischen Netze[1] ist stark an die Graphentheorie angelehnt. In diesem Sinne ist ein **semantisches Netz** ein nichtzyklischer, gerichteter Graph. Dabei repräsentieren die Knoten des Graphen Objekte oder Eigenschaften von Objekten, während die Kanten binäre Relationen zwischen diesen Objekten darstellen. Wichtige Relationen beschreiben die Beziehung eines Objektes zu der übergeordneten Objektklasse (etwa wie Superslot bei Frames) und die Beziehung des Objektes zu einer Objekteigenschaft (etwa wie 'normale' Slots bei Frames). Das Beispiel eines semantischen Netzes zeigt Abbildung 3.3.

Die Motivation für diese Art von Wissensdarstellung liegt vor allem in der Annahme, daß das menschliche Gedächtnis ähnlich arbeitet, d.h. daß es Objekte und die Relationen zwischen ihnen speichert.

Die Arbeit mit semantischen Netzen erfolgt vor allem über eine spezielle Form der Unifikation:

1. Die bekannten Fakten des Anwendungsgebietes werden in ein semantisches Netz (Fakten-Netz) umgewandelt. Das zu lösende konkrete Problem wird in Form eines Ziel-Netzes (das meist Variablen enthält) dargestellt.

2. Dann wird versucht, das Fakten-Netz und das Ziel-Netz zu unifizieren[2]. Das Ergebnis dieses Prozesses sind zumeist Variablenbindungen, die die Lösung des Problemes angeben.

Der Vorteil semantischer Netze liegt darin, daß alle Beziehungen zwischen Objekten explizit dargestellt sind (alle Fakten über ein Objekt sind in den Knoten zu finden, die mit dem Objektknoten über Kanten verbunden sind). Durch die Vererbung von Eigenschaften zwischen Objekten ist eine effiziente Informationsdarstellung möglich. Von Vorteil ist auch, daß für das Gebiet der Graphen und insbesondere für Graphensuchalgorithmen eine gut ausgebaute Theorie existiert.

Der größte Nachteil der semantischen Netze ist meiner Ansicht nach die Kompliziertheit des Inferenzprozesses. Es mag ja zutreffen, daß die Struktur und Funktion der semantischen Netze der internen Arbeitsweise des menschlichen Gedächtnisses ähnelt, aber sie entspricht auf jeden Fall nicht dem menschlichen Problemlösen wie es dem Fachexperten selbst bewußt wird.

[1] ursprünglich entwickelt für die Repräsentation natürlichsprachlicher Aussagen (Quillian 1968)

[2] Eine genaue Beschreibung des Unifikationsalgorithmus würde an dieser Stelle zu weit führen.

Deshalb wird es schwierig sein, die Erfahrungen eines Fachexperten in Form eines semantischen Netzes zu speichern.

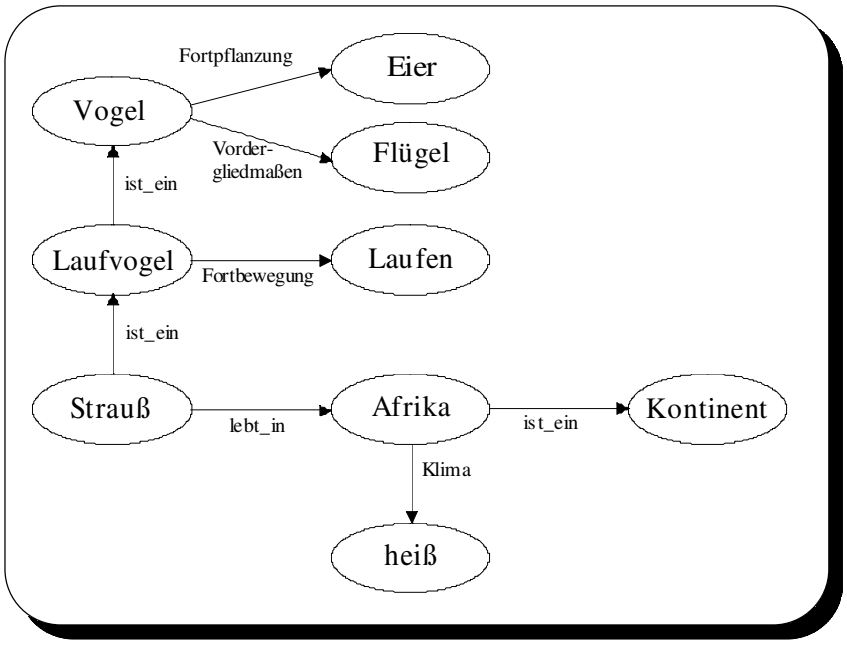

Abbildung 3.3: Beispiel eines semantischen Netzes

3.4 Anwendungsgebiete wissensbasierter Systeme

Im Zusammenhang mit dem Paradigma der wissensbasierten Programmierung muß man sich natürlich auch die Frage stellen, für welche Problemklassen wissensbasierte Systeme überhaupt vorteilhaft anwendbar sind. Nach Widman und Loparo [Wid89] existieren folgende Hauptanwendungsgebiete:

Steuerung Die Durchführung von Eingriffen in die reale Welt, um gewünschte Ziele zu erreichen.

Entwurf Das Erzeugen von 'Bauplänen' für Objekte, die bestimmten Erfordernissen genügen.

Diagnose Der Prozeß der Fehlerfindung in einem System (auch in einem lebenden System) durch Interpretation von potentiell unscharfen oder unvollständigen Daten.

Unterweisung Das Vermitteln neuer Konzepte und Informationen an Nichtexperten.

Interpretation Die Analyse von Daten, um ihre Bedeutung zu ermitteln.

Kontrolle Die fortlaufende Interpretation von Signalen und die Anzeige, wann (menschliche) Eingriffe erforderlich sind.

Planung Das Erzeugen von Aktionsplänen, deren Ausführung zum Erreichen gewünschter Ziele führt.

Prognose Die Vorhersage des Verlaufes der Zukunft durch Modelle der Vergangenheit und Gegenwart.

Reparatur Das Erzeugen von Vorschriften zur Behebung von Problemen in der realen Welt.

Natürlich ist der Gewinn, den diese Anwendungsgebiete aus der KI ziehen (werden), nicht überall gleich. Viel hängt davon ab, welche Fortschritte auf den einzelnen Gebieten auf algorithmisch/prozeduralem Wege möglich sind. Man kann jedoch davon ausgehen, daß in der Zukunft auf allen Gebieten die verwendeten Modelle immer umfangreicher und die auszuwertende Datenmasse immer größer werden wird. Deshalb sollte eigentlich jeder, der ein Problem aus einem der oben genannten Aufgabengebiete zu lösen hat, die Verwendung wissensbasierter Methoden zumindest in Betracht ziehen.

Die in dieser Arbeit zu behandelnde Frage der Dimensionierung technischer Systeme gehört zum oben genannten Aufgabenkomplex 'Entwurf'. Die Verwendung eines wissensbasierten Systemes zur Problemlösung erscheint also zumindest möglich.

3.5 Die Verbindung von wissensbasierter und numerischer Datenverarbeitung

Trotz der großen Vorteile, die die wissensbasierte Programmierung bietet, wird sie viele rein numerische Verfahren, die sich in der Vergangenheit herausgebildet haben, nicht verdrängen können. Das betrifft insbesondere solche Verfahren, denen ein effizienter Algorithmus zugrunde liegt, wie das gerade auf dem Gebiet der Mathematik oft der Fall ist.

Der Sinn der wissensbasierten Systeme kann natürlich nicht darin liegen, diese Verfahren ersetzen zu wollen, sondern sie zu integrieren. Warum soll ein wissensbasiertes Programm nicht auf mathematische Bibliotheken zurückgreifen, wenn sich diese schon seit Jahrzehnten bewährt haben.

Dabei tritt natürlich das Problem der Verbindung der dem Wesen nach verschiedenen Ansätze auf. Die numerischen Systeme sind zumeist in einer imperativen Sprache wie FORTRAN, C oder PASCAL implementiert während sich die wissensbasierten Systeme auf logische Sprachen wie PROLOG oder funktionale Sprachen wie LISP stützen. Diese Sprachen können oft wegen ihrer unterschiedlichen Grundprinzipien nicht gekoppelt werden.

In der Literatur findet man verschiedene Verfahren, um die genannten Schwierigkeiten zu umgehen. Einige davon möchte ich im folgenden kurz vorstellen.

Dateitransfer

Die wohl einfachste Möglichkeit der Kopplung ist die über die Dateiverwaltung des Betriebssystems. Sowohl imperative als auch wissensbasierte Systeme besitzen Kommandos, um Dateien zu manipulieren und um Betriebssystemrufe auszuführen.

Das eine Programm (i.a. das wissensbasierte) schreibt die zu vermittelnden Daten in eine Datei und ruft dann das andere (i.a. numerische) Programm auf. Dieses Programm liest die Daten aus der Datei, verarbeitet sie und schreibt die Ergebnisse wiederum in eine Datei, die dann von dem aufrufenden Programm gelesen werden kann.

Der Vorteil dieses Prinzips ist die gute Kontrollierbarkeit des Vorganges durch den Nutzer. Das numerische Programm kann zuerst separat getestet werden, indem der Nutzer selbst Daten in die eine Datei einträgt und dann aus der anderen Datei die Ergebnisse ausliest. Auch nach der Kopplung der beiden Programme kann durch Betrachtung der Ein- und Ausgabedateien ständig kontrolliert werden, ob die Verbindung korrekt arbeitet.

Der große Nachteil ist der Zeitaufwand, der für das Schreiben und Lesen der Dateien und für das Starten des numerischen Programmes notwendig ist.

Aus den obengenannten Gründen wird dieser Ansatz vor allem bei solchen Systemen verwendet, die einen prototypischen Charakter tragen.

Verwendung einer Sprache

Bei diesem Verfahren wird entweder der wissensbasierte Teil in der Sprache des numerischen Teiles implementiert oder umgekehrt. Oft wird dieses Vorgehen schon deshalb gewählt, weil dem Entwickler keine geeignete wissensbasierte Sprache zur Verfügung steht und er deshalb bezwungen ist, eine der verfügbaren imperativen Sprachen zu benutzen.

Der Vorteil dieser Methode ist die Einheitlichkeit der gesamten Implementation. Die gesamte Entwicklungsarbeit kann in einer Softwareumgebung erfolgen, für die meist effektive Programmierwerkzeuge zur Verfügung stehen.

Der Nachteil liegt darin, daß imperative Sprachen keine Konstrukte zur Wissensverarbeitung zur Verfügung stellen und die numerischen Fähigkeiten wissensbasierter Systeme oft nur rudimentär sind. Das erfordert eine (i.allg. ineffiziente) Verwendung der Sprache für Zwecke, für die sie eigentlich nicht gedacht ist.

Interprozeß-Kommunikation

Dieser Ansatz, der bei Multitasking-Betriebssystemen benutzt werden kann, besteht darin, den wissensbasierten und den numerischen Teil als zwei verschiedene Prozesse (im Extremfall auf zwei verschiedenen Rechnern) ablaufen zu lassen. Der Datenaustausch erfolgt über Interprozeß-Kommunikationsmechanismen (z.B. über Pipes).

Der Vorteil dieser Methode besteht in der Geschwindigkeit des Datentransfers. Da die Prozesse ihrem Wesen nach parallel laufen (auch wenn der Rechner keine echte Parallelität unterstützt), kann bei Verwendung geeigneter Hardware eine echte Parallelität des wissensbasierten und des numerischen Teiles erreicht werden, was zu einem beträchtlichen Laufzeitgewinn führen kann.

Der Nachteil ist, daß sich Interprozeß-Mechanismen vom Nutzer nur schwer überprüfen lassen, was sich in z.T. vergrößerten Fehlersuch-Zeiten in der Entwicklungsphase ausdrückt.

Einbinden des numerischen Teiles

In letzter Zeit entstehen immer mehr wissensbasierte Sprachimplementationen, die es erlauben, Prozeduren aus imperativen Programmiersprachen einzubinden. Ein Beispiel dafür ist die PROLOG-Implementation SNI-Prolog von Siemens-Nixdorf. In dieser PROLOG-Version ist es möglich, in C programmierte Funktionen so einzubinden, daß sie wie PROLOG-Prädikate genutzt werden können. Dadurch bleibt das zugrundeliegende logische Konzept von PROLOG voll erhalten und trotzdem können außerlogische Methoden (numerische Algorithmen, Datenbankanschluß, Fenstertechnik usw.) effektiv genutzt werden.

Der Vorteil dieses Konzeptes liegt in der Effektivität der Verbindung. Da die numerischen Teile direkt in den wissensbasierten Teil 'eingebettet' sind, erfolgt die Datenübergabe und die Abarbeitung sehr schnell.

Die Wahl eines der oben genannten Konzepte für eine konkrete Programmieraufgabe kann nach verschiedenen Gesichtspunkten erfolgen. Das wichtigste Kriterium wird die Verfügbarkeit der dargestellten Konzepte sein. Während Dateitransfer und Verwendung einer Sprache allgemein nutzbar sind, erfordern Interprozeß-Kommunikation und Einbinden numerischer Teile besondere Soft- bzw. Hardware. Ein weiteres Entscheidungskriterium ist das Verhältnis zwischen Effektivität und Testbarkeit des Systems. Für solche Systeme, die reinen Experimentalcharakter tragen, eignet sich der Dateitransfer, da bei ihm die Datenübergabe explizit erfolgt. Bei Systemen, die kommerziellen Charakter tragen und bei denen es deshalb auf Geschwindigkeit ankommt, empfehlen sich Interprozeß-Kommunikation oder Einbindung.

4. Die automatische Lösung von Dimensionierungsaufgaben

In diesem Kapitel wollen wir uns mit den Problemen und Entwurfsentscheidungen beschäftigen, die bei der Entwicklung eines Programmsystems zur automatischen Dimensionierung auftreten.

Da wir bei der rechnergestützten Dimensionierung nicht konkrete physische Systeme, sondern eher Beschreibungen dieser Systeme optimieren, werden wir im folgenden von **Entwurfs-Modellen** (oder genauer von **Dimensionierungs-Modellen**) sprechen. Wir abstrahieren dabei von all den Eigenschaften der technischen Systeme, die für die Dimensionierung nicht von Belang sind. Dadurch wird es möglich, mit dem vorgestellten Programmsystem völlig verschiedenartige Entwurfsobjekte zu bearbeiten.

In den folgenden Abschnitten werden wir die beiden wesentlichen Teilprobleme der rechnergestützten Dimensionierung näher behandeln. Dabei handelt es sich um die formale Beschreibung der Entwurfsmodelle sowie um die eigentliche Dimensionierung (sowohl numerisch als auch wissensbasiert).

4.1 Beschreibung von Dimensionierungsmodellen

In diesem Abschnitt wollen wir uns mit der formalen Beschreibung von Entwurfsmodellen beschäftigen, die bzgl. ihrer Parameterbelegung optimiert werden sollen. Die Wahl einer günstigen Beschreibungsform bestimmt im wesentlichen die Menge der Modelle, die dargestellt und optimiert werden können.

Die Beschreibung sollte sinnvollerweise aus zwei verschiedenen Komponenten bestehen und zwar aus

1. einer Formulierung der Dimensionierungsaufgabe (am günstigsten mittels einer speziellen Sprache) und

2. einer Modellierung des Systemverhaltens (in Form eines ausführbaren Programmes - im weiteren **Modellprogramm** genannt).

Dabei wird die erste Komponente benötigt, um das jeweils zu lösende Optimierungsproblem zu beschreiben. Die zweite Komponente dient während der Optimierung zur Ermittlung von Gütekriterien[1] der augenblicklichen Parameterbelegung, aus denen dann der Zielfunktionswert berechnet wird.

4.1.1 Die Formulierung der Dimensionierungsaufgabe

Zum Zweck der Formulierung von Dimensionierungsaufgaben wurde von mir eine spezielle Sprache mit Namen **DABS** (Dimensionierungsaufgaben-Beschreibungssprache) entwickelt. Mit dieser Sprache soll es möglich sein, beliebige Dimensionierungsaufgaben zu formulieren[2].

[1] im weitesten Sinne Leistungskenngrößen des zu untersuchenden Modelles

[2] Darüber hinaus ist es meiner Meinung nach möglich, auch andere Aufgabenstellungen, die auf der mathematischen Optimierung beruhen, darzustellen.

An eine Sprache zur Formulierung von Dimensionierungsaufgaben wird eine Anzahl von Anforderungen gestellt.

1. Die Sprache sollte dem Prinzip der **Vollständigkeit** genügen. Darunter ist zu verstehen, daß mit den Mitteln der Sprache beliebige Aufgaben der geforderten Problemklasse beschrieben werden können. Das erfordert eine Analyse der Struktur des zu behandelnden Problemraumes.

2. Die Sprache sollte eine **anwendungsfreundliche Notation** besitzen. Darunter ist zu verstehen, daß die in dieser Sprache formulierten Aufgaben vom Nutzer relativ leicht zu lesen und zu schreiben sind. Insbesondere ist zu fordern, daß es relativ einfach sein sollte, die Bedeutung der Aufgabenbeschreibung zu begreifen. Dazu ist erforderlich, daß sich alle wesentlichen Sachverhalte der Aufgabe in der Beschreibung in einer dem Menschen vertrauten Form widerspiegeln (z.B. Anlehnung an mathematische Notation).

3. Die Sprache sollte vom Rechner **'leicht' verarbeitet** werden können. Dazu gehört, daß der Rechner eine Aufgabenbeschreibung einfach auf ihre syntaktische und semantische Korrektheit prüfen und in eine rechnerinterne Darstellung transformieren kann[1]. Dabei wäre anzustreben, daß sich die Beschreibungssprache und die Implementierungssprache des Programmsystems ähneln, um eine gewisse Uniformität zu erreichen. Mir erscheint deshalb vorteilhaft, die Beschreibungssprache als Untermenge der für das Programmsystem verwendeten Implementierungssprache zu definieren.

Wie leicht zu erkennen ist, handelt es sich bei den oben aufgestellten Kriterien für die Beschreibungssprache DABS um Maximalforderungen, die sich zum Teil widersprechen. Vor allem das zweite und das dritte Kriterium (anwendungsfreundliche Notation $\leftrightarrow$ leicht verarbeitbar für Rechner) sind nur schwer gleichzeitig zu erfüllen. Spätestens an dieser Stelle muß man sich darüber klar werden, welche Implementierungssprache für das zu entwickelnde Programmsystem zu verwenden ist. Da wir die Beschreibungssprache DABS als Untermenge dieser Sprache definieren wollen, empfiehlt sich die Wahl einer **deklarativen und interpretativ abarbeitbaren Programmiersprache**. Das hat folgende Gründe:

1. Durch die Deklarativität ist es leicht möglich, Sachverhalte, die im weitesten Sinne als 'Wissen' angesehen werden können, darzustellen. Da eine Aufgabenbeschreibung auch als Wissen über einen Entwurfsgegenstand und die gewünschte Art seiner Verbesserung angesehen werden kann, läßt sich die Beschreibungssprache ohne größere Schwierigkeiten als Untermenge der deklarativen Programmiersprache formulieren.

2. Durch die interpretative Abarbeitung der Sprache wird es möglich, während der Laufzeit eines Programmes neue Sachverhalte (z.B. eine Aufgabenbeschreibung) zu integrieren, die dann als normale Bestandteile des Programmes angesehen und auch so verarbeitet werden können.

Bei der Wahl der Programmiersprache spielen natürlich nicht nur die oben genannten, eher theoretischen Gründe eine Rolle, sondern auch Erwägungen, die vorrangig praktischer Natur sind. Dazu gehört vor allem eine Betrachtung der dem normalen Nutzer zugänglichen Programmiersprachen. Das ist deshalb von Bedeutung, weil bei Verwendung einer interpretativen Programmiersprache (siehe Punkt 2) der Nutzer (der i.allg. nicht mit dem Entwickler identisch ist)

[1] An dieser Stelle treten die im Compilerbau allgemein bekannten Probleme auf.

über den entsprechenden Interpreter verfügen muß, um das Anwendungsprogramm abarbeiten zu können.

Für Aufgaben, die aus dem Gebiet der KI stammen oder mit Hilfe von KI-Methoden gelöst werden sollen, stehen als 'universelle' Programmiersprachen[1] die funktionale Programmiersprache **LISP** und die logische Programmiersprache **PROLOG** zur Verfügung. Zumindest im universitären Bereich und in Unternehmen, die sich mit KI-Methoden beschäftigen, kann deren Vorhandensein (wenn auch in unterschiedlichen Implementationen) als gegeben vorausgesetzt werden.

Beide Sprachen werden i.allg. interpretativ abgearbeitet und sind im gewissen Sinne deklarativ. Bezüglich ihrer Arbeitsweise und der verwendeten Syntax erscheinen sie jedoch recht unterschiedlich. Während bei LISP, das auf dem λ-Kalkül[2] beruht, die Programme in Form von Funktionsdefinitionen notiert werden, werden bei PROLOG, das auf der Horn-Klausel-Logik[3], einer Teilmenge der Prädikatenlogik 1. Stufe, beruht, die Programme in Form einer Menge von logischen Prädikaten formuliert.

Beispiel: Test, ob ein Element in einer Liste vorkommt

LISP:
```
( defun member ( element liste )
        ( cond (( null liste ) nil )
               (( eq ( car liste ) element ) t )
               ( t ( member element ( cdr list )))))
```

PROLOG:
```
member( Element , [Element|_] ).
member( Element , [_|Rest] ) :- member( Element , Rest ).
```

Den Ausschlag für die Wahl von PROLOG als Implementierungssprache gab vor allem die gute Lesbarkeit der Programme und die leichte Realisierbarkeit von Syntaxprüfern mit den Mitteln von PROLOG. Damit kann die weiter oben geschilderte Diskrepanz zwischen dem zweiten und dem dritten Kriterium für eine Beschreibungssprache zum großen Teil beseitigt werden. Weitere Gründe für die Wahl von PROLOG werden an den Stellen der Arbeit erläutert werden, bei denen die Vorteile von PROLOG für die Lösung bestimmter Teilaufgaben deutlich werden[4].

In den folgenden Unterpunkten werden wir uns mit der Zusammenstellung der zur Aufgabenbeschreibung notwendigen Sprachkonstrukte und mit ihrer Realisierung in PROLOG beschäftigen.

4.1.1.1 Notwendige Sprachkonstrukte

Um die Gesamtheit möglicher Dimensionierungsaufgaben beschreiben zu können, muß geklärt werden, welche Gemeinsamkeiten diese besitzen. Dabei ist von großer Bedeutung, daß die

[1] im Gegensatz zu Programmiersprachen, die für spezielle Aufgabenstellungen entwickelt wurden

[2] Theorie der Konstruktion von Funktionen höherer Ordnung

[3] Eine Horn-Klausel besteht aus der Disjunktion von Atomformeln, wobei maximal eine der Atomformeln nichtnegiert auftreten darf.

[4] Ein eher unbewußter Grund könnte sein, daß in Europa und in Japan im wesentlichen mit PROLOG programmiert, in den USA aber überwiegend LISP verwendet wird (es existieren regelrechte 'Schulen' von KI-Programmierern).

Beschreibung auf einem hohen Abstraktionsniveau erfolgt, um die Menge der benötigten Sprachkonstrukte klein zu halten.

Es erweist sich als vorteilhaft, die Dimensionierung von technischen Systemen als mathematischen Optimierungsprozeß (genauer: als Prozeß der statischen Parameter-Optimierung) darzustellen (siehe Kapitel 2.3). Dadurch wird es möglich, von den konkreten Besonderheiten der verwendeten Modelle abzusehen und die Beschreibung in einer vorwiegend mathematischen Form vorzunehmen. Natürlich werden neben diesen mathematischen Sprachkonstrukten auch eher pragmatisch orientierte Konstrukte notwendig sein, die dem Programmsystem wichtige zusätzliche Informationen liefern (z.B. für den Test der Korrektheit einer konkreten Aufgabenbeschreibung).

Verdeutlichen wir uns an dieser Stelle noch einmal die in Abschnitt 2.3 vorgestellte mathematische Formulierung für Aufgaben der statischen Parameter-Optimierung. Dazu gehört:

- ein Vektor der Parameterwerte $X = \begin{bmatrix} x_1 \\ : \\ x_n \end{bmatrix}$
- eine Zielfunktion F(X)
- eine (evtl. leere) Menge von Restriktionen.

Diese drei Elemente werden wir auch als Sprachkonstrukte der Beschreibungssprache DABS verwenden. Des weiteren müssen Konstrukte zur Definition von Konstanten und Abhängigen sowie zur Gestaltung einer Schnittstelle zum Modellprogramm (siehe Kapitel 4.1), zur Verfügung gestellt werden.

Konstrukte der Beschreibungssprache

Datenobjekte

Datenobjekte dienen zur Strukturierung der Dimensionierungsaufgabe. Sie repräsentieren die uns interessierenden Eigenschaften des zu optimierenden technischen Systems. Die Werte der Datenobjekte sind konkrete Eigenschaftsausprägungen des Systems. Mit ihrer Hilfe lassen sich Konstanten, Parameter und Leistungskenngrößen beschreiben. Im mathematischen Sinne handelt es sich bei den Datenobjekten um Zahlen, Zahlenvektoren bzw. Zahlenmatrizen. Jedes Datenobjekt besitzt einen eindeutigen Namen, der über seine Bedeutung im technischen System Aufschluß geben sollte.

Beispiel: Bedienungsnetz

→ Anzahl Maschinen,
Anzahl Aufträge,
Bearbeitungsintensität,
mittlerer Durchsatz, usw.

Die Definition von Datenobjekten ist ein eher pragmatisch orientiertes Sprachkonstrukt, mit dem die folgenden Vorteile verbunden sind:

- Die Lesbarkeit der Aufgabenbeschreibung wird erhöht, da über den Namen der Datenobjekte eine Verbindung zwischen dem technischen System und der zu lösenden Dimensionierungsaufgabe deutlich wird.

- Der Rechner kann beim Einlesen der Aufgabenbeschreibung überprüfen, ob vom Nutzer wirklich nur die zuvor definierten Datenobjekte verwendet wurden (dadurch z.B. Erkennung von Schreibfehlern).

- Der Rechner erhält Informationen über die Anzahl und Dimension der verwendeten Objekte und kann entsprechende Speicherstrukturen anlegen.

Konstanten

Konstanten sind Datenobjekte oder Teile von ihnen[1], deren Werte vorgegeben sind und die sich während der Optimierung nicht ändern. Durch die Festlegung der Werte von Konstanten wird aus der Beschreibung einer Modellklasse (z.B. Bediennetz mit N Bedienzentren und beliebiger Bearbeitungsintensität) ein konkretes Modell (z.B. Bediennetz mit 3 Bedienzentren und Bearbeitungsintensität 0.5 h^{-1}). In diesem Sinne geben Konstanten konkrete Eigenschaftsausprägungen des technischen Systems bzw. seiner Elemente an.

Man könnte die Meinung vertreten, daß die Konstanten in das Modellprogramm integriert werden könnten, das das Verhalten des zu untersuchenden Systems 'simuliert'. Dagegen sprechen jedoch zwei wesentliche Gründe:

1. Durch explizite Übergabe der Konstanten kann das Modellprogramm so allgemein formuliert werden, daß es das Verhalten einer ganzen Klasse von technischen Systemen beschreibt (z.B. Klasse der Central-Server-Systeme). Somit wird es möglich, das einmal erstellte Modellprogramm für verschiedene Dimensionierungsaufgaben (von Systemen aus dieser Klasse) zu verwenden.

2. Für den Einsatz von wissensbasierten Methoden bei der Dimensionierung ist es von großer Bedeutung, direkten Zugang zu den relevanten Informationen über das Modell zu haben. Wenn die Konstanten nur intern im Modellprogramm gespeichert werden, dann stehen sie als explizites Wissen nicht zur Verfügung.

Parameter

Parameter sind Datenobjekte oder Teile von ihnen, deren Werte während der Optimierung geändert werden können. Um den Parameterraum[2] einzuschränken, werden untere und obere Schranken für die Parameterwerte angegeben, die zumeist aus technischen oder naturwissenschaftlichen Erwägungen stammen. Es stehen sowohl diskrete (d.h. ganzzahlige) als auch kontinuierliche (d.h. reelle) Parameter zur Verfügung. Das Ziel der Optimierung ist es, solche Parameterwerte zu finden, für die der Zielfunktionswert ein Optimum erreicht und alle Restriktionen (einschließlich Definitionsbereich der Parameter) erfüllt sind.

[1] Die einzelnen Elemente eines Datenobjektes können verschiedenen Objektarten (Parameter, Konstanten, ..) angehören.

[2] Menge der möglichen Parameterbelegungen

Die Gesamtheit der bei einer konkreten Aufgabenstellung auftretenden Parameter entspricht dem Parametervektor X der mathematischen Optimierung.

Abhängige

Abhängige sind Datenobjekte oder Teile von ihnen, deren Werte sich eindeutig aus den Werten anderer Datenobjekte berechnen lassen. Wenn im zu dimensionierenden Modell bestimmte Parameter direkt von anderen Parametern abhängig sind und eine explizite mathematische Berechnungsvorschrift existiert, dann kann man diese als Abhängige darstellen. Sie hören damit intern auf, Parameter zu sein, und die Dimension des Optimierungsproblemes reduziert sich entsprechend.

Zielfunktion

Die **Zielfunktion** gibt das eigentliche Ziel der Optimierung an. Sie besteht aus der Berechnungsformel für den Zielfunktionswert und der Art der Optimierung (Minimierung oder Maximierung). Die in der Berechnungsformel auftretenden Datenobjekte sind i.allg. Leistungskenngrößen[1] des technischen Systems, deren Werte durch das Modellprogramm ermittelt werden. Die Berechnung der Zielfunktion erfolgt in zwei Schritten:

1. Bestimmung der Leistungskenngrößen des zu untersuchenden Systems durch Verwendung des ausführbaren Modellprogrammes

2. Berechnung der Zielfunktion unter Einbeziehung der ermittelten Leistungskenngrößen und evtl. weiterer Datenobjekte (Konstanten, Parameter, Abhängige).

Die Berechnungsformel für die Zielfunktion entspricht der Zielfunktion F(X) der mathematischen Optimierung.

Restriktionen

Restriktionen sind arithmetische Gleichungen bzw. Ungleichungen, die alle erfüllt sein müssen, damit eine Parameterbelegung als gültig angesehen werden kann. Sie spiegeln i.allg. gesetzliche, finanzielle, technische bzw. naturwissenschaftliche Gegebenheiten wider. Durch die Menge der Restriktionen wird der Parameterraum i.allg. verkleinert, es kann sogar vorkommen, daß der Parameterraum in mehrere, nicht mehr zusammenhängende Teilräume aufgeteilt wird.

Während bei der mathematischen Optimierung meistens nur Restriktionen der Form $g_i(X) \geq 0$ verwendet werden, wurde in DABS der Begriff 'Restriktion' auf alle arithmetischen Vergleiche ausgedehnt. Das erfordert jedoch eine zusätzliche rechnerinterne Bearbeitung der Restriktionen

Schnittstelle

Zur Kommunikation mit dem ausführbaren Modellprogramm muß eine **Schnittstelle** definiert werden. Sie gibt den Namen des Programmes sowie seine Eingabe- und Ausgabegrößen (Datenobjekte) an.

[1] Finanzielle Kosten für das System können im weiteren Sinne auch als Leistungskenngrößen aufgefaßt werden.

Für dieses Konstrukt gibt es bei der Formulierung einer mathematischen Optimierungsaufgabe kein Äquivalent. Das liegt daran, daß die Art, wie die Zielfunktionswerte ermittelt werden, bei der mathematischen Optimierung völlig offengelassen wird (z.B. Berechnung 'per Hand', Experiment, Rechenstab, Computer).

Meiner Auffassung nach genügen die oben beschriebenen Sprachkonstrukte, um beliebige Dimensionierungsaufgaben formulieren zu können.

4.1.1.2 Realisierung in DABS

Wie bereits weiter oben erwähnt, erweist es sich als günstig, die Syntax der Sprache DABS als Teilmenge der Syntax der Implementierungssprache (also PROLOG) zu definieren. Sämtliche Sprachkonstrukte der Beschreibungssprache werden deshalb in Form von speziellen PROLOG-Termen notiert, wobei sich folgende Vorteile ergeben:

- PROLOG bietet Prädikate zum Einlesen von Termen[1]. Dadurch kann auf ein zeichenweises Einlesen der Aufgabenbeschreibung verzichtet werden. Bei jedem Lesevorgang kann ein vollständiges Element der Aufgabenbeschreibung gelesen werden (insbesondere werden Leerzeichen, Tabulatoren, Kommentare usw. automatisch überlesen).

- Durch die Möglichkeit der Definition neuer Operatoren in PROLOG kann das Aussehen von PROLOG-Termen in weiten Grenzen verändert werden. Dadurch kann man ein hohes Maß an Lesbarkeit der Aufgabenbeschreibung erreichen.

- Der Unifikationsmechanismus, den PROLOG zur Verfügung stellt, ermöglicht es in Verbindung mit der Definition neuer Operatoren sehr leicht, PROLOG-Terme in ihre syntaktischen Bestandteile zu zerlegen. Damit wird ein großer Teil der Arbeit, die sonst ein spezieller Syntaxprüfer ausführen müßte, durch das PROLOG-System erledigt.

Um die Beschreibungselemente besser strukturieren zu können, werden mehrere neue Infix-Operatoren definiert. Diese haben keine arithmetische Bedeutung (wie z.B. '+', '*' usw.), sondern dienen nur zur besseren Strukturierung der Modellbeschreibung. Die wichtigsten neuen Operatoren sind '::', ':' und '..' .

Der Operator '::'

Dieser Operator wird zur Trennung von syntaktischen Elementen innerhalb eines PROLOG-Termes benutzt.

Beispiel: ankunftsintensitaet :: vektor(5).

(Definition eines Datenobjektes mit dem Namen 'ankunftsintensitaet' als Vektor mit 5 Elementen)

[1] Dazu gehören vor allem die Prädikate read/1, read/2, read_term/2 und read_term/3 (SNI-Prolog).

Der Operator ':'

Dieser Operator wird zum Zugriff auf die Elemente eines Vektor- bzw. Matrix-Datenobjektes benutzt.

Beispiel: ankunftsintensitaet:4

(Zugriff auf 4. Element des Vektor-Datenobjektes 'ankunftsintensitaet')

kosten:3:5

(Zugriff auf das 5. Element der 3. Zeile des Matrix-Datenobjektes 'kosten')

Der Operator '..'

Dieser Vektor wird zur Angabe von Elementgruppen innerhalb von Vektor- bzw. Matrix-Datenobjekten benutzt.

Beispiel: ankunftsintensitaet:2..4

(Angabe einer Elementgruppe, die aus dem 2. bis 4. Element des Vektor-Datenobjektes 'ankunftsintensitaet' besteht)

kosten:3:1..3

(Angabe einer Elementgruppe, die aus dem 1. bis 3. Element der 3. Zeile des Matrix-Datenobjektes 'kosten' besteht)

Im folgenden wird die Syntax der im Abschnitt 4.1.1.1 zusammengestellten Konstrukte der Beschreibungssprache DABS unter Benutzung der oben beschriebenen Operatoren erläutert. Dabei wird als Beispiel das Modell eines Central-Servers benutzt, auf das im Kapitel 6 nochmals genauer eingegangen wird. Eine exakte Beschreibung der Syntax von DABS finden Sie zusätzlich im Anhang dieser Arbeit.

Grundlegende Bemerkungen

Da wir uns entschlossen haben, die Sprachkonstrukte in Form von PROLOG-Termen darzustellen, müssen wir auch einige Konventionen beachten, die deren Syntax vorschreibt.

- Alle Namen von Datenobjekten und anderen Bezeichnern müssen klein geschrieben werden.
- Aufzählungen von Zahlenwerten bzw. Datenobjekten sind in eckige Klammern einzuschließen und durch Komma zu trennen (Listendarstellung in PROLOG).
- Jeder PROLOG-Term (d.h. jedes Konstrukt) ist mit einem Punkt abzuschließen.
- In PROLOG-Termen dürfen keine Umlaute benutzt werden.

Wenn wir uns konsequent an diese vier Forderungen halten, dann umgehen wir eine wesentliche Quelle von Syntaxfehlern, deren Lokalisation ansonsten nur schwer möglich ist.

1. Deklaration der Datenobjekte

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **datenobjekte** eingeleitet. Wie bereits erwähnt müssen alle Datenobjekte, die zur Problembeschreibung benötigt werden, dem Programmsystem zusammen mit ihrer Dimension bekanntgegeben werden. Als Dimensionsangaben stehen **skalar**, **vektor**(N) und **matrix**(Z,Sp) zur Verfügung, wobei es sich bei N, Z bzw. Sp um natürliche Zahlen größer Null handeln muß[1].

Beispiel: datenobjekte.

```
anz_dev    :: skalar.
bearb_int  :: vektor(4).
job_vert   :: vektor(4).
w_zeit     :: vektor(4).
```

Durch die Deklaration wird für jedes Datenobjekt eine physische Struktur im Speicher angelegt, die Informationen über dieses Objekt (z.B. Name, Dimension, Wert) enthält. Mit Datenobjekten, die nicht deklariert wurden, kann deshalb nicht gearbeitet werden.

2. Definition der Konstanten

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **konstanten** eingeleitet. Neben dem Datenobjekt oder Teil davon, das als Konstante definiert werden soll, muß auch eine Liste mit den Werten dieser Konstanten angegeben werden. Diese Liste muß genauso viele Zahlen enthalten, wie das angegebene Datenobjekt oder Teil davon Elemente besitzt.

Beispiel: konstanten.

```
anz_dev        :: [4].
bearb_int:2..4 :: [0.3, 0.1, 0.2].
job_vert:1     :: [0.0].
```

3. Definition der Parameter

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **parameter** eingeleitet. Neben dem Datenobjekt oder Teil davon, das als Parameter definiert werden soll, müssen auch spezielle Angaben über die Parameter der Form **bereich**(UG,OG,Art) gemacht werden. Dabei sind UG und OG Zahlenwerte, die die untere bzw. obere Schranke des Parameters angeben, während Art die Werte **integer** (für ganzzahlige Parameter) und **real** (für reelle Parameter) annehmen kann. Zusätzlich muß eine Liste mit den Startwerten dieser Parameter angegeben werden, deren Aufbau analog der Konstantenliste ist. Die Startwerte geben an, mit welcher Parameterbelegung die Optimierung beginnen soll. Sie müssen eine gültige Parameterbelegung (bzgl. der Restriktionen) beschreiben.

[1] Die obere Grenze dieser Zahlen wird i.a. durch das verwendete PROLOG-System im Zusammenhang mit dem vorhandenen Speicherplatz bestimmt.

Beispiel: parameter.

```
bearb_int:1    ::  bereich(0.1,0.6,real)  ::  [0.2].
job_vert:2..4  ::  bereich(0.0,1.0,real)  ::  [0.3, 0.2, 0.5].
```

4. Definition von Abhängigen

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **abhaengige** eingeleitet. Für das Datenobjekt oder Teil davon, das als Abhängige definiert werden soll, muß ein berechenbarer arithmetischer Ausdruck angegeben werden, mit dessen Hilfe sich deren Wert berechnen läßt. Die Syntax eines solchen Ausdruckes kann Punkt 5 entnommen werden.

Beispiel: abhaengige.

```
durchsatz:1 = auslastung:1 * bearb_int:1.
durchsatz:2 = durchsatz:1 * job_vert:2.
```

Wenn während der Dimensionierung des Modelles der Wert einer Abhängigen benötigt wird, dann wird dieser Wert mittels des angegebenen Ausdruckes berechnet.

5. Definition der Zielfunktion

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **zielfunktion** eingeleitet. Die Zielfunktion besteht aus der Formel zur Berechnung des Zielfunktionswertes und der Art des zu suchenden optimalen Zielfunktionswertes (**maximum** oder **minimum**).

Die Formel ist ein berechenbarer arithmetischer Ausdruck. Dieser Ausdruck kann aus Zahlen und Datenobjekten bzw. Elementen von Datenobjekten, vordefinierten Funktionen und arithmetischen Operatoren bestehen. Ihr allgemeiner Aufbau entspricht (bis auf wenige Ausnahmen) der Notation von arithmetischen Ausdrücken in imperativen Programmiersprachen wie PASCAL und C.

Funktionen:

round(Ausdruck)	berechnet den Ausdruck und liefert den auf die nächste ganze Zahl gerundeten Wert zurück
trunc(Ausdruck)	berechnet den Ausdruck und liefert den auf die nächstkleinere ganze Zahl gerundeten Wert zurück
abs(Ausdruck)	berechnet den Ausdruck und liefert den Betrag des Wertes zurück
exp(Ausdruck)	berechnet den Ausdruck und liefert e^{Wert} zurück
log(Ausdruck)	berechnet den Ausdruck und liefert ln(Wert) zurück
sqrt(Ausdruck)	berechnet den Ausdruck und liefert die Quadratwurzel des Wertes zurück

min(Ausdruck1**,**Ausdruck2**)**	berechnet die beiden Ausdrücke und liefert den kleineren Wert zurück
max(Ausdruck1**,**Ausdruck2**)**	berechnet die beiden Ausdrücke und liefert den größeren Wert zurück
sum(I=UG**..**OG**,**Ausdruck**)**	entspricht einer Laufanweisung mit der Laufvariablen I von UG bis OG, wobei die Werte des Ausdrucks berechnet und addiert werden
prod(I=UG**..**OG**,**Ausdruck**)**	entspricht einer Laufanweisung mit der Laufvariablen I von UG bis OG, wobei die Werte des Ausdrucks berechnet und multipliziert werden

Operatoren:

+ , -	Vorzeichenoperatoren (einstellig)
+ , - , * , /	wie bekannt
div , //	ganzzahlige Division
mod	Rest der ganzzahligen Division
**	Potenz (2**5 = 25 = 32)

Beispiel: zielfunktion.

(sum(i=1..4,w_zeit:i) / 4) :: minimum.

Das Angebot an vordefinierten Funktionen und arithmetischen Operatoren ist vom verwendeten PROLOG-System abhängig. Bei einfachen Systemen stehen oft nur die Grundrechenarten zur Verfügung, während bei komplexen Systemen der Leistungsumfang an den von numerisch orientierten Programmiersprachen heranreicht (Potenz, Wurzel, trigonometrische Funktionen usw.). Wenn die benötigten Funktionen und Operatoren nicht vom System zur Verfügung gestellt werden, dann müssen sie entweder 'von Hand' programmiert werden oder man läßt sie weg, wodurch die arithmetische Leistungsfähigkeit jedoch eingeschränkt wird.

ACHTUNG: Natürlich darf für eine konkrete Aufgabenbeschreibung nur genau eine Zielfunktion definiert werden !

6. Definition der Restriktionen

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **nebenbedingungen** eingeleitet. Restriktionen sind arithmetische Vergleiche, die entweder wahr oder falsch sind. Damit eine Parameterbelegung als gültig angesehen werden kann, müssen alle Restriktionen wahr sein. Restriktionen bestehen aus zwei arithmetischen Ausdrücken, wie sie im Punkt 5 beschrieben wurden, die durch einen Vergleichsoperator verknüpft sind.

Vergleichsoperatoren: < , <= , = , >= , > , <> (ungleich)

Beispiel: nebenbedingungen.

```
( sum(i=1..4,job_vert:i) = 1.0 ).
( bearb_int:1 > max(bearb_int:2,bearb_int:3) ).
```

Das Programm muß die angegebenen Restriktionen auf ihre Korrektheit überprüfen. Gleichungs-Restriktionen, die Parameter enthalten, müssen so umgeformt werden, daß ein Parameter zur Abhängigen wird und sich die Dimension des Problemes verringert.

7. Definition der Schnittstelle

Dieser Teil der Beschreibung wird mit dem Schlüsselwort **schnittstelle** eingeleitet. Die Schnittstellendefinition besteht aus dem Namen des Modellprogrammes, das die Leistungskenngrößen des Systemes bestimmt, aus der Liste der Datenobjekte, die dem Programm übergeben werden (Konstanten, Parameter, Abhängige) und der Liste der Datenobjekte, die das Programm zurückliefert (Ergebnisse/Leistungskenngrößen).

Beispiel: schnittstelle.

```
c_server :: [anz_dev, bearb_int, job_vert] :: [w_zeit].
```

ACHTUNG: Natürlich darf für eine konkrete Aufgabenbeschreibung nur genau eine Schnittstelle definiert werden !

Wenn die obengenannten Schritte ausgeführt wurden (d.h. wenn mittels eines Texteditors eine Datei mit der entsprechenden Beschreibung erzeugt wurde), dann ist die Formulierung der Dimensionierungsaufgabe abgeschlossen. Der entsprechende Dateiname muß mit der Extension '.dab' versehen werden (z.B. c_server.dab).

4.1.2 Die Modellierung des Systemverhaltens

In diesem Abschnitt wollen wir uns damit beschäftigen, auf welche Weise die Leistungskenngrößen bzw. Gütekriterien für eine konkrete Parameterbelegung des zu untersuchenden technischen Systems ermittelt werden können. Dafür gibt es drei Möglichkeiten, von denen aber nur die letzten beiden von praktischer Bedeutung für uns sind:

1. Messungen am konkreten technischen System

2. Analytische Berechnung anhand eines Modelles

3. Simulative Ermittlung anhand eines Modelles

Die **erste Methode** ist deshalb nicht praktikabel, weil die Messungen am konkreten System im weitesten Sinne zu kostspielig sind oder das System noch gar nicht existiert (bei Entwurfsaufgaben eigentlich der Normalfall).

Die **zweite Methode** ist nur dann anwendbar, wenn es für die Leistungskenngrößen entsprechende analytische Berechnungsvorschriften gibt. Das ist zum heutigen Zeitpunkt nur für

wenige und i.allg. einfache Klassen von Problemen der Fall. Existieren entsprechende Formeln, so sollte unbedingt analytisch gearbeitet werden, da Simulationen meist eine weit größere Zeit benötigen, um signifikante Ergebnisse zu liefern.

Die **dritte Methode** ist die in der Praxis am meisten benutzte zur Ermittlung von Leistungskenngrößen. Sie erfordert den Entwurf und die Implementation eines Simulationsexperimentes. Im Prinzip lassen sich mit ihr auch hochkomplexe Modelle untersuchen, was jedoch geeignete Simulationswerkzeuge (spezielle Simulationssprachen bzw. -systeme) erfordert. Oft existieren bereits Simulationsprogramme für die betrachtete Problemklasse, die dann nur noch angepaßt werden müssen.

Für die Ermittlung der Leistungskenngrößen nach den Methoden 2 und 3 gäbe es in Bezug auf das in dieser Arbeit beschriebene Programmsystem zwei mögliche Herangehensweisen:

Erstens. Das Berechnungs- bzw. Simulationsprogramm wird in PROLOG geschrieben. Das hat den Vorteil, daß sich das gesamte Programmsystem durch eine große Homogenität in der Implementierung und Abarbeitung auszeichnet. Im Sinne der wissensbasierten Programmierung könnte man das Wissen über die Ermittlung der Ergebnisse nutzen, um einen tieferen Einblick in die Natur des Modelles zu erreichen[1]. Diesen Vorteilen steht jedoch eine große Anzahl von Nachteilen gegenüber. Die meisten Nutzer, die sich mit der Dimensionierung von technischen Systemen beschäftigen, beherrschen die Programmiersprache PROLOG nicht, sondern eher imperative Programmiersprachen wie FORTRAN, PASCAL oder C. Die typischen Algorithmen, die in Berechnungs- und Simulationsprogrammen verwendet werden, lassen sich in PROLOG nur schwer oder gar nicht implementieren. Sollte man es doch geschafft haben, ein entsprechendes PROLOG-Programm zu entwickeln, dann wird man feststellen, daß dessen Laufzeit ein Mehrfaches über der vergleichbarer imperativer Programme liegt.

Zweitens. Das Berechnungs- bzw. Simulationsprogramm wird in einer der 'gebräuchlichen' imperativen Programmiersprachen bzw. in einer speziellen Simulationssprache geschrieben. Das hat den Vorteil, daß eine aus einer großen Anzahl von Programmiersprachen verwendet werden kann, wobei jeder Nutzer die Programmiersprache wählt, die ihm zur Verfügung steht bzw. die er am besten beherrscht. Die 'konventionellen' Programmiersprachen sind bei numerischen Berechnungen meist bedeutend schneller als vergleichbare PROLOG-Systeme. Außerdem kann man, wie weiter oben bereits kurz erwähnt, schon existierende Simulationsprogramme nutzen, wobei oft nur wenige Modifikationen notwendig sind. Der Nachteil dieser Methode liegt im größeren Aufwand für den Datenaustausch zwischen dem PROLOG-Programmsystem und dem Berechnungs- bzw. Simulationsprogramm.

Wie aus den geschilderten Vor- und Nachteilen der beiden Herangehensweisen leicht nachvollziehbar, habe ich mich bei dem in dieser Arbeit vorgestellten Programmsystem für die zweite Methode, d.h. für die Implementierung der Berechnungs- bzw. Simulationsprogramme in einer konventionellen Programmiersprache entschieden.

Im folgenden möchte ich die allgemeine Struktur eines solchen Modellprogrammes beschreiben. Das Programm besteht aus den drei allgemeinen Komponenten

1. Lesen der Eingabe-Datenobjekte (Konstanten + Parameter)

2. Ermittlung der Ausgabe-Datenobjekte (Leistungskenngrößen)

[1] So arbeiten modellbasierte Expertensysteme wie z.B. TEX-B.

3. Speichern der Ausgabe-Datenobjekte.

Die Implementierung der zweiten Komponente unterscheidet sich nicht vom Vorgehen bei der allgemeinen Modellierung von Systemen, das als allgemein bekannt vorausgesetzt wird (für weitere Informationen siehe z.B. [Ber89] bzw. [Sch90]).

Die erste und dritte Komponente, die der Kommunikation dienen, müssen etwa genauer erklärt werden. Ihr Aufbau hängt vor allem von der Art der kommunikativen Verbindung des Dimensionierungsprogrammes (wissensbasiert) mit dem Modellprogramm (numerisch) ab (siehe Kapitel 3.5). Da das zu erstellende Programmsystem eher prototypischen Charakter tragen soll, erscheint die Durchführung der Kommunikation durch Dateitransfer am günstigsten[1].

Bei der Realisierung der Dateitransfer-Schnittstelle wurde auf eine große Einfachheit des Konzeptes geachtet. Deshalb erfolgt die Kommunikation über ASCII-Dateien, in die die entsprechenden Informationen (Zahlen) eingetragen werden. Dadurch erreichen wir eine gute Testbarkeit des Kommunikationsvorganges (Informationen sind für Menschen direkt lesbar!).

Der Vorgang der Ermittlung der Leistungskenngrößen eines Modelles für eine konkrete Parameterbelegung erfolgt in sechs Schritten.

1. Die Datenobjekte, die bei der Definition der Schnittstelle als Eingabeobjekte angegeben wurden, werden nacheinander elementweise (genau eine Zahl in einer Zeile) in die Eingabedatei <modelldatei-name>.in eingetragen.

2. Die Modelldatei wird aus dem Dimensionierungsprogramm heraus durch einen Betriebssystemruf gestartet.

3. Das Modellprogramm liest die Eingabewerte aus der ASCII-Eingabedatei <modelldatei-name>.in .

4. Das Modellprogramm ermittelt durch Modellierung des Verhaltens des zu untersuchenden technischen Systems (analytisch bzw. simulativ) die den Eingabewerten entsprechenden Ergebnisse (Leistungskenngrößen).

5. Das Modellprogramm schreibt die Ergebnisse nacheinander elementweise (genau eine Zahl in einer Zeile) in die ASCII-Ausgabedatei <modelldatei-name>.out und endet (die Steuerung wird an das rufende Dimensionierungsprogramm zurückgegeben).

6. Das Dimensionierungsprogramm liest die Ergebnisse aus der Ausgabedatei <modelldatei-name>.out und beginnt mit der Berechnung des Zielfunktionswertes.

4.2 Dimensionierung

Nachdem wir uns in den vorangegangenen Abschnitten mit der formalen Beschreibung von Dimensionierungsmodellen (Dimensionierungsaufgabe + Modelldatei) befaßt haben, wollen wir uns in diesem Kapitel mit der eigentlichen Durchführung der Dimensionierung beschäftigen.

[1] Das verwendete SNI-PROLOG besitzt auch Prädikate zur Realisierung von Interprozeß-Kommunikationsmechanismen, jedoch erscheint mir die daraus resultierende Festlegung auf Rechner mit multitaskingfähigem Betriebssystem als zu große Einschränkung.

Dabei setze ich die in Kapitel 2.1 geschilderte Vorgehensweise eines Entwurfsingenieurs bei der Dimensionierung als Grundlage voraus.

Das globale Ziel dieser Arbeit soll ein Programmsystem sein, das den Vorgang der Dimensionierung technischer Systeme durch Verwendung wissensbasierter Methoden automatisiert. Dabei soll das Wissen des Entwurfsingenieurs über die Zusammenhänge zwischen Parameteränderung und Zielfunktionsänderung in Form einer Menge von Regeln dargestellt werden. Um den Erfolg der Automatisierung beurteilen zu können, müßten eigentlich die Resultate der wissensbasierten Methode und eines Entwurfsingenieurs verglichen und beurteilt werden. Da jedoch bei dieser Arbeit ohne die Hilfe eines entsprechenden Fachmanns ausgekommen werden mußte, wurde ein anderer Weg der Ergebnisbeurteilung gewählt.

Da, wie bereits mehrmals erwähnt, eine enge Beziehung zwischen Dimensionierung und mathematischer Optimierung besteht, wurde ein numerisches Optimierungsverfahren implementiert, das die Dimensionierung durchführt und uns Werte für den optimalen Zielfunktionswert und die optimale Parameterbelegung liefert. Im Vergleich dieser Ergebnisse mit denen der wissensbasierten Methode können Rückschlüsse auf die Güte der verwendeten Regelmenge und des benutzten Verfahrens gezogen werden. In diesem Sinne dient die numerische Optimierung als 'Benchmark'.

In den folgenden Abschnitten werden wir uns mit dem verwendeten mathematischen Optimierungsverfahren zur Dimensionierung und ausführlich mit der wissensbasierten Dimensionierung beschäftigen.

4.2.1 Mathematische Dimensionierung

Mit dem Begriff **mathematische Dimensionierung** wollen wir solche Verfahren bezeichnen, die bei der Dimensionierung eines Modelles nicht nach im weitesten Sinne 'menschlichen' Strategien (z.B. Engpaßbeseitigung) vorgehen, sondern allein die mathematische Struktur der Aufgabe (Finden des optimalen Punktes einer Fläche im (n+1)-dimensionalen Raum) zur Lösungsfindung verwenden. Sie haben den Vorteil, daß durch die Arbeit auf einer sehr hohen Abstraktionsebene eine große Zahl von Modellklassen behandelt werden kann. Trotzdem ergeben sich durch die Art und Menge der benötigten Informationen zur Problemlösung jedoch mehr oder weniger große Einschränkungen des bearbeitbaren Aufgabenspektrums. Als Faustregel könnte formuliert werden:

> Je mehr spezielle Informationen ein Dimensionierungsverfahren zur Lösungsfindung benötigt, desto kleiner wird die Menge der bearbeitbaren Aufgabenstellungen (und umgekehrt).

Da uns bei den behandelten Aufgabenstellungen zwar die Zielfunktion symbolisch zur Verfügung steht, sich diese i.allg. aber aus Daten zusammensetzt, die durch das Modellprogramm ermittelt wurden (z.B. durch Simulation), stehen uns keine Ableitungen zur Verfügung. Deshalb kamen nur Verfahren in die engere Wahl, die als Information nur Zielfunktionswerte benötigen. Sie haben auch den Vorteil, daß sie eine einfachere algorithmische Struktur als 'besserinformierte' Verfahren besitzen.

Nach einigen unbefriedigenden Versuchen mit der Methode der Koordinatenrichtungen und der Methode der konjugierten Richtungen (die insbesondere Schwierigkeiten bei der Verwendung

von Restriktionen zeigten), entschloß ich mich, ein **stochastisches Verfahren** zu benutzen (siehe [Whi70]). Dabei spielten zwei Gründe eine Rolle:

- sie sind leicht zu programmieren
- sie verhalten sich auch in unregelmäßigen Parameter-'Landschaften' effektiv

Die allgemeine Vorgehensweise bei stochastischen Optimierungsverfahren besteht darin, in einem Schritt eine Anzahl von Probepunkten nach einer bestimmten Wahrscheinlichkeitsverteilung im Parameterraum (oder bestimmten Teilräumen davon) auszuwählen und deren Funktionswerte zu ermitteln. Je nach Abbruchkriterium wird entweder der beste gefundene Probepunkt als Approximation des optimalen Punktes angenommen oder der Schritt wird wiederholt, wobei sich die Anzahl und Verteilung der Probepunkte und der betrachtete Raum ändern können (dazu werden die Informationen, die der letzte Schritt geliefert hat, ausgewertet).

Das von mir gewählte Verfahren trägt in der englischsprachigen Literatur den Namen **Creeping Random Search**. Da sich dieser Name nur sehr ungenau ins Deutsche übertragen läßt (etwa: 'Kriechende/Schleichende Zufallssuche'), werde ich im weiteren die Abkürzung **CRS-Methode** benutzen.

Bei der CRS-Methode werden sukzessive Probepunkte X_i in einer gewissen Umgebung eines bisher besten Probepunktes X_{best} ausgewählt und deren Funktionswerte $F(X_i)$ bestimmt. Findet man einen Probepunkt X_j, dessen Funktionswert besser als der von X_{best} ist, dann wird X_j zu X_{best} und der Vorgang wiederholt sich. Beendet wird das Verfahren, wenn mehr als eine bestimmte Zahl von Probepunkten geprüft wurden, ohne eine Verbesserung zu erzielen. Der Punkt X_{best} wird dann als Approximation des optimalen Punktes angenommen. Eine graphische Darstellung dieses Verfahrens bietet Abbildung 4.1.

Die konkrete Implementation der CRS-Methode enthält einige Besonderheiten. Ist eine Aufgabe mit n Parametern zu optimieren, dann wird in einer Umgebung von X_{best}, die einen n-dimensionalen quaderförmigen Teilraum bildet, nach besseren Punkten gesucht. Die Kantenlängen des Quaders entsprechen etwa 10% des Bereiches, den der zugehörige Parameter durchlaufen kann (begrenzt durch untere und obere Schranke). Innerhalb dieses Teilraumes werden die Probepunkte gleichverteilt ausgewählt. Das Verfahren wird nach einer bestimmten Anzahl vergeblicher Versuche abgebrochen, die linear zu n ist[1].

[1] Selbst durch White, der in [Whi70] eine Übersicht der verschiedenen stochastischen Verfahren zur Parameteroptimierung bietet, werden keine konkreten Angaben über die Dimensionierung des Abbruchkriteriums gemacht.

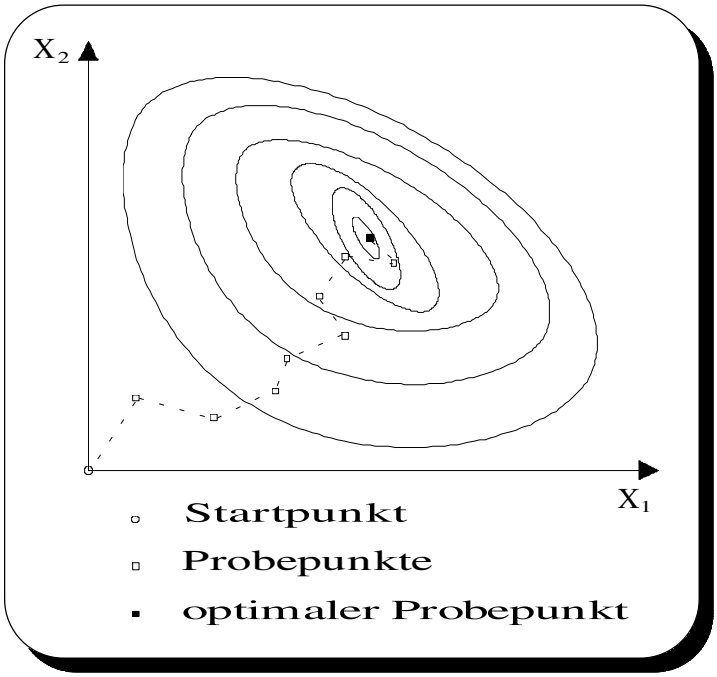

Abbildung 4.1: Graphische Darstellung der CRS-Methode

Neben den bereits weiter oben genannten allgemeinen Vorteilen stochastischer Optimierungsverfahren besitzt die CRS-Methode weitere Vorzüge, die sich auch in der Testphase bestätigten.

- Die Behandlung der in der Aufgabenstellung gegebenen Restriktionen ist sehr leicht möglich. Dabei wird jeder Probepunkt (d.h. jede Parameterbelegung), der erzeugt wurde, auf seine Gültigkeit bzgl. der Restriktionen getestet. Ist er nicht gültig, d.h. ist eine Restriktion verletzt, dann wird dieser Probepunkt nicht verwendet, sondern ein neuer erzeugt. Hier zeigt sich der Vorteil gegenüber Verfahren, die die n-dimensionale Optimierung durch eine sukzessive Folge eindimensionaler Optimierungen (auf einer Geraden im Parameterraum) realisieren. Sie müssen bei Verletzungen der Restriktionen spezielle Methoden anwenden, um die Optimierung korrekt fortsetzen zu können.

- Die Lösung 'gemischter' Optimierungsaufgaben ist ohne Probleme möglich. Dabei wollen wir unter einer **gemischten Optimierungsaufgabe** eine solche Aufgabe verstehen, die sowohl diskrete als auch kontinuierliche Parameter besitzt. Da die Gleichverteilung der Probepunkte durch gleichverteilte Generierung der einzelnen Parameter in der Umgebung von X_{best} erfolgt, muß bei diskreten Parametern nur darauf geachtet werden, daß nur ganzzahlige gleichverteilte Werte generiert werden. Im Gegensatz dazu kann es bei den im obigen Punkt beschriebenen Verfahren, die auf einer eindimensionale Optimierung entlang einer Geraden beruhen, bei gemischten Optimierungsaufgaben zu großen Schwierigkeiten kommen. Das liegt daran, daß auf einer nicht parallel zu einer Koordinatenachse verlaufenden Geraden im Parameterraum nur sehr wenige Punkte liegen, bei denen die diskreten Parameter wirklich ganzzahlige Werte besitzen. Deshalb wird die Ganzzahligkeitsforderung i.allg. während der Optimierung verletzt und erst nach dem Erreichen des Abbruchkriteriums durch Runden der entsprechenden Parameterwerte des ermittelten optimalen Punktes wieder erfüllt. Das kann zu größeren Abweichungen des Ergebnisses führen.

4.2.2 Wissensbasierte Dimensionierung

Mit dem Begriff **wissensbasierte Dimensionierung** wollen wir solche Verfahren bezeichnen, die zur Dimensionierung von Modellen explizites und i.allg. heuristisches Wissen über diese Modelle und insbesondere Wissen über die Beziehung

Parameteränderung $\rightarrow$ Zielfunktionsänderung

verwenden. Diese Verfahren versuchen in gewissem Sinne, das Vorgehen eines menschlichen Experten bei der Dimensionierung nachzubilden. Sie werden durch wissensbasierte Systeme realisiert.

Wie bereits im Kapitel 3.3 erwähnt, entscheidet die Wahl einer geeigneten Wissensrepräsentation in großem Maße über die Leistungsfähigkeit eines wissensbasierten Systems. Da die oben genannte Beziehung zwischen der Änderung von Parametern und der Änderung des Zielfunktionswertes in gewisser Weise bereits eine WENN-DANN-Struktur impliziert, habe ich mich dafür entschieden, als Wissensrepräsentationsform die **Regel** zu verwenden.

Daraus ergeben sich zwei notwendige Entwurfsentscheidungen:

1. Art der Regeldarstellung (Syntax der Regeln)

2. Art der Regelverarbeitung (Semantik der Regeln)

Mit der Wahl geeigneter Regeldarstellungs- und -verarbeitungsmechanismen und ihrer Realisierung im vorzustellenden Programmsystem werden wir uns in den nächsten Abschnitten beschäftigen.

4.2.2.1 Regeldarstellung

Wie für die Beschreibung von Dimensionierungsaufgaben wurde von mir auch für die Darstellung von Regeln eine spezielle Sprache entworfen. Diese Sprache trägt den Namen **RDS** (Regeldarstellungs-Sprache) und wurde wie DABS als Teilmenge der Sprache PROLOG definiert. Die daraus resultierenden syntaktischen Besonderheiten wurden bereits in Kapitel 4.1.1.2 (Realisierung in DABS) erläutert.

Konstrukte der Regeldarstellung

Um die Sprachkonstrukte definieren zu können, die zur Darstellung von Regeln (und besonders von Dimensionierungsregeln) benötigt werden, wollen wir uns noch einmal die bereits in Kapitel 3.3.1 (Regeln) beschriebene allgemeine Regelform

WENN Bedingung(en) **DANN** Schlußfolgerung(en)

verdeutlichen. Für die von uns zu beschreibenden Regeln zur Dimensionierung kann diese Regelform folgendermaßen abgewandelt werden:

WENN Situation **DANN** Aktion(en)

Der **Situationsteil** enthält eine oder mehrere konjunktiv verknüpfte Bedingungen, die eine gewisse Situation beschreiben. Nur wenn der Situationsteil einer Regel wahr ist (die geforderte Situation liegt vor), kann diese Regel angewendet werden. In diesem Fall wird der **Aktionsteil** der Regel, genauer gesagt die in ihm enthaltenen Aktionen, ausgeführt.

Im Zusammenhang mit der Dimensionierung erhalten diese beiden Teile folgende spezielle Bedeutung:

- Der **Situationsteil** beschreibt einen während der Dimensionierung auftretenden Entwurfszustand (allg. Modell + akt. Parameterbelegung + akt. Leistungskenngrößen). Nur wenn während einer konkreten Dimensionierung genau dieser Entwurfszustand eintritt, kann die entsprechende Regel angewendet werden.

- Der **Aktionsteil** beschreibt eine Handlungsfolge, die durchzuführen ist, um einen (wahrscheinlich) besseren Entwurfszustand (charakterisiert durch den Zielfunktionswert) zu erreichen. Die wesentliche Aktion ist dabei die Wertänderung von Parametern.

Insgesamt betrachtet repräsentiert eine Regel dieser Form ein Stück (heuristisches) Wissen, das der Entwurfsingenieur benutzt, um ein bestimmtes technisches System zu dimensionieren.

Aus diesen Betrachtungen ergibt sich die Notwendigkeit der Bereitstellung von folgenden wesentlichen Sprachkonstrukten:

- Konstrukt zur Benennung von Regeln

- Konstrukte des Situationsteiles

  - Zielfunktion und Optimalitätskriterium bestimmen
  - Eigenschaften von Datenobjekten ermitteln
  - Werte von Datenobjekten ermitteln
  - spezielle Datenobjekte finden
  - Summenbildung
  - arithmetische Vergleiche
  - lokalen Variablen Werte zuweisen
  - Werte von globalen Variablen ermitteln

- Konstrukte des Aktionsteiles

  - Parametern neue Werte zuweisen
  - Bestimmen der Leistungskenngrößen und des Zielfunktionswertes
  - lokalen Variablen Werte zuweisen
  - globale Variablen erzeugen, löschen, neue Werte zuweisen

Realisierung in RDS

Bei der Darstellung von Regeln in PROLOG erweist es sich als vorteilhaft, auch PROLOG-Variablen zu verwenden. In PROLOG werden Variablen an einen Wert gebunden, während bei 'konventionellen' Programmiersprachen Variablen Werte zugewiesen werden. Der Unterschied besteht darin, daß in PROLOG eine an einen Wert gebundene Variable aufhört, eine Variable zu sein (sie dient dann nur noch als Synonym für den Wert), während bei der in anderen

Programmiersprachen üblichen Wertzuweisung der Wert einer Variablen beliebig oft geändert werden kann. Der Prozeß der Wertbindung erfolgt in PROLOG durch **Unifikation**.

In einfachen Worten ausgedrückt besteht die Unifikation zweier PROLOG-Terme darin, diese Terme durch Variablenbindungen zu vereinheitlichen. Diese Variablenbindungen sind das eigentliche Ergebnis der Unifikation[1].

Beispiel: Unifikation von test(1 , X , 3) und
test(Y , 2 , 3)

→ X wird an Zahl 2 gebunden
Y wird an Zahl 1 gebunden

Der Vorteil der Nutzung von PROLOG-Variablen besteht vor allem in folgenden Punkten:

- Durch PROLOG-Variablen ist eine einfache Datenübertragung vom Situationsteil zum Aktionsteil einer Regel möglich. Dabei wird eine Variable im Situationsteil an einen Wert gebunden (z.B. an das Ergebnis einer arithmetischen Berechnung) und im Aktionsteil kann dann dieser Wert weiterverwendet werden.

- Durch das Ersetzen von Konstanten durch Variablen[2] ist eine Verallgemeinerung von Regeln möglich. Wenn z.B. eine Regel existiert, die den Durchsatz der Maschine 1 verbessert und es existieren noch weitere gleichartige Maschinen 2 .. 5, dann kann durch Ersetzen der Maschinennummer 1 in der Regel durch eine Variable (z.B. X) die Regel verallgemeinert werden.

- Allgemein erhalten die Sprachkonstrukte, die im weitesten Sinne der Ermittlung eines Wertes dienen, durch die Verwendung von Variablen einen neuen Charakter. Wenn in diesen Sprachkonstrukten konkrete Werte (Zahlen, arithmetische Ausdrücke, usw.) angegeben werden, dann ist die Bedingung wahr, wenn der ermittelte Wert und der angegebene Wert übereinstimmen (**Wertvergleich**). Werden in diesen Konstrukten jedoch Variablen angegeben, dann ist die Bedingung immer wahr und der ermittelte Wert wird an die entsprechende Variable gebunden (**Wertermittlung**).

Variablennamen in PROLOG können aus Buchstaben, Zahlen und dem Unterstrich '_' bestehen, wobei das erste Zeichen ein Großbuchstabe oder der Unterstrich sein muß.

In RDS hat jede Regel die allgemeine Form

<regelname> **::** <situationsteil> **::** <aktionsteil> **.**

1. Regelname

Regelnamen sind eindeutige Bezeichner, die den Regeln zugewiesen werden und die dazu dienen, diese Regeln in Kurzform zu repräsentieren. Sie sollten Aufschluß über die Bedeutung der Regel geben.

[1] Natürlich läßt sich das Konzept von PROLOG nicht in wenigen Worten erklären. Für eine weitergehende Beschäftigung mit PROLOG empfehle ich eines der zahlreichen PROLOG-Lehrbücher (z.B. [Bot91], [Dod90], [Sai90], [Ste88] u.a.).

[2] Das ist eine der wichtigsten Generalisierungsregeln des **induktiven Schließens** (siehe [Hel91]).

Regelnamen können aus Kleinbuchstaben, Zahlen und dem Unterstrich '_' bestehen, wobei das erste Zeichen ein Kleinbuchstabe sein muß.

Beispiel: regel1 (wenig aussagekräftig)
verbessere_durchsatz
verkleinere_wartezeit

2. Situationsteil

Die Bedingungen[1] des Situationsteiles einer Regel werden in eckige Klammern eingeschlossen und durch Komma getrennt. Die zur Verfügung stehenden Bedingungen werden im folgenden beschrieben.

2.1 Zielfunktion und Optimalitätskriterium bestimmen

Da ein bestimmtes Modell bezüglich verschiedener Zielfunktionen optimiert werden kann, muß eine Regel verdeutlichen, für welche Zielfunktion sie gedacht ist. Zusätzlich kann auch das Optimalitätskriterium (entweder **minimum** oder **maximum**) festgelegt werden (obwohl es i.allg. zu jeder konkreten Zielfunktion nur genau ein sinnvolles Optimalitätskriterium gibt).

Syntax: **zielfunktion ist** <arithm. ausdruck>

opt_kriterium ist <optimalitätskriterium>

Beispiel: **zielfunktion ist** (material_kosten + lohn_kosten)

opt_kriterium ist minimum

zielfunktion ist (material_kosten + Sonstige_Kosten)

opt_kriterium ist Opt_Krit

2.2 Eigenschaften von Datenobjekten ermitteln

Wenn bei einem Modell jeweils verschiedene Datenobjekte als Parameter bzw. Konstanten verwendet werden können, dann muß die Regel prüfen können, von welcher Art ein bestimmtes Datenobjekt im aktuellen Modell ist. Der Hauptanwendungsfall wird sein, daß geprüft wird, ob ein bestimmtes Datenobjekt ein Parameter ist, bevor es im Aktionsteil geändert wird. Darüber hinaus sind noch einige andere 'Eigenschaften' von Datenobjekten bestimmbar.

Syntax: <eigenschaft> **von** <datenobjekt> **ist** <eigenschaftswert>

art → **parameter**
konstante
ergebnis (Leistungskenngröße)

[1] Genau gesagt kann man zwischen 'echten' Bedingungen (z.B. arithmetische Vergleiche) und Wertermittlungsfunktionen unterscheiden. Während erstere sowohl WAHR als auch FALSCH sein können, besitzen letztere immer den Wahrheitswert WAHR.

untere_grenze	→	<zahl>
obere_grenze	→	<zahl>
datentyp	→	**integer** **real**

Beispiel: **art von** job_vert:1 **ist parameter** (Ist 'job_vert:1' ein Parameter ?)

art von Param **ist parameter** (Binde 'Param' an einen Parameter !)

untere_grenze von job_vert:1 **ist** UG (Binde 'UG' an die untere Schranke des Parameters 'job_vert:1' !)

2.3 Werte von Datenobjekten ermitteln

Um die Anwendbarkeit einer Regel überprüfen zu können, ist es oft notwendig, die konkreten Werte von Datenobjekten (z.B. Leistungskenngrößen) zu bestimmen (evtl. in Verbindung mit einem arithmetischen Vergleich).

Syntax: <datenobjekt> **ist** <wert>

Beispiel: job_vert:1 **ist** 0.5 (Hat 'job_vert:1' den Wert 0.5 ?)

kosten_matrix:2:3 **ist** Kosten (Binde 'Kosten' an den Wert des 3. Elementes der 2. Zeile von 'kosten_matrix')

2.4 Spezielle Datenobjekte finden

Bei der Dimensionierung kann es erforderlich sein, solche Datenobjekte aus einer bestimmten Menge zu finden, die innerhalb dieser Menge den größten bzw. kleinsten Wert besitzen (z.B. 'Finde die Maschine mit dem kleinsten Auslastungsgrad !').

Syntax: **maximum von** <datenobjekt> **mit** <mengenbeschreibung> **ist** <wert>
minimum von <datenobjekt> **mit** <mengenbeschreibung> **ist** <wert>

Dabei beschreibt <mengenbeschreibung> die Menge der Datenobjekte, die zur Minimum- bzw. Maximumsuche herangezogen werden. Diese Mengenbeschreibung entspricht syntaktisch dem Situationsteil einer RDS-Regel. Es werden genau die Datenobjekte betrachtet, für die dieser Situationsteil wahr ist.

Beispiel: **maximum von** job_vert:I **mit** [art von job_vert:I ist parameter] **ist** Max

(Suche das größte Element des Vektors 'job_vert', das als Parameter definiert wurde !)

2.5 Summenbildung

Manchmal wird eine Funktion benötigt, die die Aufsummierung der Werte einer Menge von Datenobjekten ausführt (vielleicht um danach den Mittelwert zu bilden).

Syntax: **summe von** <datenobjekt> **mit** <mengenbeschreibung> **ist** <wert>

Beispiel: **summe von** job_vert:I **mit** [art von job_vert:I ist parameter] **ist** Sum

(Summiere die Werte aller Elemente des Vektors 'job_vert', die als Parameter definiert wurden !)

2.6 Arithmetische Vergleiche

Nach der Ermittlung von Datenobjekt-Werten kann die Notwendigkeit bestehen, diese mit anderen Datenobjekt-Werten bzw. mit Zahlen zu vergleichen.
Ein arithmetischer Vergleich besteht aus zwei berechenbaren arithmetischen Ausdrücken, die mit einem Vergleichsoperator verknüpft sind. Er entspricht im Prinzip der Darstellung einer Restriktion in DABS, wobei in den arithmetischen Ausdrücken zusätzlich noch gebundene[1] Variablen vorkommen können.

Syntax: <arithm. ausdruck> <vergleichsoperator> <arithm. ausdruck>

Vergleichsoperatoren: **<** , **<=** , **=** , **>=** , **>** , **<>**

Beispiel: (job_vert:1 > job_vert:2)

(job_vert:1 + 0.1 <= OG) (OG z.B. an 0.8 gebunden)

2.7 Lokalen Variablen Werte zuweisen

Wenn im Situationsteil einer Regel eine Berechnung erfolgt, deren Ergebnis im Aktionsteil derselben Regel benötigt wird (z.B. um einem Parameter als neuer Wert zugewiesen zu werden), dann kann man diesen Wert einer (zu diesem Zeitpunkt noch ungebundenen) PROLOG-Variablen zuweisen. Sie wird als **lokale Variable** bezeichnet, da sie nur innerhalb dieser speziellen Regel definiert ist. Lokale Variablen können an Werte beliebigen Typs gebunden werden, wobei darauf zu achten ist, daß das System immer versucht, den rechts stehenden Ausdruck zu berechnen. Soll nicht der Wert des Ausdruckes sondern der Ausdruck selbst gespeichert werden, dann ist der Ausdruck zu quotieren.
Prinzipiell können in einer Regel beliebig viele (verschiedene) PROLOG-Variablen verwendet werden.

Syntax: <prolog-variable> **<-** <arithm. ausdruck> oder
<prolog-variable> **<- quote(**<ausdruck>**)**

Beispiel: Gesamtkosten **<-** sum(i = 1..5 , kosten:1)

Bester_Param **<-** quote(job_vert:1)

[1] Die Variable muß erst im Augenblick der Ausdrucksberechnung an einen (Zahlen-)Wert gebunden sein.

2.8 Werte von globalen Variablen ermitteln

Falls es nötig werden sollte, Informationen zwischen verschiedenen Regeln auszutauschen oder Informationen bis über eine Regelanwendung hinaus aufzubewahren, dann können **globale Variablen** verwendet werden. Dieses Konzept wird nicht von PROLOG zur Verfügung gestellt und mußte daher explizit programmiert werden. Einer globalen Variablen kann ein beliebiger Wert (Zahl, Zeichenkette, usw.) zugewiesen werden.
Im Aktionsteil einer Regel können die Werte von globalen Variablen ermittelt bzw. getestet werden.
Bei 'normalen' Regeln wird dieses Konzept i.allg. nicht benötigt.

ACHTUNG: Die Namen von globalen Variablen entsprechen den Konventionen für lokale Variablen bis auf den Unterschied, daß das erste Zeichen des Namens ein Kleinbuchstabe sein muß. Achten Sie auch darauf, daß es zu keinen Namenskonflikten mit den Datenobjekten des Modelles kommt !

Syntax: <globale variable> **ist** <wert>

Beispiel: zustand **ist** start (Hat die globale Variable 'zustand' den Wert 'start' ?)

zustand **ist** Wert (Binde 'Wert' an den Wert der globalen Variable 'zustand' !)

3. Aktionsteil

Die Aktionen des Aktionsteiles einer Regel werden in eckige Klammern eingeschlossen und durch Komma getrennt. Die zur Verfügung stehenden Aktionen werden im folgenden beschrieben.

3.1 Parametern neue Werte zuweisen

Die einzige Möglichkeit, die Zielfunktion eines parametrisierten Modelles zu verbessern, besteht in der Festlegung von Werten für die Modellparameter. Dabei ist darauf zu achten, daß der neue Wert eines Parameters innerhalb des für diesen Parameter definierten Bereiches [**untere_grenze** , **obere_grenze**] liegt und vom selben Zahlentyp ist.

Syntax: <parameter> **<-** <arithm. ausdruck>

Beispiel: job_vert:1 **<-** job_vert:1 + 0.1

3.2 Bestimmung der Leistungskenngrößen und des Zielfunktionswertes

Um herauszufinden, ob die Änderung von Parameterwerten tatsächlich zur Verbesserung der Leistungskenngrößen bzw. des Zielfunktionswertes des Modelles führt (es handelt sich schließlich um heuristische Regeln), kann das Modell mit der neuen Parameterbelegung getestet und der neue Zielfunktionswert berechnet werden.

Syntax: **berechne_zielfunktion**

3.3 Lokalen Variablen Werte zuweisen

Dieser Punkt entspricht inhaltlich Punkt 2.7 mit der Besonderheit, daß die lokalen Variablen im Aktionsteil ausschließlich zur Speicherung von Zwischenergebnissen verwendet werden (eine andere Verwendung erübrigt sich, da nach dem Verlassen des Aktionsteiles diese Variablen nicht mehr existieren).

3.4 Globale Variablen erzeugen

Da das Konzept der globalen Variablen nicht vom PROLOG-System zur Verfügung gestellt wird, muß das physische Erzeugen und Löschen dieser Variablen explizit vorgenommen werden. Dabei ist auf die im Punkt 2.8 genannten Namenskonventionen zu achten.
Bevor eine globale Variable benutzt werden kann, muß sie erst erzeugt werden. Vor der ersten Wertzuweisung an diese Variable besitzt sie noch keinen Wert[1].

Syntax: **erzeuge(** <globale Variable> **)**

Beispiel: **erzeuge(** zustand **)** (Lege eine physische Struktur für die globale Variable 'zustand' an !)

3.5 Globale Variablen löschen

Diese Funktion wurde nur der Vollständigkeit halber in den Sprachumfang aufgenommen, da nach vollständiger Dimensionierung eines konkreten Systems sowieso alle globalen Variablen gelöscht werden.

Syntax: **loesche(** <globale variable> **)**

Beispiel: **loesche(** zustand **)** (Lösche die für die globale Variable 'zustand' angelegte physische Struktur !)

3.6 Globalen Variablen Werte zuweisen

Über die Anwendung globaler Variablen wurde bereits im Punkt 2.8 gesprochen.
Eine globale Variable behält einen ihr zugewiesenen Wert solange, bis ihr ein neuer Wert zugewiesen oder sie gelöscht wird. Als Werte für globale Variablen kommen alle gültigen PROLOG-Terme in Frage (Zahlen, Atome, Zeichenketten, Strukturen, usw.).

Syntax: <globale variable> **<-** <wert>

Beispiel: zustand **<-** "Jetzt geht's los !" (Weise 'zustand' diese Zeichenkette zu !)

anz_versuche **<-** 5 (Weise 'anz_versuche' die Zahl 5 zu !)

Damit wären sämtliche notwenigen Sprachkonstrukte zur Darstellung von RDS beschrieben. Eine genaue Beschreibung der Syntax von RDS finden Sie im Anhang dieser Arbeit.

[1] Genaugenommen enthält die globale Variable dann den Wert 'unbelegt_'.

Die in RDS dargestellten Regeln zu einem bestimmten Modell oder einer Modellklasse werden hintereinander in eine Textdatei mit der Extension '.rd' eingetragen. Diese Datei kann dann in das Dimensionierungsprogramm geladen werden.

Zum Abschluß dieses Kapitels folgt das etwas komplexere Beispiel einer in RDS geschriebenen Regel, die für die Maximierung des Durchsatzes von Central-Servern formuliert wurde.

```
cs_durchsatz1 :: [ zielfunktion ist durchsatz_dev : I,
                   opt_kriterium ist maximum,
                   art von anz_jobs ist parameter,
                   obere_grenze von anz_jobs ist OG,
                   anz_jobs < OG,
                   a_jobs : I < 10 ]
              :: [ anz_jobs <- anz_jobs + trunc( (OG + 1 - anz_jobs) / 2 ),
                   berechne_zielfunktion ] .
```

Erklärung:

WENN das Ziel der Dimensionierung ist, den Durchsatz von Bedienzentrum I zu maximieren **UND**
in diesem Modell die Anzahl Jobs im Netz veränderbar ist **UND**
die Maximalzahl Jobs im Netz noch nicht erreicht ist **UND**
sich im Mittel weniger als 10 Jobs im Bedienzentrum I aufhalten

DANN erhöhe die Anzahl Jobs im Netz um die Hälfte des Abstandes zur Maximalzahl Jobs **UND**
ermittle die daraus resultierenden Veränderungen von Leistungskenngrößen und Zielfunktionswert .

4.2.2.2 Regelverarbeitung

Mit Hilfe eines geeigneten Regelverarbeitungsmechanismus soll das in Form von RDS-Regeln dargestellte Fachwissen des Entwurfsingenieurs zur Dimensionierung bestimmter Modelle so angewendet werden, daß als Ergebnis eine Belegung der Modellparameter bestimmt wird, die dieser Fachmann so oder ähnlich auch gewählt hätte. Die Regelverarbeitung (im weiteren auch **Inferenz** genannt) muß also in gewisser Weise das Vorgehen des Experten bei der Anwendung seines Wissens 'simulieren'.

Der von mir gewählte Ansatz geht von einer gewissen **Trial-Error-Methode** des Experten aus. Dabei betrachtet der Experte die bisher 'beste' Parameterbelegung und die zugehörigen Leistungskenngrößen und die Zielfunktion des Modelles. Anhand dieser Daten und seines Wissens über Ursache-Wirkung-Beziehungen im Modell verändert er den Wert von einem oder mehreren Modellparametern. Dabei hegt er die Hoffnung (da das Wissen über das Modell i.allg. nur heuristischer Natur ist), daß sich die Zielfunktion bei dieser neuen Parameterbelegung verbessert. Ist das wirklich der Fall, dann beendet er die Dimensionierung (das Ergebnis war optimal[1]) oder versucht, die Zielfunktion weiter zu verbessern. Erfolgt für die gewählte Parameterbelegung keine Verbesserung des Zielfunktionswertes, dann muß der Experte ausgehend von der bisher besten Parameterbelegung eine zu den erfolglosen Versuchen alternative Belegung finden, die dann ebenfalls getestet wird.

[1] 'Optimal' muß natürlich in der Praxis relativiert werden. Oft wird bereits eine 'gute' Parameterbelegung akzeptiert, bei der sämtliche oder die wesentlichen Restriktionen erfüllt sind.

Das allgemeine Vorgehen des von mir gewählten Regelverarbeitungsmechanismus entspricht in etwa der oben geschilderten Methode.

1. Ermittle alle Regeln, die in der augenblicklichen Situation anwendbar sind
   (Test der Situationsteile aller Regeln)

2. Wenn keine der Regeln anwendbar ist, dann gehe zu Schritt 3
   sonst

   2.1 Wenn bereits alle anwendbaren Regeln probiert wurden, dann gehe zu Schritt 3
   sonst

   2.1.1 Wähle nach einer bestimmten Strategie eine noch nicht probierte anwendbare Regel aus

   2.1.2 Führe die ausgewählte Regel aus
   (Führe alle Aktionen des Aktionsteils dieser Regel aus)

   2.1.3 Wenn sich der Zielfunktionswert verbessert hat, dann gehe zu Schritt 1
   sonst gehe zu Schritt 2.1

3. Beende die wissensbasierte Dimensionierung

Der Schritt 1 (Ermitteln aller anwendbaren Regeln) wird auch **Bildung der Konfliktmenge** und der Schritt 2.1.1 (Auswahl einer anwendbaren Regel) wird auch **Konfliktauflösung** genannt.

Während des Inferenzprozesses können drei wesentliche Probleme bzw. Aufgaben auftreten:

- Rücksetzen von Wertebelegungen des Modelles, falls eine angewendete Regel nicht zu einem Erfolg führte (die Regel muß 'zurückgenommen' werden).

- Verhinderung von (unendlichen) Schleifen
  (wenn innerhalb eines Inferenzprozesses eine konkrete Situation mehrmals auftritt und in dieser Situation immer die gleiche Regel verwendet wird, kann es zu unendlichen Schleifen bei der Regelverarbeitung kommen)

- Sammeln von Informationen über den Prozeß der Regelverarbeitung (Inferenz), um dem Nutzer das Ergebnis dieses Prozesses erklären zu können.

Rücksetzen von Wertebelegungen

Da die von uns betrachteten Regeln heuristischer Natur sind, kann es in bestimmten Situationen vorkommen, daß eine Regelanwendung nicht zum Erfolg (Verbesserung des Zielfunktionswertes) führt. Zusätzlich muß natürlich auch der Fall in Betracht gezogen werden, daß die angewendete Regel grundsätzlich falsch ist. Wenn diese Fälle eintreten, dann darf sich kein negativer Einfluß auf die Anwendung evtl. noch vorhandener alternativer Regeln ergeben. Das beinhaltet insbesondere, daß sämtliche Wertebelegungen (Parameter, Leistungskenngrößen, Zielfunktionswert, globale Variablen) auf den Stand vor Ausführung der 'falschen' Regel zurückgesetzt werden müssen.

Dieses Problem wurde von mir durch eine **Protokoll-Funktion für Werteänderungen** gelöst. Immer dann, wenn im Aktionsteil einer Regel[1] der Wert eines der oben genannten Objekte geändert wird, wird der alte Wert zuvor gesichert. Tritt der Fall ein, daß diese Regel keinen Erfolg hat, dann werden die alten Werte zurückgesetzt und es wird evtl. eine alternative Regel angewendet.

Verhinderung von Schleifen

Wie bereits oben erwähnt kann es zur Schleifenbildung kommen, wenn in gleichen Situationen während eines Inferenzprozesses immer die gleichen Regeln verwendet werden (in diesem Fall wird eine bestimmte Folge von Regeln ständig wiederholt). Um diese Gefahr zu beseitigen, muß eine **Protokoll-Funktion für Regelanwendungen** realisiert werden. Dabei ist es nicht ausreichend, die angewendeten Regeln zu protokollieren, sondern zusätzlich muß eine Beschreibung der Situation gespeichert werden, in der diese Regeln zur Anwendung kamen.

Da sich eine Entwurfssituation (für ein bestimmtes Modell) eindeutig durch die Belegung der Parameter beschreiben läßt, ist es ausreichend, die angewendeten Regeln zusammen mit einer Liste der aktuellen Parameterwerte zu speichern. Bei der Ermittlung der in einer konkreten Situation potentiell anwendbaren Regeln werden dann nur noch solche Regeln in Betracht gezogen, die in derselben Situation noch nicht angewendet wurden.

Sammeln von Informationen

Um dem Nutzer während oder nach dem Inferenzprozeß die Folge der Regelanwendungen erklären zu können, müssen nach erfolgreicher Anwendung einer Regel bestimmte Informationen gespeichert werden, die dann der Erklärungskomponente des wissensbasierten Systemes zur Verfügung stehen[2].

Dabei erachte ich die Speicherung folgender Informationen über eine Regelanwendung als wesentlich:

- Nummer des Inferenzschrittes
- Name der Regelwerte der lokalen Variablen der Regel
- geänderte Parameterwerte
- geänderte Werte globaler Variablen
- Werte der Leistungskenngrößen
- Wert der Zielfunktion

Ein genaueres Eingehen auf die Implementation des Regelverarbeitungsmechanismus erscheint mir in dieser Arbeit nicht notwendig, da für den Benutzer eigentlich nur das Wissen über die Darstellung der RDS-Regeln von Belang ist. Zum Verständnis des internen Aufbaus des Verarbeitungsmechanismus wären außerdem fundierte Kenntnisse und möglichst auch Programmiererfahrungen in PROLOG notwendig, die beim Leser nicht vorausgesetzt werden können. Der Quelltext des Regelverwaltungs- und Regelverarbeitungsmoduls befindet sich im Anhang dieser Arbeit ('regel.pro').

[1] Die Bedingungen des Situationsteils von Regeln sind grundsätzlich so gestaltet, daß es zu keinen Wertänderungen kommen kann.

[2] Ob die Erklärungskomponente wirklich alle diese Informationen zur Generierung von Erklärungen benutzt, ist im Augenblick ohne Belang.

5. Das Programmsystem

5.1 Die Nutzer-Schnittstelle

Nachdem in den vergangenen Kapiteln die wesentlichen Entwurfsentscheidungen für die interne Funktion des Programmsystems begründet wurden, stellt sich an dieser Stelle die nicht unwesentliche Frage nach der Gestaltung der **Nutzer-Schnittstelle**. Diese entscheidet im hohen Maße darüber, ob das Softwareprodukt vom Nutzer 'akzeptiert' wird. Deshalb wird beim Vergleich verschiedener Konzepte als wichtigste Forderung die **Nutzerfreundlichkeit**[1] genannt. Trotzdem wird diese Anforderung gerade auf dem Gebiet der KI zumeist relativiert[2]. Merritt sagt dazu in [Mer89]:

> *Obwohl das Bewußtsein über die Notwendigkeit guter externer Interfaces ständig zunimmt, sollte nicht vergessen werden, daß es immer noch die interne Funktion ist, die das Herz der Anwendung bildet.*

Hinzu kommt noch, daß die Verwendung bestimmter Schnittstellenkonzepte die Portierung der Anwendung auf andere Rechner oft einschränkt, auf denen diese Konzepte nicht oder nur schwer realisiert werden können.

Aus den genannten Gründen und um mich auf die wesentlichen Komponenten des Programmsystemes zu konzentrieren, entschloß ich mich zur Realisierung einer Kommandozeilenoberfläche statt einer Fensteroberfläche. Ein Teil der dadurch entstehenden Nachteile wurde durch die Entwicklung einer **natürlichsprachlich orientierten Eingabe** wieder ausgeglichen. Diese ist zwar nicht in der Lage, grammatikalische Zusammenhänge zu erkennen, doch wird es dem Nutzer dadurch möglich, einfache Sätze einzugeben, wobei für die Kommandos intuitiv naheliegende Worte[3] gewählt werden können. Die Erkennung der Kommandos erfolgt durch einfachen Zeichenkettenvergleich, während nach evtl. notwendigen Argumenten nur an bestimmten Stellen der Eingabe gesucht wird. Werden die Argumente nicht gefunden, dann wird der Nutzer explizit gefragt. Um dieses Konzept etwas deutlicher zu machen, betrachten Sie bitte folgendes Beispiel:

Eingabezeile: Lade mir bitte die Aufgabenbeschreibung aus der Datei RECHNER1.

1. Schritt: Suche die allgemeine Kommandoart

"Lade" gefunden $\rightarrow$ es handelt sich um ein LADE-Kommando

---

[1] Dieser Begriff erscheint mir irreführend, den es geht nicht um die Freundlichkeit des Nutzers, sondern um eine dem Nutzer entgegenkommende Benutzbarkeit des Programmes.

[2] Ein Grund dafür könnte sein, daß auf dem Gebiet der 'konventionellen' Softwareprodukte eine gewisse Sättigung des Marktes eingetreten ist (siehe z.B. Textverarbeitungen), so daß das gefällige Aussehen der Programmoberfläche oft über den Kauf entscheidet, während bei KI-Anwendungen (zum heutigen Zeitpunkt) die Auswahl bedeutend kleiner ist und deshalb eher die Funktionalität entscheidet.

[3] Zur Bezeichnung eines konkreten Kommandos kann dabei ein Wort aus einer Menge von Synonymen verwendet werden (z.B. lade - { lade, lese, konsultiere }).

2. Schritt: Suche eine Spezifizierung eines Kommandos dieser Art

"Aufgabe" gefunden → es handelt sich um das LADE_AUFGABE-Kommando
(dieses Kommando erfordert einen Dateinamen)

3. Schritt: Suche ein Argument, das einen Dateinamen repräsentiert

→ dieses Argument könnte nach Wörtern wie "Datei", "File" bzw. "Aufgabe" stehen; ansonsten frage den Nutzer

"RECHNER1" gefunden → Kommando: LADE_AUFGABE RECHNER1

Natürlich ist es sehr leicht möglich, die Eingabe zu 'überlisten', indem man kompliziertere Sätze benutzt oder die Stellung der Argumente unüblich wählt. Trotzdem erscheint mir dieser Mechanismus ausreichend, um die im Programmsystem enthaltenen Kommandos relativ komfortabel eingeben zu können.

5.2 Die Beschreibung der wichtigsten Kommandos

Wie bereits oben kurz erwähnt können zur Bezeichnung von Kommandos i.allg. mehrere synonyme Worte verwendet werden. Im folgenden werden bei der Angabe der Syntax für diese Kommandos nur allgemeine Platzhalter angegeben, die anschließend genauer erklärt werden. Wenn innerhalb von Kommandonamen das Metazeichen '*' auftritt, dann kann an dieser Stelle eine beliebige Zeichenkette (auch eine leere) stehen.

Die verfügbaren Kommandos können in folgende allgemeine Arten zusammengefaßt werden:

- Laden von Dateien
- Speichern von Dateien
- Anzeige von Daten bzw. Informationen
- Eingeben/Ändern von Daten
- Löschen von Daten
- Dimensionierung
- Beenden des Programmes

1. Laden von Dateien

1.1 Laden von Aufgabenbeschreibungen

Mit dieser Funktion ist es möglich, in der Beschreibungssprache DABS erstellte und in einer Textdatei mit der Extension '.dab' gespeicherte Dimensionierungsaufgaben zu laden. Das ist die

Grundvoraussetzung für die Benutzung des Programmes. Sollte die Aufgabenbeschreibung nicht der Syntax von DABS entsprechen, werden entsprechende Fehlermeldungen angezeigt.

Achtung: Im Programm kann immer nur eine Aufgabenbeschreibung verwaltet werden !

Syntax: <lade-aufgabe> <dateiname>

mit <lade-aufgabe> - [**lade*** | **lese*** | **konsult***] +
[**aufgab*** | **modell*** | **system***]

<dateiname> - Dateiname der Beschreibung (ohne Extension)

Beispiel: **lade** die **aufgab**enbeschreibung central_server

1.2 Laden von Regeln

Mit dieser Funktion ist es möglich, Dateien zu laden, die nach der RDS-Syntax erstellte Regeln zur Dimensionierung konkreter Modelle enthalten. Diese Textdateien müssen die Extension '.rd' besitzen und können beliebig viele Regeln enthalten. Beim Laden wird ein Test auf korrekte Syntax und auf offensichtliche Schreibfehler durchgeführt.

Da die Regeln einer geladenen Regeldatei der intern verwalteten Regelmenge hinzugefügt werden, können nacheinander mehrere Regeldateien geladen werden.

Syntax: <lade-regeln> <dateiname>

mit <lade-regeln> - [**lade*** | **lese*** | **konsult***] + [**regel*** | **heuristik***]

<dateiname> - Dateiname der Regeldatei (ohne Extension)

Beispiel: **lade** die **regel**datei regel1

2. Speichern von Dateien

2.1 Speichern von Regeldateien

Mit dieser Funktion ist es möglich, die vom Programm intern verwaltete Regelmenge in einer Textdatei mit der Extension '.rd' zu speichern. Dadurch ist es möglich, eine durch Laden von Regeldateien und Ändern bzw. Löschen bestimmter Regeln entstandene Regelmenge (die sich vielleicht als brauchbar für bestimmte Modelle erwiesen hat) zusammen abzulegen und sie in einer späteren Sitzung mit dem Regellade-Kommando (Punkt 1.2) wieder komplett zu laden.

Syntax: <speichere-regeln> <dateiname>

mit <speichere-regeln> - [**speich*** | **schreib***] + [**regel*** | **heuristik***]

<dateiname> - Dateiname der Regeldatei (ohne Extension)

Beispiel: **speich**ere die **regel**n in der datei regel2

2.2 Speichern der Lösung

Mit dieser Funktion ist es möglich, die erzeugte Lösung der Dimensionierungsaufgabe (numerisch und/oder wissensbasiert) in einer Textdatei mit der Extension '.erg' abzuspeichern. Diese Datei soll in gewisser Weise als Protokoll für die Lösung dieser konkreten Aufgabe dienen. Auf diese Art ist es auch möglich, mit verschiedenen Regelmengen erhaltene Lösungen einer Aufgabe zu vergleichen.

Syntax: <speichere-lösung> <dateiname>

mit <speichere-lösung> - [**speich*** | s**chreib***] + [**loes*** | **ergeb***]

<dateiname> - Dateiname der Ergebnisdatei (ohne Extension)

Beispiel: **speich**ere die **loes**ung in datei loesung1

3. Anzeige von Daten bzw. Informationen

3.1 Anzeige der geladenen Aufgabe

Mit dieser Funktion ist es möglich, die geladene Aufgabenbeschreibung anzuzeigen.

Syntax: <zeige-aufgabe>

mit <zeige-aufgabe> - [**zeig*** | **ausgab*** | **ausgeb*** | **gib*** **aus*** | **geb*** **aus***]
+ [**aufgab*** | **modell*** | **system***]

Beispiel: **zeig**e mir die aktuelle **aufgab**enstellung

3.2 Anzeige einer konkreten Regel

Mit dieser Funktion ist es möglich, sich eine einzelne Regel, deren Regelname bekannt ist, anzusehen. Bei der Anzeige wird versucht, die Regel so lesbar wie möglich zu gestalten (Ausgabeformat entspricht nicht genau der RDS-Syntax).

Syntax: <zeige-regel> <regelname>

mit <zeige_regel> - [**zeig*** | **ausgab*** | **ausgeb*** | **gib*** **aus*** | **geb*** **aus***] +
[**regel*** | **heuristik***]

<regelname> - Name der anzuzeigenden Regel

Beispiel: **zeig**e mir die **regel** erhoehe_durchsatz1

3.3 Anzeige aller Regeln

Mit dieser Funktion ist es möglich, sich die gesamte interne Regelmenge in der Reihenfolge ihrer Speicherung anzusehen. Dabei erfolgt nach jeder ausgegebenen Regel die Anfrage, ob

noch weitere Regeln angezeigt werden sollen. Das Ausgabeformat entspricht dem der Anzeige einer einzelnen Regel.

Syntax: <zeige_regeln>

mit <zeige-regeln> - [**zeig*** | **ausgab*** | **ausgeb*** | **gib*** **aus*** | **geb*** **aus***] + [**alle regel*** | **alle heuristik***]

Beispiel: **zeig**e mir **alle** gespeicherten **regel**n

3.4 Anzeige der Lösung

Mit dieser Funktion ist es möglich, sich die numerische und/oder wissensbasierte Lösung der geladenen Aufgabe anzusehen (vorausgesetzt, sie wurde vorher ermittelt). Damit kann ein Vergleich der durch verschiedene Methoden erzielten Lösungen durchgeführt werden.

Syntax: <zeige-lösung>

mit <zeige-lösung> - [**zeig*** | **ausgab*** | **ausgeb*** | **gib*** **aus*** | **geb*** **aus***] + [**loes*** | **ergeb***]

Beispiel: **zeig**e mir die ermittelten **loes**ungen

3.5 Hilfe

Mit dieser Funktion ist es möglich, Hilfstexte zu verschiedenen Themen anzuzeigen. Es erscheint ein Auswahlmenü mit folgenden Punkten:

- Hilfe zu den verfügbaren Kommandos
- Hilfe zum augenblicklichen Zustand des Programmes
- Hilfe zum zuletzt aufgetretenen Fehler
- Hilfe zum zuletzt ausgeführten Kommando

Syntax: <hilfe>

mit <hilfe> - [**hilf*** | **helf*** | **erklaer*** | **erlaeuter***]

Beispiel: **hilf** mir bitte

3.6 Erläuterung eines Inferenzprozesses

Mit dieser Funktion ist es möglich, sich Informationen über einen vorher durchgeführten Inferenzprozeß (wissensbasierte Dimensionierung) anzeigen zu lassen. Dabei werden sukzessive alle angewendeten Regeln und die durch sie vorgenommenen Parameteränderungen und erzielten Ergebnisse angezeigt.

Syntax: <erklaere-inferenz>

mit <erklaere-inferenz> - [**erklaer*** | **erlaeuter*** | **begruend***] +
[**inferenz*** | **ergeb*** | **loes***]

Beispiel: **erklaer**e mir bitte diese **inferenz**

4. Löschen von Daten

4.1 Löschen der Aufgabenbeschreibung

Mit dieser Funktion ist es möglich, eine im Programm gespeicherte Aufgabenbeschreibung zu löschen. Diese Funktion braucht i.allg. nicht verwendet werden, da beim Laden einer neuen Aufgabenbeschreibung die bisher gespeicherte automatisch gelöscht wird.

Syntax: <lösche-aufgabe>

mit <lösche-aufgabe> - [**loesch*** | **entfern***] +
[**aufgab*** | **modell*** | **system***]

Beispiel: **loesch**e die aktuelle **aufgab**e

4.2 Löschen einer konkreten Regel

Mit dieser Funktion ist es möglich, eine konkrete Regel, deren Name bekannt ist, aus der intern verwalteten Regelmenge zu entfernen. Das kann erwünscht sein, wenn diese Regel nie eine Verbesserung der Zielfunktion bewirkt bzw. wenn getestet werden soll, welche Lösung der Inferenzprozeß ohne diese Regel erzielt.

Syntax: <lösche-regel> <regelname>

mit <lösche-regel> - [**loesch*** | **entfern***] + [**regel*** | **heuristik***]

<regelname> - Name der zu löschenden Regel

Beispiel: **loesch**e die **regel** erhoehe_durchsatz1

4.3 Löschen aller Regeln

Mit dieser Funktion ist es möglich, die gesamte intern verwaltete Regelmenge zu löschen. Das kann notwendig werden, wenn eine andere Regelmenge getestet werden soll.

Syntax: <lösche-regeln>

mit <lösche-regeln> - [**loesch*** | **entfern***] +
[**alle regel*** | **alle heuristik***]

Beispiel: **loesch**e **alle** gespeicherten **regel**n

5. Dimensionierung

Das ist die Kernfunktion des Programmes. Sie ermöglicht es, die geladene Aufgabe sowohl numerisch als auch wissensbasiert zu lösen (Auswahl über Menü). Dazu ist erforderlich, daß eine syntaktisch korrekte Aufgabenbeschreibung geladen wurde (siehe Punkt 1.1), daß ein ausführbares Programm zur Modellierung des Systemverhaltens vorliegt (entsprechend der Schnittstellen-Definition) und daß bei wissensbasierter Dimensionierung eine nichtleere Menge von Dimensionierungsregeln geladen wurde (siehe Punkt 1.2).

Syntax: <dimensioniere>

mit <dimensioniere> - [**dimens*** | **loes*** | **opti*** | **rechne*** | **simul***]

Beispiel: **dimens**ioniere das vorliegende modell

6. Beenden des Programmes

Mit dieser Funktion wird das Programm beendet. Dabei gehen sämtliche während der Sitzung im Programm gesammelten Daten und Informationen verloren (zum Abspeichern siehe Punkt 2). Es erfolgt eine Sicherheitsabfrage, ob Sie das Programm wirklich verlassen wollen.

Syntax: <beende>

mit <beende> - [**stop*** | **ende*** | **schluss*** | **exit***]

Beispiel: lass uns fuer heute **schluss** machen

Wie Sie sicher bemerkt haben, stellt daß Programm keinerlei Funktionen zur Modifikation der Aufgabenbeschreibung oder der Regeln zur Verfügung. Das hat vor allem zwei Gründe:

- Problem des Testes der Korrektheit der Modifikation
- hoher Programmieraufwand für komfortable Editierfunktionen

Bei der Arbeit auf einem Rechner mit multitaskingfähigem Betriebssystem und Fenstertechnik stellen diese fehlenden Funktionen keine Nachteile dar, da es möglich ist, zur gleichen Zeit sowohl das Programm als auch einen Texteditor (oder mehrere) laufen zu lassen. In diesem Fall wird die Datei, die die Aufgabenbeschreibung bzw. Regeldarstellung enthält, im Texteditor modifiziert und anschließend wieder ins Programm geladen (siehe Punkt 2).

Damit sind die wesentlichen Kommandos des Programmes zur wissensbasierten Dimensionierung beschrieben. Während einer Sitzung mit dem Programm stehen zu den verfügbaren Kommandos Online-Hilfstexte zur Verfügung.

5.3 Eine typische Sitzung

Um das Aussehen und die Funktion der Programmoberfläche sowie das typische Vorgehen bei der Benutzung des Programmes deutlich zu machen, erscheint es mir angebracht, eine 'typische' Sitzung zu beschreiben. Dabei werde ich die Nutzereingaben in Kursivschrift, die Programmausgaben in Fettschrift und Kommentare in normaler Schrift nach einem '%' notieren.

Um das Programm zu starten, muß man sich in dem Verzeichnis befinden, das sowohl die Quelldateien des Programmes als auch die Aufgabenbeschreibungs- und Regeldarstellungsdateien enthält. Gestartet wird das Programm mit

prolog -c dim_experte

Dieses Kommando ruft den PROLOG-Interpreter auf und lädt die angegebene Datei. Anschließend wird das Programm automatisch gestartet.

Wissensbasierte Dimensionierung technischer Systeme v1.0 von Sven Hader 1993

Ihr Vorname bitte : *Sven* % Eingabe des Nutzernamens

Guten Tag, Sven ! Heute ist der 17.4.1993. Es ist 14.35 Uhr.

> *lade die Aufgabe aus der Datei m_halle1* % Laden einer Dimensionierungsaufgabe

Laden der Aufgabenbeschreibungsdatei m_halle1.dab .

> *lade die Regeldatei m_halle3* % Laden der Dimensionierungsregeln

Laden der Regeldatei m_halle3.rd .

> *dimensioniere* % Lösung der Dimensionierungsaufgabe

Dimensionierung mit

1. numerischer Methode
2. wissensbasierter Methode

3. Abbruch

Auswahl : *1* % zuerst numerisch dimensionieren, um eine Vergleichslösung zu erhalten

Numerische Loesung der Aufgabe m_halle1.

: % hier werden Zwischenergebnisse angezeigt

Dimensionierung nach 87 Schritten beendet !

Zeitdauer: 6.3 sec % Dauer der Lösung in CPU-Sekunden

Optimale Loesung % Ausgabe der optimalen Parameterbelegung

ZFW : 234978
Parameter anz_jobs = 23

> *dimensioniere*

Dimensionierung mit

1. numerischer Methode
2. wissensbasierter Methode

3. Abbruch

Auswahl : *2* % jetzt mit den geladenen Regeln dimensionieren

Regelbasierte Loesung der Aufgabe m_halle1 .

: % hier werden Zwischenergebnisse angezeigt

Dimensionierung nach 6 Inferenzschritten beendet !

Zeitdauer: 2.48333 sec % Dauer der Lösung in CPU-Sekunden

Optimale Loesung % Ausgabe der optimalen Parameterbelegung
ZFW : 234978 % es wurde dasselbe Ergebnis ermittelt wie
Parameter anz_jobs = 23 % bei der numerischen Methode

anz_jobs1 wurde 1 mal benutzt. % Ausgabe der verwendeten Regeln
anz_jobs5 wurde 2 mal benutzt.
anz_jobs4 wurde 3 mal benutzt.

> *speichere die Loesungen in Datei m_halle1* % Ergebnisse protokollieren

Speichern der Loesungen in Datei m_halle1.erg .

> *ende* % Programm beenden

Wollen Sie wirklich beenden, Sven [j/n] : *j*

Im hier gezeigten Beispiel war die verwendete Regelmenge vollständig, da mit ihr wissensbasiert das gleiche Ergebnis ermittelt wurde wie mit dem mathematischen Verfahren. Meistens ist es jedoch so, daß die Regelmenge noch nicht vollständig ist, so daß mit der regelbasierten Methode schlechtere Ergebnisse erzielt werden. In diesem Fall können ausgehend von einer Analyse der Ergebnisse der numerischen Methode neue Regeln zur Regelmenge hinzugefügt bzw. bestehende modifiziert werden. Das wird solange fortgesetzt, bis der Nutzer mit den Ergebnissen der regelbasierten Methode zufrieden ist.

6. Ein Beispiel

In diesem Kapitel möchte ich ein Beispiel für eine Dimensionierungsaufgabe und die vom Programm ermittelten Lösungen vorstellen. Dieses Beispiel wurde von mir verwendet, um die Funktion des Programmes zu testen. Das ist meiner Ansicht nach bei wissensbasierten Systemen eine durchaus angemessene Testmethode, da eine theoretisch fundierte Verifikation nur schwer möglich ist. Dodd bemerkt zu diesem Thema folgendes (siehe [Dod90]):

> *In der Praxis ist der einzige erfolgreiche Ansatz zum Testen eines großen Systems als Ganzes der, einige Beispieltexte aufzuschreiben und sich hinzusetzen und sie einzugeben, während man beobachtet, ob Fehlermeldungen auf dem Bildschirm erscheinen oder ob sich allgemein ein unerwartetes Verhalten zeigt.*

Das von mir gewählte Beispiel beruht auf der Nachbildung der Arbeit in einer Maschinenhalle durch ein Central-Server-Modell. Das bringt den Vorteil, daß die Ermittlung der Leistungskenngrößen analytisch[1] erfolgen kann und somit i.allg. Zeit gespart wird. Außerdem besitzt ein Central-Server-Modell eine gut überschaubare Menge von das System charakterisierenden Kenngrößen.

In einer **Maschinenhalle** befinden sich n Maschinen, die dieselbe Funktionalität besitzen, aber i.allg. nicht dieselbe Bearbeitungsintensität haben (die Maschinen arbeiten unterschiedlich schnell). Die Werkstückrohlinge werden in einer speziellen Austauscheinheit ('Eingang' der Halle) auf Bearbeitungsschlitten montiert; fertige Werkstücke werden von den Schlitten abmontiert und verlassen die Halle. Der Transport der Bearbeitungsschlitten zu den Maschinen und zur Austauscheinheit sei 'ideal', d.h. erfolge in der Zeit 0. In der Maschinenhalle befinden sich genau m Bearbeitungsschlitten. Es wird angenommen, daß am Eingang der Halle immer genügend Werkstückrohlinge zur Verfügung stehen[2]; der Abtransport der fertigen Werkstücke erfolge ohne Stockungen. Jedes Werkstück wird vollständig von einer Maschine bearbeitet und verläßt die Halle danach. Die Bearbeitungszeiten der Maschinen sowie die Montierzeiten der Austauscheinheit werden als exponentialverteilt angenommen. Das angegebene System läßt sich durch folgende Kenngrößen eindeutig beschreiben (Abbildung 6.1 zeigt eine Darstellung des Beispielsystemes) :

- Anzahl Maschinen n
- Anzahl Bearbeitungsschlitten m
- Montierintensität der Austauscheinheit λ_0
  (umfaßt Abmontieren des fertigen Werkstückes und Montieren eines Rohlings)
- Bearbeitungsintensitäten der Maschinen λ_1 .. λ_n
- Verteilung der Werkstücke auf die einzelnen Maschinen p_1 .. p_n
  (Übergangswahrscheinlichkeiten)

Diese Maschinenhalle soll dimensioniert werden. Dazu ist es notwendig, eine zu optimierende Zielfunktion und die zu dimensionierenden Parameter anzugeben. Die gewählte Zielfunktion ist der Gewinn pro Zeiteinheit, den die Maschinenhalle erbringt. Dieser Gewinn soll sich folgendermaßen berechnen:

---

[1] Dazu ist eine Reihe von Vereinfachungen nötig (z.B. Exponentialverteilung der Bediendauern), auf die an dieser Stelle jedoch nicht näher eingegangen werden soll.

[2] Dadurch wird erreicht, daß sich in der Halle immer genau m Werkstücke bzw. -rohlinge befinden.

Gewinn pro ZE = Verkaufserlös pro ZE
- Kosten der Bearbeitungsschlitten pro ZE
- Arbeitskosten pro ZE

(die Materialkosten je Werkstück sollen mit 0 angenommen werden)

Die in der Zielfunktion enthaltenen Teilkosten werden folgendermaßen berechnet:

Verkaufserlös pro ZE = fertige Werkstücke pro ZE * Verkaufspreis pro Stück

Kosten der Bearbeitungsschlitten pro ZE = Anzahl Bearbeitungsschlitten *
Kosten pro Bearbeitungsschlitten

Arbeitskosten pro ZE = fertige Werkstücke pro ZE *
mittlere Verweildauer eines Werkstücks in der Halle *
Arbeitskosten pro Stück und ZE

Diese Kostenfunktionen wurden von mir ohne größere Beachtung der in der Ökonomie verwendeten Berechnungsvorschriften gewählt. Sie sollen nur garantieren, daß die Zielfunktion ein 'nichttriviales'[1]Maximum besitzt. Für die konstanten Kostenfaktoren wählte ich folgende Werte:

Verkaufspreis pro Stück : 5000 Einheiten
Kosten pro Bearbeitungsschlitten: 500 Einheiten
Arbeitskosten pro Stück und ZE: 10 Einheiten

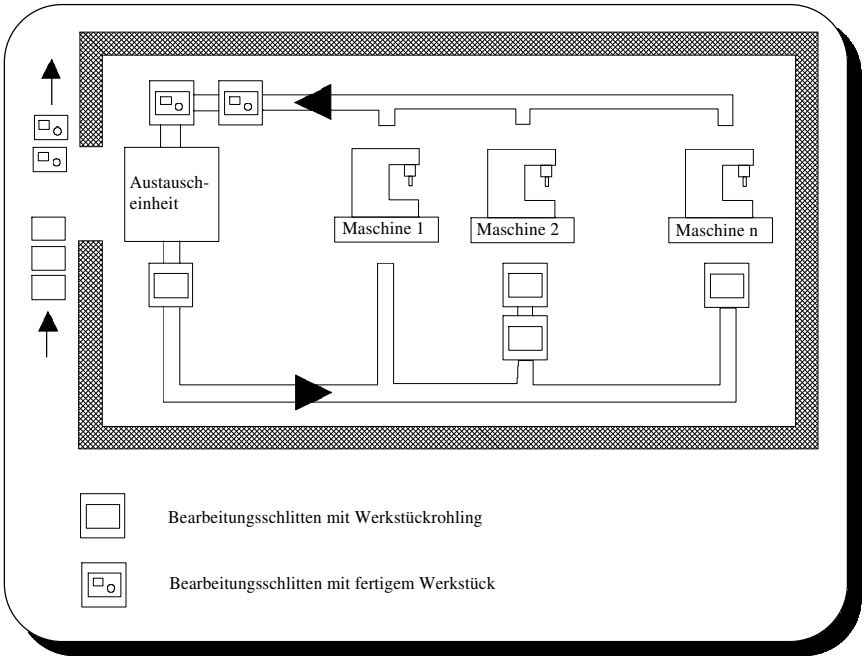

Abbildung 6.1: Darstellung der Maschinenhalle

[1] Unter trivial würde ich in diesem Zusammenhang optimale Parameterbelegungen verstehen, deren Werte den unteren bzw. oberen Intervallgrenzen der Parameter entsprechen.

Vom geschilderten allgemeinen Beispiel ausgehend entwickelte ich 8 konkrete Beispielaufgaben, die sich sowohl in den Werten der Kenngrößen als auch in der Wahl der Parameter unterscheiden. (In allen Beispielen wurde $\lambda_0 = 50$ gewählt.)

Alle Beispiele wurden sowohl numerisch als auch wissensbasiert dimensioniert. Die für die wissensbasierte Methode notwendige Regelmenge war bei allen Beispielen gleich. Die Menge umfaßte 11 Regeln, wobei sich 7 Regeln mit der optimalen Dimensionierung der Anzahl Bearbeitungsschlitten m und 4 Regeln mit der optimalen Dimensionierung der Übergangswahrscheinlichkeiten p_i beschäftigten. Die verwendete Regelmenge finden Sie im Anhang dieser Arbeit.

Ermittelte Ergebnisse

Im folgenden habe ich die ermittelten Ergebnisse in Tabellenform dargestellt. Tabelle 1 enthält die ermittelten optimalen Parameterbelegungen und Zielfunktionswerte für die einzelnen Beispielaufgaben. Dabei bedeutet '?' , daß die entsprechende Kenngröße in diesem Beispiel als Parameter gilt; 'n:' und 'w:' bezeichnen die numerische bzw. wissensbasierte Lösung. Tabelle 2 enthält die benötigten Rechenzeiten für die Lösung der Beispielaufgaben in CPU-Sekunden.

Beispiel	n	m	$\lambda_{1..n}$	$p_{1..n}$	ZFW	opt. Belegung
m_halle1	3	?	20, 20, 30	0.333, 0.333, 0.333	n: 234978 w: 234978	n: m = 23 w: m = 23
m_halle2	3	30	20, 20, 30	?, ?, ?	n: 234574 w: 234580	n: p_1 = 0.28839 p_2 = 0.264831 p_3 = 0.44678 w: p_1 = 0.285714 p_2 = 0.285714 p_3 = 0.428571
m_halle3	5	30	10, 15, 20, 10, 15	?, ?, ?, ?, ?	n: 232948 w: 233775	n: p_1 = 0.11311 p_2 = 0.237017 p_3 = 0.337795 p_4 = 0.125882 p_5 = 0.186196 w: p_1 = 0.142857 p_2 = 0.214286 p_3 = 0.285714 p_4 = 0.142857 p_5 = 0.214286
m_halle5	5	30	10, 15, 20, 10, 15	?, ?, 0.12, 0.17, ?	n: 224343 w: 223311	n: p_1 = 0.172647 p_2 = 0.270482 p_5 = 0.266872 w: p_1 = 0.18 p_2 = 0.2789 p_5 = 0.2511
m_halle8	5	?	10, 15, 20, 10, 15	?, ?, 0.12, 0.17, ?	n: 220350 w: 224024	n: m = 54 p_1 = 0.160914 p_2 = 0.277419 p_5 = 0.271667 w: m = 36 p_1 = 0.18 p_2 = 0.2789 p_5 = 0.2511
m_halle7	5	?	10, 15, 20, 10, 15	?, ?, ?, ?, ?	n: 229546 w: 234624	n: m = 39 p_1 = 0.111706 p_2 = 0.243907 p_3 = 0.341329 p_4 = 0.109932 p_5 = 0.193127 w: m = 25 p_1 = 0.142857 p_2 = 0.214286 p_3 = 0.285714 p_4 = 0.142857 p_5 = 0.214286

m_halle4	7	30	5, 7, 8, 10, 7, 8, 5	?, ?, ?, ?, ?, ?, ?	n: 177089 w: 187796	n: $p_1 = 0.11535$ $p_2 = 0.133315$ $p_3 = 0.170023$ $p_4 = 0.222714$ $p_5 = 0.158451$ $p_6 = 0.168864$ $p_7 = 0.031283$ w: $p_1 = 0.09$ $p_2 = 0.14$ $p_3 = 0.16$ $p_4 = 0.21$ $p_5 = 0.14$ $p_6 = 0.16$ $p_7 = 0.1$
m_halle6	7	30	5, 7, 8, 10, 7, 8, 5	?, ?, ?, 0.12, 0.18, ?, ?	n: 165056 w: 159708	n: $p_1 = 0.106865$ $p_2 = 0.136631$ $p_3 = 0.160767$ $p_6 = 0.208601$ $p_7 = 0.087136$ w: $p_1 = 0.107576$ $p_2 = 0.15$ $p_3 = 0.221103$ $p_6 = 0.15$ $p_7 = 0.071321$

Tabelle 1: Ermittelte optimale Parameterbelegungen

Beispiel	verbrauchte Zeit in CPU-Sekunden	
	numerisch	wissensbasiert
m_halle1	6.300	2.483
m_halle2	5.383	0.667
m_halle3	12.550	1.333
m_halle4	93.900	3.233
m_halle5	9.300	0.867
m_halle6	22.283	1.067
m_halle7	28.850	13.917
m_halle8	18.950	6.600

Tabelle 2 : Zur Dimensionierung benötigte Zeit (in CPU-Sekunden)

Auswertung

Zur Beurteilung der Leistung der wissensbasierten Dimensionierung erscheint der direkte Vergleich mit den 'numerisch' erzielten Ergebnissen angebracht. Dabei darf natürlich nicht

vergessen werden, daß die numerischen Ergebnisse in gewissem Sinne bloß Approximationen der optimalen Parameterbelegung darstellen.

Die Beispielaufgaben m_halle1 und m_halle2 werden numerisch und wissensbasiert gleich gut gelöst. Diese Übereinstimmung ist dadurch zu erklären, daß es sich um relativ 'kleine' Aufgaben handelt (ein bzw. drei Parameter).

Die Beispielaufgaben m_halle3, m_halle4, m_halle7 und m_halle8 werden wissensbasiert besser gelöst als numerisch, wobei bei den letzten drei genannten Aufgabe ein sehr deutlicher Unterschied der Lösungen bemerkbar ist. Das ist wahrscheinlich darauf zurückzuführen, daß bei diesen eher 'großen' Aufgaben (vier bis sieben Parameter) die numerische Lösung aus Zeitgründen (siehe Tabelle 2) sehr ungenau ist.

Die Beispielaufgaben m_halle5 und m_halle6 werden numerisch besser gelöst als wissensbasiert, wobei bei der zweiten Aufgabe die wissensbasierte Lösung deutlich schlechter abschneidet. Diese Fälle deuten darauf hin, daß zu bestimmten dort auftretenden Dimensionierungssituationen keine oder nur ungenügende Regeln formuliert wurden. Um diese Defizite zu beseitigen, empfiehlt es sich, die mit den ermittelten Parameterbelegungen erzielbaren Leistungskenngrößen der Maschinenhalle zu vergleichen. Treten dort größere Unterschiede auf (z.B. beim Durchsatz oder der Auslastung der Maschinen), dann sollten solche Regeln formuliert werden, die diesen Unterschieden entgegenwirken[1].

Bezüglich der verbrauchten Rechenzeit zeigt sich der deutliche Trend, daß die wissensbasierte Methode bedeutend schneller ist als die numerische (z.T. mehr als eine Zehnerpotenz). Besonders deutlich wird das bei der Lösung der Beispielaufgabe m_halle4 (3.233 sec zu 93.9 sec). Dabei darf natürlich nicht vergessen werden, daß das Abbruchkriterium der numerischen Methode eine gewisse Anzahl vergeblich geprüfter Parameterbelegungen ist (siehe Kapitel 4.2.1). Selbst wenn die optimale Parameterbelegung bereits beim ersten Versuch gefunden wird, müssen also noch eine Anzahl weiterer Parameterbelegungen geprüft werden, ehe die Dimensionierung beendet werden kann.

Das wichtigste Ergebnis der Beschäftigung mit den Beispielaufgaben ist für mich die Erkenntnis, daß selbst mit einer kleinen Regelmenge eine große Zahl von verschiedenen Dimensionierungsaufgaben erfolgreich und schnell behandelt werden kann. Das einzige Hindernis, um eine Regelmenge auf eine ganze Klasse von Aufgaben anwenden zu können (z.B. auf alle Systeme, die sich durch Central-Server-Modelle darstellen lassen), ist wahrscheinlich die unterschiedliche Benennung der Datenobjekte in den verschiedenen Aufgaben. Wenn man Möglichkeiten finden würde, die Datenobjekte nicht durch ihren konkreten Namen sondern durch ihre funktionelle Bedeutung[2] im System zu referenzieren, dann würde sich die Anwendbarkeit von konkreten Regelmengen wahrscheinlich auf ganze Aufgabenklassen ausdehnen lassen.

---

[1] Es erscheint mir ein wirklich nichttriviales Problem, eine allgemeine Bildungsvorschrift für Dimensionierungsregeln anzugeben. Selbst in der Fachliteratur zu diesem Thema werden die Regeln i.a. nur genannt und nicht ihre Herleitung erläutert.

[2] In einer Aufgabe zur Dimensionierung einer Maschinenhalle tritt z.B. ein Datenobjekt 'anzahl_werkstücke' und in einer Aufgabe zur Dimensionierung eines Rechnersystems tritt z.B. ein Datenobjekt 'anzahl_programme' auf. Bei Abstraktion von den physischen Besonderheiten stellt sich jedoch heraus, daß es sich in beiden Fällen im Sinne der Bedienungstheorie um eine **Forderung** handelt.

7. Zusammenfassung und Ausblick

Die Aufgabe dieser Arbeit bestand in der Suche nach geeigneten wissensbasierten Methoden zur Automatisierung der Entwurfsoptimierung parametrisierter Modelle. Dabei wurde ein Ansatz gewählt, der von einer 'Simulation' des Verhaltens des Entwurfsingenieurs ausgeht. Es wurde die Vermutung aufgestellt, daß sich das entsprechende Wissen in Form von WENN-DANN-Regeln darstellen läßt. Für diese Regeln wurde eine Beschreibungssprache entwickelt, mit der es möglich sein soll, daß vorhandene Dimensionierungswissen darzustellen und dem Rechner zugänglich zu machen. Weiterhin wurde ein Mechanismus zur Verarbeitung dieser Regeln entwickelt und implementiert. Dieser Mechanismus (Regelinterpreter) ist das eigentliche Kernstück des geschaffenen Programmsystems. Er soll in der Lage sein, bei Vorhandensein geeigneter Regeln den Entwurfsingenieur zu unterstützen und im Extremfall ganz zu ersetzen.

Zur formalen Beschreibung von Dimensionierungsaufgaben wurde eine Beschreibungssprache entwickelt, mit der sich beliebige Dimensionierungsaufgaben beschreiben lassen. Diese Sprache unterstützt eine einfache Dateischnittstelle, die die Verbindung zu ausführbaren Programmen herstellt, mit deren Hilfe das Verhalten der zu untersuchenden Systeme modelliert wird. Dabei ist unerheblich, ob diese Modellierung analytisch oder simulativ erfolgt.

Da an der Arbeit kein Entwurfsingenieur beteiligt war, wurde ein mathematisches Optimierungsverfahren implementiert, das als Gütekriterium für die Optimalität der wissensbasierten Lösungen dienen soll. Dadurch kann das geschaffene Programm auch zur 'numerischen' Lösung von Dimensionierungsaufgaben benutzt werden, ohne daß Regeln formuliert werden müssen.

Die Funktion des Programmes wurde anhand von 8 Beispielen auf der Grundlage eines Central-Server-Modelles getestet. Dabei zeigte sich, daß bereits relativ kleine Regelmengen geeignet sind, einfachere Aufgaben schnell und relativ genau zu lösen. Aus Zeitmangel konnte das Programm nicht anhand von komplexeren Beispielen getestet werden. Deshalb können insbesondere nur Vermutungen darüber geäußert werden, wie sich das Programm beim Verarbeitung großer Regelmengen (z.B. mehr als 100 Regeln) verhält. Aus der Literatur ist ersichtlich, daß bei großen Regelmengen die Effizienz der Regelverarbeitung stark abnimmt, wenn nicht spezielle Verarbeitungsverfahren angewendet werden. Auf eine Implementation solcher i.allg. recht komplexen Verfahren wurde verzichtet.

In der vorliegenden Arbeit wurde besonders großer Wert darauf gelegt, den Entwicklungsprozeß des Programmsystems deutlich zu machen. Deshalb wurden viele Entwurfsentscheidungen ausführlich besprochen und die Vor- und Nachteile der verschiedenen Alternativen genannt. Damit wurde das Ziel verfolgt, für das bearbeitete Thema eine gewisse Übersicht über die heute verwendeten Verfahren zu geben und so die Arbeit an ähnlichen Projekten zu erleichtern. Auf die Erläuterung spezieller Einzelheiten des Systems, mit denen der Nutzer nur indirekt in Berührung kommt, wurde weitgehend verzichtet. Das lag vor allem daran, daß zum Verständnis dieser Details erhebliche PROLOG-Kenntnisse erforderlich sind. Zu diesen Details gehören zum Beispiel die symbolische Bearbeitung der Restriktionen (um Abhängigkeiten explizit zu machen), der gesamte arithmetische Apparat (der in dieser Form nicht von PROLOG zur Verfügung gestellt wird) und die Realisierung von Syntaxprüfern für die entwickelten Sprachen.

Als das wesentliche Problem des Einsatzes von Regelinterpretern zur Dimensionierung von technischen Systemen wurde nicht die Realisierung dieser Interpreter, sondern die Bereitstellung und Formalisierung des Regelwissens erkannt. Bei diesem Wissen handelt es sich i.a. nicht um 'Lehrbuchwissen' sondern um Erfahrungswissen. Selbst den Personen, die täglich diese

Erfahrungen nutzen, fällt es zumeist schwer, dieses Wissen explizit zu formulieren. Deshalb entscheidet das Maß, zu dem das Wissen des Entwurfsingenieurs wirklich formalisiert werden kann, darüber, wie leistungsfähig die Regelverarbeitung arbeiten und den menschlichen Experten ersetzen kann.

Bei der Beschäftigung mit den Beispielaufgaben und insbesondere bei der Erstellung der verwendeten Regelmenge zeigte sich neben dem primären Nutzen (der Lösung der Aufgabe) auch noch ein sekundärer, eher pädagogischen Nutzen. Dabei handelt es sich um die Erweiterung des Wissens des Nutzers über das zu betrachtende System und sein Verhalten. Dadurch, daß der Nutzer gezwungen ist, sein Wissen über das System explizit zu formulieren und bei unbefriedigenden Ergebnissen nach den Ursachen zu suchen, gewinnt er eine größere Einsicht in das konkrete Aufgabengebiet. Durch Aufstellen von Hypothesen (in Form von Regeln) und anschließendem Test mit Hilfe des Programmes kann der Nutzer seine Vermutungen auf ihre Richtigkeit überprüfen.

Die für die Zukunft vorstellbaren Entwicklungsmöglichkeiten lassen sich grob in funktionelle und nichtfunktionelle Erweiterungen einteilen. Dabei sollen unter funktionellen Erweiterungen solche verstanden werden, die die implementierten Lösungsverfahren verbessern bzw. die Menge der lösbaren Aufgaben erweitern, während nichtfunktionelle Erweiterungen sich vor allem auf Fragen der Benutzungsfreundlichkeit beziehen.

Als funktionelle Erweiterungen wären denkbar:

- Verwendung eines anderen mathematischen Optimierungsverfahrens zur 'numerischen' Lösung der Dimensionierungsaufgabe
- Verbesserung des Regelverarbeitungs-Mechanismus im Hinblick auf die effektivere Verwendung großer Regelmengen
- Implementation eines 'schnelleren' Kommunikationsmechanismus zum Modellprogramm (z.B. über Pipes)

Als nichtfunktionelle Erweiterungen wären denkbar:

- Verwendung eines Fenstersystems (z.B. Motif) für die Bedienoberfläche
- bessere Erklärungsfähigkeit des Inferenzprozesses
  (z.B. 'Warum wurde Regel x nicht angewendet !')
- Unterstützung des Nutzer bei der Formulierung von Regeln

Zusammenfassend kann gesagt werden, daß sich das entstandene Programmsystem gut als Prototyp eines komplexeren und leistungsfähigeren Systemes eignet, da man an ihm die Auswirkungen der einzelnen getroffenen Entwurfsentscheidungen gut beobachten kann. Dabei kann an dem vorliegenden 'kleinen' System auch getestet werden, welches Wissen sich überhaupt in Regelform darstellen läßt und welche Sprachkonstrukte dazu erforderlich sind. In diesem Sinne empfiehlt sich vor allem ein Einsatz in Forschung und Lehre.

Literaturverzeichnis

[Ber89] Bergholz, G.: Leistungsmodellierung von Rechnersystemen. Akademie-Verlag, Berlin, 1989.

[Bie90] Bierig, O.: Konzeption eines Systems für die regelbasierte Wissensverarbeitung. Diplomarbeit, TU Karl-Marx-Stadt, Sektion Informatik, 1990.

[Bol89] Bolch, G.: Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle. Teubner, Stuttgart, 1989.

[Bot91] Bothe, K.; Stojanow, S.: Praktische Prolog-Programmierung. Verlag Technik GmbH, Berlin, 1991.

[Böh88] Böhringer, B.; Chiopris, C.; Futo, I.: Wissensbasierte Systeme mit Prolog. Addison-Wesley, Bonn, 1988.

[Bra92] Brandt, S.: Datenanalyse. BI Wissenschaftsverlag, Mannheim, 1992.

[Bro86] Brooking, A.G.: Japan's Fifth Generation Project. In: [Mit86], S. 27-33.

[Buc84] Buchanan, B.G.; Shortliffe, E.H. (eds.): Rule-Based Expert Systems. Addison Wesley, Reading, Mass., 1984.

[Cer90] Ceri, S.; Gottlob, G.; Tanca, L.: Logic Programming and Databases. Springer-Verlag, Berlin, 1990.

[Cha85] Charniak, E.; McDermott, D.: Introduction to Artificial Intelligence. Addison Wesley, Reading, Mass., 1985.

[Dav84] Davis, R.; King, J.J.: The Origin of Rule-Based Systems in AI. In: [Buc84], S.20-52.

[Dav89] Davenport, J.H.; Siret, Y.; Tournier, E.: Computer Algebra - Systems and Algorithms for Algebraic Computation. Academic Press 1989.

[Die91] Diepenbrock, F.-R.: Prolog-Werkzeugkasten. Carl Hanser Verlag, München, 1991.

[Dod90] Dodd, T.: Prolog: a logical approach. Oxford University Press, Oxford, 1990.

[Elz86] Elzas, M.S.: The kinship between Artificial Intelligence, Modelling & Simulation: An Appraisal. In: Elzas, M.S.; Ören, T.I.; Zeigler, B.P. (eds.): Modelling and Simulation Methodology in the Artificial Intelligence Era. Elsevier Science Publishers B.V., North-Holland, 1986.

[Fis91] Fishwick, P.A.; Modjeski, R.B. (eds.): Knowledge-Based Simulation: Methodology and Application. Springer Verlag, New York, 1991.

[Fli87] Flitman, A.M.; Hurrion, R.D.: Linking Discrete-Event Simulation Models with Expert Systems. Journal of the Operational Research Society, Vol. 38, No. 8, 1987, S. 723-733.

[For87] Ford, N.: How Machines Think - A general introduction to Artificial Intelligence. John Wiley & Sons, Chichester, 1987.

[For82] Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. Artificial Intelligence, 19(1982), S. 17-37.

[Fri90] Friedrich, C.: Basiskonzepte der Wissensverarbeitung für Expertensysteme. TU Chemnitz, Wissenschaftliche Schriftenreihe, Heft 8/1990.

[Gen89] Genesereth, M.R.; Nilsson, N.J.: Logische Grundlagen der Künstlichen Intelligenz. Vieweg, Braunschweig, 1989.

[Gob89] Goble, T.: Structured Systems Analysis Through Prolog. Prentice Hall 1989.

[Got90] Gottlieb, G.; Frühwirth, Th.; Horn, W. (Hrsg.): Expertensysteme. Springer-Verlag, Wien, 1990.

[Hei91] Hein, M.: Effizientes Lösen von Konfigurierungsaufgaben. Dissertation, Technische Universität Berlin, Fachbereich Systemanalyse und EDV, 1991.

[Hel91] Helbig, H.: Künstliche Intelligenz und Automatische Wissensverarbeitung. Verlag Technik GmbH, Berlin, 1991.

[Hes80] Hestenes, M.: Conjugate Direction Methods in Optimization. Springer Verlag, New York, 1980.

[Jac87] Jackson, P.: Expertensysteme - Eine Einführung. Addison-Wesley, Bonn, 1987.

[Jan89] Janson, A.: Expertensysteme und Prolog: Der sichere Weg zum eigenen Programm. Franzis-Verlag, München, 1989.

[Kow86] Kowalik, J.S. (ed.): Coupling Symbolic and Numerical Computing in Expert Systems. North-Holland, Amsterdam, 1986.

[Kru76] Krug, W.; Schönfeld, S.; Wolf, C.-D.: Programmsysteme zur rechnergestützten Optimierung von Konstruktionen. Kammer der Technik, Karl-Marx-Stadt, 1976.

[Kus89] Kusiak, A.; Heragu, S.S.: Expert Systems and Optimization. In: IEEE Transactions on Software Engineering, Vol. 15, No. 8, August 1989.

[Leh89] Lehmann, C.M.: Wissensbasierte Unterstützung von Konstruktionsprozessen. Carl Hanser Verlag, München, 1989.

[Len83] Lenat, D.B.: EURISKO: A Program That Learns New Heuristics And Domain Concepts. Artificial Intelligence 21(1983), S. 61-98.

[Loc91] Lock, H.; Martins, A.: Issues in the Implementation of Prolog and their Optimization. GMD-Studien Nr. 187, 1991.

[Lou86] Lounamaa, P.; Tse, E.: The Simulation and Expert Environment. In: [Kow86], S. 47-57.

[Lus90] Lusti, M.: Wissensbasierte Systeme: Algorithmen, Datenstrukturen und Werkzeuge. BI Wissenschaftsverlag, Mannheim, 1990.

[Mar90] Marshall, G.: Advanced student's guide to expert systems. Heinemann Professional Publishing Ltd., Halley Court, 1990.

[Mel89] Mellichamp, J.M.; Park, Y.H.: A Statistical Expert System for Simulation Analysis. Simulation 52(1989)4, S. 134-139.

[Mer92] Merkuryev, Y.A.; Visipkov, V.L.: Two-Stage Optimization of Simulation Models. Proceedings of the 1992 European Simulation Symposium, Dresden, Germany, November 5-8, 1992, S. 480-484.

[Mer89] Merritt, D.: Building Expert Systems in Prolog. Springer-Verlag, New York, 1989.

[Mit86] Mitra, G. (ed.): Computer Assisted Decision Making. North-Holland, Amsterdam, 1986.

[Neu88] Neumann, G.: Meta-Programmierung und Prolog. Addison-Wesley, Bonn, 1988.

[OKe86] O'Keefe, R.: Simulation and Expert Systems - A Taxonomy and some Examples. Simulation 46(1986)1, S. 10-16.

[Pea84] Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison Wesley, Reading, Mass., 1984.

[Pin89] Pinto, I. C.: Wissensbasierte Unterstützung bei der Lösung von Optimierungsaufgaben. Dissertation, Universität Dortmund, Fachbereich Informatik, 1989.

[Pos90] Posthoff, C.; Schubert, S.; Rätz, D.: Einführung in die Programmierung mit Prolog. TU Karl-Marx-Stadt, Wissenschaftliche Schriftenreihe, Heft 1/1990.

[Sai90] Saint-Dizier, P.: An Introduction to Programming in Prolog. Springer Verlag, New York, 1990.

[Sch89] Schöning, U.: Logik für Informatiker. BI Wissenschaftsverlag, Mannheim, 1989.

[Sch90] Schwarze, G.: Digitale Simulation. Akademie-Verlag, Berlin, 1990.

[Sim86] Simmons, M.K.; Dixon, J.R.: Reasoning about Quantitative Methods in Engineering Design. In: [Kow86], S. 47-57.

[Sol89] Soltysiak, R.: Wissensbasierte Prozeßregelung. Oldenbourg Verlag, München, 1989.

[Ste88] Sterling, L.; Shapiro, E.: Prolog - Fortgeschrittene Programmiertechniken. Addison-Wesley, Bonn, 1988.

[Sto91] Stoyan, H.: Programmiermethoden der Künstlichen Intelligenz - Band 2. Springer-Verlag, Berlin, 1991.

[Swi90] Swift, K.G.: Fertigungsgerechter Entwurf mit Expertensystemen. VCH Verlagsgesellschaft mbH, Weinheim, Deutschland, 1990.

[Thu89] Thuy, N.; Schnupp, P.: Wissensverarbeitung und Expertensysteme. Oldenbourg Verlag, München, 1989.

[Whi70] White, R.C.: A Survey of Random Methods for Parameter Optimization. Eindhoven University of Technology, Netherlands, Department of Electrical Engineering, TH-Report 70-E-16, 1970.

[Wid89] Widman, L.E.; Loparo, K.A.; Nielsen, N.R. (eds.): Artificial Intelligence, Simulation and Modeling. John Wiley & Sons, New York, 1989.

[Wit92] Wittmann, J.: The Linkage of Simulation and Optimization by the SIMPLEX II Experiment Description Language. Proceedings of the 1992 European Simulation Symposium, Dresden, Germany, November 5-8, 1992, S. 470-474.

[Yam92] Yampolsky, L.S.; Yampolsky, S.L.; Lavrov, A.A.: Elements of Intelligent Control of Simulation Experiments. Proceedings of the 1992 European Simulation Symposium, Dresden, Germany, November 5-8, 1992, S. 600-604.

Verzeichnis der Abbildungen

Abbildung 2.1: Phasen der Auftragsabwicklung Seite 4

Abbildung 3.1: Aufbau eines wissensbasierten Systems Seite 12

Abbildung 3.2: Beispiel für eine Teilmengenhierarchie Seite 18

Abbildung 3.3: Beispiel eines semantischen Netzes Seite 20

Abbildung 4.1: Graphische Darstellung der CRS-Methode Seite 40

Abbildung 6.1: Darstellung der Maschinenhalle Seite 62

Anhang

A. Syntax der Dimensionierungsaufgaben-Beschreibungssprache DABS

<aufgabenbeschreibung> ::= <datenobjekt-block>
[<konstanten-block>]
<parameter-block>
<zielfunktions-block>
[<nebenbedingungs-block>]
<schnittstellen-block> .

<datenobjekt-block> ::= **datenobjekte.**
[<datenobjekt-definition>]$^{+}$.

<datenobjekt-definition> ::= <datenobjekt> **::** <datentyp> **.** .

<datenobjekt> ::= <bezeichner> .

<bezeichner> ::= <kleinbuchstabe> [<buchstabe> | <ziffer> | <unterstrich>]* .

<datentyp> ::= **skalar** |
vektor(<pos. ganze zahl>**)** |
matrix(<pos. ganze zahl>,<pos. ganze zahl>**)** .

<konstanten-block> ::= **konstanten.**
[<konstanten-definition>]$^{+}$.

<konstanten-definition> ::= <datenbeschreibung> **::** <wertliste> **.** .

<datenbeschreibung> ::= <datenindizierung> |
<datenbereich>.

<datenindizierung> ::= <datenobjekt> |
<datenobjekt> : <index> |
<datenobjekt> : <index> : <index> .

<index> ::= <pos. ganze zahl>.

<datenbereich> ::= <datenobjekt> : <bereich> |
<datenobjekt> : <bereich> : <index> |
<datenobjekt> : <index> : <bereich> |
<datenobjekt> : <bereich> : <bereich> .

<bereich> ::= <pos. ganze zahl> **..** <pos. ganze zahl> .

<wertliste> ::= <vektor> | **[** <vektor> [**,** <vektor>]* **]** .

<vektor> ::= **[** <zahl> [**,** <zahl>]* **]** .

<parameter-block> ::= **parameter.**
[<parameter-definition>]+ .

<parameter-definition> ::= <datenbeschreibung> **::** <datenart> **::** <wertliste> **.** .

<datenart> ::= **bereich**(<pos. ganze zahl>**,**<pos.ganze zahl>**,**<datentyp>**)** .

<datentyp> ::= **integer** | **real** .

<zielfunktions-block> ::= zielfunktion.
<zielfunktions-definition>.

<zielfunktions-definition> ::= <zielfunktion> **::** <opt_kriterium> **.** .

<zielfunktion> ::= <term>.

<term> ::= <zahl> | <datenindizierung> |
sum(<bezeichner>**=**<bereich>**,**<term>**)** |
prod(<bezeichner>**=**<bereich>**,**<term>**)** |
min(<term>**,**<term>**)** | **max**(<term>**,**<term>**)** |
round(<term>**)** | **trunc**(<term>**)** |
abs(<term>**)** | **exp**(<term>**)** |
log(<term>**)** | **sqrt**(<term>**)** |
<term> **+** <term> | <term> **-** <term> |
<term> ***** <term> | <term> **/** <term> |
<term> **//** <term> | <term> **div** <term> |
<term> **mod** <term> | <term> ****** <term> .

<opt_kriterium> ::= **maximum** | **minimum** .

<nebenbedingungs-block> ::= **nebenbedingungen.**
[<nebenbedingungs-definition>]+ .

<nebenbedingungs-definition> ::= <vergleich> .

<vergleich> ::= <term> <v_op> <term> **.** .

<v_op> ::= **=** | **<>** | **>** | **<** | **>=** | **<=** .

<schnittstellen-block> ::= **schnittstelle.**
<schnittstellen-definition> .

<schnittstellen-definition> ::= <dateiname> **::** <e_liste> **::** <a_liste> **.** .

<dateiname> ::= <bezeichner>.

<e_liste> ::= **[** <datenobjekt> [, <datenobjekt>]* **]** .

<a_liste> ::= **[** <datenobjekt> [, <datenobjekt>]* **]** .

Bedeutungen: Worte bzw. Zeichen in Fettschrift gehören zur Sprache

| - zeigt alternative Möglichkeiten an

[...] - null- oder einmal
[...]$^{+}$ - beliebig oft, aber mindestens einmal
[...]* - beliebig oft

B. Syntax der Regeldarstellungs-Sprache RDS

<regelmenge> ::= [<regel-definition>]$^{+}$.

<regel-definition> ::= <regelname> **::** <situation> **::** <aktionsliste> **.** .

<regelname> ::= <bezeichner> .

<situation> ::= **[ ]** | **[** <bedingung> [, <bedingung>]* **]** .

<bedingung> ::= <prolog-variable> **<-** **quote(**<ausdruck>**)** |
<prolog-variable> **<-** <term> |
<eigenschaft> **von** <datenindizierung> **ist** <ausdruck> |
<funktion> **von** <ausdruck> **mit** <situation> **ist** <ausdruck> |
<datenindizierung> **ist** <ausdruck> |
<globale variable> **ist** <ausdruck> |
<spezialprädikat> **ist** <ausdruck> |
<vergleich> .

<ausdruck> ::= <prolog-variable> | <bezeichner> | <term> .

<eigenschaft> ::= **art** | **untere_grenze** | **obere_grenze** | **datentyp** .

<funktion> ::= **summe** | **maximum** | **minimum** .

<globale variable> ::= <bezeichner> .

<spezialprädikat> ::= **zielfunktion** | **opt_kriterium** .

<aktionsliste> ::= **[ ]** | **[** <aktion> [**,** <aktion>]* **]** .

<aktion> ::= <prolog-variable> **<-** <ausdruck> |
<datenindizierung> **<-** <ausdruck> |
<globale variable> **<-** <ausdruck> |
erzeuge(<globale variable>**)** |
loesche(<globale variable>**)** |
berechne_zielfunktion .

Bemerkung: Alle nicht näher erklärten Sprachkonstrukte entsprechen den gleichnamigen Konstrukten von DABS.

C. Quelltexte des Programmsystems

Das in PROLOG implementierte Programmsystem besteht aus einer Anzahl von Modulen, die spezielle Funktionen bereitstellen. Jedes Modul kann PROLOG-Prädikate an die 'Umwelt' exportieren bzw. aus anderen Modulen importieren. Die zum Programm gehörenden Module sind bzgl. ihrer Import/Export-Beziehungen hierarchisch geordnet und bilden folgende Hierarchieklassen:

- Spracherweiterungen für PROLOG
  - Modul **var2** stellt spezielle arithmetische Operationen zur Verfügung
  - Modul **util2** stellt Listenfunktionen und allgemeine I/O-Funktionen zur Verfügung
  - Modul **zustand** stellt ein Konzept für globale Variablen zur Verfügung
- Grundfunktionen des Programmes
  - Modul **aufgabe** verwaltet geladene Dimensionierungsaufgaben
  - Modul **regel** verwaltet die Regelmenge und stellt den Regelinterpreter zur Verfügung
  - Modul **gl_manip** stellt Funktionen zur Gleichungsmanipulation zur Verfügung
  - Modul **i_face2** stellt eine Dateitransfer-Schnittstelle zur Verfügung
  - Modul **fehler** stellt Funktionen zur Fehleranzeige zur Verfügung
  - Modul **hilfe** stellt Funktionen zur Online-Hilfe zur Verfügung
- Kommandos der Benutzeroberfläche
  - Modul **datei** stellt Funktionen zur Dateiarbeit zur Verfügung
  - Modul **dimens** stellt Funktionen zur Aufgabenlösung zur Verfügung
  - Modul **anzeige** stellt spezielle Ausgabefunktionen zur Verfügung
  - Modul **experte** stellt den Kommandointerpreter des Programmes zur Verfügung

Das Startprogramm **dim_experte** dient zum Laden der Module in die PROLOG-Umgebung. Nachdem alle Module ordnungsgemäß geladen sind, wird das Programmsystem automatisch gestartet.

Das Modul var2

```
/* Modul mit Funktionen zur phys. Erzeugung und zur Arbeit  mit 'globalen Variablen'   v2.0 */
/*                                                                                          */
/* Sven Hader, 01REA88,  18.2.1993  ..  30.3.1993                                            */

:- module(var2).

:- export([ create_skalar/2, create_vektor/3, create_matrix/4, delete_gvar/1, delete_owner/1,
            get_vartyp/2, get_varwert/2, set_varwert/2, set_dependency/2, get_dependency/2,
            delete_dependency/1, delete_dependency/0, lese_vektor_i/3, lese_matrix_ij/4,
            inc/1, inc/2, dec/1, dec/2, bExpand/6, randomize/1, (<-)/2, (#<)/2, (#<=)/2, (#=)/2, (#>=)/2, (#>)/2, (#<>)/2]).

:- op(100,yfx,:).
:- op(90,xfx,..).
:- op(400,yfx,div).
:- op(600,xfx,<-).
:- op(600,xfx,#=).
:- op(600,xfx,#<>).
:- op(600,xfx,#<).
:- op(600,xfx,#<=).
:- op(600,xfx,#>).
:- op(600,xfx,#>=).

:- body(var2).
```

```
:- dynamic([wert/4, dependency/2]).

:- init(init_var).


/* initialisiert das Modul */

init_var :-    abolish(wert/4), abolish(dependency/2),
               assertz(wert(randseed_S,skalar,1.0,var)), !.


/* physisches Erzeugen globaler Variablen */

/*** fuer globale Groessen */

create_skalar(VarName,Owner) :-      ( wert(VarName,_,_,_),
                                        retract(wert(VarName,_,_,_)) ;
                                      true ),
                                     assertz(wert(VarName,skalar,0,Owner)), !.

/*** fuer Vektoren */

create_vektor(VarName,N,Owner) :-  integer(N), N > 0,
                                     ( wert(VarName,_,_,_),
                                        retract(wert(VarName,_,_,_)) ;
                                      true ),
                                     erzeuge_vektor(N,V),
                                     assertz(wert(VarName,vektor(N),V,Owner)), !.

/***** Erzeugen eines Vektors (Liste) mit N Nullen */

erzeuge_vektor(N,V) :- ev(N,[],V).

ev(0,V,V).
ev(I,V1,V2) :- I1 is I-1, ev(I1,[0|V1],V2).

/*** fuer Matrizen */

create_matrix(VarName,Z,Sp,Owner) :-    integer(Z), Z > 0,  integer(Sp), Sp > 0,
                                        ( wert(VarName,_,_,_),
                                            retract(wert(VarName,_,_,_)) ;
                                          true ),
                                        erzeuge_matrix(Z,Sp,M),
                                        assertz(wert(VarName,matrix(Z,Sp),M,Owner)), !.

/***** Erzeugen einer Matrix (Liste von Listen) mit Z*Sp Nullen */

erzeuge_matrix(Z,Sp,M) :- em(Z,Sp,[],M).

em(0,_,M,M).
em(I,Sp,M1,M2) :- ev(Sp,[],V), I1 is I-1, em(I1,Sp,[V|M1],M2).


/* physisches Zerstoeren einer globalen Variablen */

delete_gvar(VarName) :- ( retract(wert(VarName,_,_,_)) ;  true ), !.


/* physisches Zerstoeren aller globalen Variablen mit  bestimmtem Besitzertyp */

delete_owner(Owner) :- ( retract(wert(_,_,_,Owner)), fail ;  true ), !.


/* liefert den Typ der angegebenen globalen Variable */

get_vartyp(VarName,VarTyp) :- wert(VarName,VarTyp,_,_).


/* liefert die Wertliste der angeg. globalen Variable */

get_varwert(VarName,Wert) :-   get_vartyp(VarName,Typ),
                               eintragen_abhaengigkeiten(VarName,Typ),
                               wert(VarName,_,Wert,_).

/*** berechnet die fuer diese Variable gespeicherten */
/*** Abhaengigkeiten und traegt sie in Wertliste ein */

eintragen_abhaengigkeiten(VarName,skalar) :-     dependency(VarName,Funktion),
                                                 VarName <- Funktion.
eintragen_abhaengigkeiten(VarName,vektor(_)) :- dependency(VarName:I,Funktion),
                                                    VarName:I <- Funktion,  fail.
eintragen_abhaengigkeiten(VarName,matrix(_,_)) :-   dependency(VarName:I:J,Funktion),
                                                    VarName:I:J <- Funktion,  fail.
eintragen_abhaengigkeiten(_,_).


/* aendert die Wertliste der angeg. globalen Variable */
```

```
set_varwert(VarName,Wert) :-    nonvar(Wert),
                                retract(wert(VarName,Typ,_,Owner)),
                                assertz(wert(VarName,Typ,Wert,Owner)), !.


/* Setzen einer Abhaengigkeitsbeziehung */

set_dependency(DObj,Funktion) :-  assertz(dependency(DObj,Funktion)).


/* liefert eine Abhaengigkeitsbeziehung */

get_dependency(DObj,Funktion) :-  dependency(DObj,Funktion).


/* loescht eine Abhaengigkeitsbeziehung */

delete_dependency(DObj) :-  retract(dependency(DObj,_)).


/* loescht alle Abhaengigkeitsbeziehungen */

delete_dependency :- abolish(dependency/2).


/* Ergibtanweisung '<-' fuer Ausdruecke mit globalen Variablen  (vergleichbar mit is/2) */

(Var <- Term) :- berechneTerm(Term,Wert), erg_op(Var,Wert), !.

/*** physisches Veraendern des Wertes einer globalen Variablen */

erg_op(Var,Term) :- var(Var), Var = Term, !.

erg_op(Var:I:J,Term) :-     berechneTerm(I,W1), berechneTerm(J,W2),
                            retract(wert(Var,matrix(Z,Sp),MAlt,Owner)),
                            schreibe_matrix_ij(MAlt,W1,W2,Term,MNeu),
                            assertz(wert(Var,matrix(Z,Sp),MNeu,Owner)), !.
erg_op(Var:I,Term) :-       berechneTerm(I,W),
                            retract(wert(Var,vektor(N),VAlt,Owner)),
                            schreibe_vektor_i(VAlt,W,Term,VNeu),
                            assertz(wert(Var,vektor(N),VNeu,Owner)), !.
erg_op(Var,Term) :-             retract(wert(Var,skalar,_,Owner)),
                            assertz(wert(Var,skalar,Term,Owner)), !.

/*** Berechnung von Termen mit globalen Variablen */

berechneTerm(Term,Wert) :- bT(Term,L), Wert is L, !.

bT(N,N) :- number(N).

bT(Term,Wert) :-    Term =.. [Op,Term1,Term2],
                    member(Op,['+','-','*','/','//',div,mod,'**']),
                    ( Op == div, OpN = '//' ;  OpN = Op ),
                    bT(Term1,Wert1),  bT(Term2,Wert2),
                    Wert =.. [OpN,Wert1,Wert2].

bT(DObj,Wert) :-    dependency(DObj,Funktion),  !, bT(Funktion,Wert).

bT(X:I:J,Wert) :-   !, wert(X,matrix(_,_),M,_),
                    berechneTerm(I,W1), berechneTerm(J,W2),
                    lese_matrix_ij(M,W1,W2,Wert).
bT(X:I,Wert) :-     !, wert(X,vektor(_),V,_),
                    berechneTerm(I,W),
                    lese_vektor_i(V,W,Wert).

bT(sum(I=N1..N2,Term),Wert) :-          !, berechneTerm(N1,UG),
                                    berechneTerm(N2,OG),
                                    bExpand(+,I,UG,OG,Term,Ausdruck),
                                    bT(Ausdruck,Wert).

bT(prod(I=N1..N2,Term),Wert) :-     !, berechneTerm(N1,UG),
                                    berechneTerm(N2,OG),
                                    bExpand(*,I,UG,OG,Term,Ausdruck),
                                    bT(Ausdruck,Wert).

bT(min(Term1,Term2),Wert) :-        !, berechneTerm(Term1,Wert1),
                                    berechneTerm(Term2,Wert2),
                                    ( Wert1 < Wert2, Wert = Wert1 ;  Wert = Wert2 ).

bT(max(Term1,Term2),Wert) :-    !, berechneTerm(Term1,Wert1),
                                    berechneTerm(Term2,Wert2),
                                    ( Wert1 > Wert2, Wert = Wert1 ; Wert = Wert2 ).

bT(round(Term),Wert) :-     !, berechneTerm(Term,Zahl), Wert is round(Zahl).

bT(trunc(Term),Wert) :-     !, berechneTerm(Term,Zahl), Wert is truncate(Zahl).
```

```
bT(abs(Term),Wert) :-          !, berechneTerm(Term,Zahl), Wert is abs(Zahl).

bT(exp(Term),Wert) :-          !, berechneTerm(Term,Zahl), Wert is exp(Zahl).

bT(log(Term),Wert) :-          !, berechneTerm(Term,Zahl), Wert is log(Zahl).

bT(sqrt(Term),Wert) :-         !, berechneTerm(Term,Zahl), Wert is sqrt(Zahl).

bT(random(OG),Wert) :-       !, ( integer(OG), Wert <- trunc(random*(OG+1)) ;
                              real(OG), Wert <- random*OG ), !.

bT(random,Wert) :-  !, RS <- randseed_S,
                    K is truncate(RS) // 127773,
                    RS1 is 16807.0 * (RS-K*127773.0) - K*2836.0,
                    ( RS1 >= 0.0,
                       randseed_S <- RS1,
                      Wert is RS1 / 2147483647.0 ;
                     RS2 is RS1 + 2147483647.0,
                       randseed_S <- RS2,
                      Wert is RS2 / 2147483647.0 ), !.

bT(X,Wert) :- wert(X,skalar,Wert,_).

/***** Zugriffsfunktionen auf Elemente von Vektoren und Matrizen */

/******* i-tes Element eines Vektors (Liste) lesen */

lese_vektor_i([H|_],1,H) :- !.
lese_vektor_i([_|T],I,E) :- I1 is I-1, lese_vektor_i(T,I1,E), !.

/******* i-tes Element eines Vektors (Liste) schreiben */

schreibe_vektor_i(VAlt,I,E,VNeu) :-   svi(VAlt,I,VVor,VNach), append(VVor,[E|VNach],VNeu), !.

svi([_|Rest],1,[],Rest).
svi([A|Rest],I,[A|VVor],VNach) :- I1 is I-1,  svi(Rest,I1,VVor,VNach).

/******* Element (i,j) einer Matrix (Liste von Listen) lesen */

lese_matrix_ij(M,I,J,E) :- lese_vektor_i(M,I,E1), lese_vektor_i(E1,J,E), !.

/******* Element (i,j) einer Matrix (Liste von Listen) schreiben */

schreibe_matrix_ij(MAlt,I,J,E,MNeu) :-   lese_vektor_i(MAlt,I,SAlt),
                                          schreibe_vektor_i(SAlt,J,E,SNeu),
                                          schreibe_vektor_i(MAlt,I,SNeu,MNeu), !.

/***** Expandieren eines sum- oder prod-Ausdruckes in einen einfachen */
/***** Ausdruck (Einsetzen der konkreten Werte der Laufvariable)      */

bExpand(Op,I,UG,OG,Term,Ausdruck) :- ersetze(I,UG,Term,TermNeu), UG1 is UG+1,
                                          bExp(Op,I,UG1,OG,Term,TermNeu,Ausdruck), !.

ersetze(Alt,Neu,Alt,Neu).
ersetze(_,_,N,N) :- number(N).
ersetze(_,_,A,A) :- atom(A).
ersetze(Alt,Neu,AltTerm,NeuTerm) :-      AltTerm =.. [F,X1,X2],
                                      ersetze(Alt,Neu,X1,X1Neu),
                                      ersetze(Alt,Neu,X2,X2Neu),
                                      NeuTerm =.. [F,X1Neu,X2Neu].

bExp(_,_,Akt,OG,_,Ausdruck,Ausdruck) :-          Akt > OG.
bExp(Op,I,Akt,OG,Term,AusdruckAlt,Wert) :-   ersetze(I,Akt,Term,TermNeu),
                                             AusdruckNeu =.. [Op,AusdruckAlt,TermNeu],
                                             AktNeu is Akt+1,
                                             bExp(Op,I,AktNeu,OG,Term,AusdruckNeu,Wert), !.


/* Initialisieren des Zufallszahlengenerators */

randomize(Seed) :- number(Seed),  randseed_S <- Seed, !.


/* Inkrement- und Dekrementfunktionen fuer globale Variablen */

inc(DBeschr) :-        DBeschr <- DBeschr + 1, !.
inc(DBeschr,I) :-      DBeschr <- DBeschr + I, !.

dec(DBeschr) :-        DBeschr <- DBeschr - 1, !.
dec(DBeschr,I) :-      DBeschr <- DBeschr + I, !.


/* Vergleichsoperatoren fuer Terme mit globalen Variablen */

(Term1 #< Term2) :-      berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 < Wert2, !.
```

```
(Term1 #> Term2) :-      berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 > Wert2, !.

(Term1 #= Term2) :-      berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 =:= Wert2, !.

(Term1 #<> Term2) :-     berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 =\= Wert2, !.

(Term1 #<= Term2) :-     berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 =< Wert2, !.

(Term1 #>= Term2) :-     berechneTerm(Term1,Wert1), berechneTerm(Term2,Wert2),
                         Wert1 >= Wert2, !.
```

Das Modul util2

```
/* Modul mit Hilfsfunktionen (Ein/Ausgabe usw.)  v2.0 */
/*                                                    */
/* Sven Hader, 01REA88, 19.2.1993  ..  25.3.1993      */

:- module(util2).

:- export([ write_neu/1, readln/1, ja_nein_eingabe/2, string_eingabe/2, menue_eingabe/2, retractall/1,
            list_length/2, dim_matrix/3, dim_vektor/2, insert_vv/4, insert_mm/5, vektor2matrix/2, max_atomlength/2]).

:- meta_predicate([lelement_mit_eigenschaft/3]).

:- body(util2).

:- op(0,xfx,:).    % Loeschen der Systemdefinition von ':'

:- import(var2).

:- op(200,yfx,'::').
:- op(700,xfx,<=).
:- op(700,xfx,<>).


/* write-Fkt. mit beliebig vielen Argumenten (in Liste) sowie  Ersetzung   */
/* globaler Variablen durch ihre Werte (wenn nicht  gequotet)              */

write_neu([]) :-          flush_output.       % Ausgabepuffer leeren
write_neu([H|T]) :-    once(w_neu(H)),        write_neu(T), !.

/*** Ausgabe eines Argumentes der oben beschriebenen Liste */

w_neu(S) :- S = [_|_], atom_codes(A,S), write(A).  % Strings
w_neu(nl) :- nl.
w_neu(tab(X)) :- tab(X).
w_neu(quote(Var)) :- write(Var).
w_neu(Var) :- Wert <- Var, write(Wert).
w_neu(Ausgabe) :- write(Ausgabe).


/* readln-Fkt., die alle bis zum Enter eingegebenen Zeichen als */
/* String interpretiert                                         */

readln(String) :- rln([],String), !.

rln(S1,S2) :-     get_code(B),
                  ( ( B == -1 ; B == 10 ), reverse(S1,S2) ;
                    rln([B|S1],S2) ).


/* Ausgabe einer Liste entspr. write_neu/1 und Einlesen einer  ja-nein-Antwort  */

ja_nein_eingabe(TListe,Antwort) :-    repeat,
                                        write_neu(TListe),
                                        readln(A),
                                        ( member(A,["j","J","ja","Ja","yes","Yes","y","Y"]), Antwort = ja ;
                                          member(A,["n","N","nein","Nein","no","No"]), Antwort = nein ), !.


/* Ausgabe einer Liste entspr. write_neu/1 und Einlesen eines Strings */

string_eingabe(TListe,String) :- write_neu(TListe),  readln(String), !.


/* Ausgabe einer Liste von write_neu-Listen (numeriert) und Einlesen */
/* einer der gegebenen Zahlen                                        */
```

```
menue_eingabe(TTListe,Zahl) :-  menue_ausgabe(TTListe,0,N),
                                repeat,
                                  string_eingabe([nl,"       Auswahl : "],ZString),
                                   number_codes(Zahl,ZString),
                                ( Zahl >= 1, Zahl =< N ), !.

menue_ausgabe([],N,N).
menue_ausgabe([TListe|R],I,N) :-      I1 is I+1,
                                      write_neu(["   ["',I1,"] "|TListe]),
                                      menue_ausgabe(R,I1,N), !.


% ------------------------ Listenfunktionen --------------------------


/* bestimmt die Anzahl Elemente einer Liste */

list_length(Liste,N) :- ll(Liste,0,N), !.

ll([],N,N).
ll([_|T],I,N) :- I1 is I+1, ll(T,I1,N).


/*** bestimmt die Anzahl Elemente (müssen Zahlen sein) in einer eindim. Liste */

anz_elem(Liste,Anz) :- ae(Liste,0,Anz).

ae([],Anz,Anz).
ae([H|T],I,Anz) :- number(H), I1 is I+1, ae(T,I1,Anz).


/* Dimension der angeg. Wertmatrix */

dim_matrix(Mat,I,J) :- dim_liste(Mat,I,J), !.


/* Dimension des angeg. Wertvektors */

dim_vektor(Vek,I) :- dim_liste(Vek,1,I), !.


/*** ermittelt die Dimension einer Liste von Zahlen (Vektor) bzw. */
/*** von Listen von Zahlen (Matrix)                               */

dim_liste([H|T],Z,Sp) :-      H = [_|_],      % H ist nichtleere Liste
  /* Matrix */                anz_elem(H,Sp),   % Anzahl Zahlen in H
                              d_liste(T,Z1,Sp),
                              Z is Z1+1,!.
dim_liste([H|T],1,Sp) :-      number(H),
  /* Vektor */                anz_elem(T,Sp1),
                              Sp is Sp1+1, !.


/***** ermittelt die Anzahl Unterlisten (bestehend aus Zahlen) */
/***** einer Liste (müssen alle dieselbe Elementanzahl haben)  */

d_liste(Liste,Z,Sp) :- dl(Liste,0,Z,Sp).

dl([],Z,Z,_).
dl([H|T],I,Z,Sp) :- anz_elem(H,Sp), I1 is I+1, dl(T,I1,Z,Sp).


/* fügt in Vektor V1 ab Position Pos den Vektor V2 ein,  Ergebnis ist VNeu  */

insert_vv(VAlt,[],_,VAlt).
insert_vv(VAlt,VIn,Pos,VNeu) :-  list_length(VIn,Le),          % Länge des einzufüg. Vektors
                                 Pos1 is Pos-1,
                                 split_v(VAlt,Pos1,Le,VVor,VNach),  % Teilen des Originalvekt. in Vekt.
                                                                    % vor und Vektor nach Einfügebereich
                                 append(VVor,VIn,VTmp),             % alles wieder zusammenfügen
                                 append(VTmp,VNach,VNeu),!.

split_v(V,Pos,Le,VV,VN) :-    sv(V,0,Pos,[],VV,VTmp), sv(VTmp,0,Le,[],_,VN).

sv(VN,Pos,Pos,VTmp,VV,VN) :- reverse(VTmp,VV).
sv([H|T],I,Pos,VTmp,VV,VN) :- I1 is I+1, sv(T,I1,Pos,[H|VTmp],VV,VN).


/* fügt in Matrix M1 ab Position (Z,Sp) die Matrix M2 ein,  Ergebnis ist MNeu  */

insert_mm(MAlt,[],_,_,MAlt).
insert_mm(MAlt,MIn,Z,Sp,MNeu) :-  list_length(MIn,Le),        % Länge der einzufügenden Matrix
                                  Z1 is Z-1,
                                  sv(MAlt,0,Z1,[],MVor,MTmp),   % MVor - Mat. vor dem Einfügebereich
                                                                % MTmp - Mat. ab dem Einfügebereich
                                  sv(MTmp,0,Le,[],MM,MNach),    % MM - zu ersetzender Bereich
                                                                % MNach - Mat. nach dem Einfügebereich
```

```
                                    im(MM,MIn,Sp,[],MM1),          % Einfügen von MIn ab Spalte Sp in MM
                                    append(MVor,MM1,MM2),          % alles wieder zusammenfügen
                                    append(MM2,MNach,MNeu),!.

im([],[],_,M1,M2) :-  reverse(M1,M2).
im([H1|T1],[H2|T2],Sp,M1,M2) :-       insert_vv(H1,H2,Sp,H3),
                                    im(T1,T2,Sp,[H3|M1],M2).


/*   wandelt einen Vektor in eine Matrix um (jedes Element des     */
/*   Vektors wird Unterliste (Zeile) der Matrix)                   */

vektor2matrix(Vektor,Matrix) :- v2m(Vektor,[],Matrix).

v2m([],M1,M2) :- reverse(M1,M2).
v2m([H|T],M1,M2) :- v2m(T,[[H]|M1],M2).


/*   liefert die Elemente einer Liste, für die die Eigenschaft         */
/*   zutrifft                                                         */
/*   (Eigenschaft muß der Name eines einstell. Prädikates sein)    */

lelement_mit_eigenschaft(Liste,Eigenschaft,NeuListe) @ Modul :-
  lme(Liste,Eigenschaft,Modul,[],NeuListe), !.

lme([],_,_,L,LR) :- reverse(L,LR).
lme([H|T],E,Modul,L1,L) :-   ET =.. [E,H],
                             ( call(ET) @ Modul, lme(T,E,Modul,[H|L1],L) ;
                               lme(T,E,Modul,L1,L) ).


/* findet die Laenge des laengsten Atomes einer Liste von Atomen */

max_atomlength(L,Length) :- max_al(L,0,Length), !.

max_al([],N,N).
max_al([Atom|Rest],NTmp,N) :-  atom_length(Atom,I),
                               ( I > NTmp, max_al(Rest,I,N) ;
                                 I =< NTmp, max_al(Rest,NTmp,N) ).
```

Das Modul zustand

```
/* Modul mit Fkt. zur Verwaltung globaler Zustaende   v1.0 */
/*                                                          */
/* Sven Hader, 01REA88, 24.2.1993                           */

:- module(zustand).

:- export([set_zustand/2, get_zustand/2]).

:- body(zustand).

:- dynamic([system_zustand/2]).

:- init(init_zustand).


/* initialisiert das Modul */

init_zustand :- abolish(system_zustand/2).


/* Speichern eines Zustandes (neu anlegen oder aendern) */

set_zustand(Zustand,Wert) :-       ( retract(system_zustand(Zustand,_)) ;  true ),
                                     assertz(system_zustand(Zustand,Wert)), !.


/* Lesen eines Zustandes (scheitert, wenn nicht def.) */

get_zustand(Zustand,Wert) :- system_zustand(Zustand,Wert).
```

Das Modul aufgabe

```
/* Modul mit Fkt. zur Verwaltung der Aufgabenbeschreibung  v1.0 */
/*                                                               */
/* Sven Hader, 01REA88,  5.3.1993  bis  14.4.1993                */

:- module(aufgabe).
```

```
:- export([ delete_aufgabe/0,  create_datenobjekt/2, delete_datenobjekt/1, get_datenobjekt/2,
            set_konstante/2, get_konstante/1, set_parameter/3, get_parameter_regel/5,
            get_parameter_num/6, set_abhaengige/3, get_abhaengige/4, set_ergebnis/1, get_ergebnis/1,
            set_zielfunktion/2, get_zielfunktion/2, set_nebenbedingung/1, get_nebenbedingung_regel/3,
            get_nebenbedingung_num/3,  pruefe_nb_num/1, pruefe_nb_regel/1,
            set_schnittstelle/3, get_schnittstelle/3, bearbeite_aufgabe/0, zeige_aufgabe/0]).

:- body(aufgabe).

:- op(0,xfx,:).

:- import(var2).
:- import(util2).
:- import(zustand).
:- import(fehler).
:- import(gl_manip).

:- dynamic([ datenobjekt/2, constant/1, parameter_num/6, parameter_regel/5, abhaengige/4, ergebnis/1,
             zielfunktion/2, nebenbedingung_regel/3, nebenbedingung_num/3, schnittstelle/3 ]).

:- init(init_aufgabe).


/* initialisiert das Modul */

init_aufgabe :-   create_skalar(anz_param_r_S,modell),   % Anzahl Param. fuer rb. Dim.
                  create_skalar(anz_param_n_S,modell),   % Anzahl Param. fuer num. Dim.
                  create_skalar(anz_abhaeng_S,modell),   % Anzahl Abhaengige
                  create_skalar(anz_nbed_r_S,modell),    % Anzahl Nebenbed. (regel)
                  create_skalar(anz_nbed_n_S,modell),    % Anzahl Nebenbed. (num.)
                  delete_aufgabe.


/* loescht die augenblicklich gespeicherte Aufgabenbeschreibung */

delete_aufgabe :-     abolish(datenobjekt/2), delete_owner(akt_aufgabe),
                      abolish(constant/1), abolish(parameter_regel/5),
                      abolish(parameter_num/6),  abolish(abhaengige/4),
                      abolish(ergebnis/1),  abolish(zielfunktion/2),
                      abolish(nebenbedingung_regel/3),  abolish(nebenbedingung_num/3),
                      abolish(schnittstelle/3),
                      anz_param_r_S <- 0,  anz_param_n_S <- 0,
                      anz_abhaeng_S <- 0,  anz_nbed_r_S <- 0,
                      anz_nbed_n_S <- 0, !.


/* erzeugt ein Datenobjekt der angegebenen Art */

create_datenobjekt(DObj,skalar) :-          assertz(datenobjekt(DObj,skalar)),
                                            create_skalar(DObj,akt_aufgabe), !.
create_datenobjekt(DObj,vektor(N)) :-       assertz(datenobjekt(DObj,vektor(N))),
                                            create_vektor(DObj,N,akt_aufgabe), !.
create_datenobjekt(DObj,matrix(Z,Sp)) :- assertz(datenobjekt(DObj,matrix(Z,Sp))),
                                            create_matrix(DObj,Z,Sp,akt_aufgabe), !.


/* liefert ein Datenobjekt */

get_datenobjekt(DObj,Art) :- datenobjekt(DObj,Art).


/* loescht das angegebene Datenobjekt */

delete_datenobjekt(DObj) :- retract(datenobjekt(DObj,_)), delete_gvar(DObj), !.


/* definiert das angegebene Datenobjekt oder Teile davon als */
/* Konstante mit dem angegebenen Wert                        */

set_konstante(DObj,Wert) :-  ( DObj = DName:_:_ ;
                               DObj = DName:_ ;
                               DObj = DName ),
                             get_vartyp(DName,DTyp),
                             get_varwert(DName,DWertAlt),
                             set_konst(DObj,Wert,DTyp,DWertAlt,DWertNeu),
                             set_varwert(DName,DWertNeu),
                             assertz(constant(DObj)), !.

set_konst(_:B1:B2,WMatrix,matrix(_,_),WAlt,WNeu) :-          ( B1 = UG1.._,
                                                               ( B2 = UG2.._,
                                                                 insert_mm(WAlt,WMatrix,UG1,UG2,WNeu) ;
                                                                 insert_mm(WAlt,WMatrix,UG1,B2,WNeu) ) ;
                                                               ( B2 = UG2.._,
                                                                 insert_mm(WAlt,WMatrix,B1,UG2,WNeu) ;
                                                                 insert_mm(WAlt,WMatrix,B1,B2,WNeu) ) ).
set_konst(_,WMatrix,matrix(_,_),_,WMatrix).
```

```
set_konst(_:B,WVektor,vektor(_),WAlt,WNeu) :-  ( B = UG.._,
                                                 insert_vv(WAlt,WVektor,UG,WNeu) ;
                                                insert_vv(WAlt,WVektor,B,WNeu) ).
set_konst(_,WVektor,vektor(_),_,WVektor).
set_konst(_,[WSkalar],skalar,_,WSkalar).


/* prueft, ob ein Datenobjekt oder Teil davon eine Konstante ist */

get_konstante(DObj:I:J) :-      constant(DObj),     % Elem. einer Matrix
                                get_datenobjekt(DObj,matrix(Z,Sp)),
                                for(1,I,Z), for(1,J,Sp).
get_konstante(DObj:I:J) :-      constant(DObj:B1:B2),
                                ( B1 = UG1..OG1, for(UG1,I,OG1) ;
                                  integer(B1), for(B1,I,B1) ),
                                ( B2 = UG2..OG2, for(UG2,J,OG2) ;
                                  integer(B2), for(B2,J,B2) ).

get_konstante(DObj:I) :-        constant(DObj),     % Elem. eines Vektors
                                get_datenobjekt(DObj,vektor(N)),
                                for(1,I,N).
get_konstante(DObj:I) :-    constant(DObj:B),
                            ( B = UG..OG,  for(UG,I,OG) ;
                              integer(B), for(B,I,B) ).

get_konstante(DObj) :-      constant(DObj),       % Skalar
                            get_datenobjekt(DObj,skalar).


/* definiert das angegebene Datenobjekt oder Teile davon als  Parameter  */

set_parameter(DObj:UG1..OG1:UG2..OG2,Art,WMatrix) :-
  ( for(UG1,I,OG1),
      for(UG2,J,OG2),
        Z is I-UG1+1, Sp is J-UG2+1,
        lese_matrix_ij(WMatrix,Z,Sp,Wert),
        inc(anz_param_r_S),
        Anz <- anz_param_r_S,
        assertz(parameter_regel(Anz,DObj:I:J,Art,Wert,[])),
        fail ;
    true ), !.
set_parameter(DObj:UG1..OG1:J,Art,WMatrix) :-
  ( for(UG1,I,OG1),
      Z is I-UG1+1,
      lese_matrix_ij(WMatrix,Z,1,Wert),
      inc(anz_param_r_S),
      Anz <- anz_param_r_S,
      assertz(parameter_regel(Anz,DObj:I:J,Art,Wert,[])),
      fail ;
    true ), !.
set_parameter(DObj:I:UG2..OG2,Art,WMatrix) :-
  ( for(UG2,J,OG2),
      Sp is J-UG2+1,
      lese_matrix_ij(WMatrix,I,Sp,Wert),
      inc(anz_param_r_S),
      Anz <- anz_param_r_S,
      assertz(parameter_regel(Anz,DObj:I:J,Art,Wert,[])),
      fail ;
    true ), !.
set_parameter(DObj:I:J,Art,[[Wert]]) :-
   inc(anz_param_r_S),
   Anz <- anz_param_r_S,
   assertz(parameter_regel(Anz,DObj:I:J,Art,Wert,[])), !.

set_parameter(DObj:UG..OG,Art,WVektor) :-
  ( for(UG,I,OG),
      N is I-UG+1,
      lese_vektor_i(WVektor,N,Wert),
      inc(anz_param_r_S),
      Anz <- anz_param_r_S,
      assertz(parameter_regel(Anz,DObj:I,Art,Wert,[])),
      fail ;
    true ), !.
set_parameter(DObj:I,Art,[Wert]) :-
   inc(anz_param_r_S),
   Anz <- anz_param_r_S,
   assertz(parameter_regel(Anz,DObj:I,Art,Wert,[])), !.

set_parameter(DObj,Art,WListe) :-
  ( datenobjekt(DObj,skalar),
      inc(anz_param_r_S),
      Anz <- anz_param_r_S,
      assertz(parameter_regel(Anz,DObj,Art,WListe,[])) ;
    datenobjekt(DObj,vektor(N)),
      set_parameter(DObj:1..N,Art,WListe) ;
    datenobjekt(DObj,matrix(Z,Sp)),
      set_parameter(DObj:1..Z,1..Sp,Art,WListe) ), !.
```

```
/* prueft, ob das angegebene Datenobjekt oder Teil davon ein */
/* Parameter fuer regelbas. Optimierung ist                  */

get_parameter_regel(PName,Nr,Art,Start,NBL) :-  parameter_regel(Nr,PName,Art,Start,NBL).


/* prueft, ob das angegebene Datenobjekt oder Teil davon ein */
/* Parameter fuer num. Optimierung ist                       */

get_parameter_num(PName,Nr,Art,Start,Schritt,NBL) :-  parameter_num(Nr,PName,Art,Start,Schritt,NBL).


/* definiert den angegebenen Parameter als abhaengig und */
/* speichert seine Berechnungsvorschrift                 */

set_abhaengige(PName,Funktion,Art) :-    inc(anz_abhaeng_S), Anz <- anz_abhaeng_S,
                                         assertz(abhaengige(Anz,PName,Funktion,Art)).


/* liefert eine Abhaengige */

get_abhaengige(PName,Nr,Funktion,Art) :-  abhaengige(Nr,PName,Funktion,Art).


/* definiert das angegebene Datenobjekt als Ergebnis */

set_ergebnis(DObj) :- assertz(ergebnis(DObj)).


/* prueft, ob ein Datenobjekt ein Ergebnis ist oder liefert ueber */
/* Backtracking alle Ergebnis-Objekte                             */

get_ergebnis(DObj) :- ergebnis(DObj).


/* definiert die Zielfunktion und das Opt.kriterium der Aufgabe */

set_zielfunktion(ZFkt,OptKrit) :-  assertz(zielfunktion(ZFkt,OptKrit)).


/* liefert die Zielfunktion und das Opt.kriterium der Aufgabe */

get_zielfunktion(ZFkt,OptKrit) :- zielfunktion(ZFkt,OptKrit).


/* definiert eine Nebenbedingung der Aufgabe */

set_nebenbedingung(Ausdruck) :-
  Ausdruck =.. [Op,Term1,Term2],
  ( Op == '#=',                  % Nebenbedingung ist eine Gleichung
      objekte_in_term(Term1,L1),
      objekte_in_term(Term2,L2),
      append(L1,L2,L3),
      liste_zu_quantmenge(L3,QM),    % QM ist Liste aller Objekte in NB
      lelement_mit_eigenschaft(QM,ist_param,PM),
      ( PM == [],                    % keine Parameter in NB
          lelement_mit_eigenschaft(QM,ist_const,CM),  % Liste der Konstanten
          ( QM == CM,                % Ausdruck sofort berechenbar
             ( Ausdruck ;
               !, merke_fehler(224,Ausdruck) ) ;
             set_nbed_regel(Ausdruck),
               set_nbed_num(Ausdruck) ) ;
        ( member([P1,1],PM),         % ein Parameter nur einmal
             ( stelle_um_nach(P1,Ausdruck,P1#=Term) ;
               umstellen_nutzer(P1,Ausdruck,P1#=Term) ),
             get_parameter_regel(P1,_,bereich(UG,OG,_),_,_),
             set_abhaengige(P1,Term,nebenbedingung),
             set_nbed_regel(Ausdruck),
             set_nbed_num(UG #<= Term),
             set_nbed_num(OG #>= Term) ) ;
          set_nbed_regel(Ausdruck),
            set_nbed_num(Ausdruck) ) ;
     set_nbed_regel(Ausdruck),
        set_nbed_num(Ausdruck) ).        % Nebenbedingung ist eine Ungleichung

/*** speichert allg. Nebenbedingung (fuer Regelanwendung) */

set_nbed_regel(NBed) :-      inc(anz_nbed_r_S), Anz <- anz_nbed_r_S,
                             assertz(nebenbedingung_regel(Anz,NBed,dummy)), !.

/*** speichert spezielle Nebenbedingung (fuer num. dim.) */

set_nbed_num(NBed) :-  inc(anz_nbed_n_S), Anz <- anz_nbed_n_S,
                       assertz(nebenbedingung_num(Anz,NBed,dummy)), !.
```

```
/*** Eigenschaftspraedikate */

ist_param([PName,_]) :- get_parameter_regel(PName,_,_,_,_).
ist_const([Const,_]) :- get_konstante(Const).


/* liefert Nebenbedingungen (regel) */

get_nebenbedingung_regel(NBed,Nr,Art) :- nebenbedingung_regel(Nr,NBed,Art).


/* liefert Nebenbedingungen (num.) */

get_nebenbedingung_num(NBed,Nr,Art) :- nebenbedingung_num(Nr,NBed,Art).


/* definiert die Schnittstelle zur Modelldatei */

set_schnittstelle(DName,EParam,AParam) :- assertz(schnittstelle(DName,EParam,AParam)), !.


/* liefert die Schnittstelle */

get_schnittstelle(DName,EParam,AParam) :- schnittstelle(DName,EParam,AParam).


/* bearbeitet die eingelesene Aufgabe, indem es zusaetzlich benoetigte */
/* Datenstrukturen anlegt oder vorhandene  modifiziert                 */

bearbeite_aufgabe :-
  PRAnz <- anz_param_r_S,
  create_vektor(opt_param_r_S,PRAnz,akt_aufgabe),
  create_skalar(opt_zfw_n_S,akt_aufgabe),
  create_skalar(opt_zfw_r_S,akt_aufgabe),

  ( get_parameter_regel(PName,_,bereich(UG,OG,Art),Start,_),
     not get_abhaengige(PName,_,_,_),
     inc(anz_param_n_S),
     Nr <- anz_param_n_S,
     ( Art == real,
        Schritt is (OG-UG)/10.0 ;
       Art == integer,
        Schritt1 is round((OG-UG)/10),
        ( Schritt1 =:= 0, Schritt = 1 ;
          Schritt1 > 0, Schritt = Schritt1 ) ),
     assertz(parameter_num(Nr,PName,bereich(UG,OG,Art),Start,Schritt,[])),
     fail ;
   true ),

  PNAnz <- anz_param_n_S,
  create_vektor(opt_param_n_S,PNAnz,akt_aufgabe),

  registriere_nebenbedingungen, !.


/*** teilt die Nebenbedingungen in waehrend der Parameterbelegung,    */
/*** vor der Berechnung und nach der Berechnung ueberpruefbare        */
/*** ausserdem werden fuer jeden Parameter die zugehoerigen Neben- */
/*** bedingungen ermittelt                                           */
/*** ACHTUNG: erst die gesamte Aufgabe einlesen und erst dann die   */
/***          Nebenbedingungen registrieren                          */
/*** drei Arten von NB: parameter, vor_berechnung, nach_berechnung  */

registriere_nebenbedingungen :- reg_nb_regel,  reg_nb_num, !.

reg_nb_regel :-
  retract(nebenbedingung_regel(Nr,Term,dummy)),   % eine noch unbearbeitete NB loeschen
  objekte_in_term(Term,L),                        % Bestimmen enthaltener DObj.
  liste_zu_quantmenge(L,QM),                      % Quantifizieren
  ermittle_ueb(regel,Nr,QM,[],UebArt),            % Ermitteln der Ueberpruef.art
  assertz(nebenbedingung_regel(Nr,Term,UebArt)),  % bearbeitete NB speichern
  reg_nb_regel.
reg_nb_regel.

reg_nb_num :-
  retract(nebenbedingung_num(Nr,Term,dummy)),   % eine noch unbearbeitete NB loeschen
  objekte_in_term(Term,L),                      % Bestimmen enthaltener DObj.
  liste_zu_quantmenge(L,QM),                    % Quantifizieren
  ermittle_ueb(num,Nr,QM,[],UebArt),            % Ermitteln der Ueberpruef.art
  assertz(nebenbedingung_num(Nr,Term,UebArt)),  % bearbeitete NB speichern
  reg_nb_num.
reg_nb_num.

ermittle_ueb(_,_,[],L,nach_berechnung) :-         % Ergebniswerte benoetigt
   member(erg,L).
ermittle_ueb(regel,Nr,[],[par(PName)],parameter) :-  % nur ein Parameter
  retract(parameter_regel(I,PName,Art,Start,NBL)),
```

```
  append(NBL,[Nr],NBLNeu),
  assertz(parameter_regel(I,PName,Art,Start,NBLNeu)).
ermittle_ueb(num,Nr,[],[par(PName)],parameter) :-  % nur ein Parameter
  retract(parameter_num(I,PName,Art,Start,Schritt,NBL)),
  append(NBL,[Nr],NBLNeu),
  assertz(parameter_num(I,PName,Art,Start,Schritt,NBLNeu)).
ermittle_ueb(_,_,[],_,vor_berechnung).          % mehrere Parameter

ermittle_ueb(Art,Nr,[[Obj,_]|T],L,UebArt) :-
  ( get_konstante(Obj),
      ermittle_ueb(Art,Nr,T,L,UebArt) ;
    (  Art == regel,
       get_parameter_regel(Obj,_,_,_,_),
       ermittle_ueb(Art,Nr,T,[par(Obj)|L],UebArt) ;
     Art == num,
       get_parameter_num(Obj,_,_,_,_,_),
       ermittle_ueb(Art,Nr,T,[par(Obj)|L],UebArt) ) ;
    get_ergebnis(Obj),
      ermittle_ueb(Art,Nr,T,[erg|L],UebArt) ).


/* prueft alle NB fuer die num. Dimensionierung, deren Nummern in */
/* der uebergebenen Liste stehen                                  */

pruefe_nb_num([]) :- !.
pruefe_nb_num([NBNr|T]) :-          nebenbedingung_num(NBNr,NBTerm,_),  NBTerm,  pruefe_nb_num(T), !.


/* prueft alle NB fuer die regelbas. Dimensionierung, deren Nummern */
/* in der uebergebenen Liste stehen                                 */

pruefe_nb_regel([]) :- !.
pruefe_nb_regel([NBNr|T]) :- nebenbedingung_regel(NBNr,NBTerm,_),  NBTerm,  pruefe_nb_regel(T), !.


/* zeigt die augenblicklich geladene Aufgabenstellung */

zeige_aufgabe :-
  get_zustand(ad_geladen,ja),
  write_neu(['   Datenobjekte :',nl,nl]),
  findall(Obj,get_datenobjekt(Obj,_),ObjListe),
  max_atomlength(ObjListe,I1),
  write_datenobjekte(1,ObjListe,I1),
  findall(Konst,get_konstante(Konst),KonstListe),
  ( KonstListe == [],
      write_neu(['   Konstanten :   keine',nl,nl]);
    write_neu(['   Konstanten :',nl,nl]),
      write_konstanten(1,KonstListe,17) ),
  write_neu(['   Parameter :',nl,nl]),
  findall(Param,get_parameter_regel(Param,_,_,_,_),ParamListe),
  write_parameter(1,ParamListe,17),
  findall(Abh,get_abhaengige(Abh,_,_,echt),AbhListe),
  ( AbhListe == [],
      write_neu(['   Abhaengige :   keine',nl,nl]) ;
    write_neu(['   Abhaengige :',nl,nl]),
      write_abhaengige(1,AbhListe,17) ),
  get_zielfunktion(ZFkt,OptKrit),
  write_neu(['   Zielfunktion :',nl,nl,'     ',quote(ZFkt),'  -->  ',quote(OptKrit),nl,nl]),
  findall(NB,get_nebenbedingung_regel(NB,_,_),NBListe),
  ( NBListe == [],  write_neu(['   Nebenbedingungen :   keine',nl,nl]) ;
    write_neu(['   Nebenbedingungen :',nl,nl]),
      write_nebenbedingungen(1,NBListe) ),
  write_neu(['   Schnittstelle :',nl,nl]),
  get_schnittstelle(MDatei,EParam,AParam),
  write_neu([tab(5),'Modelldatei:  ',quote(MDatei),nl, tab(5),'Eingabewerte: ',quote(EParam),nl,
             tab(5),'Ausgabewerte: ',quote(AParam),nl,nl]),!.

zeige_aufgabe :-    get_zustand(ad_geladen,nein),
                    ( merke_fehler(102) ; true ), !.


/*** formatierte Ausgabe der Datenobjekte */

write_datenobjekte(I,[],_) :- nl,  ( I // 2 =:= I / 2 , nl ; true ), !.
write_datenobjekte(I,[H|T],Tab) :-
  get_datenobjekt(H,Art),
  atom_length(H,OL),
  length_art(Art,AL),
  write_neu([tab(5),quote(H),tab(Tab-OL+1),':: ',quote(Art),tab(13-AL)]),
  I1 is I+1,
  ( I // 2 =:= I / 2 , nl ; true ),
  write_datenobjekte(I1,T,Tab), !.

length_art(skalar,6).
length_art(vektor(I),L) :- ( I < 10, L = 9 ; L = 10 ).
length_art(matrix(I,J),L) :- ( I < 10, ( J < 10 , L = 11 ; L = 12 ) ;
                               I > 10, ( J < 10 , L = 12 ; L = 13 ) ).
```

```
/*** formatierte Ausgabe der Konstanten */

write_konstanten(I,[],_) :- nl,  ( (I+2) // 3 =\= (I+2) / 3, nl ; true ), !.
write_konstanten(I,[H|T],Tab) :-
  Wert <- H,
  length_param(H,KL),
  number_codes(Wert,WListe),
  list_length(WListe,WL),
  write_neu([tab(5),quote(H),tab(Tab-KL+1),'= ',Wert,tab(6-WL)]),
  I1 is I+1,
  ( I // 3 =:= I / 3 , nl ;  true ),
  write_konstanten(I1,T,Tab), !.


/*** formatierte Ausgabe der Parameter */

write_parameter(_,[],_) :- nl, !.
write_parameter(I,[H|T],Tab) :-
  get_parameter_regel(H,_,Art,Wert,_),
  length_param(H,PL),
  write_neu([tab(5),quote(H),tab(Tab-PL+1),':: ',quote(Art),'  :: ', Wert,nl]),
  I1 is I+1,
  write_parameter(I1,T,Tab), !.

length_param(Obj:I:J,L) :-   atom_length(Obj,N),
                             ( I < 10, ( J < 10 , L is N+8 ;  L is N+9 ) ;
                               I > 10, ( J < 10 , L is N+9 ; L is N+10 ) ), !.
length_param(Obj:I,L) :-  atom_length(Obj,N), (  I < 10, L is N+4 ; L is N+5 ), !.
length_param(Obj,L) :- atom_length(Obj,L), !.


/*** formatierte Ausgabe der Abhaengigen */

write_abhaengige(_,[],_) :- nl, !.
write_abhaengige(I,[H|T],Tab) :-
  get_abhaengige(H,_,Fkt,_),
  length_param(H,PL),
  write_neu([tab(5),quote(H),tab(Tab-PL+1),':: ',quote(Fkt),nl]),
  I1 is I+1,
  write_abhaengige(I1,T,Tab), !.


/*** formatierte Ausgabe der Nebenbedingungen */

write_nebenbedingungen(_,[]) :- nl, !.
write_nebenbedingungen(I,[H|T]) :-
  write_neu([tab(5),'( ',quote(H),' )',nl]),
  I1 is I+1,
  write_nebenbedingungen(I1,T), !.
```

Das Modul regel

```
/* Modul mit Fkt. fuer einen Regelinterpreter (vorwaerts- */
/* verkettend mit PROLOG-Backtracking)  v1.0               */
/*                                                         */
/* Sven Hader, 01REA88,  24.2.1993  ..  16.4.1993          */

:- module(regel).

:- export([ regelinterpreter/0, erklaere_inferenz/0, set_regel/4, get_regel/3, change_regel/4, delete_regel/1,
            delete_regeln/0, ausgabe_regel/2, korrekte_regel/4, regelausdruck/1, ausgabe_regelbenutzung/1]).

:- op(800,xfx,mit).
:- op(850,xfx,von).
:- op(900,xfx,ist).

:- body(regel).

:- op(0,xfx,:).
:- op(700,xfx,<>).

:- import(var2).
:- import(util2).
:- import(zustand).
:- import(gl_manip).
:- import(i_face2).
:- import(fehler).
:- import(aufgabe).

:- dynamic([regel/3, aenderung/1, zfkt/2, zfkt/3, verwendete_regel/5, veraend_elem/2, wm_elem/2, regelvar/2]).
```

```
:- init(init_regel).


/* initialisiert das Modul */

init_regel :- create_skalar(inf_schritte_S,regel).


/* vorwaertsverkettender Regelinterpreter mit Backtracking */

regelinterpreter :-
  abolish(aenderung/1),
  abolish(zfkt/2),
  abolish(zfkt/3),
  abolish(verwendete_regel/5),
  abolish(veraend_elem/2),
  abolish(wm_element/2),
  AP <- anz_param_r_S,
  asserta(wm_element(anzahl_parameter,AP)),
  asserta(wm_element(stop,nein)),
  inf_schritte_S <- 0,
  findall(Nr,get_nebenbedingung_regel(_,Nr,vor_berechnung),NBL1),
  findall(Nr,get_nebenbedingung_regel(_,Nr,nach_berechnung),NBL2),
  r_interpreter(NBL1,NBL2),
  findall(RId,verwendete_regel(_,RId,_,_,_),RL1),
  liste_zu_quantmenge(RL1,QM),
  reverse(QM,QM1),
  set_zustand(regelnutzung,QM1), !.

r_interpreter(NBL1,NBL2) :-
  set_zustand(bessere_lsg,nein),
  get_umgebung(AlteUmgeb),                    % aktuelle Umgebung speichern
  erzeuge_konfliktmenge(KM,AlteUmgeb),  % sucht alle beim augenblicklichen Systemzustand anwendbaren
                                                % Regelinstantiierungen
  waehle_regel(KM,regel(RId,LHS,RHS)),  % waehlt eine der Regelinstant. aus (bei Backtracking weitere)
  ruecksetzen_beim_backtracking,          % Aenderungen zuruecknehmen
  fuehre_aus(RHS,NBL1,NBL2),                % Ausfuehrung des Aktionsteils der gewaehlten Regelinstant.
  registriere_regel(RId,LHS,RHS,AlteUmgeb),    % Regelinstant. als 'benutzt'  kennzeichnen
  !, wm_element(stop,nein),
     r_interpreter(NBL1,NBL2).                % neuer Aufruf, falls kein Ende-Komm.

r_interpreter(_,_).


/*** liefert eine sortierte Umgebungsliste (Parameter + WM-Elemente) */

get_umgebung(UmListe) :-
  findall(parameter(PName,PWert), ( get_parameter_regel(PName,_,_,_,_),PWert <- PName ), PListe),
  sort(PListe,PListe1),
  findall(wm_elem(WMName,WMWert), wm_elem(WMName,WMWert), WMListe),
  sort(WMListe,WMListe1),
  append(PListe1,WMListe1,UmListe), !.


/*** Bilden der Konfliktmenge (Menge aller Regeln, deren Beding. */
/*** erfuellt sind und die angewendet werden koennten)                */

erzeuge_konfliktmenge(KM,Umgebung) :-
  findall(regel(RId,LHS,RHS),
          ( regel(RId,LHS,RHS), pruefe_LHS(LHS), not verwendete_regel(_,RId,LHS,_,Umgebung) ),
          KM), !.

/***** Pruefen, ob alle Bedingungen der Bed.-seite der Regel */
/***** erfuellt sind (+ Erzeugen von Bindungen)                    */
/***** (backtrackbar)                                                  */

pruefe_LHS([LVar <- quote(Source)|Rest]) :-  % lok. Variable Wert zuweisen
  var(LVar),
  LVar = Source,
  pruefe_LHS(Rest).

pruefe_LHS([LVar <- Source|Rest]) :-        % lok. Variable Wert zuweisen
  var(LVar),
  LVar <- Source,
  pruefe_LHS(Rest).

pruefe_LHS([Eig von Obj ist Wert|Rest]) :-   % Eigenschaft eines Obj. ermitteln
  Pred =.. [Eig,Obj,Wert],
  Pred,
  pruefe_LHS(Rest).

pruefe_LHS([WM_Name ist Wert|Rest]) :-       % Wert eines WM-Elem. ermitteln
  wm_element(WM_Name,Wert),
  pruefe_LHS(Rest).

pruefe_LHS([SPred ist Wert|Rest]) :-         % Wert eines Spez.praed. ermitteln
  spezialpraedikat(SPred),
```

```
  Pred =.. [SPred,Wert],
  Pred,
  pruefe_LHS(Rest).

pruefe_LHS([DObj:I:J ist Wert|Rest]) :-   % Wert eines Matrixelem. ermitteln
  get_vartyp(DObj,matrix(Z,Sp)),
  for(1,I,Z), for(1,J,Sp),
  ( nonvar(Wert), Wert #= DObj:I:J ; var(Wert), Wert <- DObj:I:J ),
  pruefe_LHS(Rest).

pruefe_LHS([DObj:I ist Wert|Rest]) :-     % Wert eines Vektorelem. ermitteln
  get_vartyp(DObj,vektor(N)),
  for(1,I,N),
  ( nonvar(Wert), Wert #= DObj:I ;  var(Wert), Wert <- DObj:I ),
  pruefe_LHS(Rest).

pruefe_LHS([DObj ist Wert|Rest]) :-       % Wert eines Skalars ermitteln
  get_vartyp(DObj,skalar),
  ( nonvar(Wert), Wert #= DObj ;  var(Wert), Wert <- DObj ),
  pruefe_LHS(Rest).

pruefe_LHS([Vergleich|Rest]) :-           % arithmetische Vergleiche
  Vergleich =.. [Op,T1,T2],
  member(Op,[<,<=,=,<>,>,>=]),
  atom_chars(Op,OpListe),
  atom_chars(OpNeu,[#|OpListe]),
  VergleichNeu =.. [OpNeu,T1,T2],
  VergleichNeu,
  pruefe_LHS(Rest).

pruefe_LHS([]).


/***** Eigenschaften von Datenobjekten */

eigenschaft(art).
eigenschaft(untere_grenze).
eigenschaft(obere_grenze).
eigenschaft(datentyp).
eigenschaft(summe).
eigenschaft(maximum).
eigenscahft(minimum).

art(DObj,Art) :-
  ( get_parameter_regel(DObj,_,_,_,_),
      Art1 = parameter ;
    get_konstante(DObj),
      Art1 = konstante ;
    get_ergebnis(DObj),
      Art1 = ergebnis ),
   ( var(Art), Art = Art1 ; nonvar(Art), Art == Art1 ).

untere_grenze(PName,UG) :- get_parameter_regel(PName,_,bereich(UG,_,_),_,_).

obere_grenze(PName,OG) :- get_parameter_regel(PName,_,bereich(_,OG,_),_,_).

datentyp(PName,DTyp) :- get_parameter_regel(PName,_,bereich(_,_,DTyp),_,_).

summe(Formel mit BL,Sum) :-
  findall(Formel,pruefe_LHS(BL),FL),  % Formel bleibt ungebunden !
  FL = [_|_],                         % Fehler, wenn kein Element in Liste
  berechne_summe(FL,0,S),
  ( var(Sum), Sum = S ; nonvar(Sum), Sum =:= S ), !.

berechne_summe([],S,S).
berechne_summe([H|T],STmp,S) :- S1 <- STmp + H,  berechne_summe(T,S1,S).

maximum(DObj mit BL,Max) :-
  findall([Wert,DObj],( pruefe_LHS(BL), Wert <- DObj ),ML),
  ML = [H|T],                         % Fehler, wenn kein Element in Liste
  finde_max(T,H,[WMax,DOMax]),
  DObj = DOMax,                       % Bindung an max. Datenobjekt
  ( var(Max), Max = WMax ;  nonvar(Max), Max =:= WMax ), !.

finde_max([],Max,Max).
finde_max([[Wert,DObj]|T],[WTmp,DOTmp],Max) :-
  ( Wert > WTmp, finde_max(T,[Wert,DObj],Max) ;
    finde_max(T,[WTmp,DOTmp],Max) ).

minimum(DObj mit BL,Min) :-
  findall([Wert,DObj],( pruefe_LHS(BL), Wert <- DObj ),ML),
  ML = [H|T],                         % Fehler, wenn kein Element in Liste
  finde_min(T,H,[WMin,DOMin]),
  DObj = DOMin,                       % Bindung an min. Datenobjekt
  ( var(Min), Min = WMin ; nonvar(Min), Min =:= WMin ), !.

finde_min([],Min,Min).
```

```
finde_min([[Wert,DObj]|T],[WTmp,DOTmp],Min) :-
  ( Wert < WTmp, finde_min(T,[Wert,DObj],Min) ;
    finde_min(T,[WTmp,DOTmp],Min) ).


/***** in den Regeln verwendbare Spezialpraedikate */

spezialpraedikat(zielfunktion).
spezialpraedikat(opt_kriterium).

zielfunktion(ZFkt) :- get_zielfunktion(ZFkt,_).

opt_kriterium(OptKrit) :- get_zielfunktion(_,OptKrit).


/*** waehlt aus der Konfliktmenge eine Regel aus (und zwar die mit den */
/*** meisten Bedingungen  (backtrackbar)                               */

waehle_regel(KM,Regel) :- member(Regel,KM).


/* setzt bei Backtracking Aenderungen an globalen Datenobjekten und */
/* WM-Elementen zurueck                                             */

ruecksetzen_beim_backtracking :-
  ( abolish(aenderung/1), abolish(zfkt/2) ;
    rbb, abolish(zfkt/2), fail ).

rbb :- retract(aenderung(Art)), setze_zurueck(Art), rbb.
rbb :- !.

setze_zurueck(parameter(PName,Wert,_)) :- PName <- Wert, !.
setze_zurueck(wm_elem(WM_Name,Wert,_,aendere)) :-
                                        retract(wm_elem(WM_Name,_)),
                                        assertz(wm_elem(WM_Name,Wert)), !.
setze_zurueck(wm_elem(WM_Name,Wert,_,loesche)) :-
                                        assertz(wm_elem(WM_Name,Wert)), !.
setze_zurueck(wm_elem(WM_Name,_,_,erzeuge)) :-
                                        retract(wm_elem(WM_Name,_)), !.


/*** Ausfuehren der Aktionen eines Regelaktionsteiles (scheitert, wenn  */
/*** Wert eines Parameters das def. Intervall verlaesst, nach einer     */
/*** ZF-Berechnung die Nebenbed. nicht erfuellt oder wenn ZF-Berechnung */
/*** keinen Gewinn erzeugt)                                            */

fuehre_aus([Dest <- Source|Rest],NBL1,NBL2) :-  % lok. Var. Wert zuweisen
  var(Dest),
  Dest <- Source,
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([Dest <- Source|Rest],NBL1,NBL2) :-  % Parameter Wert zuweisen
  get_parameter_regel(Dest,_,bereich(UG,OG,Art),_,NBL),
  AWert <- Dest,
  Wert <- Source,
  Dest <- Wert,
  ( Wert >= UG,
      Wert =< OG,
      ( Art == integer, integer(Wert) ; Art == real ),
      pruefe_nb_regel(NBL),
      asserta(aenderung(parameter(Dest,AWert,Wert))) ;
    Dest <- AWert, fail ),
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([WM_Name <- Source|Rest],NBL1,NBL2) :-  % WM-Elem. Wert zuweisen
  retract(wm_elem(WM_Name,AWert)),
  asserta(aenderung(wm_elem(WM_Name,AWert,Source,aendere))),
  assertz(wm_elem(WM_Name,Source)),
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([erzeuge(WM_Name)|Rest],NBL1,NBL2) :-   % Erzeugen eines WM-Elem.
  ( wm_elem(WM_Name,_) ;
    asserta(aenderung(wm_elem(WM_Name,unbelegt_,dummy_,erzeuge))),
      assertz(wm_elem(WM_Name,unbelegt_)) ),
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([loesche(WM_Name)|Rest],NBL1,NBL2) :-   % Loeschen eines WM-Elem.
  ( retract(wm_elem(WM_Name,Wert)),
      asserta(aenderung(wm_elem(WM_Name,Wert,dummy_,loesche))) ;
    true ),
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([berechne_zielfunktion|Rest],NBL1,NBL2) :-  % ZF-Berechnung
  get_schnittstelle(Datei,InListe,OutListe),
  !, pruefe_nb_regel(NBL1),
  dobj2mehrliste(OutListe,AltErgListe),  % alte Ergebnisswerte sichern
  dobj2einliste(InListe,EWListe),        % Liste der Eingabewerte erstellen
  run_modell(Datei,EWListe,AWListe),
  einliste2dobj(OutListe,AWListe),       % neue Ergebnisse speichern
  ( pruefe_nb_regel(NBL2),
```

```
      get_zielfunktion(ZFormel,OptKrit),
      ZF <- ZFormel,  zfw_S <- ZF,
      ( OptKrit == maximum,
          zfw_S #> opt_zfw_r_S,
          speichere_belegung,
          dobj2mehrliste(OutListe,OptEListe),
          set_zustand(regel_erg,OptEListe),
          set_zustand(bessere_lsg,ja) ;
       OptKrit == minimum,
          zfw_S #< opt_zfw_r_S,
          speichere_belegung,
          dobj2mehrliste(OutListe,OptEListe),
          set_zustand(regel_erg,OptEListe),
          set_zustand(bessere_lsg,ja) ),
     dobj2mehrliste(OutListe,ErgListe),
     assertz(zfkt(ZF,ErgListe)) ;
    mehrliste2dobj(OutListe,AltErgListe),  % alte Ergebnisse speichern
     !, fail ),                            % (es trat keine Verbess. ein)
  !, fuehre_aus(Rest,NBL1,NBL2).

fuehre_aus([],_,_).

/***** Speichern einer ermittelten optimalen Belegung */

speichere_belegung :-
   ( get_parameter_regel(PName,I,_,_,_), opt_param_r_S:I <- PName, fail ;
     true ),
   opt_zfw_r_S <- zfw_S.


/*** speichert Informationen ueber den Inferenzprozess */

registriere_regel(RId,LHS,RHS,Umgebung) :-
   inc(inf_schritte_S),  N <- inf_schritte_S,
   assertz(verwendete_regel(N,RId,LHS,RHS,Umgebung)),
   ( aenderung(Aend), asserta(veraend_elem(N,Aend)), fail ;
     true ),
   ( zfkt(ZF,AusListe), assertz(zfkt(N,ZF,AusListe)) ;
     true ),
   ( get_zustand(trace,aus),
       get_zustand(bessere_lsg,ja),
       ausgabe_akt_loesung ;
     get_zustand(trace,ein),
       get_zustand(trace_ausgabe,Art),
       er_inf([[N,RId,LHS,RHS]],Art),
       ( get_zustand(bessere_lsg,ja), ausgabe_akt_loesung ;
         true ),
       string_eingabe([' Weiter mit <Return> ... '],_), nl,nl ;
     true ), !.

/***** Ausgabe der aktuellen Lösung */

ausgabe_akt_loesung :-
   write_neu(['   Temporaere Loesung',nl,'     ZFW : ',zfw_S,nl]),
   ausgabe_akt_parameter, nl, !.


/***** Ausgabe der aktuellen Parameterwerte */

ausgabe_akt_parameter :-
   ( get_parameter_regel(PName,_,_,_,_),
       write_neu(['     Parameter ',quote(PName),' =  ',PName,nl]),  fail ;
     true), !.


/* Erklaerung eines abgeschlossenen Inferenzprozesses */

erklaere_inferenz :-
   findall([N,RId,LHS,RHS], verwendete_regel(N,RId,LHS,RHS,_), RL),
   write_neu([' Die opt. Parameterbelegung ergab sich folgendermassen:', nl,nl]),
   get_zustand(trace_ausgabe,Art),
   er_inf(RL,Art),
   get_zustand(regelnutzung,QM1),
   write_neu([nl,' Regelbenutzung: ',nl,nl]),
   ausgabe_regelbenutzung(QM1), !.
erklaere_inferenz :-
   write_neu([' Es wurde keine Inferenz durchgefuehrt !',nl,nl]).

er_inf([[N,RId,LHS,RHS]|Rest],Art) :-
   write_neu([' ',N,'. Schritt:',nl,nl]),
   ausgabe_regel(RId,user(3)), nl,
   ausgabe_belegung(RId,LHS,RHS), nl,
   ausgabe_aenderungen(N), nl,
   ( Art == lang, member(berechne_zielfunktion,RHS), ausgabe_berech_ergebnis(N), nl ;
     true ),
   ( Rest \== [], string_eingabe([' Weiter mit <Return> ... '],_), nl,nl ;
     true ),
```

```
  er_inf(Rest,Art).
er_inf([],_).


/*** Ausgabe, wie oft die Regeln im Laufe des Inferenzprozesses */
/*** benutzt wurden                                             */

ausgabe_regelbenutzung([]) :- nl.
ausgabe_regelbenutzung([[RId,Anz]|Rest]) :-
  write_neu(['   ',quote(RId),' wurde ',Anz,' mal benutzt.',nl]),
  ausgabe_regelbenutzung(Rest).


/*** Ausgabe der Regel mit Regelident. RId, wobei alle internen Var. */
/*** ( _xxx) durch die urspruenglichen ersetzt werden                */

ausgabe_regel(RId,Art) :-
  regel(RId,LHS,RHS),
  regelvar(RId,VarListe),
  namevars_liste(LHS,VarListe,VarListe1),
  namevars_liste(RHS,VarListe1,_),
  write_regel(regel(RId,LHS,RHS),Art).

/***** Ersetzen der internen Var.namen einer Liste von Termen durch */
/***** in einer Liste gespeicherte Pseudo-Variablennamen (Atome)    */

namevars_liste([],VL,VL) :- !.
namevars_liste([H|T],VL,VLRest) :- namevars(H,VL,VL1), namevars_liste(T,VL1,VLRest), !.

namevars(Var,[VName = _|Rest],Rest) :- var(Var), Var = VName.
namevars(Term,VL,VL) :- atomic(Term).
namevars(Term,VL,VLRest) :- Term =.. [_|AListe], namevars_liste(AListe,VL,VLRest).

/***** Ausgabe der Regel auf die angegebene Weise */
/***** (LZ = Anzahl linker Leerzeichen)           */

write_regel(regel(RId,LHS,RHS),user(LZ)) :-
  write_neu([tab(LZ),'REGEL  ',quote(RId),nl,nl]),
  write_neu([tab(LZ),'  WENN: ']),
  ( LHS == [], write_neu(['<keine Bedingungen>']) ;  write_bed(LHS,user(LZ)) ),
  write_neu([nl,tab(LZ),'  DANN: ']),
  ( RHS == [], write_neu(['<keine Aktionen>',nl]) ;  write_bed(RHS,user(LZ)), nl ), !.

write_regel(regel(RId,LHS,RHS),prolog(LZ)) :-
  atom_codes(RId,RIdListe),       % Laenge des Regelnamens
  list_length(RIdListe,RIdLen),
  write_neu([tab(LZ),quote(RId),' :: [ ']),
  ( LHS == [], write_neu([' ]',nl]) ;  write_bed(LHS,prolog(LZ+RIdLen)), nl ),
  write_neu([tab(LZ+RIdLen),' :: [ ']),
  ( RHS == [], write_neu([' ].',nl]) ;  write_bed(RHS,prolog(LZ+RIdLen)),
    write('.'), nl ), !.

/******* Schreiben einer Liste von Bed. oder Aktionen */

write_bed([],user(_)).
write_bed([H|T],user(LZ)) :-
  write_term(H,[ignore_ops(false)]),
  ( T \== [],  write_neu(['  UND',nl,tab(8+LZ)]) ;
    true ),
  write_bed(T,user(LZ)).

write_bed([],prolog(_)) :- write_neu([' ']).
write_bed([H|T],prolog(LZ)) :-
  write_term(H,[ignore_ops(false)]),
  ( T \== [], write_neu([' ,',nl,tab(LZ+6)]) ;
    true ),
  write_bed(T,prolog(LZ)).

/*** Ausgabe der Belegung der Variablen der angegebenen Regelinstantiierung */

ausgabe_belegung(RId,LHS,RHS) :-
  write_neu(['   Variablenbelegung :',nl,nl]),
  regel(RId,LHS1,RHS1),                    % Originalregel holen
  regelvar(RId,VarListe),                  % Originalvar.namen holen
  append(LHS,RHS,TermListe),
  append(LHS1,RHS1,TermListe1),
  valuevars_liste(TermListe1,TermListe,VarListe,_,[],ValListe),
  write_valliste(ValListe).

/***** Bilden einer Liste der in der Regel vorkomm. Variablen (als      */
/***** Pseudo-Variablennamen) und der Belegung in der angegebenen       */
/***** Regelinstantiierung                                              */

valuevars_liste([],_,VL,VL,ValL,ValLR) :- reverse(ValL,ValLR).
valuevars_liste([H|T],[H1|T1],VL,VLRest,ValL,ValLNeu) :-
  valuevars(H,H1,VL,VL1,ValL,ValL1),
  valuevars_liste(T,T1,VL1,VLRest,ValL1,ValLNeu).
```

```
valuevars(Var,Wert,[VName = _|Rest],Rest,ValL, [[VName,Wert]|ValL]) :-
  var(Var),  Var = VName.
valuevars(Term,_,VL,VL,ValL,ValL) :- atomic(Term).
valuevars(Term,Term1,VL,VLRest,ValL,ValLNeu) :-
  Term =.. [Func|AListe],
  Term1 =.. [Func|AListe1],
  valuevars_liste(AListe,AListe1,VL,VLRest,ValL,ValLNeu).

/***** schreibt die in der Liste angegebenen Variablenbelegungen */

write_valliste([]).
write_valliste([[VName,Wert]|Rest]) :-  write_neu(['     ',VName,' = ',quote(Wert),nl]), write_valliste(Rest).


/*** Ausgabe der waehrend der Regelaktion veraenderten Parameter und WM-Elemente */

ausgabe_aenderungen(N) :-
  bagof(Art,veraend_elem(N,Art),AListe),
  write_neu(['   Veraenderte Parameter und Speicherelemente :',nl,nl]),
  write_aenderungen(AListe), !.

write_aenderungen([]).
write_aenderungen([parameter(PName,AW,W)|Rest]) :-
  write_neu(['     Parameter ',quote(PName),' von ',AW,' auf ',W,nl]),
  write_aenderungen(Rest).
write_aenderungen([wm_elem(WM_Name,_,_,erzeuge)|Rest]) :-
  write_neu(['     Speicherelement ',quote(WM_Name),' wurde erzeugt',nl]),
  write_aenderungen(Rest).
write_aenderungen([wm_elem(WM_Name,_,_,loesche)|Rest]) :-
  write_neu(['     Speicherelement ',quote(WM_Name),' wurde geloescht',nl]),
  write_aenderungen(Rest).
write_aenderungen([wm_elem(WM_Name,AW,W,aendere)|Rest]) :-
  write_neu(['     Speicherelement ',quote(WM_Name),' von ', quote(AW),' auf ',quote(W),nl]),
  write_aenderungen(Rest).


/*** Ausgabe der Berechnungsergebnisse der Zielfunktion, falls diese */
/*** im Regelkoerper berechnet wurde                                 */

ausgabe_berech_ergebnis(N) :-
  zfkt(N,ZF,AusListe),
  get_schnittstelle(_,_,AusObjekte),
  write_neu(['   Die Ermittlung des Zielfkt.wertes ergab :',nl,nl]),
  max_atomlength(AusObjekte,I), I2 is I+2,
  write_ObjListen(AusObjekte,I2,AusListe),
  write_neu([nl,'     Neuer opt. Zielfkt.wert :  ',ZF,nl]), !.

write_ObjListen([],_,_).
write_ObjListen([DObj|T1],N,[WListe|T2]) :-
  atom_length(DObj,I),
  write_neu(['     ',quote(DObj),' :',tab(N-I)]), write(WListe), nl,
  write_ObjListen(T1,N,T2).


/* ----------------- Test der Regeln auf Korrektheit ---------------- */


/* Pruefen einer Regel auf syntaktische Korrektheit */

korrekte_regel(RId,LHS,RHS,VListe) :-
  ( atom(RId) ;
    write_neu([' Fehler: ',quote(RId),' ist kein gueltiger Regelname !', nl]), !, fail ),
  korrekte_LHS(LHS,1,FZeileL),
  ( FZeileL =:= 0 ;
      write_neu([' Fehler: Falsche Syntax in der ',FZeileL,'. Bedingung von Regel ',quote(RId),' !',nl]),
        !, fail ),
  korrekte_RHS(RHS,1,FZeileR),
  ( FZeileR =:= 0 ;
      write_neu([' Fehler: Falsche Syntax in der ',FZeileR,'. Aktion von Regel ',quote(RId),' !',nl]),
        ! , fail ),
  append(LHS,RHS,RListe),
  anz_var_vorkommen(RListe,VListe,QListe),
  warnung_single_var(RId,QListe), !.


/*** prueft, ob die linke Seite einer Regel (Regel-Bedingung) syntaktisch korrekt ist */

korrekte_LHS([],_,0).
korrekte_LHS([Var <- quote(Ausdruck)|R],I,FZ) :-
  ( var(Var),
      regelausdruck(Ausdruck),
      I1 is I+1,
      korrekte_LHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_LHS([Var <- Ausdruck|R],I,FZ) :-
  ( var(Var),
```

```
        regelausdruck(Ausdruck),
        I1 is I+1,
        korrekte_LHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_LHS([Eig von Formel mit BL ist _|R],I,FZ) :-
  ( nonvar(Eig),
        ( Eig == summe, regelausdruck(Formel) ;
          member(Eig,[maximum,minimum]), dobjekt(Formel) ),
        korrekte_LHS(BL,1,_),
         I1 is I+1,
         korrekte_LHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_LHS([Eig von DObj ist _|R],I,FZ) :-
  ( nonvar(Eig),
        eigenschaft(Eig),
        dobjekt(DObj),
        I1 is I+1,
        korrekte_LHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_LHS([_ ist _|R],I,FZ) :- I1 is I+1, korrekte_LHS(R,I1,FZ), !.
korrekte_LHS([Vergleich|R],I,FZ) :-
  ( Vergleich =.. [Op,Aus1,Aus2],
        member(Op,[<,<=,=,>=,>,<>]),
        regelausdruck(Aus1), regelausdruck(Aus2),
        I1 is I+1,
        korrekte_LHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_LHS([_|_],I,I).


/*** prueft, ob die rechte Seite einer Regel (Regel-Aktion) syntaktisch korrekt ist */

korrekte_RHS([],_,0).
korrekte_RHS([Obj <- Ausdruck|R],I,FZ) :-
  ( ( var(Obj) ; dobjekt(Obj) ),
      regelausdruck(Ausdruck),
      I1 is I+1,
      korrekte_RHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_RHS([erzeuge(WM_Elem)|R],I,FZ) :-
  ( ( var(WM_Elem) ; atom(WM_Elem) ),
      I1 is I+1,
      korrekte_RHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_RHS([loesche(WM_Elem)|R],I,FZ) :-
  ( ( var(WM_Elem) ; atom(WM_Elem) ),
       I1 is I+1,
       korrekte_RHS(R,I1,FZ) ;
    FZ = I ), !.
korrekte_RHS([berechne_zielfunktion|R],I,FZ) :- I1 is I+1, korrekte_RHS(R,I1,FZ), !.
korrekte_RHS([_|_],I,I).


/*** zaehlt, wie oft jede Variable in einer Regel vorkommt */
/*** ( wenn nur einmal, dann i.a. Schreibfehler)                */

anz_var_vorkommen(Liste,VarListe,QListe) :-
  namevars_liste(Liste,VarListe,_),         % Binden der Nutzer-Var.namen an die entspr. System-Var.namen
  avv(Liste,[],VListe),                     % Liste aller Var.vorkommen erzeugen
  liste_zu_quantmenge(VListe,QListe),  % Liste quantifizieren
  set_zustand(var_liste,QListe),            % Ergebnis speichern
  fail.                                     % Backtracking ausloesen  -> Var.bindungen rueckgaengig

anz_var_vorkommen(_,_,QListe) :-  get_zustand(var_liste,QListe).       % Ergebnis holen

avv([],VL,VL).
avv([Ausdruck|Rest],VLTmp,VL) :-
  avv1(Ausdruck,VL1),
  append(VLTmp,VL1,VL2),
  avv(Rest,VL2,VL), !.

avv1(VName,[VName]) :- atom(VName),                    % funkt. nicht, wenn der
                       atom_codes(VName,[EB|_]),  % Nutzer grossgeschriebene
                       for(65,EB,90).                  % und gequotete Atome benutzt
avv1(Atomic,[]) :- atomic(Atomic).
avv1(Ausdruck,VL) :- Ausdruck =.. [_,Aus], avv1(Aus,VL).
avv1(Ausdruck,VL) :- Ausdruck =.. [_,Aus1,Aus2], avv1(Aus1,VL1),
                     avv1(Aus2,VL2), append(VL1,VL2,VL).


/*** prueft, ob in einer quantifizierten Var.liste Variablen nur */
/*** einmal vorkommen und meldet das                             */

warnung_single_var(_,[]).
warnung_single_var(RId,[[VName,1]|Rest]) :-
  write_neu([' Warnung: Die Variable ',quote(VName),' kommt in der Regel ', quote(RId),' nur einmal vor !',nl]),
  warnung_single_var(RId,Rest).
```

```
warnung_single_var(RId,[_|Rest]) :-  warnung_single_var(RId,Rest).


/***** prueft, ob ein arithmetischer Ausdruck in der Regel korrekt */
/***** ist  (soweit man das jetzt schon beurteilen kann)            */

regelausdruck(Var) :- var(Var), !.
regelausdruck(Zahl) :- number(Zahl), !.
regelausdruck(DObj) :- dobjekt(DObj), !.
regelausdruck(sum(I=UG..OG,Ausdruck)) :-
  !, atom(I),
  regelausdruck(UG), regelausdruck(OG),
  regelausdruck(Ausdruck).
regelausdruck(prod(I=UG..OG,Ausdruck)) :-
  !, atom(I),
  regelausdruck(UG), regelausdruck(OG),
  regelausdruck(Ausdruck).
regelausdruck(Ausdruck) :-
  Ausdruck =.. [Op,Aus],
  !, member(Op,[+,-,round,trunc,abs,exp,log,sqrt]),
  regelausdruck(Aus).
regelausdruck(Ausdruck) :-
  Ausdruck =.. [Op,Aus1,Aus2],
  !, member(Op,[+,-,*,/,//,div,mod,**,min,max]),
  regelausdruck(Aus1), regelausdruck(Aus2).


/***** prueft, ob der angegebene Ausdruck ein Datenobjekt sein koennte */

dobjekt(DObj:I:J) :- ( atom(DObj) ; var(DObj) ),
                     regelausdruck(I),regelausdruck(J).
dobjekt(DObj:I) :- ( atom(DObj) ; var(DObj) ), regelausdruck(I).
dobjekt(DObj) :- ( atom(DObj) ; var(DObj) ).



/* ------------------- Verwaltung der Regeln ---------------------- */


/* Hinzufuegen einer Regel zur Regelmenge */

set_regel(RId,LHS,RHS,VL) :-
  regel(RId,_,_),
  write_neu([' Der Regelidentifikaktor ',quote(RId),' existiert bereits !',nl,nl]),
  menue_eingabe([['neue Regel nicht laden',nl], ['alte Regel ueberschreiben',nl],
                 ['neuer Regelidentifikator',nl]],Wahl), nl,
  (  Wahl =:= 1 ;
     Wahl =:= 2,  delete_regel(RId), set_regel(RId,LHS,RHS,VL) ;
     Wahl =:= 3,
       string_eingabe([' Neuer Regelidentifikator : '],String), nl,
       atom_codes(RIdNeu,String),
       set_regel(RIdNeu,LHS,RHS,VL) ), !.

set_regel(RId,LHS,RHS,VL) :-
  assertz(regel(RId,LHS,RHS)),
  ( var(VL), assertz(regelvar(RId,[])) ; assertz(regelvar(RId,VL)) ), !.


/* Loeschen einer Regel aus der Regelmenge */

delete_regel(RId) :- retract(regel(RId,_,_)), retract(regelvar(RId,_)), !.


/* Loeschen aller gespeicherten Regeln */

delete_regeln :- abolish(regel/3), abolish(regelvar/2).


/* Lesen einer Regel der Regelmenge */

get_regel(RId,LHS,RHS) :- regel(RId,LHS,RHS).


/* Aendern einer bereits existierenden Regel der Regelmenge */

change_regel(RId,LHS,RHS,VL) :- retract(regel(RId,_,_)), retract(regelvar(RId,VL)),
                                assertz(regel(RId,LHS,RHS)), assertz(regelvar(RId,VL)), !.
```

Das Modul gl_manip

```
/* Modul mit Fkt. zur Manipulation von Gleichungen  v1.0 */
/*                                                      */
/* Sven Hader, 01REA88,  19.2.1993 ..  15.3.1993        */
```

```
:- module(gl_manip).

:- export([objekte_in_term/2, stelle_um_nach/3,liste_zu_quantmenge/2]).

:- body(gl_manip).

:- op(0,xfx,:).    % Loeschen der Systemdefinition von ':'

:- import(var2).


/*   erzeugt zu einer geg. Gleichung den zugehör. binären Baum     */

gleichung2baum(Objekt:I,[Objekt:I,[],[]]).
gleichung2baum(Objekt:I:J,[Objekt:I:J,[],[]]).
gleichung2baum(G,B) :- G =.. [sum,I=UG..OG,Term], bExpand(+,I,UG,OG,Term,ExpTerm),
                       gleichung2baum(ExpTerm,B), !.
gleichung2baum(G,B) :- G =.. [prod,I=UG..OG,Term], bExpand(*,I,UG,OG,Term,ExpTerm),
                       gleichung2baum(ExpTerm,B), !.
gleichung2baum(G,[Op,LB,RB]) :- G =.. [Op,LT,RT],
                                gleichung2baum(LT,LB), gleichung2baum(RT,RB), !.
gleichung2baum(Objekt,[Objekt,[],[]]).


/*   erzeugt zu einem geg. binären Baum die zugehörige Gleichung   */

baum2gleichung([Objekt,[],[]],Objekt).
baum2gleichung([Op,LB,RB],G) :- baum2gleichung(LB,LT),  baum2gleichung(RB,RT), G =.. [Op,(LT),(RT)].


/*   Ist das angeg. Objekt im Baum enthalten?   */

enthalten_in(Objekt,[Objekt,_,_]).
enthalten_in(Objekt,[_,[Objekt,_,_],_]).
enthalten_in(Objekt,[_,_,[Objekt,_,_]]).
enthalten_in(Objekt,[_,LB,RB]) :- ( enthalten_in(Objekt,LB) ; enthalten_in(Objekt,RB) ).


/*   Wie oft ist das angeg. Objekt im Baum enthalten?   */

enthalten_in(_,[],0).
enthalten_in(Objekt,[Obj,[],[]],0) :- Objekt \== Obj, !.
enthalten_in(Objekt,[Obj,LB,RB],Anz) :-
  enthalten_in(Objekt,LB,Anz1), enthalten_in(Objekt,RB,Anz2),
  ( Objekt == Obj, Anz is Anz1+Anz2+1 ; Anz is Anz1+Anz2 ), !.


/*   führt, wenn möglich, durch num. Berechnung eine Baumverklei-  */
/*   nerung durch  (Baum ist Term und keine Gleichung)              */

optimiere_baum([Objekt,[],[]],[Objekt,[],[]]).
optimiere_baum([Op,[Z1,[],[]],[Z2,[],[]]],[Z3,[],[]]) :-
  number(Z1), number(Z2),  Ber =.. [Op,Z1,Z2], Z3 is Ber.
optimiere_baum([Op,LB,RB],BOpt) :-
  optimiere_baum(LB,LBOpt), optimiere_baum(RB,RBOpt),
  ( LBOpt == LB, RBOpt == RB, BOpt = [Op,LB,RB] ;
    optimiere_baum([Op,LBOpt,RBOpt],BOpt) ),! .


/*   stellt Gleichung GAlt nach 'Objekt' um   -->  GNeu   */

stelle_um_nach(Objekt,GAlt,GNeu) :-
  gleichung2baum(GAlt,[Op,LB,RB]),
  enthalten_in(Objekt,LB,Anz1), enthalten_in(Objekt,RB,Anz2), !,
  ( Anz1 =:= 1, Anz2 =:= 0, sun(Objekt,[Op,LB,RB],[OpNeu,LBNeu,RBNeu]) ;
    Anz1 =:= 0, Anz2 =:= 1, sun(Objekt,[Op,RB,LB],[OpNeu,LBNeu,RBNeu]) ),
  /* optimiere_baum(RBNeu,RBOpt), */
  baum2gleichung([OpNeu,LBNeu,RBNeu],GNeu), !.

sun(Objekt,[Op,[Objekt,[],[]],RB],[Op,[Objekt,[],[]],RB]).
sun(Objekt,[Op,[OpL,LBL,LBR],RB],BNeu) :-
  inv_op(OpL,OpInv),
  ( enthalten_in(Objekt,LBL),                  % Var. steht links
      sun(Objekt,[Op,LBL,[OpInv,RB,LBR]],BNeu) ;
    ( member(OpL,['-','/']),                   % Var. steht rechts
        sun(Objekt,[Op,LBR,[OpL,LBL,RB]],BNeu) ;
      sun(Objekt,[Op,LBR,[OpInv,RB,LBL]],BNeu) ) ),! .

inv_op(+,-).
inv_op(*,/).
inv_op(-,+).
inv_op(/,*).


/*   findet alle Objekte eines Terms  (keine Zahlen, keine Operatoren)  */
```

```
objekte_in_term(Zahl,[]) :- number(Zahl), !.
objekte_in_term(sum(I=UG..OG,Term),L) :-   bExpand(+,I,UG,OG,Term,Ausdruck), objekte_in_term(Ausdruck,L), !.
objekte_in_term(prod(I=UG..OG,Term),L) :-  bExpand(*,I,UG,OG,Term,Ausdruck),  objekte_in_term(Ausdruck,L), !.
objekte_in_term(Term,L) :- Term =.. [Op,Term1,Term2], not (Op == ':'),
                           objekte_in_term(Term1,L1), objekte_in_term(Term2,L2),
                           append(L1,L2,L), !.
objekte_in_term(Objekt,[Objekt]).


/*   wandelt Liste in eine Menge von [Element,Anzahl]-Paaren um    */

liste_zu_quantmenge(Liste,QMenge) :- l2qm(Liste,[],QMenge), !.

l2qm([],QM,QM).
l2qm([H|T],QM1,QM) :- ( member([H,Anz],QM1), delete_element([H,Anz],QM1,QM2),
                        Anz1 is Anz+1, l2qm(T,[[H,Anz1]|QM2],QM) ;
                      l2qm(T,[[H,1]|QM1],QM) ).

delete_element(Elem,AltListe,NeuListe) :- de(Elem,AltListe,[],NeuListe).

de(_,[],L,L).
de(Elem,[Elem|T],L1,L) :- de(Elem,T,L1,L).
de(Elem,[H|T],L1,L) :- de(Elem,T,[H|L1],L).
```

Das Modul i_face2

```
/* Modul mit Funktionen zur Kommunikation mit dem Programm,    */
/* das die Ergebniswerte berechnet (ueber ASCII-Dateien)  v2.0     */
/*                                                                 */
/* Sven Hader, 01REA88, 24.2.1993  bis  19.3.1993                  */

:- module(i_face2).

:- export([dobj2einliste/2, dobj2mehrliste/2, einliste2dobj/2, mehrliste2dobj/2, run_modell/3, machflach_liste/2]).

:- body(i_face2).

:- op(0,xfx,:).

:- import(var2).


/* formt eine mehrfach verschachtelte Liste in eine eindim.  Liste um  */

machflach_liste(MListe,EListe) :- mf_l(MListe,[],EListe), !.

mf_l([],L,L).
mf_l([H|T],LTmp,L) :- ( H == [], mf_l(T,LTmp,L) ;
                        H = [_|_], mf_l(H,[],L1),
                          append(LTmp,L1,L2), mf_l(T,L2,L) ;
                        append(LTmp,[H],L2), mf_l(T,L2,L) ).


/* erzeugt aus den Werten der angegebenen Datenobjekte eine eindim. Liste

dobj2einliste(ObjListe,WertListe) :-  d2l(ObjListe,[],L), machflach_liste(L,WertListe), !.

d2l([],L,LRev) :- reverse(L,LRev).
d2l([H|T],LTmp,L) :- get_varwert(H,WListe), d2l(T,[WListe|LTmp],L).


/* erzeugt aus den Werten der angegebenen Datenobjekte */
/* eine mehrdim. Liste  (entspr. DObj-Struktur)        */

dobj2mehrliste(ObjListe,WertListe) :- d2l(ObjListe,[],WertListe), !.


/* belegt die angegebenen Datenobjekte mit den Werten aus der eindim. Liste */

einliste2dobj(ObjListe,ErgListe) :- l2E(ObjListe,ErgListe), !.

l2E([],_).
l2E([Obj|T],Liste) :- get_vartyp(Obj,Typ),
                      ( Typ = skalar, s_skalar(Obj,Liste,L) ;
                        Typ = vektor(N), s_vektor(Obj,1,N,Liste,L) ;
                        Typ = matrix(Z,Sp), s_matrix(Obj,1,Z,Sp,Liste,L) ),
                      l2E(T,L).

s_skalar(Obj,[Wert|L],L) :- Obj <- Wert.

s_vektor(_,I,N,L,L) :- I > N.
s_vektor(Obj,I,N,[Wert|T],L) :- Obj:I <- Wert, I1 is I+1, s_vektor(Obj,I1,N,T,L).
```

```
s_matrix(_,M,Z,_,L,L) :- M > Z.
s_matrix(Obj,I,Z,Sp,LTmp,L) :- s_vektor(Obj:I,1,Sp,LTmp,L1),  I1 is I+1, s_matrix(Obj,I1,Z,Sp,L1,L).


/* belegt die angegebenen Datenobjekte mit den Werten aus  der mehrdim. Liste */

mehrliste2dobj([],_) :- !.
mehrliste2dobj([H|T],[Wert|Rest]) :- set_varwert(H,Wert), mehrliste2dobj(T,Rest), !.


/* schreibt eine eindim. Liste von Werten in die angegebene Datei  (einen pro Zeile) */

schreibe_liste(Datei,Liste) :- file_test(Datei,write),
                               telling(AlteAusgabe), open(Datei,write,EinDatei), tell(EinDatei),
                               aus_liste(Liste),
                               tell(AlteAusgabe), close(EinDatei), !.

aus_liste([]).
aus_liste([H|T]) :- integer(H), write_formatted('%d\n',[H]), aus_liste(T).
aus_liste([H|T]) :- real(H), write_formatted('%e\n',[H]), aus_liste(T).


/* liest eine eindim. Liste von Werten aus der angegebenen Datei  (ein Wert pro Zeile)  */

lese_liste(Datei,Liste) :- file_test(Datei,read),
                           seeing(AlteEingabe), open(Datei,read,AusDatei), see(AusDatei),
                           ein_liste([],Liste),
                           see(AlteEingabe), close(AusDatei), !.

ein_liste(LTmp,L) :- read(Wert),
                     ( Wert == end_of_file, reverse(LTmp,L) ;
                       number(Wert), ein_liste([Wert|LTmp],L) ;
                       ein_liste(LTmp,L) ).


/* Berechnungsprogramm ablaufen lassen */

run_modell(Datei,EinListe,AusListe) :- concat_atom([Datei,'.in'],EinDatei), concat_atom([Datei,'.out'],AusDatei),
                                       schreibe_liste(EinDatei,EinListe), system(Datei), lese_liste(AusDatei,AusListe), !.
```

Das Modul fehler

```
/* Modul mit Fkt. zur Fehlerregistrierung und -auswertung  v2.0 */
/*                                                              */
/* Sven Hader,01REA88,  31.3.1993  ..  13.4.1993                */

:- module(fehler).

:- export([ init_fehler/0, loesche_fehler/0, merke_fehler/1, merke_fehler/2, push_kontext/1, pop_kontext/1,
            zeige_fehler/0, zeige_fehler/3,get_fehler/3, zeige_letzten_fehler/0, get_letzten_fehler/3]).

:- body(fehler).

:- op(0,xfx,:).

:- import(util2).

:- op(200,yfx,'::').
:- op(700,xfx,<=).
:- op(700,xfx,<>).

:- dynamic([fehler/3, letzter_fehler/3]).


/* initialisiert das Fehlermodul */

init_fehler :- abolish(fehler/3),  abolish(letzter_fehler/3),
               assertz(fehler(0,[],dummy_)), assertz(letzter_fehler(0,[],dummy_)), !.


/* löscht gespeicherte Fehlernummer und zugehoerigen Kontext */

loesche_fehler :- retract(fehler(FNr,KontextListe,Info)), asserta(fehler(0,[],dummy_)),
                  ( FNr =\= 0, retract(letzter_fehler(_,_,_)),
                       asserta(letzter_fehler(FNr,KontextListe,Info)) ;
                    true ), !.


/* registriert die Fehlernummer einen aufgetretenen Fehlers und  scheitert dann */

merke_fehler(FehlerNr) :-  retract(fehler(_,KontextListe,_)),
                           asserta(fehler(FehlerNr,KontextListe,dummy_)), !, fail.
```

```
merke_fehler(FehlerNr,Info) :-  retract(fehler(_,KontextListe,_)),
                                asserta(fehler(FehlerNr,KontextListe,Info)), !, fail.


/* fuegt der Kontextliste einen neuen Eintrag hinzu (vorn) */

push_kontext(Kontext) :- retract(fehler(FNr,KontextListe,Info)), asserta(fehler(FNr,[Kontext|KontextListe],Info)), !.


/* entfernt den vordersten Eintrag der Kontextliste */

pop_kontext(Kontext) :- retract(fehler(FNr,KontextListe,Info)),
                        ( KontextListe = [Kontext|Rest], asserta(fehler(FNr,Rest,Info)) ;
                          Kontext = empty, asserta(fehler(FNr,[],Info)) ), !.


/* zeigt den aufgetretenen Fehler (oder nichts) an */

zeige_fehler :- fehler(FNr,KontextListe,Info),
                ( FNr =\= 0, zeige_kontext(_,KontextListe), nl,
                     zeige_info(FNr,KontextListe,Info), zeige_fehlertext(FNr) ;
                  true ), !.

zeige_fehler(FNr,KontextListe,Info) :-  ( FNr =\= 0, zeige_kontext(_,KontextListe), nl,
                                              zeige_info(FNr,KontextListe,Info), zeige_fehlertext(FNr) ;
                                        true ), !.

/*** zeigt den Fehlerkontext an */

zeige_kontext(1,[]).
zeige_kontext(I1,[H|T]) :- zeige_kontext(I,T), write_neu([' ',I,'. Kontext : ',quote(H),nl]), I1 is I+1, !.

/*** zeigt evtl. vorhandene weitere Informationen */

zeige_info(_,_,dummy_).
zeige_info(1,_,Zeile) :-  write_neu([' Fehler in Zeile:  ',Zeile,nl,nl]).
zeige_info(_,[_,lese_aufgabe|_],FZeile) :-  write_neu([' Fehlerhafte Zeile:  ',quote(FZeile),nl,nl]).
zeige_info(_,_,_).

/*** erlaeutert den Fehler */

zeige_fehlertext(FNr) :-  fehler_text(FNr,FText), write_neu([' FEHLER:  ',FText,nl,nl]).


/* liefert Fehlernummer und Kontext des aktuellen Fehlers */

get_fehler(FNr,Kontext,Info) :- fehler(FNr,Kontext,Info).


/* zeigt den letzten aufgetretenen Fehler (oder nichts) an */

zeige_letzten_fehler :- letzter_fehler(FNr,KontextListe,Info),
                        ( FNr =\= 0,  zeige_kontext(_,KontextListe), nl,
                             zeige_info(FNr,KontextListe,Info), zeige_fehlertext(FNr) ;
                          true ), !.


/* liefert Fehlernummer und Kontext des letzten aufgetretenen Fehlers */

get_letzten_fehler(FNr,Kontext,Info) :- letzter_fehler(FNr,Kontext,Info).



/* ------------------------- Fehlertexte ---------------------------- */

   % allgemeine Fehler

fehler_text(1,'PROLOG-Syntaxfehler !!').
fehler_text(10,'Unbekanntes Kommando !!').
fehler_text(20,'Falsche oder fehlende Argumente eines Kommandos !!').
fehler_text(30,'Nicht behebbarer Fehler bei Betriebssystemruf !!').

   % Modelldatei-Arbeit

fehler_text(100,'Diese Aufgabendatei kann nicht gelesen werden !').
fehler_text(101,'Zu frühes Dateiende !').
fehler_text(102,'Noch keine Aufgabenbeschreibung geladen !').

fehler_text(120,'Der Datenobjekte-Block (datenobjekte.) fehlt !').
fehler_text(121,'Falsches Format einer Datenobjekt-Definition !').
fehler_text(122,'Falsche Syntax eines Datenobjekt-Namens !').
fehler_text(123,'Dieser Datenobjekt-Name wurde schon verwendet !').
fehler_text(124,'Dieses Datenobjekt wurde noch nicht definiert !').
fehler_text(125,'Falsche Zugriffsart auf Datenobjekt (Typkonflikt) !').
fehler_text(126,'Diese Dimensionsart existiert nicht !').
fehler_text(127,'Falsche Elementanzahl eines Vektors !').
fehler_text(128,'Falsche Zeilenanzahl einer Matrix !').
```

```
fehler_text(129,'Falsche Spaltenanzahl einer Matrix !').

fehler_text(141,'Falsches Format einer Konstanten-Definition !').
fehler_text(142,'Falsche Indizierung des Datenobjektes !').
fehler_text(143,'Die Konstanten-Wertliste besitzt eine falsche Dimension !').

fehler_text(161,'Falsches Format einer Parameter-Definition !').
fehler_text(162,'Falsche Bereichsangabe fuer einen Parameter !').
fehler_text(163,'Die Startwertliste besitzt eine falsche Dimension !').

fehler_text(181,'Falsches Format einer Abhaengigen-Definition !').
fehler_text(182,'Falsche Indizierung des Datenobjektes !').
fehler_text(183,'Die angegebene Formel muss berechenbar sein !').

fehler_text(200,'Der Zielfunktions-Block (zielfunktion.) fehlt !').
fehler_text(201,'Falsches Format der Zielfunktions-Definition !').
fehler_text(202,'Die Zielfunktion muß ein berechenbarer Term sein !').
fehler_text(203,'Es wurde ein falsches Optimalitaetskriterium angegeben !').

fehler_text(221,'Falsches Format einer Nebenbedingungs-Definition !').
fehler_text(222,'Unbekannter Vergleichsoperator !').
fehler_text(223,'Die Terme der Nebenbedingung müssen berechenbar sein !').
fehler_text(224,'Widerspruechliche Nebenbedingung !').

fehler_text(240,'Der Schnittstellen-Block (schnittstelle.) fehlt !').
fehler_text(241,'Falsches Format der Schnittstellendefinition !').
fehler_text(242,'Falsche Syntax des Modellprogramm-Namens !').
fehler_text(243,'Das Modellprogramm kann nicht gefunden werden !').
fehler_text(244,'Fehler in der Eingabeliste des Modellprogrammes !').
fehler_text(245,'Fehler in der Ausgabeliste des Modellprogrammes !').

   % Regeldatei-Arbeit

fehler_text(300,'Diese Regeldatei kann nicht gelesen werden !').
fehler_text(301,'In diese Regeldatei kann nicht geschrieben werden !').
fehler_text(302,'Noch kein Regelwissen geladen !').

fehler_text(321,'Falsches Format einer Regel-Definition !').

fehler_text(340,'Es existiert keine Regel mit diesem Namen !').

   % Nutzer-Fehler

fehler_text(400,'Es wurde noch keine Dimensionierung durchgefuehrt !').

fehler_text(410,'Falsche Trace-Art !').
fehler_text(411,'Falsche Trace-Ausgabe !').

   % Hilfsdatei-Arbeit

fehler_text(500,'Die Hilfe-Datei konnte nicht gefunden werden !').
fehler_text(501,'Zu diesem Thema existiert keine Hilfe !').
```

Das Modul hilfe

```
/* Modul enthaelt Fkt. zur Realisierung eines Hilfesystems */
/*                                                         */
/* Sven Hader, 01REA88,  31.3.1993                         */


:- module(hilfe).

:- export([get_hilfe/2]).

:- body(hilfe).

:- op(200,yfx,::).

:- import(util2).
:- import(fehler).


/* bei Eingabe eines Hilfe-Identifikators (Atom,Zahl) und evtl. einer        */
/* Liste von Werten wird der entsprechende Hilfstext ausgegeben, wobei       */
/* im Hilfstext vorkommende Variablen durch die uebergebenen Werte           */
/* ersetzt werden                                                            */

get_hilfe(fehler500,[]) :-
  write_neu(['   Die Datei, die saemtliche Hilfstexte enthaelt, befindet sich wahrscheinlich',nl,
             '   nicht im aktuellen Verzeichnis. Versuchen Sie, die Hilfe-Datei "experte.hlp"',nl,
             '   in  einem anderen Verzeichnis zu finden und in das aktuelle Verzeichnis zu',nl,
             '  kopieren !',nl,nl,
             '   ACHTUNG: Wenn Sie die Hilfe-Datei nicht finden, dann steht Ihnen keinerlei',nl,
```

```
                '          interaktive Hilfe zur Verfuegung !!',nl]), !.

get_hilfe(HId,WListe) :- lies_hilfe(HId,WListe,HText),
                         ( HText \== not_found, write_neu(HText) ; merke_fehler(501,HId) ), !.
get_hilfe(_,_).


/*** versucht, den verlangten Hilfstext aus einer Datei 'experte.hlp'  zu lesen */

lies_hilfe(_,_,_) :- not file_test('experte.hlp',read), !, merke_fehler(500).

lies_hilfe(HId,WListe,HText) :-  seeing(AlteEingabe),  open('experte.hlp',read,Hilfedatei),  see(Hilfedatei),
                                 suche_hilfstext(HId,WListe,HText),
                                 see(AlteEingabe), close(Hilfedatei), !.


/***** liest die in der Hilfedatei stehenden Terme und versucht den richtigen zu finden */

suche_hilfstext(HId,WListe,HText) :-  repeat,
                                      read(Term),
                                     ( Term == end_of_file,  HText = not_found ;
                                      Term = HId :: HText :: WListe ).
```

Das Modul datei

```
/* Modul mit Fkt. zur speziellen Dateiarbeit  v1.0    */
/*                                                    */
/* Sven Hader, 01REA88, 22.2.1993  bis  13.4.1993  */


:- module(datei).

:- export([lese_aufgabendatei/1, loesche_alte_aufgabe/0, lese_regeldatei/1, schreibe_regeldatei/1, schreibe_loesungsdatei/1]).

:- body(datei).

:- op(0,xfx,:).

:- import(var2).
:- import(zustand).
:- import(util2).
:- import(fehler).
:- import(aufgabe).
:- import(regel).
:- import(anzeige).

:- dynamic([zeile/1, zeile_vars/1]).

:- op(200,yfx,'::').
:- op(700,xfx,<=).
:- op(700,xfx,<>).

:- init(init_datei).


/* Initialisierung des Modules */

init_datei :- set_zustand(ad_geladen,nein).


% ------------------------ Fkt. fuer Aufgabendatei ----------------------


/*   liest die angegebene Beschreibungsdatei 'DName'  (Atom) */
/*   und überträgt die Beschreibung in eine rechnerinterne       */
/*   Darstellung                                                 */

lese_aufgabendatei(DName) :-  push_kontext(lese_aufgabe), concat_atom([DName,'.dab'],Datei),
                              ( file_test(Datei,read) ;  merke_fehler(100) ),
                              loesche_alte_aufgabe,
   /* Einlesen */             seeing(AlteEingabe), open(Datei,read,Aufgabendatei), see(Aufgabendatei),
                              lese_alle_zeilen(Status),
                              see(AlteEingabe),  close(Aufgabendatei),
   /* Übersetzen */           ( Status == ok,
                                ( compile_aufgabe, bearbeite_aufgabe,
                                    set_zustand(ad_geladen,ja),  set_zustand(ad_name,DName) ;
                                  true ) ;
                              Status == pro_err,
                                zeile(error(Err,_)), ( merke_fehler(1,Err) ; true ) ).
lese_aufgabendatei(_).

/*** Einlesen aller Prolog-Terme der geoeffneten Datei und Ablegen in zeile/1-Fakten */
```

```
lese_alle_zeilen(Status) :-
  read_term(X,[varnames(L)]),
  assertz(zeile(X)),  assertz(zeile_vars(L)),
  ( X \== end_of_file, lese_alle_zeilen(Status) ;  Status = ok ), !.

lese_alle_zeilen(pro_err) :-  read_error(Z,FNr), assertz(zeile(error(Z,FNr))), !.


/*  Löschen der alten Aufgabenbeschreibung und  */
/*  Initialisierung bestimmter Systemvariablen  */

loesche_alte_aufgabe :-  abolish(zeile/1),  abolish(zeile_vars/1), delete_aufgabe,
                         set_zustand(ad_geladen,nein), set_zustand(nl_vorhanden,nein),
                         set_zustand(rl_vorhanden,nein), set_zustand(ld_gesichert,ja), !.


/***  Übersetzen der in den 'zeile'-Klauseln stehenden Aufgaben- */
/***  beschreibung in eine interne Darstellung                   */

compile_aufgabe :-  ! , comp_datenobjekte, ! , comp_konstanten, ! , comp_parameter, ! , comp_abhaengige,
                    ! , comp_zielfunktion,  ! , comp_nebenbedingungen,  ! , comp_schnittstelle.


/*****  Registrieren aller in der Aufgabe benutzten Datenobjekte  */
/*****  ( Konstanten, Parameter, Ergebnisse, .. )                 */

comp_datenobjekte :- push_kontext(datenobjekte), zeile(X),
                     ( X == datenobjekte, retract(zeile(X)), !, comp_do ; !, merke_fehler(120) ).

comp_do :- zeile(X),
           ( keyword(X) ;
             X == end_of_file, ! , merke_fehler(101) ;
             !, anal_do(X), retract(zeile(X)), ! , comp_do ).

anal_do(DObj::DTyp) :-
  zeile(FZ),
  ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
  ( Status == neu ; !, merke_fehler(123,FZ) ),
  ( datentyp(DTyp) ; !, merke_fehler(126,FZ) ),
  ( DTyp = vektor(N),
      ( for(1,N,10) ;  !, merke_fehler(127,FZ) ) ;    % max. 10 Elemente
    DTyp = matrix(Z,Sp),
      ( for(1,Z,10) ;  !, merke_fehler(128,FZ) ),      % max. 10 Zeilen
      ( for(1,Sp,10) ;  !, merke_fehler(129,FZ) ) ;    % max. 10 Spalten
    true ),
   create_datenobjekt(DObj,DTyp).    % Aktion

anal_do(Term) :- merke_fehler(121,Term).


/*****  Festlegen der Konstanten und ihrer Werte in der Aufgabe */

comp_konstanten :- pop_kontext(_),  push_kontext(konstanten),
                   zeile(X),
                   ( X == konstanten, retract(zeile(X)), !, comp_ko ; true ).

comp_ko :- zeile(X),
           ( keyword(X) ;
             X == end_of_file, ! , merke_fehler(101) ;
             !, anal_ko(X), retract(zeile(X)), ! , comp_ko ).

anal_ko(DObj:DBer1:DBer2::WMatrix) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    ( get_vartyp(DObj,matrix(Z,Sp)) ; !, merke_fehler(125,FZ) ),
    ( datenbereich(Z,DBer1),  datenbereich(Sp,DBer2) ; !, merke_fehler(142,FZ) ),
    ( dber2length(DBer1,Len1), dber2length(DBer2,Len2), dim_matrix(WMatrix,Len1,Len2) ;
      !, merke_fehler(143,FZ) ),
    set_konstante(DObj:DBer1:DBer2,WMatrix), !.   % Aktion

anal_ko(DObj:DBer::WVektor) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    ( get_vartyp(DObj,vektor(N)) ; !, merke_fehler(125,FZ) ),
    ( datenbereich(N,DBer) ; !, merke_fehler(142,FZ) ),
    ( dber2length(DBer,Len), dim_vektor(WVektor,Len) ; !, merke_fehler(143,FZ) ),
    set_konstante(DObj:DBer,WVektor), !.          % Aktion

anal_ko(DObj::WListe) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    get_vartyp(DObj,DTyp),
    ( DTyp == skalar,
        ( [Wert] = WListe ; !, merke_fehler(143,FZ) ),
```

```
        ( number(Wert) ;  !, merke_fehler(143,FZ) ),
        WNeu = WListe ;
      DTyp = vektor(N),
        ( dim_vektor(WListe,N) ; !, merke_fehler(143,FZ) ),
        WNeu = WListe ;
      DTyp = matrix(Z,Sp),
        ( dim_matrix(WListe,Z,Sp) ; !, merke_fehler(143,FZ) ),
        WNeu = WListe ),
    set_konstante(DObj,WNeu), !.              % Aktion

anal_ko(Term) :- merke_fehler(141,Term).


/*****   Festlegen der Parameter (und ihrer Art) in der Aufgabe */
/*****   !!! nur an Parametern kann 'gedreht' werden              */

comp_parameter :- pop_kontext(_),  push_kontext(parameter),
                  zeile(X),
                  ( X == parameter, retract(zeile(X)), !, comp_pa ;  true ).

comp_pa :- zeile(X),
           ( keyword(X) ;
             X == end_of_file, ! , merke_fehler(101) ;
             !, anal_pa(X), retract(zeile(X)), ! , comp_pa ).

anal_pa(DObj:DBer1:DBer2 :: Bereich :: WMatrix) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    ( def_bereich(Bereich) ; !, merke_fehler(162,FZ) ),
    ( get_vartyp(DObj,matrix(Z,Sp)) ; !, merke_fehler(126,FZ) ),
    ( datenbereich(Z,DBer1),  datenbereich(Sp,DBer2) ; !, merke_fehler(142,FZ) ),
    ( dber2length(DBer1,Len1), dber2length(DBer2,Len2), dim_matrix(WMatrix,Len1,Len2) ;
      !, merke_fehler(163,FZ) ),
    set_parameter(DObj:DBer1:DBer2,Bereich,WMatrix).  % Aktion

anal_pa(DObj:DBer :: Bereich :: WVektor) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    ( def_bereich(Bereich) ; !, merke_fehler(162,FZ) ),
    ( get_vartyp(DObj,vektor(N)) ; !, merke_fehler(126,FZ) ),
    ( datenbereich(N,DBer) ; !, merke_fehler(142,FZ) ),
    ( dber2length(DBer,Len), dim_vektor(WVektor,Len) ; !, merke_fehler(163,FZ) ),
    set_parameter(DObj:DBer,Bereich,WVektor).    % Aktion

anal_pa(DObj :: Bereich :: WListe) :-
  !, zeile(FZ),
    ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
    ( Status == existiert ; !, merke_fehler(124,FZ) ),
    ( def_bereich(Bereich) ; !, merke_fehler(162,FZ) ),
    get_vartyp(DObj,DTyp),
    ( DTyp == skalar,
       ( [Wert] = WListe ; !, merke_fehler(163,FZ) ),
        ( number(Wert) ; !, merke_fehler(163,FZ) ),
       WNeu = Wert ;
     DTyp = vektor(N),
       ( dim_vektor(WListe,N) ; !, merke_fehler(163,FZ) ),
      WNeu = WListe ;
     DTyp = matrix(Z,Sp),
        ( dim_matrix(WListe,Z,Sp) ; !, merke_fehler(163,FZ) ),
        WNeu = WListe ),
    set_parameter(DObj,Bereich,WNeu).        % Aktion

anal_pa(Term) :- merke_fehler(161,Term).


/*****   Registrieren aller in der Aufgabe benutzten Abhaengigen      */
/*****   (werden aus anderen Datenobjekten automatisch berechnet)  */

comp_abhaengige :- pop_kontext(_), push_kontext(abhaengige),
                   zeile(X),
                   ( X == abhaengige, retract(zeile(X)), !, comp_ah ; true ).

comp_ah :- zeile(X),
           ( keyword(X) ;
             X == end_of_file, ! , merke_fehler(101) ;
             !, anal_ah(X), retract(zeile(X)), ! , comp_ah ).

anal_ah(DObj:I:J = Formel) :-
  zeile(FZ),
  ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
  ( Status == existiert ; !, merke_fehler(124,FZ) ),
  ( get_vartyp(DObj,matrix(Z,Sp)) ; !, merke_fehler(126,FZ) ),
  ( for(1,I,Z),  for(1,J,Sp) ; !, merke_fehler(182,FZ) ),
  ( regelausdruck(Formel) ; !, merke_fehler(183,FZ) ),
  set_abhaengige(DObj:I:J,Formel,echt).   % Aktion
```

```
anal_ah(DObj:I = Formel) :-
  zeile(FZ),
   ( datenobjekt(DObj,Status) ;  !, merke_fehler(122,FZ) ),
   ( Status == existiert ; !, merke_fehler(124,FZ) ),
   ( get_vartyp(DObj,vektor(N)) ;  !, merke_fehler(126,FZ) ),
   ( for(1,I,N) ; !, merke_fehler(182,FZ) ),
   ( regelausdruck(Formel) ; !, merke_fehler(183,FZ) ),
   set_abhaengige(DObj:I,Formel,echt).   % Aktion

anal_ah(DObj = Formel) :-
  zeile(FZ),
   ( datenobjekt(DObj,Status) ; !, merke_fehler(122,FZ) ),
   ( Status == existiert ; !, merke_fehler(124,FZ) ),
   ( get_vartyp(DObj,skalar) ; !, merke_fehler(126,FZ) ),
   ( regelausdruck(Formel) ; !, merke_fehler(183,FZ) ),
   set_abhaengige(DObj,Formel,echt).   % Aktion

anal_ah(Term) :- merke_fehler(181,Term).


/*****   Festlegen der Zielfunktion (Kostenfunktion im weitesten */
/*****   Sinne) sowie des Optimalitätskriteriums                 */

comp_zielfunktion :- pop_kontext(_),  push_kontext(zielfunktion),
                     zeile(X),
                      ( X == zielfunktion, retract(zeile(X)), !, comp_zf ; !, merke_fehler(200) ).

comp_zf :- zeile(X),
           ( keyword(X), ! , merke_fehler(200) ;
           X == end_of_file, ! , merke_fehler(101) ;
           ! , anal_zf(X), retract(zeile(X)) ).

anal_zf(ZFkt::OptKrit) :-
  zeile(FZ),
  ( regelausdruck(ZFkt) ; !, merke_fehler(202,FZ) ),
  ( opt_kriterium(OptKrit) ; !, merke_fehler(203,FZ) ),
  set_zielfunktion(ZFkt,OptKrit).            % Aktion

anal_zf(Term) :- merke_fehler(201,Term).


/*****   Festlegen der Nebenbedingungen, die für gültige Lösungen  */
/*****   erfüllt sein müssen                                       */

comp_nebenbedingungen :- pop_kontext(_), push_kontext(nebenbedingungen),
                         zeile(X),
                          ( X == nebenbedingungen, retract(zeile(X)), !, comp_nb ; true ).

comp_nb :- zeile(X),
           ( keyword(X) ;
              X == end_of_file, ! , merke_fehler(101) ;
              !, anal_nb(X), retract(zeile(X)), ! , comp_nb ).

anal_nb(Ausdruck) :-
  zeile(FZ),
  ( Ausdruck =.. [Op,Term1,Term2] ; !, merke_fehler(221,FZ) ),
  ( member(Op,['<','<=','>','>=','=','<>']) ; !, merke_fehler(222,FZ) ),
  ( term(Term1,NT1), term(Term2,NT2) ; !, merke_fehler(223,FZ) ),
  concat_atom([#,Op],NeuOp),
  NeuAusdruck =.. [NeuOp,NT1,NT2],
  !, set_nebenbedingung(NeuAusdruck).


/*****   Festlegen der Schnittstelle zur Programmdatei, die das  */
/*****   analytische oder stochastische Modell repräsentiert     */

comp_schnittstelle :- pop_kontext(_),  push_kontext(schnittstelle),
                      zeile(X),
                      ( X == schnittstelle, retract(zeile(X)), !, comp_ss ; !, merke_fehler(240) ).

comp_ss :- zeile(X),
           ( keyword(X), ! , merke_fehler(240) ;
             X == end_of_file, ! , merke_fehler(101) ;
             !, anal_ss(X), retract(zeile(X)) ).

anal_ss(DName::EListe::AListe) :-
  zeile(FZ),
  ( atom(DName) ; !, merke_fehler(242,FZ) ),
  ( file_test(DName,read) ;  !, merke_fehler(243,FZ) ),
  ( pruefe_IListe(EListe) ;  !, merke_fehler(244,FZ) ),
  ( pruefe_OListe(AListe) ; !, merke_fehler(245,FZ) ),
  def_ergebnisse(AListe),       % Aktionen
  set_schnittstelle(DName,EListe,AListe).

anal_ss(Term) :- merke_fehler(241,Term).
```

```
/******* Hilfsfunktionen fuer das Einlesen der Aufgabendatei */

% Schluesselworte fuer Aufgabendatei

keyword(datenobjekte).
keyword(konstanten).
keyword(parameter).
keyword(abhaengige).
keyword(zielfunktion).
keyword(nebenbedingungen).
keyword(schnittstelle).


% Ist X ein Datenobjekt (wenn ja, existiert es?) ?

datenobjekt(X,Status) :- atom(X) , ( get_datenobjekt(X,_), Status = existiert ; Status = neu ), !.

% Ist X ein Datentyp ?

datentyp(skalar).
datentyp(vektor(N)) :- integer(N).
datentyp(matrix(Z,Sp)) :- integer(Z), integer(Sp).


% Korrekter Datenbereich ?

datenbereich(N,UG..OG) :- integer(UG), UG > 0, integer(OG), OG > 0, UG =< OG, for(1,OG,N), !.
datenbereich(N,I) :- integer(I), I > 0, for(1,I,N), !.


% Datenbereich -->  Länge des Datenbereiches

dber2length(UG..OG,Len) :- Len is OG-UG+1, !.
dber2length(_,1).


% Ist die Parameterart korrekt ?

def_bereich(bereich(UG,OG,real)) :- ( number(UG), number(OG), UG =< OG ; !, fail ).
def_bereich(bereich(UG,OG,integer)) :- ( integer(UG), integer(OG) ; UG =< OG ; !, fail ).


% korrektes Optimalitaetskriterium ?

opt_kriterium(minimum).
opt_kriterium(maximum).


% Ist der arithmetische Ausdruck korrekt und wie kann er vereinfacht werden ?

term(X,_) :- var(X), !, fail.
term(Term,NT) :-     Term =.. [Op,T1,T2], member(Op,['+','-','*','/']),
                     term(T1,NT1), term(T2,NT2),
                     ( number(NT1), number(NT2),
                         A =.. [Op,NT1,NT2], NT is A ;
                       Op = '+',  ( (NT1 = 0; NT1 = 0.0), NT = NT2 ;  (NT2 = 0; NT2 = 0.0), NT = NT1 ) ;
                       Op = '-',  ( (NT1 = 0; NT1 = 0.0), NT =.. [-,NT2] ;  (NT2 = 0; NT2 = 0.0), NT = NT1 ) ;
                       Op = '*',  ( (NT1 = 1; NT1 = 1.0), NT = NT2 ;  (NT2 = 1; NT2 = 1.0), NT = NT1 ) ;
                       Op = '/',  ( (NT2 = 0; NT2 = 0.0), !, fail ; (NT2 = 1; NT2 = 1.0), NT = NT1 ) ;
                       NT =.. [Op,NT1,NT2] ), !.
term(sum(I=UG..OG,X),NT) :- !, atom(I), integer(UG), integer(OG), for(1,UG,OG),
                                bExpand(+,I,UG,OG,X,Term), term(Term,NT).
term(prod(I=UG..OG,X),NT) :- !, atom(I), integer(UG), integer(OG), for(1,UG,OG),
                                bExpand(*,I,UG,OG,X,Term), term(Term,NT).
term(DObj,Zahl) :- get_konstante(DObj), Zahl <- DObj, !.
term(DObj:Z:Sp,DObj:Z:Sp) :- !, get_vartyp(DObj,matrix(_,_)), term(Z,_), term(Sp,_).
term(DObj:I,DObj:I) :- !, get_vartyp(DObj,vektor(_)), term(I,_).
term(DObj,DObj) :- get_vartyp(DObj,skalar).
term(Zahl,Zahl) :- number(Zahl), !.


% prüft, ob die Liste korrekte Eingabeparameter für die Berechnungsdatei enthält

pruefe_IListe([]) :- !.
pruefe_IListe([H|T]) :- get_vartyp(H,_), pruefe_IListe(T).


% prüft, ob die Liste korrekte Ausgabeparameter für die Berechnungsdatei enthält

pruefe_OListe([]) :- !.
pruefe_OListe([H|T]) :- get_vartyp(H,_), pruefe_OListe(T).


% definiert die in der Liste stehenden Datenobjekte als Ergebnisse

def_ergebnisse([]).
```

```
def_ergebnisse([E|T]) :- set_ergebnis(E), def_ergebnisse(T).


% -------------------- Fkt. fuer Regeldatei --------------------------


/*   liest die angegebene Regeldatei 'DName'  (Atom ohne Ext.) und */
/*   überträgt die Regelbeschreibungen in eine rechnerinterne          */
/*   Darstellung                                                       */

lese_regeldatei(DName) :-    abolish(zeile/1),  abolish(zeile_vars/1),
                             push_kontext(lese_regeln), concat_atom([DName,'.rd'],Datei),
                             ( file_test(Datei,read) ;  merke_fehler(300) ),
   /* Einlesen */            seeing(AlteEingabe),  open(Datei,read,Regeldatei), see(Regeldatei),
                             lese_alle_zeilen(Status),
                             see(AlteEingabe),  close(Regeldatei),
   /* Übersetzen */          ( Status = ok,
                                 ( compile_regeln, nl,
                                     ( get_regel(_,_,_), set_zustand(rd_geladen,ja),
                                       get_zustand(rd_namen,RDNamen), set_zustand(rd_namen,[DName|RDNamen]),
                                       ( RDNamen = [], set_zustand(rd_gesichert,ja) ; set_zustand(rd_gesichert,nein) ) ;
                                   set_zustand(rd_geladen,nein),  set_zustand(rd_namen,[]),
                                     set_zustand(rd_gesichert,ja) ) ;
                                 true ) ;
                              Status = pro_err, zeile(error(Err,_)), ( merke_fehler(1,Err) ; true ) ).

lese_regeldatei(_).


/***   Festlegen von Heuristiken in Form von Regeln, die helfen  */
/***   sollen, ein bestimmtes Modell zu optimieren                */

compile_regeln :- zeile(X),  zeile_vars(L),
                    ( X == end_of_file ;
                      !, anal_regeln(X,L), retract(zeile(X)), retract(zeile_vars(L)), ! , compile_regeln ).

anal_regeln(RId::LHS::RHS,VL) :-
  ( korrekte_regel(RId,LHS,RHS,VL), set_regel(RId,LHS,RHS,VL) ;
    write_neu([' *** Regel ',quote(RId),' konnte nicht geladen werden !',nl]) ).

anal_regeln(Term,_) :- merke_fehler(321,Term).


/*   schreibt die dem System bekannten Regeln in die angegebene    */
/*   Regeldatei 'DName'  (Atom ohne Ext.)                           */

schreibe_regeldatei(DName) :-
  push_kontext(schreibe_regeln),
  concat_atom([DName,'.rd'],Datei),
  ( file_test(Datei,write) ;  merke_fehler(301) ),
  ( file_test(Datei,read),  ja_nein_eingabe(['   Diese Datei existiert bereits ! Soll sie ',
                                             'ueberschrieben werden [j/n]: '],Antwort), nl ;
    Antwort = ja ),
  ( Antwort == ja,
      telling(AlteAusgabe), open(Datei,write,Regeldatei), tell(Regeldatei),
      ( ausgabe_regel(_,prolog(2)), nl, fail ; true ),
      tell(AlteAusgabe), close(Regeldatei), set_zustand(rd_gesichert,ja) ;
    Antwort == nein,
      write_neu(['   Die Regeln wurden nicht gespeichert !',nl,nl]) ), !.


/*   schreibt die Loesung(en) der aktuellen Aufgabe in die */
/*   angegebene Ergebnisdatei                              */

schreibe_loesungsdatei(DName) :-
  push_kontext(schreibe_loesung),
  concat_atom([DName,'.erg'],Datei),
  ( file_test(Datei,write) ;  merke_fehler(301) ),
  ( file_test(Datei,read),  ja_nein_eingabe(['   Diese Datei existiert bereits ! Soll sie ',
                                             'ueberschrieben werden [j/n]: '],Antwort), nl ;
    Antwort = ja ),
  ( Antwort == ja,
      telling(AlteAusgabe), open(Datei,write,Loesungsdatei), tell(Loesungsdatei),
      localtime(time,Jahr,Monat,Tag,_,_,Stunde,Minute,_),
      get_zustand(user_name,UName), get_zustand(ad_name,ADName), get_zustand(rd_namen,RDNamen),
      write_neu(['Ergebnisdatei von DIM_EXPERTE v1.0',nl,'-----------------------------------',nl,nl]),
      write_neu(['Datum:    ',Tag,'.',Monat,'.19',Jahr,nl,nl,'Uhrzeit:  ',Stunde,'.']),
      ( Minute < 10, write(0) ;  true ),
      write_neu([Minute,' Uhr',nl,nl]),  write_neu(['Nutzer:   ',UName,nl,nl]),
      write_neu(['Aufgabendatei:   ',quote(ADName),'.dab',nl,nl]),
      ( get_zustand(rd_geladen,ja),  write_neu(['Regeldatei(en):  ',quote(RDNamen),nl,nl,nl]) ;
        get_zustand(rd_geladen,nein),  write_neu(['Regeldatei(en):  keine geladen',nl,nl,nl]) ),
      write_neu(['Zu loesende Aufgabe:',nl,'--------------------',nl,nl]),
      zeige_aufgabe, nl,
      write_neu(['Loesungen:',nl,'----------',nl,nl]),
      ( get_zustand(nl_vorhanden,ja), write_neu(['   Ergebnis der numerischen Dimensionierung:',nl,nl]),
          ausgabe_opt_loesung(numerisch) ;
```

```
        true ),
      ( get_zustand(rl_vorhanden,ja), write_neu(['   Ergebnis der wissensbas. Dimensionierung:',nl,nl]),
          ausgabe_opt_loesung(regeln) ;
        true ),
      ( get_zustand(nl_vorhanden,ja), get_zustand(rl_vorhanden,ja),
          P <- opt_zfw_r_S * 100.0 / opt_zfw_n_S,
          write_neu(['   Der optimale ZFW der wissensbas. Dimensionierung erreichte  ',P,' %  des Wertes',nl,
                     '   des optimalen ZFW der numerischen Dimensionierung !', nl,nl]) ;
        true ),
     tell(AlteAusgabe), close(Loesungsdatei), set_zustand(ld_gesichert,ja) ;
   Antwort == nein,  write_neu(['   Die Loesungen wurden nicht gespeichert !',nl,nl]) ), !.
```

Das Modul dimens

```
/* Modul mit Fkt. zur numerischen und wissensbasierten Dimensionierung*/
/*                                                                    */
/* Sven Hader, 01REA88,  9.3.1993  bis  2.4.1993            */

:- module(dimens).

:- export([dimens/0]).

:- body(dimens).

:- op(0,xfx,:).

:- import(var2).
:- import(util2).
:- import(zustand).
:- import(fehler).
:- import(regel).
:- import(i_face2).
:- import(aufgabe).
:- import(anzeige).

:- init(init_dimens).

/* Initialisierung des Modules */

init_dimens :- create_skalar(anz_runs_S,dimens),
               create_skalar(anz_probe_S,dimens),
               create_skalar(zfw_S,dimens).


/* Dimensionierungsmenue */

dimens :- ( get_zustand(ad_geladen,ja),
              write_neu([' Dimensionierung mit',nl,nl]),
              menue_eingabe([['numerischer Methode',nl],['wissensbasierter Methode',nl,nl],['Abbruch',nl]],Zahl), nl,
             ( Zahl =:= 1, set_zustand(dim_art,numerisch), starte_dim(numerisch) ;
               Zahl =:= 2, set_zustand(dim_art,regeln), starte_dim(regeln) ;
              Zahl =:= 3 ) ;
           get_zustand(ad_geladen,nein), merke_fehler(102) ; merke_fehler(30)), !.
dimens.


/*** Loesung der geladenen Dimensionierungsaufgabe mittels numerischen Optimierung  */

starte_dim(numerisch) :-
  get_zustand(nl_vorhanden,ja),  get_zustand(ad_name,DName),
  write_neu([' Die Dimensionierungsaufgabe ',DName,' haben Sie bereits numerisch geloest !',nl,nl]),
  ausgabe_opt_loesung(numerisch), !.

starte_dim(numerisch) :-
  get_zustand(nl_vorhanden,nein), get_zustand(ad_name,AName),
  get_schnittstelle(DName,InListe,OutListe),
  findall(Nr,get_nebenbedingung_num(_,Nr,nach_berechnung),NBL),
  AnzParam <- anz_param_n_S,
  write_neu([' Numerische Loesung der Aufgabe ',AName,' .',nl,nl]),
  Time1 is cputime,
  init_dimensionierung(numerisch),
  repeat,
     veraendere_parameter_num(AnzParam),
     dobj2einliste(InListe,EinListe),
     run_modell(DName,EinListe,ErgListe),
     inc(anz_runs_S),
     einliste2dobj(OutListe,ErgListe),
     berechne_zielfunktion,
     pruefe_nb_num(NBL),
     werte_aus_num,
  ( anz_probe_S #> 20 + 15 * anz_param_n_S),
  Time2 is cputime,
  set_zustand(gueltige_lsg,ja),
```

```
  write_neu([' Dimensionierung nach ',anz_runs_S,' Schritten beendet !',nl,nl]),
  Time is Time2-Time1,
  set_zustand(nl_zeit,Time),
  ( get_zustand(gueltige_lsg,ja),  set_zustand(nl_vorhanden,ja),  set_zustand(ld_gesichert,nein) ;
    true ),
  ausgabe_opt_loesung(numerisch), !.

starte_dim(regeln) :-
  get_zustand(rd_geladen,ja),  get_zustand(ad_name,AName),
  write_neu([' Regelbasierte Loesung der Aufgabe ',AName,' .',nl,nl]),
  Time1 is cputime,
  init_dimensionierung(regeln),
  regelinterpreter,
  Time2 is cputime,
  set_zustand(gueltige_lsg,ja),
  write_neu([' Dimensionierung nach ',inf_schritte_S,' Inferenzschritten beendet !',nl,nl]),
  Time is Time2-Time1,
  set_zustand(rl_zeit,Time),
  ( get_zustand(gueltige_lsg,ja), set_zustand(rl_vorhanden,ja), set_zustand(ld_gesichert,nein) ;
    true ),
  ausgabe_opt_loesung(regeln), !.

starte_dim(regeln) :- get_zustand(rd_geladen,nein), ( merke_fehler(302) ; true ), !.


/***** Anfangsinitialierung der Parameter                          */
/***** (vom Nutzer in der Aufgabenbeschreibung vorgegeben) */

init_dimensionierung(numerisch) :-
  randomize(1),                  % ZZ-Generator initialisieren
  delete_dependency,             % Loeschen aller Abhaengigkeiten
  ( get_abhaengige(AName,_,Formel,_),  % Abhaengigkeiten setzen
      set_dependency(AName,Formel), fail ;
    true ),
  ( get_parameter_num(PName,_,_,Start,_,_),   % Parameterstartwerte setzen
      PName <- Start,  fail ;
    true ),
  get_schnittstelle(DName,InListe,OutListe),  % Zfkt. fuer diese Belegung
  dobj2einliste(InListe,EinListe),
  run_modell(DName,EinListe,AusListe),
  einliste2dobj(OutListe,AusListe),
  berechne_zielfunktion,
  speichere_belegung(numerisch),          % bisher opt. Belegung
  dobj2mehrliste(OutListe,OptEListe),
  set_zustand(num_erg,OptEListe),
  anz_runs_S <- 1, anz_probe_S <- 0,
  ausgabe_akt_loesung(numerisch), !.        % Ausgabe dieser Belegung

init_dimensionierung(regeln) :-
  delete_dependency,             % Loeschen aller Abhaengigkeiten
  ( get_abhaengige(AName,_,Formel,echt),  set_dependency(AName,Formel), fail ;
    true ),
  ( get_parameter_regel(PName,_,_,Start,_),   % Parameterstartwerte setzen
      PName <- Start, fail ;
    true ),
  get_schnittstelle(DName,InListe,OutListe),  % Zfkt. fuer diese Belegung
  dobj2einliste(InListe,EinListe),
  run_modell(DName,EinListe,AusListe),
  einliste2dobj(OutListe,AusListe),
  berechne_zielfunktion,
  speichere_belegung(regeln),          % bisher opt. Belegung
  dobj2mehrliste(OutListe,OptEListe),
  set_zustand(regel_erg,OptEListe),
  anz_runs_S <- 1,
  ausgabe_akt_loesung(regeln), !.         % Ausgabe dieser Belegung


/******* Speichern der Belegung der Parameter */

speichere_belegung(numerisch) :-  ( get_parameter_num(PName,I,_,_,_,_),  opt_param_n_S:I <- PName,  fail ;  true ),
                                  opt_zfw_n_S <- zfw_S.

speichere_belegung(regeln) :- ( get_parameter_regel(PName,I,_,_,_), opt_param_r_S:I <- PName, fail ; true ),
                              opt_zfw_r_S <- zfw_S.

/***** Verändern der Parameter, um das Opt.kriterium zu erfüllen */

veraendere_parameter_num(AnzParam) :-
  findall(Nr,get_nebenbedingung_num(_,Nr,vor_berechnung),NBL),
  repeat,
    veraend_parameter(1,AnzParam),
  pruefe_nb_num(NBL),
  inc(anz_probe_S), !.

veraend_parameter(I,Anz) :- I > Anz.
veraend_parameter(I,Anz) :-
  I =< Anz, I1 is I+1,
```

```
  get_parameter_num(PName,I,bereich(UG,OG,_),_,Schritt,NBL),
  for(1,_,10),
    NWert <- opt_param_n_S:I + 2*random(Schritt) - Schritt,
    NWert >= UG, NWert =< OG,       % Bereichsgrenzen
    pruefe_nb_num(NBL),             % spez. Nebenbed.
    PName <- NWert,
 veraend_parameter(I1,Anz).


/***** Berechnung der Zielfunktion */

berechne_zielfunktion :- get_zielfunktion(Formel,_),  zfw_S <- Formel, !.


/***** Auswertung der aktuellen Berechnung */

werte_aus_num :-
  get_zielfunktion(_,OptArt),
  ( ( OptArt == minimum, zfw_S #< opt_zfw_n_S ;
      OptArt == maximum, zfw_S #> opt_zfw_n_S ),
       speichere_belegung(numerisch),
       get_schnittstelle(_,_,OutListe),
       dobj2mehrliste(OutListe,OptEListe),
       set_zustand(num_erg,OptEListe),
       anz_probe_S <- 0,
       ausgabe_akt_loesung(numerisch) ;
    true ), !.
```

Das Modul anzeige

```
/* Modul mit Fkt. zur Anzeige von Werten, Ergebnissen usw. */
/*                                                         */
/* Sven Hader,  01REA88,  16.3.1993  ..  14.4.1993         */

:- module(anzeige).

:- export([ ausgabe_akt_loesung/1, ausgabe_akt_parameter/1, ausgabe_opt_loesung/1, ausgabe_opt_parameter/1,
            zeige_eine_regel/1, zeige_alle_regeln/0]).

:- body(anzeige).

:- op(0,xfx,:).

:- import(var2).
:- import(util2).
:- import(zustand).
:- import(fehler).
:- import(aufgabe).
:- import(regel).


/* Ausgabe der aktuellen Lösung */

ausgabe_akt_loesung(Art) :-  write_neu(["   Temporaere Loesung",nl,"     ZFW : ",zfw_S,nl]),
                             ausgabe_akt_parameter(Art), nl, !.


/* Ausgabe der aktuellen Parameterwerte */

ausgabe_akt_parameter(_) :-
  ( get_parameter_regel(PName,_,_,_,_),  write_neu(["     Parameter ",quote(PName)," =  ",PName,nl]),  fail ;  true), !.


/* Ausgabe der optimalen Lösung */

ausgabe_opt_loesung(numerisch) :-   get_zustand(nl_vorhanden,nein),
                                    write_neu(["  Es konnte keine gueltige Loesung gefunden werden !",nl,nl]), !.

ausgabe_opt_loesung(numerisch) :-
  get_zustand(nl_vorhanden,ja),  get_zustand(nl_zeit,Time),
  write_neu(['   Zeitdauer:  ',Time,' sec',nl,nl,'   Optimale Loesung',nl,'     ZFW : ',opt_zfw_n_S,nl]),
  ausgabe_opt_parameter(numerisch), nl,
  write_neu(['     Erzielte Ergebnisse:',nl,nl]),
  get_zustand(num_erg,NEListe),
  ausgabe_LKG(NEListe), !.

ausgabe_opt_loesung(regeln) :-
  get_zustand(rl_vorhanden,nein),
  write_neu(["  Es konnte keine gueltige Loesung gefunden werden !",nl,nl]), !.

ausgabe_opt_loesung(regeln) :-
  get_zustand(rl_vorhanden,ja),  get_zustand(rl_zeit,Time),
  write_neu(['   Zeitdauer:  ',Time,' sec',nl,nl,'   Optimale Loesung',nl,'     ZFW : ',opt_zfw_r_S,nl]),
```

```
  ausgabe_opt_parameter(regeln), nl,
  write_neu(['     Erzielte Ergebnisse:',nl,nl]),
  get_zustand(regel_erg,REListe),
  ausgabe_LKG(REListe),
  get_zustand(regelnutzung,RN),
  ausgabe_regelbenutzung(RN), !.


/* Ausgabe der ermittelten optimalen Parameter */

ausgabe_opt_parameter(numerisch) :-
  ( get_parameter_regel(PName,_,_,_,_),
      ( get_parameter_num(PName,Nr,_,_,_,_),
          write_neu(['     Parameter ',quote(PName),' =  ',opt_param_n_S:Nr,nl]) ;
        get_abhaengige(PName,_,Formel,_),
          ersetze_parameter(numerisch,Formel,Formel1), Wert <- Formel1,
          write_neu(['     Parameter ',quote(PName),' =  ',Wert,nl]) ), fail ;
    true ), !.

ausgabe_opt_parameter(regeln) :-
  ( get_parameter_regel(PName,Nr,_,_,_),
      write_neu(['     Parameter ',quote(PName),' =  ', opt_param_r_S:Nr,nl]), fail ;
    true ), !.

/*** Ausgabe der Leistungskenngroessen bei optimaler Parameterbelegung */

ausgabe_LKG(AusListe) :-
  get_schnittstelle(_,_,AusObjekte),
  max_atomlength(AusObjekte,I), I2 is I+2,
  write_LKG(AusObjekte,I2,AusListe), !.

write_LKG([],_,_) :- nl.
write_LKG([DObj|T1],N,[WListe|T2]) :-
  atom_length(DObj,I),
  write_neu(['     ',quote(DObj),' :',tab(N-I)]),
  write(WListe), nl,
  write_LKG(T1,N,T2).

/*** ersetzt in einem Ausdruck alle vorkommenden Parameter durch ihre */
/*** Optimalwerte                                                     */

ersetze_parameter(Art,Ausdruck,Ausdruck1) :-
  Ausdruck =.. [Op,A1,A2],  Op \== ':',
  ersetze_parameter(Art,A1,A11), ersetze_parameter(Art,A2,A22),
  Ausdruck1 =.. [Op,A11,A22], !.
ersetze_parameter(numerisch,P,Wert) :- get_parameter_num(P,Nr,_,_,_,_), Wert <- opt_param_n_S:Nr, !.
ersetze_parameter(numerisch,A,Wert) :-
  get_abhaengige(A,_,Ausdruck,_),
  ersetze_parameter(Ausdruck,Ausdruck1),
  Wert <- Ausdruck1, !.
ersetze_parameter(_,Objekt,Objekt).


/* zeigt eine bestimmte Regel */

zeige_eine_regel(RId) :- get_zustand(rd_geladen,ja),
                         ( ausgabe_regel(RId,user(3)),nl ; merke_fehler(340) ; true ), !.
zeige_eine_regel(_) :- get_zustand(rd_geladen,nein), ( merke_fehler(302) ; true ), !.


/* zeigt alle gespeicherten Regeln */

zeige_alle_regeln :- get_zustand(rd_geladen,ja),
                     ( ausgabe_regel(_,user(3)),
                         ja_nein_eingabe([nl,' Weiter [j/n]: '],Taste), nl,  ( Taste == nein ) ;
                       true ), !.
zeige_alle_regeln :- get_zustand(rd_geladen,nein), ( merke_fehler(302) ; true ), !.
```

Das Modul experte

```
/* Hauptprogramm des Wissensbasierten Dimensionierungssystems  v1.0 */
/*                                                                  */
/* Sven Hader, 01REA88,  9.3.1993  ..  15.4.1993                    */


:- op(0,xfx,:).

:- import(var2).
:- import(zustand).
:- import(util2).
:- import(fehler).
:- import(hilfe).
:- import(regel).
```

```
:- import(dimens).
:- import(aufgabe).
:- import(datei).
:- import(dimens).
:- import(anzeige).

:- init(init_experte).


/* initialisiert das Modul */

init_experte :-     set_zustand(user_name,    dummy_),  set_zustand(ad_geladen,    nein),
                    set_zustand(ad_name,      dummy_),  set_zustand(rd_geladen,    nein),
                    set_zustand(rd_namen,     []),  set_zustand(rd_gesichert,  ja),
                    set_zustand(ld_gesichert,  ja),  set_zustand(nl_vorhanden,  nein),
                    set_zustand(rl_vorhanden,  nein),  set_zustand(trace,        aus),
                    set_zustand(trace_ausgabe, lang),  set_zustand(letztes_komm,  dummy_),
                    loesche_alte_aufgabe,  delete_regeln,  init_fehler.


/* Fenstermanagment  (Init., Abfrageschleife) */

start :- init_experte, begruessung,  interaktion, ende.


/*** Begruessung des Nutzers */

begruessung :-
  localtime(time,Jahr,Monat,Tag,_,_,Stunde,Minute,_),
  write_neu([nl,nl,' Wissensbasierte Dimensionierung technischer Systeme  v1.0  von Sven Hader  1993',nl,nl]),
  string_eingabe([' Ihr Vorname bitte : '],Name),
  ( Name == [],  set_zustand(user_name,"Nutzer") ;
    set_zustand(user_name,Name) ),
  write_neu([nl,' Guten Tag, ',Name,' !  Heute ist der ',Tag,'.',Monat,'.19',Jahr,'.  Es ist ',Stunde,'.']),
  ( Minute < 10, write(0) ; true ),
  write_neu([Minute,' Uhr.',nl,nl,nl]).


/*** Interaktionsschleife */

interaktion :-      repeat,
                      loesche_fehler,
                      lies_eingabe(EZeile),
                uebersetze_eingabe(EZeile,Kommando),
                      fuehre_aus(Kommando),
                      registriere_kommando(Kommando),
                      zeige_fehler,
                    ( Kommando == stop ).


/*****   liest eine Kommandozeile des Nutzers ein  */

lies_eingabe(Zeile) :- string_eingabe([' > '],Zeile),nl.


/*****   übersetzt eine (z.T. natürlichsprachl.) Eingabe in ein Kommando */

uebersetze_eingabe(EZeile,Kommando) :-
  str2strliste(EZeile,SListe),                          % Eingabe -> Liste von Strings
  strliste2atomliste(SListe,[],AListe),                 %         -> Liste von Atomen
  suche_aktnamen(AListe,AktName),                       % Suche nach Aktionsnamen
  suche_fktnamen(AListe,AktName,FktName,ArgZahl),  % Suche nach Funktionsnamen
  ( FktName == keiner,
      Kommando = keiner, ( merke_fehler(10,EZeile) ; true ) ;
    suche_argumente(AListe,FktName,ArgListe),           % Suche nach Argumenten
      ( list_length(ArgListe,ArgZahl), Kommando =.. [FktName|ArgListe] ;
        Kommando = keiner, ( merke_fehler(20,EZeile) ; true ) ) ), !.


/******* Umwandlung String -> Liste von Strings  (Wörter mit Kleinbuchstaben) */

str2strliste(S,SListe) :- cl2sl(S,[],SListe), !.

/********* Umwandlung  Liste von Zeichen -> Liste von Strings (Wörter mit Kleinbuchstaben)  */

cl2sl([],SL,SLR) :- reverse(SL,SLR).
cl2sl(CL,SLTmp,SL) :-
  uebersprinqe_ws(CL,CL1),                % überlese Trennzeichen
  separiere_string(CL1,CL2,[],[H|T]),    % lese zus.hängendes Wort
  ( member(H,[33,46,63]), T \== [],       % entferne Satzendzeichen
      reverse(T,CL3) ;                   % ( '!' '.' '?' )
    reverse([H|T],CL3) ),
  gross2klein(CL3,[],CL4),                % Wort in Kleinbuchstaben
  cl2sl(CL2,[CL4|SLTmp],SL).             % nächstes Wort

/********** überlese Trennzeichen   ( TAB SPACE ',' ';' ) */
```

```
ueberspringe_ws([WS|T],CL) :- member(WS,[9,32,44,59]), ueberspringe_ws(T,CL).
ueberspringe_ws(CL,CL) :- !.

/********** lese zusammenhängendes Wort */

separiere_string([],[],CL,CL).
separiere_string([WS|T],[WS|T],CL,CL) :- member(WS,[9,32,44,59]), !.
separiere_string([C|T],CLRest,CLString,CL) :-  separiere_string(T,CLRest,[C|CLString],CL).

/********** Liste von Zeichen  -> Liste von Zeichen  (Groß- in Kleinbuchstaben)  */

gross2klein([],CL,CLR) :- reverse(CL,CLR), !.
gross2klein([GB|T],CLI,CLO) :- GB >= 65, GB =< 90,   % Großbuchstabe
                               KB is GB+32,  gross2klein(T,[KB|CLI],CLO).
gross2klein([C|T],CLI,CLO) :- gross2klein(T,[C|CLI],CLO).


/******* Umwandlung  Liste von Strings (Wörter) -> Liste von Atomen */

strliste2atomliste([],AListe,ARListe) :- reverse(AListe,ARListe), !.
strliste2atomliste([S|T],ATmp,AL) :-  atom_codes(At,S),  strliste2atomliste(T,[At|ATmp],AL).


/******* sucht in Wortliste (Atome) nach Namen von Aktionen,         */
/******* die der Nutzer ausführen kann (auch innerhalb von Worten) */

suche_aktnamen(WListe,AName) :-  def_akt(AName,BListe),        % BListe - mögl. Bezeichner
                                 s_fnamen(WListe,BListe), !.
suche_aktnamen(_,keiner).

/********* sucht in einer Liste nach einer Unterliste von Bezeichnern, */
/********* die alle (!) in einer Wortliste vorkommen                    */

s_fnamen(WL,[BL|T]) :- ( sfn1(WL,BL) ; s_fnamen(WL,T) ), !.

/********** prüft, ob alle (!) Bezeichner einer Liste in einer Wortliste vorkommen */

sfn1(_,[]) :- !.
sfn1(WL,[Bez|T]) :- sfn2(WL,Bez),  sfn1(WL,T), !.

/************* prüft, ob ein Bezeichner (als Zeichenliste) in einer */
/************* Wortliste vorkommt  (auch teilweise)                  */

sfn2([Wort|T],Bez) :- ( index(Wort,Bez,_) ;   % Bezeichner in Wort enthalten
                        sfn2(T,Bez) ), !.


/******* sucht in Wortliste (Atome) nach Funktionsnamen (ausgehend von */
/******* bestimmtem Aktionsnamen)  (auch innerhalb von Worten)          */

suche_fktnamen(_,keiner,keiner,0).
suche_fktnamen(WListe,AktName,FName,ArgZahl) :-
  def_komm(FName,AktName,BListe,ArgZahl),   % BListe - mögl. Bezeichner
  ( BListe == [] ;  s_fnamen(WListe,BListe) ), !.
suche_fktnamen(_,_,keiner,0).


/******* sucht in Wortliste (Atomen) nach Argumenten für */
/******* vorgeg. Funktionsnamen                          */

suche_argumente(_,keiner,[]) :- !.

suche_argumente(AListe,lade_regeln,[DName]) :-
   suche_nachwort(datei,AListe,DName) ;
   suche_nachwort(file,AListe,DName) ;
   suche_nachwort(aus,AListe,DName) ;
   get_zustand(user_name,Name),
      string_eingabe([' Geben Sie bitte den Dateinamen ein, ',Name,' : '],DNameS), nl, atom_codes(DName,DNameS), !.

suche_argumente(AListe,lade_aufgabe,[DName]) :-
   suche_nachwort(datei,AListe,DName) ;
   suche_nachwort(file,AListe,DName) ;
   suche_nachwort(aus,AListe,DName) ;
   suche_nachwort(aufgabe,AListe,DName) ;
   suche_nachwort(modell,AListe,DName) ;
   get_zustand(user_name,Name),
      string_eingabe([' Geben Sie bitte den Dateinamen ein, ',Name,' : '],DNameS), nl, atom_codes(DName,DNameS), !.

suche_argumente(AListe,speichere_regeln,[DName]) :-
   suche_nachwort(datei,AListe,DName) ;
   suche_nachwort(file,AListe,DName) ;
   suche_nachwort(in,AListe,DName) ;
   get_zustand(user_name,Name),
      string_eingabe([' Geben Sie bitte den Dateinamen ein, ',Name,' : '],DNameS), nl, atom_codes(DName,DNameS), !.

suche_argumente(AListe,speichere_loesung,[DName]) :-
   suche_nachwort(datei,AListe,DName) ;
```

```
  suche_nachwort(file,AListe,DName) ;
  suche_nachwort(in,AListe,DName) ;
  get_zustand(user_name,Name),
     string_eingabe([' Geben Sie bitte den Dateinamen ein, ',Name,' : '],DNameS), nl, atom_codes(DName,DNameS), !.

suche_argumente(AListe,setze_trace,[Zustand]) :-
  ( ( member(ja,AListe) ; member(ein,AListe) ), Zustand = ein ;
    ( member(nein,AListe) ; member(aus,AListe) ), Zustand = aus ), !.

suche_argumente(AListe,setze_traceausgabe,[Zustand]) :-
  ( member(lang,AListe), Zustand = lang ;
    member(kurz,AListe), Zustand = kurz ), !.

suche_argumente(AListe,zeige_eine_regel,[RId]) :-
  suche_nachwort(name,AListe,RId) ;
  suche_nachwort(identif,AListe,RId) ;
  suche_nachwort(regel,AListe,RId) ;
  get_zustand(user_name,Name),
     string_eingabe([' Geben Sie bitte den Regelnamen ein, ',Name,' : '],RIdS), nl, atom_codes(RId,RIdS), !.

suche_argumente(AListe,loesche_eine_regel,[RId]) :-
  suche_nachwort(name,AListe,RId) ;
  suche_nachwort(identif,AListe,RId) ;
  suche_nachwort(regel,AListe,RId) ;
  get_zustand(user_name,Name),
     string_eingabe([' Geben Sie bitte den Regelnamen ein, ',Name,' : '],RIdS), nl, atom_codes(RId,RIdS), !.

suche_argumente(_,_,[]).

/********* sucht in Wortliste (Atomen) nach einem Argument, daß */
/********* nach einem best. Schlüsselwort stehen könnte         */

suche_nachwort(Schl,[Pre,Argument|_],Argument) :- index(Pre,Schl,_), !.
suche_nachwort(Schl,[_|T],Argument) :-  suche_nachwort(Schl,T,Argument).


/***** fuehre_aus(Kommando)  -  selbsterklärend */

fuehre_aus(keiner) :- !.
fuehre_aus(Kommando) :- Kommando \== stop,
                        functor(Kommando,Funktor,Aritaet),
                        ( def_komm(Funktor,_,_,Aritaet),  Kommando ; merke_fehler(10,Kommando) ), !.
fuehre_aus(_) :- !.


/***** Registrieren des letzten gueltigen Kommandos mit evtl. */
/***** aufgetretenen Fehlern                                  */

registriere_kommando(keiner) :- !.
registriere_kommando(_) :- get_fehler(10,_,_), !.
 registriere_kommando(Kommando) :-
  get_fehler(FNr,Kontext,Info),
  ( FNr =:= 0,  set_zustand(letztes_komm,[Kommando,fehlerfrei]) ;
    set_zustand(letztes_komm,[Kommando,fehler(FNr,Kontext,Info)]) ), !.


/* die allgemeinen Nutzeraktionen */

def_akt(laden,[[lade],[lese],[konsult]]).
def_akt(zeigen,[[zeig],[ausgab],[ausgeb],[gebe,aus],[gib,aus]]).
def_akt(speichern,[[speich],[schreib],[sicher]]).
def_akt(eingeben,[[eingeb],[eingab],[gebe,ein],[gib,ein]]).
def_akt(setzen,[[setze],[belege],[schalte]]).
def_akt(loeschen,[[loesch],[entfern]]).
def_akt(dimensioniere,[[dimens],[opti],[rechne],[simul]]).
def_akt(erklaere,[[erklaer],[begruend],[erlaeuter]]).
def_akt(hilfe,[[hilf],[helf]]).
def_akt(stop,[[stop],[ende],[schluss],[exit],[halt]]).


/* die bis jetzt definierten Kommandos */

def_komm(lade_regeln,laden,[[regel],[heuristik]],1).
def_komm(lade_aufgabe,laden,[[aufgab],[modell],[system]],1).

def_komm(speichere_loesung,speichern,[[ergeb],[loes]],1).
def_komm(speichere_regeln,speichern,[[regel],[heuristik]],1).

def_komm(zeige_alle_regeln,zeigen,
       [[alle,regel],[alle,heuristik]],0).
def_komm(zeige_eine_regel,zeigen,[[regel],[heuristik]],1).
def_komm(zeige_loesung,zeigen,[[ergeb],[loes]],0).
def_komm(zeige_zustand,zeigen,[[zustand],[status]],0).
def_komm(zeige_aufgabe,zeigen,[[aufgab],[modell],[system]],0).

def_komm(setze_traceausgabe,setzen,[[trace,ausgabe],[trace,anzeige]],1).
def_komm(setze_trace,setzen,[[trace],[schritt]],1).
```

```
def_komm(loesche_alle_regeln,loeschen,
       [[alle,regel],[alle,heuristik]],0).
def_komm(loesche_eine_regel,loeschen,[[regel],[heuristik]],1).
def_komm(loesche_aufgabe,loeschen,[[aufgab],[modell],[system]],0).

def_komm(dimensioniere,dimensioniere,[],0).

def_komm(erklaere_inferenz,erklaere,
       [[wissen],[regel],[ergeb],[loes],[inferenz]],0).

def_komm(hilfe,hilfe,[],0).

def_komm(stop,stop,[],0).


/*** Beenden des Programmes */

ende :- get_zustand(user_name,Name),
         ( get_zustand(rd_geladen,ja),  get_zustand(rd_gesichert,nein),
             write_neu([' Die Regelmenge wurde modifiziert und ist noch nicht gesichert !!',nl,nl]) ;
           true ),
         ( get_zustand(ad_geladen,ja),  get_zustand(ld_gesichert,nein),
             write_neu([' Es existieren Ergebnisse, die noch nicht gesichert wurden !!',nl,nl]) ;
           true ),
         ja_nein_eingabe([' Wollen Sie wirklich beenden, ',Name,'  [j/n] : '],Antwort), nl,
        Antwort == ja,  write_neu([' Bis bald, ',Name,' !',nl,nl,nl]), !.


/***** Benutzerkommandos */

% Lade-Kommandos ----------------------------------------------------------

lade_aufgabe(DName) :-  write_neu([' Laden der Aufgabenbeschreibungsdatei ',quote(DName),'.dab .', nl,nl]),
                        lese_aufgabendatei(DName).
lade_aufgabe(_).

lade_regeln(DName) :-  write_neu([' Laden der Regeldatei ',quote(DName),'.rd .', nl,nl]),
                       lese_regeldatei(DName).
lade_regeln(_).


% Speicher-Kommandos ------------------------------------------------------

speichere_loesung(_) :- get_zustand(ad_geladen,nein), merke_fehler(102).   % keine Aufgabe geladen
speichere_loesung(_) :-  get_zustand(ad_geladen,ja),  get_zustand(ld_gesichert,ja),
                         write_neu([' Entweder existieren noch keine Loesungen oder Sie haben sie',nl,
                                    '  bereits gesichert !',nl,nl]).
speichere_loesung(DName) :-  get_zustand(ad_geladen,ja),
                             write_neu([' Speichern der Loesungen in Datei ',quote(DName),'.erg .', nl,nl]),
                             schreibe_loesungsdatei(DName).
speichere_loesung(_).

speichere_regeln(DName) :-  write_neu([' Speichern aller Regeln in Datei ',quote(DName),'.rd .', nl,nl]),
                            schreibe_regeldatei(DName).
speichere_regeln(_).


% Anzeige-Kommandos -------------------------------------------------------

zeige_loesung :- get_zustand(ad_geladen,nein), merke_fehler(102).    % keine Aufgabe geladen
zeige_loesung :-
  get_zustand(ad_geladen,ja),
    ( get_zustand(nl_vorhanden,nein),  get_zustand(rl_vorhanden,nein),
        merke_fehler(400) ;               % noch keine Loesung
      ( get_zustand(nl_vorhanden,ja),  write_neu(['   Ergebnis der numerischen Dimensionierung:',nl,nl]),
          ausgabe_opt_loesung(numerisch),nl ;
        true ),
      ( get_zustand(rl_vorhanden,ja), write_neu(['   Ergebnis der wissensbas. Dimensionierung:',nl,nl]),
          ausgabe_opt_loesung(regeln) ;
        true ),
      ( get_zustand(nl_vorhanden,ja),  get_zustand(rl_vorhanden,ja),
          P <- opt_zfw_r_S * 100.0 / opt_zfw_n_S,
          write_neu(['   Der optimale ZFW der wissensbas. Dimensionierung erreichte  ',P,' %  des Wertes',nl,
                     '  des optimalen ZFW der numerischen Dimensionierung !', nl,nl]) ;
        true ) ), !.
zeige_loesung.

zeige_zustand :-
  write_neu([' Augenblicklicher Programmzustand:',nl,nl]),
  get_zustand(user_name,UName), atom_codes(NName,UName),
  write_neu(['   Der augenblickliche Nutzer des Programmes heisst ', quote(NName),'.',nl,nl]),
  ( get_zustand(ad_geladen,ja),  get_zustand(ad_name,AD_Name),
      write_neu(['   Die Aufgabe ',quote(AD_Name),' ist geladen.',nl,'   ( weitere Inform. mit "zeige aufgabe" )',nl,nl]),
      get_zustand(nl_vorhanden,Ant1),
      ( Ant1 == ja,   write_neu(['     Es wurde bereits numerisch dimensioniert.',nl]) ;
        Ant1 == nein,  write_neu(['     Es wurde noch nicht numerisch dimensioniert.',nl]) ),
```

```
      get_zustand(rl_vorhanden,Ant1),
      ( Ant1 == ja,  write_neu(['     Es wurde bereits regelbasiert dimensioniert.',nl]) ;
        Ant1 == nein,  write_neu(['     Es wurde noch nicht regelbasiert dimensioniert.',nl]) ), nl ;
    write_neu(['   Es ist noch keine Aufgabe geladen.',nl,nl]) ),
      ( get_zustand(rd_geladen,ja),  get_zustand(rd_namen,RD_Name),
          write_neu(['   Die Regeldatei(en) ',quote(RD_Name), ' wurde(n) geladen.',nl]),
          get_zustand(rd_gesichert,RDS),
          write_neu(['   Systeminterne Regelmenge in Datei gesichert: ', quote(RDS),nl,nl]) ;
        write_neu(['   Es sind noch keine Regeln geladen.',nl,nl]) ),
  get_zustand(trace,TArt),
  write_neu(['   Der TRACE-Modus ist ',quote(TArt),'geschaltet.',nl]),
  get_zustand(trace_ausgabe,TAus),
  write_neu(['   Die TRACE-Ausgabe ist auf ',quote(TAus),' gesetzt.',nl,nl]), !.


% Setz-Kommandos ------------------------------------------------------

setze_trace(Zustand) :- write_neu([' Setzen des Trace-Modus auf ',quote(Zustand),' .',nl,nl]),
                        ( member(Zustand,[ein,aus]), set_zustand(trace,Zustand) ; merke_fehler(410) ), !.
setze_trace(_).

setze_traceausgabe(Zustand) :-  write_neu([' Setzen der Trace-Ausgabe auf ',quote(Zustand),' .',nl,nl]),
                                ( member(Zustand,[lang,kurz]), set_zustand(trace_ausgabe,Zustand) ;
                                  merke_fehler(411) ), !.
setze_traceart(_).


% Loesch-Kommandos ----------------------------------------------------

loesche_aufgabe :-
  get_zustand(user_name,Name),
  ja_nein_eingabe([' Wollen Sie wirklich die geladene Aufgabe loeschen, ',Name,'  [j/n] : '],Antwort), nl,
  ( Antwort == ja, loesche_alte_aufgabe, write_neu([' Die geladene Aufgabe wurde geloescht !',nl,nl]) ;
    true ), !.

loesche_alle_regeln :-
  get_zustand(user_name,Name),
  ja_nein_eingabe([' Wollen Sie wirklich alle geladenen Regeln loeschen, ',Name,'  [j/n] : '],Antwort), nl,
  ( Antwort == ja,
      delete_regeln,  set_zustand(rd_geladen,    nein), set_zustand(rd_namen,      []),
      set_zustand(rd_gesichert,  ja),  set_zustand(neue_regeln,   nein),
      write_neu([' Die geladenen Regeln wurden geloescht !',nl,nl]) ;
    true ), !.

loesche_eine_regel(RId) :-
  ( get_regel(RId,_,_),       % Regel existiert
      get_zustand(user_name,Name),
      ja_nein_eingabe([' Wollen Sie die Regel "',quote(RId),'" wirklich loeschen, ',Name,'  [j/n] : '],Antwort), nl,
     ( Antwort == ja,
         delete_regel(RId), set_zustand(rd_gesichert, nein),
         write_neu([' Die Regel "',quote(RId),'" wurde geloescht !',nl,nl]),
         ( get_regel(_,_,_) ;
           set_zustand(rd_geladen,    nein), set_zustand(rd_namen,      []),
             set_zustand(rd_gesichert,  ja),
              write_neu([' Im System sind jetzt keine Regeln mehr gespeichert !', nl,nl]) ) ;
      true ) ;
    ( merke_fehler(340) ; true ) ), !.


% Dimensionier-Kommandos ----------------------------------------------

dimensioniere :- dimens.
dimensioniere.


% Hilfe-Kommandos -----------------------------------------------------

hilfe :- write_neu([' Folgende Hilfsmoeglichkeiten bietet das System:',nl,nl]),
          menue_eingabe([['Anzeige der verfuegbaren Kommandos',nl],
                         ['Erklaerung des augenblicklichen Systemzustandes',nl],
                         ['Erklaerung des letzten aufgetretenen Fehlers',nl],
                         ['Erklaerung des letzten ausgefuehrten Kommandos',nl,nl],
                         ['Hilfe beenden',nl]],Zahl), nl,
          ( Zahl =:= 1, hilfe_kommandos ;
            Zahl =:= 2, write_neu([' NOCH NICHT VERFUEGBAR !!',nl,nl]) ;
            Zahl =:= 3,  hilfe_fehler ;
            Zahl =:= 4,  hilfe_letztes_komm ;
            Zahl =:= 5 ), !.

hilfe_kommandos :-
  write_neu([' Folgende Kommandos sind ausfuehrbar:',nl,nl]),
  menue_eingabe([['lade die Aufgabendatei <Name>',nl],
                 ['lade die Regeldatei <Name>',nl],
                 ['speichere die Regeln in Datei <Name>',nl],
                 ['speichere die Loesung in Datei <Name>',nl],
                 ['schalte Trace ein|aus',nl],
                 ['schalte Traceausgabe lang|kurz',nl],
```

```
                    ['dimensioniere',nl],
                    ['erklaere die Inferenz',nl],
                    ['zeige die Regel <Regelname>',nl],
                    ['zeige alle Regeln',nl],
                    ['zeige die Aufgabe',nl],
                    ['zeige die Loesung',nl],
                    ['zeige den Systemzustand',nl],
                    ['loesche die Aufgabe',nl],
                    ['loesche die Regel <Regelname>',nl],
                    ['loesche alle Regeln',nl],
                    ['hilfe',nl],
                    ['beende',nl,nl],
                    ['ABBRUCH',nl]],Zahl),nl,
  ( Zahl =:= 1, Kommando = lade_aufgabe ;
    Zahl =:= 2, Kommando = lade_regeln ;
    Zahl =:= 3, Kommando = speichere_regeln ;
    Zahl =:= 4, Kommando = speichere_loesung ;
    Zahl =:= 5, Kommando = setze_trace ;
    Zahl =:= 6, Kommando = setze_traceausgabe ;
    Zahl =:= 7, Kommando = dimensioniere ;
    Zahl =:= 8, Kommando = erklaere_inferenz ;
    Zahl =:= 9, Kommando = zeige_eine_regel ;
    Zahl =:= 10, Kommando = zeige_alle_regeln ;
    Zahl =:= 11, Kommando = zeige_aufgabe ;
    Zahl =:= 12, Kommando = zeige_loesung ;
    Zahl =:= 13, Kommando = zeige_zustand ;
    Zahl =:= 14, Kommando = loesche_aufgabe ;
    Zahl =:= 15, Kommando = loesche_eine_regel ;
    Zahl =:= 16, Kommando = loesche_alle_regeln ;
    Zahl =:= 17, Kommando = hilfe ;
    Zahl =:= 18, Kommando = stop ;
    true ),
  ( nonvar(Kommando), get_hilfe(Kommando,[]),nl ; true ), !.


hilfe_fehler :- get_letzten_fehler(0,_,_), write_neu([' Sie haben noch gar keinen Fehler gemacht !', nl,nl]), !.

hilfe_fehler :- get_letzten_fehler(FNr,_,_),  zeige_letzten_fehler,
                write_neu([' Erlaeuterung: ',nl,nl]), concat_atom(['fehler',FNr],FId),
                get_hilfe(FId,[]), nl, !.

hilfe_letztes_komm :- get_zustand(letztes_komm,dummy_),
                      write_neu([' Sie haben noch gar kein Kommando ausgefuehrt !',nl,nl]), !.

hilfe_letztes_komm :-
  get_zustand(letztes_komm,[Kommando,Fehler]),
  functor(Kommando,KName,AnzArg),
  write_neu([' Als letztes wurde das Kommando "',quote(KName),'" ausgefuehrt !',nl,nl]),
  ( AnzArg =:= 0,  write_neu([' Dieses Kommando besass keine Argumente !',nl,nl]) ;
    write_neu([' Dieses Kommando besass die Argumente: ']),
      ( for(1,I,AnzArg),
          arg(I,Kommando,Argument), write_neu([quote(Argument),'  ']), fail ;
        nl,nl ) ),
  ( Fehler = fehlerfrei,  write_neu([' Das Kommando wurde fehlerfrei ausgefuehrt !',nl,nl]) ;
    Fehler = fehler(FNr,Kontext,Info),
      ja_nein_eingabe([' Es trat ein Fehler auf ! Soll der Fehler erklaert werden [j/n]: '],Antwort), nl,
      ( Antwort == ja,
          zeige_fehler(FNr,Kontext,Info), write_neu([' Erlaeuterung: ',nl,nl]),
        concat_atom(['fehler',FNr],FId), get_hilfe(FId,[]), nl ;
        true ) ),
  ja_nein_eingabe([' Soll das Kommando erlaeutert werden [j/n]: '],Antwort1), nl,
  ( Antwort1 == ja, get_hilfe(KName,[]), nl ;true ), !.
```

D. Beispiel einer DABS-Datei

An dieser Stelle soll gezeigt werden, wie eine DABS-Textdatei aussieht, in der eine Dimensionierungsaufgabe beschrieben wird. Dabei beziehe ich mich auf die von mir gelöste Beispielaufgabe **m_halle8**.

Die DABS-Textdatei:

```
/* Dimensionierung von Parametern einer Maschinenhalle, die */
/* durch ein CS-Modell beschrieben wird                     */
/*                                                          */
/* Sven Hader, 01REA88,  16.4.1993                          */

datenobjekte.  % hier muessen saemtliche Datenobjekte def. werden

   anz_dev      :: skalar.      % Austauscheinheit + Maschinen
   anz_jobs     :: skalar.      % Anzahl Auftraege
   bearb_int    :: vektor(6).   % Bearbeitungsintensitaet
   job_vert     :: vektor(6).   % Verteilung der Auftraege

   st_preis     :: skalar.      % Verkaufspreis (pro Stueck)
   t_kosten     :: skalar.      % Transporterkosten (pro ZE)
   a_kosten     :: skalar.      % Arbeitskosten (pro Stueck und ZE)
   st_zahl      :: skalar.      % gefertigte Produkte (pro ZE)
   b_zeit       :: skalar.      % mittl. Bearb.zeit (pro Stueck)

   a_grad          :: vektor(6).   % Auslastungsgrad
   durchsatz_dev   :: vektor(6).   % Durchsatz
   v_dauer         :: vektor(6).   % Verweildauer
   a_jobs          :: vektor(6).   % Anzahl Auftraege in Maschine
   w_zeit          :: vektor(6).   % mittlere Wartezeit

konstanten.   % konstante Datenobjekte  (oder Teile davon)

   anz_dev     ::  [6].
   bearb_int   ::  [50, 10, 15, 20, 10, 15].
   job_vert:1  ::  [0.0].
   job_vert:4  ::  [0.12].
   job_vert:5  ::  [0.17].

   st_preis    ::  [5000].
   t_kosten    ::  [500].
   a_kosten    ::  [10].

parameter.    % Parameter (koennen vom System geaendert werden)

   anz_jobs       ::  bereich(10,100,integer)  :: [100].
   job_vert:2..3  ::  bereich(0.0,1.0,real)    :: [0.2, 0.2].
   job_vert:6     ::  bereich(0.0,1.0,real)    :: [0.31].

abhaengige.
```

```
st_zahl  =  durchsatz_dev:1.
b_zeit   =  v_dauer:1 + sum(i=2..anz_dev, job_vert:i*v_dauer:i).
```

zielfunktion. % Zielfunktion und Optimalitaetskriterium

```
( st_preis*st_zahl - t_kosten*anz_jobs - a_kosten*st_zahl*b_zeit )  ::  maximum.
```

nebenbedingungen.

```
( sum(i=1..6,job_vert:i) = 1.0 ).
```

schnittstelle. % Schnittstellenbeschreibung

```
c_serv1  ::  [anz_dev,anz_jobs,bearb_int,job_vert]
         ::  [a_grad,durchsatz_dev,v_dauer,a_jobs,w_zeit].
```

E. Beispiel einer RDS-Datei

An dieser Stelle soll bezeigt werden, wie eine RDS-Textdatei aussieht, die die für die Dimensionierung notwendigen Regeln enthält. Dabei gebe ich als Beispiel die Datei an, mit der die Aufgaben **m_halle1** bis **m_halle8** gelöst wurden.

Die RDS-Textdatei:

```
/* Regelwissen zur Dimensionierung einer Maschinenhalle, die */
/* sich als CS-Modell darstellen laesst                      */
/*                                                           */
/* Sven Hader , 01REA88 ,   14.4.1993                        */

% Regeln fuer anz_jobs als Parameter

   anz_jobs1  :: [ art von anz_jobs ist parameter,
                   untere_grenze von anz_jobs ist UG,
                   obere_grenze von anz_jobs ist OG,
                   A_Jobs <- trunc(max(UG,min(OG,bearb_int:1))) ]
              :: [ anz_jobs <- A_Jobs, berechne_zielfunktion ].

   anz_jobs2  :: [ art von anz_jobs ist parameter,
                   obere_grenze von anz_jobs ist OG,
                   Add <- round( (OG-anz_jobs) / 2), Add > 1 ]
              :: [ anz_jobs <- anz_jobs + Add, berechne_zielfunktion ].

   anz_jobs3  :: [ art von anz_jobs ist parameter,
                   obere_grenze von anz_jobs ist OG,
                   Add <- round( (OG-anz_jobs) / 10), Add > 1 ]
              :: [ anz_jobs <- anz_jobs + Add, berechne_zielfunktion ].

   anz_jobs4  :: [ art von anz_jobs ist parameter,
                   obere_grenze von anz_jobs ist OG, anz_jobs < OG ]
              :: [ anz_jobs <- anz_jobs + 1, berechne_zielfunktion ].

   anz_jobs5  :: [ art von anz_jobs ist parameter,
                   untere_grenze von anz_jobs ist UG,
                   Sub <- round( (anz_jobs-UG) / 2), Sub > 1 ]
              :: [ anz_jobs <- anz_jobs - Sub, berechne_zielfunktion ].

   anz_jobs6  :: [ art von anz_jobs ist parameter,
                   untere_grenze von anz_jobs ist UG,
                   Sub <- round( (anz_jobs-UG) / 10), Sub > 1 ]
              :: [ anz_jobs <- anz_jobs - Sub, berechne_zielfunktion ].

   anz_jobs7  :: [ art von anz_jobs ist parameter,
                   untere_grenze von anz_jobs ist UG,
                   anz_jobs > UG ]
              :: [ anz_jobs <- anz_jobs - 1, berechne_zielfunktion ].
```

```
% Regeln fuer job_vert als Parameter

   job_vert3  :: [ summe von bearb_int:K mit
                   [ art von job_vert:K ist parameter ] ist SumBI,
                   art von job_vert:I ist parameter, I <> 1,
                   obere_grenze von job_vert:I ist OGi,
                   OGi > job_vert:I,
                   job_vert:I < bearb_int:I/SumBI,
                   art von job_vert:J ist parameter, J <> 1, J <> I,
                   untere_grenze von job_vert:J ist UGj,
                   UGj < job_vert:J,
                   job_vert:J > bearb_int:J/SumBI,
                   Diff <- min( min(OGi-job_vert:I,bearb_int:I/SumBI-job_vert:I),
                          min(job_vert:J-UGj,job_vert:J-bearb_int:J/SumBI) ), Diff > 0.001 ]
                :: [ job_vert:I <- job_vert:I + Diff,
                   job_vert:J <- job_vert:J - Diff,
                   berechne_zielfunktion ].

   job_vert4  :: [ maximum von a_grad:I mit
                   [ art von job_vert:I ist parameter ] ist Ai,
                   minimum von a_grad:J mit
                   [ art von job_vert:J ist parameter , J <> I ] ist Aj,
                   bearb_int:I < bearb_int:J,
                   untere_grenze von job_vert:I ist UGi,
                   obere_grenze von job_vert:J ist OGj,
                   Diff <- min(OGj-job_vert:J,job_vert:I - UGi), Diff > 0.01 ]
                :: [ job_vert:I <- job_vert:I - Diff/10,
                   job_vert:J <- job_vert:J + Diff/10,
                   berechne_zielfunktion ].

   job_vert5  :: [ maximum von a_grad:I mit
                   [ art von job_vert:I ist parameter ] ist Ai,
                   maximum von bearb_int:J mit
                   [ art von job_vert:J ist parameter ] ist JVj, I <> J,
                   untere_grenze von job_vert:I ist UGi,
                   obere_grenze von job_vert:J ist OGj,
                   Diff <- min(OGj-job_vert:J,job_vert:I - UGi), Diff > 0.01 ]
                :: [ job_vert:I <- job_vert:I - Diff/10,
                   job_vert:J <- job_vert:J + Diff/10,
                   berechne_zielfunktion ].

   job_vert6  :: [ maximum von a_grad:I mit
                   [ art von job_vert:I ist parameter ] ist Ai, Ai >= 0.97,
                   minimum von a_grad:J mit
                   [ art von job_vert:J ist parameter ] ist Aj, I <> J,
                   untere_grenze von job_vert:I ist UGi,
                   obere_grenze von job_vert:J ist OGj,
                   Diff <- min( min(OGj-job_vert:J,job_vert:I - UGi), Ai-0.97 ), Diff > 0.001 ]
                :: [ job_vert:I <- job_vert:I - Diff,
                   job_vert:J <- job_vert:J + Diff,
                   berechne_zielfunktion ].
```

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur angefertigt habe.

Chemnitz, den 20. April 1993

Thesen

1. In der Informatik bildet sich in den letzten Jahren ein neues Paradigma heraus, das der **wissensbasierten Programmierung**. Das zeigt sich vor allem in der Nutzung neuartiger Programmiertechniken in allen Bereichen der Informatik. Dazu gehören z.B. die Nutzung komfortabler Expertensystemshells und die Verwendung objektorientierter Programmiersprachen.

2. Bei der Entwicklung großer Programmsysteme sollte stets die Frage berücksichtigt werden, ob sich die Aufgabe durch Verwendung wissensbasierter Methoden nicht effizienter lösen läßt. Durch die explizite Formulierung des verwendeten Wissens und dessen leichte Modifizierbarkeit erhöht sich die Flexibilität und Wartungsfreundlichkeit des Programmes wesentlich.

3. Der **Entwurf technischer Systeme** ist ein Prozeß, der ein hohes Maß an menschlicher Intelligenz erfordert. Das dabei verwendete Wissen ist zum großen Teil heuristisches Erfahrungswissen des Konstrukteurs.

4. Die optimale **Dimensionierung technischer Systeme** ist ein Vorgang, bei dem sich wissensbasierte Methoden nutzbringend anwenden lassen, um dem Konstrukteur einen Teil seiner Arbeit abzunehmen. Dabei sollte ein Ansatz gewählt werden, der die Problemlösungsmethodik des Konstrukteurs 'simuliert'.

5. Das Erfahrungswissen des Konstrukteurs bei der Dimensionierung läßt sich in Form von **WENN-DANN-Regeln** ausdrücken. Dabei entspricht jede Regel einem in sich abgeschlossenen Wissens-'Stück'. Durch die Verwendung eines **vorwärtsverkettenden Regelinterpreters** können diese Regeln verarbeitet werden.

6. Die Transformation des Erfahrungswissens des Konstrukteurs in die Form von WENN-DANN-Regeln ist ein komplizierter Prozeß, der ein hohes Maß an Einsicht in das zu bearbeitende Aufgabengebiet erfordert. Dieser Prozeß erfolgt i.allg. inkrementell, d.h. daß die entstandene Regelmenge anhand von Beispielen auf ihre Vollständigkeit getestet und wenn notwendig erweitert wird.

7. Die Prozeß der expliziten Formulierung der Dimensionierungsregeln führt beim Konstrukteur i.allg. zu einer größeren Einsicht in die Wirkungsweise und das Verhalten der betrachteten technischen Systeme. In diesem Sinne kann die Formulierung der Regeln als Aufstellen von Hypothesen und der Test der Regeln als Prüfen der Richtigkeit dieser Hypothesen verstanden werden.

8. Das entwickelte Programmsystem ist in der Lage, bei Vorgabe einer Menge von Regeln das Problemlöseverhalten eines Konstrukteurs nachzuvollziehen und Dimensionierungsaufgaben zu lösen. Die Anwendbarkeit des Programmes wurde anhand mehrerer kleinerer Beispiele nachgewiesen. Über das Verhalten des Programmes bei großen Regelmengen bzw. komplexen Aufgaben liegen keine gesicherten Erkenntnisse vor. Jedoch ist zu vermuten, daß sich die Geschwindigkeit der Problemlösung in diesen Fällen stark verlangsamen wird.