552 | Juni 1995

SCHRIFTENREIHE SCHIFFBAU

Isa Afsin Toparlak

Wissensbasierte Verarbeitung der technischen Regeln am Beispiel der schiffbaulichen Bauvorschriften

TUHH
Technische Universität Hamburg-Harburg

Wissensbasierte Verarbeitung der technischen Regeln am Beispiel der schiffbaulichen Bauvorschriften

Isa Afsin Toparlak, Hamburg, Technische Universität Hamburg-Harburg, 1995

ISBN: 3-89220-552-3

Schriftenreihe Schiffbau
Schwarzenbergstraße 95c
D-21073 Hamburg

http://www.tuhh.de/vss

Wissensbasierte Verarbeitung der technischen Regeln am Beispiel der schiffbaulichen Bauvorschriften

Dissertation
zur Erlangung des Grades
Doktor-Ingenieur
der Universität Hamburg

vorgelegt von

İsa Afşin Toparlak
aus Söke/Türkei

Hamburg
Juni 1994

INSTITUT FÜR SCHIFFBAU DER UNIVERSITÄT HAMBURG

Bericht Nr. 552

Wissensbasierte Verarbeitung der technischen Regeln am Beispiel der schiffbaulichen Bauvorschriften

İsa Afşin Toparlak

Juni 1995

Preface

"Make it as simple as possible, but not simpler."

ALBERT EINSTEIN

A ship's steel structure represents a complex technical object, managing and supporting of the design of such objects requires sophisticated computer-based systems. The objective of my research work has been the development of facilities to support the dimensioning process which is one of the most important tasks in the steel ship construction. Two methods exist for this purpose: structure analysis with the finite element method and dimensioning according to the classification rules. One important aspect is that the finite element analysis depends on the results of a pre-dimensioning according to the classification rules.

Adding the feature of dimensioning according to the classification rules to the functionality of the current shipbuilding CAD systems is of big complexity due to the poor capability of interfaces for programming and data management. The existing interfaces are often restricted because of consistency and security aspects of the underlying CAD product.

Under the consideration of the retrievability of the needed data from CAD system, the dimensioning process according to the rules can be seen in two contexts: I) for the already dimensioned parts of the structure; it will verify and eventually modify the dimensions, i.e. if the existing dimensions can not satisfy the constraints of the classification rules, II) dimensioning according to the classification rules follows if the structure was not dimensioned previously. The import back to the CAD system is possible if the appropriate interfaces made available. The import and export aspects may be neglected if the product data is stored in a shared storage which is not yet the practice.

The existing computer programs for dimensioning the ship structure according to the codes and standards of classification societies can neither handle any standard product model nor communicate with shipbuilding CAD systems. Another important lack of these programs is the poor update capability reasoned by the underlying implementation languages and software conceptions which are mostly procedural and also of poor suitability for programming of rules.

A knowledge based rule programming system was developed. It is based on a problem oriented language which is designed to formulate the dimensioning rules with a close correspondence to the original rule text. The system is then able to interpret directly this specification which allows the programming of dimensioning rules in a declarative manner. The knowledge about how the rule is implemented is no more necessary to modify the system which is a very often case with conventional, mostly procedural systems. The architecture of the system is also based on a unified data model which makes it an open system. Having an open system architecture simplifies its integration to the most of existing engineering systems.

I thank Prof. Dr.–Ing. Eike Lehmann. His helpfulness and liberality, and his pragmatism made the best platform to carry out this research work. I thank also Dr.–Ing. Thomas Koch who helped me in every question.

I thank my wife Elif who motivated me to make my homework. Her patience was indispensable.

Inhaltsverzeichnis

Abkürzungen

AIM	*Application Interpreted Product Model*
CAD	*Computer-Aided Design*
CAE	*Computer-Aided Engineering*
CAM	*Computer-Aided Manufacturing*
DBMS	*Data Base Management System*
DBVS	*Datenbankverwaltungssystem*
DCG	*Definite Clause Grammars*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
DRL	*Dimensioning Rule Language*
dDBVS	*deduktives Datenbankverwaltungssystem*
FEA	*Finite Element Analysis*
FEM	*Finite Element Method*
GUI	*Graphical User Interface*
ISO	*International Standardisation Organization*
IT	*Informationstechnik*
Lisp	*List Processing Language*
Prolog	*Programming in Logic*
RID	*Rechnerinterne Darstellung*
RWVS	*Regelwerkverwaltungssystem*
SPS	*STEP–Parser–System*
STEP	*STandard for the Exchange of Product Model Data*
UoD	*Universe of Discourse*

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Rechnerunterstützte Systeme werden in immer weitergehendem Umfang auf Werften und dem Schiffbau nahestehenden Unternehmen sowie von Klassifikationsgesellschaften, Ingenieurbüros und Forschungsinstitutionen benutzt und weiter ausgebaut. Die Einsatzbereiche dieser Systeme umfassen den Entwurf, die Konstruktion und die Fertigung von Schiffen und meerestechnischen Anlagen. Die heute im Schiffbau verwendeten CAD–Systeme decken insbesondere die Bereiche Schiffsoberflächenentwurf und Konstruktion der Stahlstruktur ab. FEA–Systeme werden zur Bewältigung der Aufgabenstellungen wie Konstruktionsbeurteilung, Strukturoptimierung und Bemessung von Bauteilen eingesetzt.

Die Fähigkeiten dieser schiffbauspezifischen CAD–Systeme unterscheiden sich meist aus entwicklungsbedingten technischen Gründen voneinander. Im einzelnen befassen sich diese Systeme nur mit Teilaufgaben, und es werden kontinuierlich neue Funktionalitäten hinzugefügt. Aufgrund von verschiedenen Konzeptionen und Bedingungen sind zahlreiche branchen– oder problemspezifische Systeme (*Insellösungen*) entwickelt worden. Dabei war es immer notwendig, nur bestimmte eingegrenzte Anwendungsbereiche ins Auge zu fassen.

Aufgrund fehlender Kommunikationsmöglichkeiten, verschlechtert durch die nicht einheitliche Soft– und Hardware–Architektur, entsteht ein zusätzlicher Zeit– und Kostenaufwand, wenn die Systeme koordiniert werden sollen. Beispielsweise können die mittels eines CAD–Systems erstellten Daten für den Schiffsentwurf nicht ohne weiteres von einem anderen verwendet werden, das für die Konstruktion der Stahlstruktur eingesetzt wird. Dies bedingt, daß die Eingabedaten, z.B. für die Begrenzung der Bauteile durch den Schiffskörper, nochmals evtl. manuell, erstellt werden müssen. Wesentliche Vorteile dieser Systeme lassen sich daher für eine rechnergestützte Konstruktion mit den derzeitigen Möglichkeiten nicht im vollen Umfange ausnutzen.

Strukturanalyse–Systeme, die auf der Finite–Elemente–Methode beruhen, werden von

Klassifikationsgesellschaften, Ingenieurbüros, Forschungsinstitutionen und z.T. auf den Großwerften, wie bereits erwähnt, für die Beurteilung der Konstruktion, Strukturoptimierung und Bauteilbemessung eingesetzt. Die FE–Analyse liefert im allgemeinen genauere Ergebnisse, wobei sie aber eine Vordimensionierung voraussetzt. Die Vordimensionierung erfolgt nach einfachen statischen Berechnungen, nach Bauvorschriften mit überwiegend **manuellen** Berechnungen oder teilweise nach den Erfahrungen des Konstrukteurs (*Augenmaß*). So ist die Dimensionierung der Stahlstruktur mit FE–Methoden keine eigenständige Lösung, sondern nur eine Erweiterung bzw. Verbesserung der ersten Auslegung nach den Klassifikationsregeln. In der Tat hat die FE–Analyse einen *verifizierenden* Charakter.

Für die Bemessung der schiffbaulichen Stahlkonstruktion nach den Vorschriften existieren Programme, die aufgrund fehlender Schnittstellen mit den derzeit verfügbaren schiffbaulichen CAD–Systemen nicht kommunizieren können. Die meisten dieser Programme eignen sich nicht für eine eventuelle Aktualisierung, jedenfalls nicht ohne großen Aufwand, da die Bauvorschriften relativ oft geändert werden. Die Abbildung der Vorschriften auf *prozedurale* Algorithmen durch den Einsatz imperativer Programmiersprachen (z.B. FORTRAN) macht diese Programme oft unüberschaubar, da die Transparenz zwischen der Vorschrift und dem Programm wegen des aufwendigen manuell geleisteten Abbildungsprozesses verloren geht, was andererseits eine Integration der Programme zu den bestehenden Systemen verkompliziert. Hierfür ist unter dem Begriff 'Integration' z.B. eine Anpassung an das vorhandene CAD– oder FEA–System oder an das Betriebssystem zu verstehen. Die Portierbarkeit der dabei verwendeten Programmiersprache spielt daher eine sehr große Rolle, da die Teilnehmer im Genehmigungsprozeß über recht unterschiedliche Software bzw. Hardware–Plattformen verfügen.

Wegen der großen Anzahl von Vorschriften, deren Erneuerung und deren unterschiedlicher Einsatzgebiete ist der Entwickler eines solchen Programms oft nicht in der Lage, das Programm zu aktualisieren bzw. an spezielle Anforderungen mit derzeit verfügbaren Möglichkeiten anzupassen.

Diese Schwierigkeiten sind im allgemeinen dadurch bedingt, daß keine einheitliche Information über das Objekt *"Schiff"* verfügbar ist, weil bei den meisten Programmsystemen zur Unterstützung der schiffbaulichen Konstruktion die größten Anteile des Implementierungsaufwands sowie des Ausführungsaufwands in der Verwaltung der riesigen Datenmengen bestehen. Nicht selten beschäftigt sich der Anwender eines CAD–Systems mit der rechnerinternen Strukturierung der systemeigenen Daten oder mit Einzelheiten der Datenverwaltungskomponenten. So sind Nutzer sowie Entwickler solcher Systeme oft mit Randproblemen der eigentlichen Problematik, wiederum durch die nicht vereinheitlichte Soft– und Hardware verstärkt, konfrontiert.

Zur Unterstützung der Dimensionierung nach Vorschriften soll im Rahmen der vorliegenden Arbeit ein *wissensbasiertes* System entwickelt werden, das Aspekte wie *Pflege*, *Erweiterung* und *Anpassung* besonders berücksichtigt. Der wissensbasierte Programmieransatz hat gegenüber dem konventionellen, prozeduralen (*problemlösenden*) Ansatz einen

deklarativen, d.h. *problembeschreibenden*, Charakter. Aus einer ausreichenden Beschreibung des Problems läßt sich eine Lösung ableiten (*Inferenz*, Folgerung). Das geplante System muß in der Lage sein, eine für die Problematik geeignete Sprache zu verarbeiten. Die *problemorientierte* Sprache soll zur Spezifikation von Vorschriften dienen und das Programmieren der Vorschriften erleichtern. So muß sich der Entwickler eines Regelwerks[1] nicht mit einzelnen Befehlen einer allgemeinen Programmiersprache beschäftigen, die umständlich[2] zu beherrschen sind. Durch die Definition einer problemorientierten deklarativen Programmiersprache wird eine geeignete Transparenz zwischen der Vorschrift und dem Programm geschaffen. Dadurch wird erreicht, daß der Quellcode eines solchen durch diese problemorientierte Sprache erstellten Programms dem textlichen Inhalt der Vorschrift entspricht. Dies kann sogar den Weg öffnen, daß der Gesetzestext der Vorschrift als Programm erfaßt wird, wenn er in einer für den Rechner verständlichen Form vorliegt. Dies ist durchaus technisch zu bewältigen, wenn die Grundlagen dafür geschaffen sind.

Die Verknüpfung dieses Systems mit bereits existierenden CAD–Systemen muß konzipiert werden, um die Datenredundanz und mögliche Fehler bei einer wiederholten Datenaufbereitung zu vermeiden. Diese Verknüpfung soll möglichst durch die Verwendung von neutralen, d.h. *systemunabhängigen*, Schnittstellen realisiert werden. Der Zugriff eines solchen Systems auf ein genormtes Produktmodell verbessert die Wiederverwendbarkeit der erwähnten Verknüpfung mit anderen Systemen.

Über den Stand der heute verfügbaren Programme zur Dimensionierung der schiffbaulichen Stahlkonstruktion wird in Kapitel 2 diskutiert. Das sich im Rahmen der internationalen Normungsbemühungen befindliche Produktmodell für die schiffbauliche Stahlstruktur wird kurz dargestellt und auf seine Relevanz für die vorliegende Problematik hin geprüft.

Kapitel 3 beschäftigt sich mit der Analyse der technischen Bauvorschriften, die für den schiffbaulichen Konstruktionsprozeß relevant sind. Es wird versucht, das Dimensionierungsproblem mit den gemeinsamen Charakteristika der einzelnen Klassevorschriften grob zu formalisieren. Abschnitte aus dem Regelwerk des Germanischen Lloyd dienen dazu als Beispiel.

Eine Analyse der Möglichkeiten der heutigen Informationstechnik zur Realisierung von Programmsystemen für die schiffbaulichen Bauvorschriften wird im Kapitel 4 durchgeführt. Die Auswahl der verwendeten Werkzeuge (Programmieransatz, Programmiersprache, –umgebung etc.) wird begründet.

Für die Realisierung eines geeigneten Systems zur Verarbeitung der Dimensionierungsvorschriften wird in Kapitel 5 eine Systemarchitektur vorgeschlagen, bei der die Beziehung der einzelnen Komponenten eines wissensbasierten Systems zu dem Produkt-

[1] Der Begriff 'Regelwerk' findet in der vorliegenden Arbeit zweierlei Bedeutung: Regelwerk als Vorschriften selbst und Regelwerk als Programm für die Vorschriften.

[2] Die Konzentration für die zugrundeliegende Problematik wird mit technischen Details dieses Hilfsmittels, der Programmiersprache und deren jeweiliger Implementation, geschwächt.

modell berücksichtigt wird. Die Systemarchitektur besteht hauptsächlich aus einem einheitlichen Datenmodell für die Beschreibung der für das Programmieren von Vorschriften relevanten Aspekte der Stahlschiffsstruktur, das sich an dem STEP–Standard orientiert, und einem wissensbasierten System für das Programmieren von Dimensionierungsvorschriften, welches an ein *einheitliches* Datenmodell gekoppelt ist. Somit ist die hier vorgeschlagene Systemarchitektur der Grundbaustein eines **offenen** Systems.

Die Möglichkeiten des wissensbasierten Programmieransatzes werden an Hand der Programmiersprache Prolog und ihrer theoretischen Grundlage (*Logische Programmierung*) in Kapitel 6 erläutert.

Kapitel 7 beschäftigt sich mit der Abbildung und der Verarbeitung der schiffbautechnischen Vorschriften mittels Prolog, wobei eine problemorientierte Sprache als anwenderfreundliche Zwischenstufe ins Spiel kommt. Das in der vorliegenden Arbeit entwickelte wissensbasierte System zur Erstellung prototypischer Dimensionierungsprozessoren wird vorgestellt.

Die Implementation gemäß den Kapiteln 5 und 7 und ihre technischen Details werden in Kapitel 8 besprochen. An Hand einiger zusätzlicher Prototyp–Implementationen, die als Erweiterung der Funktionalität des wissensbasierten Systems anzusehen sind, wird gezeigt, wie dem System ein neues Anwendungsprofil zugeteilt werden kann.

In Kapitel 9 werden die Ergebnisse bewertet und über die zukünftigen Verbesserungen bzw. Erweiterungen der realisierten Pilotanwendung bzw. Implementation Aussagen gemacht.

Das vorliegende wissensbasierte System erlaubt mittels einer problemorientierten Regelsprache die Spezifikation von Dimensionierungsregeln, die unmittelbar von dem System interpretiert werden kann, wodurch eine direkte Korrespondenz zwischen dem Programm und dem Regeltext der Bauvorschrift erreicht wird. Das wissensbasierte Programmieren stellt für die Programmierung von Vorschriften in dem vorliegenden Fall eine sehr effektive Form dar. Die Änderung der Vorschriften bzw. das *Update* von solchen Programmen ist kein charakteristisches Problem mehr, da das Programm und die Vorschrift in einer engen Beziehung zueinander stehen. Um ein Programm modifizieren bzw. aktualisieren zu können, sind die detaillierten Kenntnisse über die Implementation und die ihr zugrundeliegende Programmiersprache nicht erforderlich.

Kapitel 2

Stand der Technik

2.1 Programme zur Verarbeitung schiffbaulicher Bauvorschriften

Um sich über den Inhalt von schiffbaulichen Regelwerken, allgemein von Texten, zu informieren, gibt es durch die Verwendung von Rechnern eine Reihe von Hilfsmitteln. Beispiele sind *Indizierung*, *Hyphertext* und *Volltextsuche*.

ABS[1] bietet als erste Klassifikationsgesellschaft ihre Vorschriften auf dem elektronischen Medium CD an. Ein mitgeliefertes Programm unterstützt die Volltextsuche nach mehreren Begriffen sowie Verzweigungen durch sogenannte *hot–links* zu anderen Stellen des Textes.

Das VEGA–System von DNV[2] deckt sowohl IMO–Vorschriften als auch die norwegischen Forderungen ab. Es ist eine Art Datenbank, in der Texte und Grafiken einzelner Regeln, in Abhängigkeit von Schiffstyp, Ladung, Fahrtgebiet, Länge zwischen den Loten, Zuladung etc., gespeichert sind. Durch die Eingabe der relevanten, jedoch sehr allgemeinen und beschränkten Angaben läßt sich z.B. eine Liste der Regeln, die erfüllt werden müssen, erstellen. Die Interpretation bzw. die Anwendung der so gefundenen Regeln ist jedoch dem Anwender selbst überlassen.

Ähnlich wie das DNV–System VEGA verhält sich das von Lloyd's Register (LR) ausgelieferte System RULE*finder*, mit dem Unterschied, daß, zumal in den Anfängen, eine Schnittstelle zu dem ebensfalls LR-eigenen Auslegungsprogramm LRPASS für die Interpretation der relevanten Regel vorgesehen ist. Diese Schnittstelle ist jedoch erweiterungsbedürftig,

[1] American Bureau of Shipping
[2] Det Norske Veritas

Programm	Organisation	Vorschrift
SFOLDS	BMT	IMO Intakt–, Leckstabilität, Längsfestigkeit
AutoHYDRO	CoastDesign	IMO Intakt–, Leckstabilität, Längsfestigkeit
AutoCLASS	CoastDesign	Bauteildimensionierung nach DNV Vorschriften
PILOT	DNV	IMO Intakt–, Leckstabilität, Längsfestigkeit, lokale Festigkeit
GLRULES	GL	Stahlschiffbau nach Vorschriften des Germanischen Lloyd
SEASAFE	LR	IMO Intakt–, Leckstabilität
LRPASS	LR	Stahlschiffbau nach Vorschriften Lloyd's Register
NAPA	NAPA	IMO Intakt–, Leckstabilität, Längsfestigkeit

Tab. 2.1: *Einige wichtige Auslegungsprogramme*

da sie nur die Regeln unterstützt, die bereits in dem Auslegungsprogramm implementiert sind. Daher ist diese Schnittstelle in der Tat nicht interpretativ. Sie dient nur noch dazu, die entsprechenden Regeln aus dem Auslegungsprogramm einfacher aufzurufen, in welchem Aufrufparameter z.T. bereits in dem LR–System RULE*finder* ermittelt sind. Immerhin stellt das RULE*finder*–System eine große Hilfe dar. Eine allgemeinere, interpretative Schnittstelle für die Auslegung nach den relevanten Regeln wird vermißt, wenn diese nicht bereits in dem Auslegungsprogramm vorliegt. Das RULE*finder*–System enthält LR–eigene Bauvorschriften und einige internationale Vorschriften wie SOLAS.

Es existiert eine ganze Reihe von individuellen Lösungen für einzelne, meist eng begrenzte Teilbereiche von Regelwerken, die zur Durchführung der Dimensionierung in den Auswerteprogrammen integriert werden. Die in der Tabelle 2.1 kurz dargestellten Programme mit Ausnahme von LRPASS haben keine Schnittstellen zu den im Eingang besprochenen Systemen. Es ist keine andere Möglichkeit vorgesehen, mit diesen Systemen in Verbindung zu kommen, so daß die Vorteile dieser Systeme nicht in vollem Umfange ausgenutzt werden können.

Alle diese Programme sind Speziallösungen in den erläuterten Teilgebieten. Ein allgemeiner Ansatz ist nicht bekannt geworden. Als Implementationssprache wurde meistens die Programmiersprache FORTRAN 77 eingesetzt. Aspekte des Software Engineering wie Portabilität, Wiederverwendbarkeit, Erweiterbarkeit sowie Pflege wurden, soweit bekannt, von keinem der Systeme berücksichtigt. Dieses ist die Aufgabenstellung in der

vorliegenden Arbeit. Ähnlich wie die vorliegende Arbeit geht JENSEN [Jensen 92] einen erstmalig allgemeineren Weg an, der die obigen Fragen der Softwaretechnik berücksichtigt. Software–Engineering–Werkzeuge wie `yacc` und `lex` des Betriebsystems UNIX werden dort eingesetzt, um eine speziell für die Vorschriften entworfene Regelsprache in Programmiersprache C zu übersetzen. Dieser Schritt reduziert den manuellen Codierungsaufwand der Regelwerke in spezielle Programmiersprachen wie FORTRAN bzw. C und verbessert somit die Chancen für zukünftige Modifikation und Pflege. Da das System in C realisiert ist, bietet es große Portabilität.

Zwar bietet das System eine schnelle Realisierung der Auslegungsroutinen durch den Einsatz einer problemorientierten Regelsprache, es kann aber wegen der erforderlichen und zum Teil mehrphasigen Compilationsschritte unflexibel werden, da der generierte Modul an eine spezielle Laufzeitbibliothek jedesmal neu gebunden werden muß. Die Redefinition der bereits vorliegenden Teile der Bibliothek muß auch entsprechend behandelt werden. Durch die Nutzung neuer Softwaretechniken wie dynamisches Binden (*dynamic linking*) sowie Überladen[3] (*overloading*) können jedoch die besprochenen Nachteile aufgewogen werden. Der dort vorgeschlagene Eingabesyntax ist z.T. nicht leicht vertändlich.

2.2 Produktmodell für die schiffbauliche Stahlstruktur

Ein wesentlicher Kritikpunkt an den im vorangegangenen Abschnitt erläuterten Systemen zur Verarbeitung schiffbaulicher Bauvorschriften ist die Nichtberücksichtigung eines standardisierten beziehungsweise einheitlichen Datenmodells, da der Austausch der Information jedweder Art zwischen den Engineering–Systemen — im schiffbaulichen Kontext — unterstützen und somit die Ausbaufähigkeit derjenigen Systeme fördern sowie deren Lebensdauer verlängern kann.

Die Notwendigkeit, eines solchen, einheitlichen Datenmodells zur Beschreibung eines extrem komplexen Sachgebildes wie das eines Schiffes, ist seit längerem bekannt und ein noch aktuelles Thema. Es wird voraussichtlich auch noch lange so bleiben.

Die ersten Versuche hierzu gehen, soweit bekannt, auf BRONSART [Bronsart 90] zurück, der vor der Initiierung der internationalen Standardisierungsbemühungen im Rahmen der ISO–Norm–Entwicklung erstmalig ein konzeptionelles Datenmodell (Produktmodell) entwickelt hat. Die ersten erfolgversprechenden Erfahrungen machte BRONSART mit der Implementation eines Produktmodells für die schiffbauliche Stahlstruktur auf der Basis eines semi–objektorientierten Datenbankverwaltungssystems DAMOKLES mit dem formalen Entity–Relationship–Modell. Dabei waren funktionale und strukturelle

[3] Das Konzept des Überladens einer Funktion ist meistens in neueren Programmiersprachen wie C++ verfügbar. Eine Portierung des Systems auf C++ ist daher notwendig beziehungsweise vorteilhaft.

Beschreibungen der Stahlkonstruktion und die Einbindung der Geometrie die Hauptmerkmale dieser Modellierarbeit bzw. des daraus entstandenen Softwaremodells.

So wird die zentrale Konzeption der Gliederung des Stahlstrukturmodells in unterschiedliche Beschreibungsebenen von KOCH [Koch 91] weiter ausgebaut, da sie die Voraussetzungen zur Definition von Beschreibungselementen schafft, wie sie für die interessierende Anwendung von zentraler Bedeutung ist beziehungsweise benötigt wird. Die Beschreibung der Stahlstruktur für Modelltransformationszwecke ist auf die Erfassung umfangreicher funktionaler und struktureller Daten angewiesen. Die Modelltransformationen und daher die unterschiedlichen Sichten eines Produktmodells sind die Folgen der benötigten Wiederverwendung der Information zwischen den verschiedenen Engineering–Systemen innerhalb der schiffbaulichen Konstruktionstätigkeit. Der CAD–Datenaustausch zwischen den Entwurf–, Konstruktion–, Analyse– sowie Fertigungssystemen kann als Beispiel genannt werden.

Im Gegensatz zu dem in [Bronsart 90] für die Modellierung eingesetzten Entity–Relationship–Modell wird in [Koch 91] die objektorientierte Modellierungsprache EXPRESS[4] verwendet, die ein wesentlicher Bestandteil der Normungsbemühungen innerhalb des ISO–STEP–Projekts ist. Die folgenden Kapitel werden darauf z.T. detaillierter eingehen. Es wird in [Koch 91] z.B. gezeigt, wie sich die konkreten Daten aus dem schiffbaulichen CAD–System STEERBEAR mit Hilfe eines Equivalenz–Modells abbilden lassen, das sich aus dem generellen Datenmodell Stahlstruktur herleiten läßt und dem internen Datenmodell entspricht. Diese Arbeit stellt einen großen Beitrag in Rahmen der Normungsbemühungen von STEP dar. Das Modell ist z.Z. Bestandteil des schiffbaulichen Anwendungsprotokolls, das im Rahmen des ESPRIT–Forschungsvorhabens MARITIME[5] ausgebaut und weiterentwickelt wird. Die derzeitigen Entwicklungen dort bezüglich des schiffbaulichen Anwendungsmodells (*Shipbuilding Application Protocol*) beinhalten folgende Teile des STEP–Standards, die von Interesse sind:

Shipbuilding Application Protocol	
Arrangements	Part 215
Outfitting and Furnishing	geplant
Moulded Forms	Part 216
Piping	Part 217
Structures	Part 218

Einige weitere Entwicklungen kommen aus dem BMFT–geförderten Verbundsvorhaben ITiS[6]. Das Querschnittsvorhaben ITiS–2B[7] beschäftigt sich z.B. speziell mit der Imple-

[4] ISO/DIS 10303-11 Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 11: *Description Methods: The EXPRESS Language Reference Manual.*

[5] Modelling and Reuse of Information Over Time – ESPRIT III 6041

[6] *Informationstechnik im Schiffbau*

[7] *Datenbankbasierte Implementation von Produktmodellen*

mentierung von Produktmodellen auf der Basis von objektorientierten Datenbankverwaltungssystemen, die möglicherweise die besten Realisierungschancen von Produktmodellen bieten. Sie sollen mit der objektorientierten Sprache EXPRESS modelliert werden.

Schlußfolgerungen

Mit großem Abstand von den Auslegungsprogrammen bietet das in [Jensen 92] vorgestellte Programm RSHELL gute Möglichkeiten, innerhalb kurzer Zeit Dimensionierungsprozessoren als eigenständige Programme zu realisieren. Jedoch ist der Ressourcenverbrauch (CPU–Anteil und Hauptspeicher) laut [Jensen 92] für die tiefgeschachtelten Regeln sehr groß, was in der Tat durchaus vorkommen wird. Einige hundert Regeln mit einer Verschachtelungstiefe von ca. 15 sollen 5 MB Hauptspeicher–Ressourcen in Anspruch nehmen, was unbedingt verbessert werden müßte. In Verdacht stehen daher als Verursacher die dort verwendeten Optimierungsverfahren und die aufwendigen Verfahren für das unscharfe Schließen. Das RSHELL–System realisiert ein eigenes Produktmodell, das hierarchisch aufgebaut ist, mit eigenständiger Datenverwaltung, die in C realisiert ist. Letzteres ist für diesen erheblichen Ressourcenverbrauch verantwortlich. Abgesehen von diesen Kritikpunkten ist der Mangel an Schnittstellen zu neutralen Produktmodellen ein erheblicher Nachteil für den Einsatz des Systems aus. Damit entsteht eine weitere Insellösung.

Die in dieser Arbeit besonders berücksichtigte Notwendigkeit, Systeme mit flexiblen Schnittstellen zu den standardisierten Produktmodellen zu entwickeln, bestätigt sich daher wieder. Jede weitere fortschrittliche Entwicklung des schiffbaulichen Produktmodells wird nur noch Systemen mit neutralen Schnittstellen zugutekommen. Komplizierte Anwendungsfälle wie Auslegungsprogramme können somit auch vor einer eigenständigen Datenverwaltung bewahrt werden, da sich sicherlich allgemein vertretbare Lösungen für die Datenhaltung einsetzen lassen, so z.B. auf der Basis von objektorientierten Datenbanken.

In der vorliegenden Arbeit wird auf Einfachheit[8] sehr großer Wert gelegt. Ein typisches Beispiel ist unscharfes Schließen. [Neumann 88] zeigt z.B., wie man mit Hilfe der Meta–Programmierung einem Prolog–Interpreter beibringen kann, unscharf zu schließen. (siehe dazu auch Abschnitt 8.4)

[8] Ein sehr typischer Anspruch an das Softwaremodell wie in allen anderen UNIX basierenden Software–Werkzeugen bzw. Software–Entwicklungen.

Kapitel 3

Analyse technischer Vorschriften

3.1 Schiffbauliche Bauvorschriften

Die Gestaltung von technischen Gebilden des Komplexitätsgrades von Schiffen ist ein iterativer langwieriger Prozeß, der mit der Einhaltung von vielen verschiedenen sowohl internationalen als auch nationalen Vorschriften bzw. Normen unterworfen ist. Die einzuhaltenden Bestimmungen sind vielfältig und einem ständigen Wandel unterworfen. Die Anwendung der einzelnen Vorschriften hängt von sehr vielen verschiedenen Faktoren ab, z.B. der Bauart, dem Einsatzgebiet und von vielen verschiedenen Eigenschaften des technischen Produktes. Die unterschiedlichen Phasen seines Lebenszyklus vom Entwurf über Konstruktion und Fertigung bis zu Lieferung und Einsatz haben einen weiteren Einfluß in dieser Hinsicht. Weitere Aspekte wie Festigkeit, Stabilität, Sicherheit sowie Umweltschutz kommen hinzu.

Neben der Vielfältigkeit der einzuhaltenden Vorschriften sowohl allgemeiner als auch schiffbauspezifischer Art gibt es eine Vielzahl von verschiedenen sogenannten Gesetzgebern, die sich international, national sowie privat formiert haben. Beispiele hierfür sind Organisationen (ISO, IMO), Institute (DIN), Klassifikationsgesellschaften (ABS, LR, GL, DNV), Berufsgenossenschaften (SBG, USCG) sowie Reedereien und Werften meistens mit unterschiedlichen Gesellschaftsformen. Die Vorschriften der Klassifikationsgesellschaften besitzen dabei einen Querschnittscharakter. Diese enthalten z.B. mehr oder weniger ausgearbeitete Ausschnitte aus den anderen Bestimmungen. In einigen Fällen sind mehrere Regelwerke zu bearbeiten.

Gemeinsam ist den Vorschriften aller Art, daß die Relevanz der zu verwendenden Vorschrift von dem vorliegenden Produkt abhängt. Beispielsweise bestimmen der Typ, das Fahrtgebiet, die Ladung und viele andere Attribute des Schiffes, welche Vorschrift anzuwenden ist. Gemeinsam ist allen Vorschriften auch, daß bestimmte Entscheidungskriterien verfügbar sind, wenn mehrere Vorschriften gleichzeitig relevant erscheinen. Die

Vereinbarungen zwischen Werften und Klassifikationsgesellschaften erleichtern z.B. den Entscheidungsprozeß über den Inhalt und die Relevanz der anzuwendenden Vorschrift bzw. der zu verwendenden Norm.

Die meisten Schwierigkeiten tauchen auf, wenn die Daten über das zu dimensionierende Objekt fehlen, da diese für die Auswahl der relevanten Vorschriften eine wichtige Rolle spielen. Beispiele sind Entwurfsparameter wie Hauptabmessungen, Bauart usw. Das Ermitteln der benötigten Informationen ist daher von besonderer Wichtigkeit für die Auswahl geeigneter Vorschriften sowie für deren Auswertung. Nicht selten dienen manche Vorschriften selbst zur Ermittlung der bestimmten Informationen, die sowohl für die Relevanz als auch für die Interpretation einiger weiterer Regeln erforderlich sind.

Ziel ist es in diesem Kapitel, die für den Dimensionierungsprozeß erforderlichen Informationen — diese sind die relevanten Vorschriften einerseits und die Daten über das zu dimensionierende Objekt andererseits — zu analysieren bzw. zu beschreiben. Diese Analyse dient als Grundlage für Formalismen, auf die zum Zwecke der rechnergestützten Verarbeitung der Dimensionierungsvorschriften nicht verzichtet werden kann. Zunächst wird die klassische Vorgehensweise des Konstrukteurs bzw. des Ingenieurs beim Auslegen einer Konstruktion nach Klassevorschriften dargestellt und untersucht, in welcher Weise ein Dimensionierungsproblem durch Bauteileigenschaften und Auswahl anzuwendender Regeln beschrieben wird.

Dabei wird an Hand der Bauvorschriften des Germanischen Lloyd [GL 92] vorgegangen. Die dabei gewonnenen Erkenntnisse lassen sich aber auf die vergleichbaren Vorschriften anderer Klassifikationsgesellschaften übertragen. Die Informationen werden klassifiziert und den entsprechenden Attributen zugeordnet. Zusätzlich werden die Gültigkeitsbereiche der Daten bzw. Attribute untersucht. Anschließend erfolgt eine Klassifizierung der Regeln in bezug auf die Daten, die das zu dimensionierende Objekt beschreiben. Die Einteilung der Regeln nimmt einen wichtigen Einfluß auf die rechnerinterne Spezifikation der Regeln, was in Kapitel 7 detailliert erläutert wird.

3.1.1 Bauvorschriften des Germanischen Lloyd

Wie jedes andere Regelwerk weisen Bauvorschriften des Germanischen Lloyd einen logischen Aufbau auf, der sich aus den verschiedenen Unterteilungen zusammensetzt. Die zur Unterteilung der Vorschriften verwendeten Kriterien unterscheiden sich, grob formuliert, z.B. in

- sachtechnische Anwendungsgebiete wie die Schiffstechnik, die Werkstoff– und Schweißtechnik, Meerestechnik, nichtmaritime Technik (z.B. Windkraftanlagen) sowie Berechnungstechnik als ein interdisziplinäres Anwendungsgebiet,

- Typ des Produktes wie See– und Binnenschiffe, Wassersportfahrzeuge, Unterwasserfahrzeuge sowie Offshore–Bauwerke,
- Bestandteile des Produktes wie den Schiffskörper, die Maschinenanlagen, die elektrischen Anlagen sowie Kühlanlagen und maschinenbauliche Komponenten,
- verwendetes Baumaterial z.B. metallische und nichtmetallische Werkstoffe etc.

Die GL–Vorschriften sind dabei schon von ihrem Aufbau und ihrer Konzeption her gut als Wissensvorlage für ein Programmsystem mit wissensbasiertem Ansatz geeignet, da sie ein typisches Beispiel für Wissen aus dem Ingenieurbereich darstellen: Sie liegen in natürlich-sprachlicher Form vor und enthalten zusätzlich Formeln, Tabellen und Bilder zur Wissensdarstellung. Die Gliederung der Bauvorschriften des Germanischen Lloyd ist konsequent, ihr Layout gut strukturiert. Sie sind klar in Bände, Teile, Kapitel, Abschnitte und Unterabschnitte eingeteilt, wobei sie nach einem alphanumerischen System geordnet sind. Jedes Element ist somit einem sogenannten Code zugeordnet, der innerhalb der gesamten Vorschrift eindeutig vergeben ist. Eine weitere Unterteilung innerhalb der Unterabschnitte folgt nach einem Dezimalsystem, wobei jeweils eine in sich geschlossene Aussage mit einer eventuellen charakteristischen Überschrift versehen wird. Auf die aussagekräftigen Vorgaben, Bedingungen, Formeln, Graphen, Bilder sowie Querverweise auf andere Stellen in der Vorschrift oder auch auf andere Vorschriften kann damit ein eindeutiger Bezug genommen werden.

Eine tabellarische Zusammenfassung des Codierungsprinzips wird in der folgenden Tabelle kurz aufgelistet:

UNTERTEILUNG	SYSTEM	BEISPIEL	CODE
Band	Römische Ziffern	Schiffstechnik	I
Teil	Dezimalsystem	Seeschiffe	I/1
Kapitel	"	Schiffskörper	I/1/1
Abschnitt	"	Wasserdichte Schotte	11,[1]
Unterabschnitt	Großbuchstaben	Abmessungen	B.
Paragraph	Dezimalsystem	Schottbeplattung	B.2.
Absatz	"	$t = c_p * a * \sqrt{p} + t_K$	B.2.1
"	"	"Die Art und Anordnung ..."	A.3.1.1

Das äußere Erscheinungsbild der GL–Vorschrift wird durch die Aufteilung des Textes in konsequent getrennte Portionen bestimmt. Jeder Abschnitt verfügt z.B. über Unterabschnitte wie "Begriffsbestimmungen" oder "Allgemeines", damit eine unabhängige Vollständigkeit für jeden Abschnitt erreicht wird. Die konsequente Wiedergabe von solchen ergänzenden Informationen erstreckt sich von Kapiteln über Abschnitte bis hin zu

[1] Als "Abschnitt 11," zu verstehen. Die Schreibweise im GL–Regelwerk wird z.T. übernommen.

da zur Anwendung der Dimensionierungsregeln von einer bis auf die Abmessungen definierten Struktur ausgegangen werden muß, d.h., die Topologie ist festgelegt.

Die Auswahl geeigneter Regeln geschieht durch die Entscheidung der Anwendbarkeit aufgrund der in Vorschriften enthaltenen Randbedingungen. In dem folgenden Beispiel sind unter den zu beurteilenden Kriterien die wichtigsten Alternativen aufgelistet, wobei Verweise auf die entsprechenden Teile des Vorschriftenwerkes angegeben werden. Diese Verweise haben z.B. die Form [11, B.2.1][2], d.h. Abschnitt 11, Unterabschnitt B, Paragraph 2.1. Der jeweils für die Regelauswahl und -auswertung ausschlaggebende Wert ist unterstrichen. Als Beispiel wurde eine Dimensionierung der Beplattung eines wasserdichten Querschottes ausgewählt.

Dimensionierung der Beplattung eines wasserdichten Querschottes

Auswahl der Regelkategorie

• Schiffstyp:	Standard	→ [1–22]	← [I/1/1, X]
	Massengut/Erzschiffe	[23]	(Inhaltsübersicht)[3]
	Tanker	[24]	

• Konstruktionsgruppe:	Außenhaut	[6]	↩
	Decks	[7]	
	Bodenkonstruktion	[8]	
	Wasserdichte Schotte	→ [11]	
	Tanks	[12]	

• Laderaumnutzung:	Ballastwasser	→ [12, C.]	← [11, B.1.1]
	Erzladung	[23]	← [11, B.1.2]
	Standard	→ [11, B.2]	

Das Schott muß somit nach den Regeln der Abschnitte [11] und [12] ausgelegt werden, die jeweils höheren[4] *Werte sind auszuwählen.*

1. *Auslegung nach den Regeln des Abschnitts [11]:*

• Konstruktionstyp:	Standardschott	→ [11, B.2]
	Knickschotte	[11, B.4]

Angabe des zu dimensionierenden Bauteils bzw. des Berechnungsziels:

[2] Diese Abschnitte sind unter Kapitel 1 – *"Klassifikations– und Bauvorschriften für den Schiffskörper"* des Germanischen Lloyd [GL 92] zu finden.

[3] Vgl. Beispiel "Suche mit dem Inhaltsverzeichnis"

[4] Die größte Plattendicke ist auszuwählen, wobei letztendlich nach einer sicheren (gemäß den Vorschriften) Plattendicke gesucht wird.

- Bauteil/Berechnungsziel:

  Schottbeplattung → [11, B.2.1]

  $$\| \; t = c_p \cdot a \cdot \sqrt{p} + t_K \; \|$$

- Benötigte Parameter: (→ [11, B.1.3])

	BENENNUNG	AUS REGEL
c_p	Bemessungsfaktor	← Tabelle 11.2
a	Steifenabstand	aus der Geometrie
t_K	Korrosionszuschlag	← [3, K.1]
p	$9.81 * h$	

- Berechnungsziel:

  $$\| \; p = 9.81 * h \; \|$$

- Benötigte Parameter:

	BENENNUNG	AUS REGEL
h	Druckhöhe	aus der Geometrie

Auszug v. [11, B.1]:

1.3 Begriffsbestimmungen

h = Abstand vom Belastungszentrum des Bauteils bis 1m über Seite Schottendeck, beim Kollisionsschott bis 1m über Oberkante Kollisionsschott

Das Ergebnis muß evtl. noch korrigiert werden, falls eine der nachfolgend aufgeführten Sonderbedingungen erfüllt wird:

- Sonderbedingungen: (beispielhafte Auszüge v. [11, B.2])

  2.2 Bei kleinen Schiffen braucht die Schottbeplattung nicht dicker als die Außenhaut bei einem dem Steifenabstand entsprechenden Spantabstand zu sein.

  2.3 Das Stopfbuchsenschott ist am Stevenrohr mit einer verstärkten Platte zu versehen.

  2.4 In Bereichen, in denen stoßartige Belastungen bei Anlegemanövern zu erwarten sind, gilt für die Beulsicherheit der an die Außenhaut grenzenden Schottfelder sinngemäß Abschnitt 9, B.4.4 und 4.5.

2. *Auslegung der Schottbeplattung nach Regeln des Abschnitts [12] (Ballastladeraum)*

- Bauteil/Berechnungsziel:

  Tankbeplattung → [12, C.2] → [12, B.2.1]

$$\left\| \begin{array}{l} t_1 = 1.1 \cdot a \cdot \sqrt{p \cdot k} + t_K \\ t_2 = 0.9 \cdot a \cdot \sqrt{p_2 \cdot k} + t_K \end{array} \right\|$$

- Benötigte Parameter:

	BENENNUNG	AUS REGEL
a	Steifenabstand	$\hookleftarrow$[5]
k	Werkstoffkennziffer	$\leftarrow$ [2, B.2]
t_K	Korrosionszuschlag	$\hookleftarrow$
p	Druck p_1 oder p_d	$\leftarrow$ [4, D]
p_2	Druck	$\leftarrow$ [4, D.1]

- Berechnungsziel:

$$\underline{p_1, p_d} \quad \rightarrow [4, D.1], [4, D.2]$$

$$\left\| \begin{array}{c} p_1 = 9.81 \cdot h_1 \cdot \rho \cdot (1 + a_v) + 100 \cdot p_v \\ p_1' = 9.81 \cdot \rho \cdot h_1 \cdot \cos\varphi + (b/2 + y) \cdot \sin\varphi + 100 \cdot p_v \\ p_d = (4 - L/150) l_t \cdot \rho + 100 \cdot p_v \\ p_d' = (5.5 - L/20) b_t \cdot \rho + 100 \cdot p_v \end{array} \right\|$$

- Benötigte Parameter:

	BENENNUNG	AUS REGEL
h_1	Abstand von Oberkante Tank bis zum Belastungszentrum	aus der Geometrie
ρ	Dichte des Tankinhaltes	aus den Belastungsmerkmalen
a_v	Beschleunigungsfaktor	$\leftarrow$ [4, C.1.1]
b	obere Tankbreite	aus der Geometrie
y	Abstand des Belastungszentrums von der Mittellängsebene des Tanks	aus der Geometrie
p_v	Einstelldruck des Überdruckventils	aus den Belastungsmerkmalen
$p_{v_{\min}}$	0.2 bar für Ladetanks	$\leftarrow$ [4, D.1.1]
f	freie Tanklänge, -breite in Quer- und Längswänden	aus der Geometrie
L	Schiffslänge	aus den Systemparametern
φ	Entwurfskrängungswinkel	i.A. 20°
l_t, b_t	Abstände ...	aus der Geometrie

- Berechnungsziel:

$$\underline{p_2} \quad \rightarrow [4, D.1.2]$$

$$\| \; p_2 = 9.81 \cdot h_2 \; \|$$

- Benötigte Parameter:

[5] Das Symbol $\hookleftarrow$ bedeutet, daß diese Werte schon berechnet wurden und weiter verwendet werden können.

	BENENNUNG	AUS REGEL
h_2	Abstand zwischen Belastungszentrum und Oberkante Überlaufrohr	aus der Geometrie

3.2.2 Analyse der Attribute

An Hand des vorliegenden Beispiels wird erkennbar, in welchem Umfang und in welcher Art Informationen benötigt werden, um eine eindeutige Zuordnung zwischen Bauteil und geeigneten Bemessungsregeln zu ermöglichen und diese Regeln auswerten zu können.

Die Analyse zahlreicher Regeln führt weiter zu den unten dargestellten Informationskategorien, innerhalb derer die benötigten Informationen durch vielfältige Attributwerte beschrieben werden. Die Informationskategorien stellen somit "Attributklassen" dar.

Konstruktionsmerkmale

Unter dieser Attributklasse sind die Attribute zur Beschreibung der konstruktiven Funktion eines Bauteils zusammengefaßt.

Die große Anzahl der zugrundeliegenden Modelle spiegelt sich dabei in den vielfältigen Typbezeichnungen wieder.

- ***Lokale Bauteilfunktion***

  beschreibt die *lokale* konstruktive Funktion der Bauteile. Hinter den vielfältigen Typbezeichnungen verbirgt sich häufig das statische Berechnungsmodell, also die Grundlage der Regel, nach der diese Bauteile ausgelegt werden sollen. Dieses Attribut kann zum Beispiel folgende Werte annehmen:

  LOKALE BAUTEILFUNKTION

  Steife, Platte, Kragträger, Beulsteife, Stützplatte, Mittellängsträger, Spant, Knieblech, Seitenträger, Balken, Stringer, Kollisionsschott, Rahmen, Anker, Schlagschott, Träger, Stütze, Stopfbuchsenschott, Unterzug, Bodenwrange, ...

- ***Globale Bauteilfunktion***

  beschreibt, ob von dem Bauteil Belastungen aus der übergeordneten Strukturbeanspruchung übernommen werden müssen (z.B. des Schiffskörpers). Die möglichen Werte sind:

  GLOBALE BAUTEILFUNKTION

  längsfestigkeitsrelevant, querfestigkeitsrelevant, torsionsfestigkeitsrelevant, ...

- *Bauweise*

  Die Werte dieses Attributs kennzeichnen dimensionierungsrelevante Bauausführungen z.B. von Bodenwrangen, Trägern im Boden, Schotten.[6]

  BAUWEISE
  offen, voll, wasserdicht, ...

Topologische Merkmale

Die Werte der hier aufgeführten Attribute beschreiben den Ort des Bauteils in bezug auf seine Anordnung innerhalb der Struktur und den Ort der dieses Bauteil umgebenden Räume.

- ***Strukturbereich***

  Die Werte dieses Attributs haben verschiedene Gültigkeitsbereiche wie:

  an der Außenhaut:

  STRUKTURBEREICH
  Boden, Aufkimmung, Kiel, Seite, Scheergang, Steven, Kimm, Innenboden, ...

  an den Decks:

  STRUKTURBEREICH
  neben den Luken, zwischen den Luken, an den Enden, n-tes Deck, ...

- ***Räume***

  beschreiben die Lage eines Bauteils raumorientiert im Schiff.

  WEITERE BEREICHE
  Vorschiff, Hinterschiff, Vorpiek, Mittschiff, Maschinenraum, Laderaum, Bilge, Hinterpiek, Aufbau, Back, Deckshaus, Poop, Zwischendecksraum, Doppelboden, ...

- ***Angrenzende Bauteile***

  Häufig haben die geometrischen Eigenschaften angrenzender Bauteile Einfluß auf die Dimensionierung. Daher muß das angeschlossene Bauteil bekannt sein.

  ANGRENZENDE BAUTEILE
  Steife, ...

[6] Es ist relativ kompliziert, diese Informationen aus den CAD–Daten herzuleiten, da über sie keine **präzisen** Aussagen gemacht werden können.

- ***Orientierung***

  von Profilen in Abhängigkeit von der Schiffslängsachse oder von lokaler Bauteilausprägung.

  ORIENTIERUNG
  längs, quer, horizontal, vertikal, diagonal, radial, ...

- ***Auflagerbedingungen***

  werden für Profile, Spanten, Balken, Steifen usw. vorgegeben. Über dieses Attribut werden in den Regeln die Auflagerbedingungen erfaßt.

  AUFLAGERBEDINGUNGEN
  beidseitig eingespannt, einseitig eingespannt, beidseitig frei gelagert, einseitig frei gelagert, ...

Belastungsmerkmale

Zu dieser Attributklasse gehören Attribute wie *Belastungstyp, Belastungsverursacher* und *Belastungskennwerte* sowie die Attribute *Raumnutzung* und *Lastbereich.* Die letzten beiden Attribute lassen sich auch unter der Attributklasse *Belastungsumgebungen* gruppieren.

- ***Belastungstyp***

  Hierunter sind Formen elementarer Belastungsbeschreibung zu verstehen wie:

  BELASTUNGSTYP
  Punktlast, Streckenlast, Flächenlast, Moment, ...

- ***Belastungsverursacher***

  Attribut zur Erfassung von Radlasten, Einzellasten (Container) oder anderer spezifischer Lasteinbringungen.

  BELASTUNGSVERURSACHER
  Ladung, PKW, LKW, Container, Trailer, Gabelstapler, Hubschrauber, Einzellast, ...

- ***Belastungskennwerte***

  BELASTUNGSKENNWERTE
  Einstelldruck von Tanküberdruckventilen, spez. Luftdruck zur Ermittlung von Radlasten, Belastungskoeffizienten (Lastanteile), Spannungsgrenzwerte und Sicherheitszahlen[7], *...*

Die Berechnung der Bauteilbelastung erfolgt bei den Klassifikationsvorschriften im allgemeinen nicht durch die Regel für die Bauteilbemessung sondern in einem eigenen Kapitel bzw. in vorangestellten Absätzen. Die Verknüpfung zwischen zutreffender Belastung und dem Bauteil erfolgt über die Zuordnung des Bauteils bzw. der Baugruppe zu einer Belastungsumgebung. So bedeutet z.B. die Belastungsumgebung *Tank*, daß die Vorschriften für die Dimensionierung von Bauteilen in Tanks angewendet werden müssen.

Es lassen sich die folgenden *Belastungsumgebungen* darstellen:

- ***Raumnutzung***[8]

  kennzeichnet die Art der Belastung eines Raums bzw. seiner Wände durch *Ladung* jedweder Art.

  RAUMNUTZUNG
  Laderaum Stückgut, Laderaum Schüttgut, Ballasttank, Leerzelle (Kofferdamm), Tank (allgemein), Verbrauchstank, Store, ...

- ***Lastbereich***

  Die Werte dieses Attributs sind i.d.R. Belastungsregionen, für die von den Bauvorschriften bestimmte Lastvorgaben gemacht werden, die ggf. durch höhere Entwurfsforderungen ersetzt werden.

  LASTBEREICH
  Außenhaut, Einrichtungsdeck, Ladungsdeck, Deck, Maschinendeck, Schott, Wetterdeck, Innenboden, ...

Materialdaten

Zu dieser Klasse können die Attribute wie *Werkstoffart, Werkstoffgüte, zulässige Spannungen* usw. gehören.

Geometriedaten

Unter dieser Attributklasse können Attribute zusammengefaßt werden, die alle geometrischen Daten liefern, die aus der Sicht der Dimensionierungsregeln erforderlich sind. Sie können hauptsächlich aus dem entsprechenden CAD–System übertragen werden [Toparlak 91a].[9]

[7] Soweit sie im Zusammenhang mit dem Belastungsverursacher stehen.

[8] auch benachbarter Räume

[9] Oft sind komplexe geometrische Operationen erforderlich, um nichtvorhandene geometrische Information, die für die Auswertung der Dimensionerungsvorschriften benötigt werden, aus der vorhandenen geometrisch-topologischen Information herzuleiten. Auf diese nicht triviale Problematik wird im Kapitel 8.1 [DEXp] ausführlicher eingegangen.

GEOMETRIEDATEN

Bauteilort in bezug auf das Schiffskoordinatensystem: *Transformation*

Bauteilgrenzen:	*Platteneinteilung (Unterkante), ...*
Bauteilabmessungen:	*Länge, Breite, Höhe, ...*
Widerstandsmomente:[10]	W_x, W_y, W_z
Querschnittswerte:[11]	I_x, I_y, I_xy, I_z, A
Schiffskontur:	*Spantausfall, Stevenverlauf, ...*
Belastungsgeometrie:	*Lastverlauf, ...*
Umgebungsdaten:	*Höhe Überlaufrohr, Höhe Lukensüll, Tankbreite, ...*

Systemparameter

Darunter werden Merkmale verstanden, die nicht bauteilorientiert vergeben werden müssen, sondern für das gesamte Dimensionierungsprojekt Gültigkeit besitzen. Die Attributklasse Systemparameter hat einige Subklassen wie *Entwurfsdaten* und Attribute wie *Schiffstyp* und *Klassezeichen*. Die meisten Computerprogramme zur Unterstützung der Dimensionierung nach den Vorschriften gehen nur noch mit Systemparametern um (s. RULE*finder* o.ä.).

- ***Entwurfsdaten***

  Unter diesem Attribut können Angaben für Hauptabmessungen, Völligkeiten, Gewichte usw. gemacht werden.

  ENTWURFSDATEN

  $L, B, c_B, \ldots$

- ***Schiffstyp***

  Diesem Attribut können Werte zugewiesen werden wie:

  SCHIFFSTYP

  Trockenfrachter, Massengutschiff, Erzschiff, Tankschiff für Ölladung, Schiffe für den Transport von Trockenladung oder Ölladung, Tankschiffe zur Beförderung gefährlicher Chemikalien, offene Schiffe, ...

- ***Klassezeichen***

  Die Werte dieses Attributs bezeichnen Schiffstyp und Klassezeichen entsprechend der Klassifizierung, festgelegt in der Bauvorschrift.

[10] Widerstandsmomente, i.a. Dimensionierungsergebnisse, müssen jedoch als Vergleichswerte darstellbar sein.

[11] für iterative Dimensionierungsprobleme

KLASSEZEICHEN	ZUSÄTZE
	Eisklassen
100 A 4	*Watt–, Küsten– oder mittlere Fahrt verstärkt für Schwergutladung*

Gültigkeitsbereiche

Attributwerte besitzen unterschiedliche Gültigkeitsbereiche bezogen auf das beschriebene Objekt. Es kann folgende Unterteilung vorgenommen werden:

- *Bauteilwerte* betreffen das beschriebene oder ein gleichartiges Bauteil.
- *Ortsabhängige* Werte haben Gültigkeit für gewisse Regionen. Beispiel: *Spantabstand vor dem Kollisionschott, hinter dem Hinterpiekschott, im Laderaum–Bereich, usw.*
- *Umgebungswerte* betreffen z.B. bestimmte Belastungsumgebungen wie Räume, die damit für viele Bauteile relevant sind.
- *Systemwerte*: siehe oben (Abschnitt 3.2.2, Punkt 3.2.2)

Allgemeingültige und regelabhängige Daten

Unter allgemeingültigen Daten sind solche zu verstehen, deren Inhalte eindeutig dem allgemeinen oder branchenspezifischen Verständnis entsprechen, wie z.B.: *Geometriedaten* oder Bezeichnungen wie *Platte, Steife, Tank, Außenhaut*. Diese Daten sind im obengenannten Sinne problemlos.

Die regelabhängigen Daten werden in zwei Typen unterschieden:

1) Daten, die mit Hilfe von Regeln aus allgemeingültigen Daten erzeugt werden können; z.B. *Berechnungslänge, Normalspantabstand*. Sie sind ebenfalls problemlos.

2) Daten, deren Bedeutung nur im Zusammenhang mit der angewendeten Klassevorschrift eindeutig ist oder deren Bedeutung nicht als Norm angesehen werden kann, z.B. die Werte des Attributs *Bauweise*: *offen* und *voll*. Solche Attribute müssen entweder eine eindeutige Ersatzbeschreibung erhalten oder durch eine Art Lexikon erfaßt werden. Ihre Bedeutung muß eindeutig festgelegt werden.

Übersicht

Es ist von großer Wichtigkeit, zwischen allgemeingültigen und regelabhängigen Daten zu unterscheiden, da möglichst keine regelabhängigen Daten zur Modelldefinition herangezogen werden sollen. Dies würde die Verwendbarkeit eines Modells auf eine Klasse, im Extremfall sogar auf eine Ausgabe der Bauvorschriften einschränken.

Diesem Ansatz steht die gelegentliche Notwendigkeit der Zuordnung von Strukturelement (Bauteil) und Regel bei der Regelauswahl entgegen, da dafür häufig regelwerksspezifische Angaben erforderlich sind.

Die hier aufgeführten Merkmale erheben nicht den Anspruch auf Vollständigkeit. Das ist auch nicht das Ziel der Untersuchung, sondern vielmehr die Analyse der Daten, die prinzipiell zur vollständigen Definition eines Bauteils aus der Sicht des Regelwerkes erforderlich sind, um diese in einem geeigneten Beschreibungsmodell darstellen zu können.

Bei der Aufzählung der Attributwerte ergeben sich gelegentlich Wiederholungen. So wird z.B. der Attributwert *Außenhaut–Seite* sowohl bei den Belastungs– als auch bei den Topologiemerkmalen vergeben. Für die Auslegung der Seitenbeplattung gibt es jedoch sowohl Regeln, für deren Anwendung die bloße Zugehörigkeit zu diesem Strukturbereich ausschlaggebend ist, als auch Regeln, die die Auslegung auf Grund der Seitenbelastung vorsehen. Da in diesem Stadium noch kein Beschreibungsmodell definiert ist, ist die Mehrfachbenennung dieser Attribute wegen unterschiedlicher Bedeutungen sinnvoll.

3.2.3 Analyse der Regeln

Die Regeln, Elemente der Regelbasis, lassen sich nach verschiedenen Kriterien differenzieren. Dabei sollen die folgenden Merkmale berücksichtigt werden:

Unterscheidung der Regeln nach ihrem Inhalt

- *Strukturauflagen* sind Regeln, die Richtlinien für den *topologischen, geometrischen* Aufbau der Struktur enthalten. Beispiel: *Spantabstand vor dem Kollisionsschott, hinter dem Hinterpiekschott.*

  Diese Regeln sollen hier nur für Überprüfungszwecke dienen, eine globale Modifikation der Topologie der Stahlstruktur auf der Basis dieser Regeln soll an dieser Stelle nicht weiter untersucht werden.

- *Regeln zur Strukturauswertung*: Als Ergebnis wird die *Anwendbarkeit* bestimmter anderer Regeln geliefert, oder es werden *Eingangsparameter* für andere Regeln berechnet. Beispiel: *Bestimmung des Faktors c_p für Kollisionsschott/anderes Schott ([11, B.1])*.

- *Direkte Dimensionierungsregeln* werden durch eine vorausgehende Strukturauswertung *ausgewählt* und enthalten Berechnungsanweisungen für konkrete Bauteile. Ihnen werden wiederum die Eingangsparameter ermittelt.

- *Regeln zur Bewertung von Berechnungsergebnissen nach diversen Kriterien.* Quantitative Beurteilung der Ergebnisse wie der Vergleich mit festen Grenzwerten oder die Auswahl des zutreffenden Ergebnisses nach einem *Extremwert*. Beispiel: *Die größere Plattendicke ist auszuwählen.*

- *Verweisende Regeln*: Dies sind meist untergeordnete Regeln, die in anderen enthalten sind oder mit anderen in engem Zusammenhang stehen oder auf andere Regeln verweisen. Sie enthalten im allgemeinen keine Berechnungs– oder Auswertungsvorschriften.

- *Verschachtelte Regeln*: Ihre Auswertung ist von anderen Regeln abhängig und kann erst dann durchgeführt werden, wenn die Auswertung dieser Regeln erfolgt ist. Verschachtelte Regeln sind oft Regeln, denen eine Belastung zugrunde liegt, deren Wert vom aktuellen Ort abhängig ist. Beispiel: *Berechnung der Plattendicke eines Schottes nach Regel [12, B.2.1]*. Hier muß der Druck eingesetzt werden, der nach Regel [4, D] berechnet wird. Die zyklische bzw. rekursive Verschachtelung der Regeln sollte weitestgehend vermieden werden. Im weiteren Sinne sind sie den Regeln zur Strukturauswertung ähnlich.

- *Belastungsregeln*: Diese Regeln ordnen den Bauteilen oder Strukturbereichen Belastungen zu. Eine erforderliche Lastgenerierung für ein Bauteil kann durch solche Regeln erfolgen.

Unterscheidung der Regeln nach ihrer Auswirkung auf die Struktur

- Regeln, die direkt die logische Struktur beeinflussen. Diese sind oft Entwurfsregeln.

- Regeln, die direkt die physikalische Struktur beeinflussen. Diese sind die Dimensionierungsregeln.

- Regeln, die indirekt die logische Struktur betreffen. Darunter fallen Regeln, deren Ergebnis für die physikalische Struktur eine Veränderung der logischen Struktur erforderlich macht.

- Regeln, die Systemwerte liefern. Damit sind Regeln gemeint, die innerhalb eines geschlossenen Systems Bedeutung haben, z.B. innerhalb der Dimensionierungsregeln des Germanischen Lloyd. Beispiel: *Berechnungslänge.*

Schlußfolgerungen

Die in den vorangegangenen Abschnitten durchgeführt Analyse bzw. die grobe Formalisierung zeigt die Komplexität der Information, die zur Auswertung der ebenso komplexvernetzten Regelstrukturen heranzuziehen ist. Bemerkenswert ist hierbei, daß die Definition der entsprechenden Produktmodelle in dieser Hinsicht erforderlich ist, zumal sie im Rahmen der verschiedenen aktuellen Untersuchungen in ähnlicher Weise benötigt wird, unabhängig von der Problematik der Dimensionierung nach Vorschriften.

Die Notwendigkeit der Neudefinition eines Produktmodells bzw. der Erweiterung der bereits existierenden schiffbaulichen Produktmodelle wird offensichtlich. Da die Definition des schiffbaulichen Produktmodells z.T. selbst in der Entwicklung ist, wird in der vorliegenden Arbeit Abstand davon genommen, eine Neudefinition vorzunehmen. Vielmehr soll versucht werden, vorhandene Definitionen zur Auswertung der Dimensionierungsregeln nützbar zu machen, d.h. Schnittstellen zu definieren. Dies soll im weiteren Sinne die Schnittstelle zu den CAD–Daten eines schiffbaulichen CAD–Systems wie TRIBON[12] ausmachen bzw. die Definition spezifischer Schnittstellen möglichst vermeiden.

Die anschließende Analyse der Regeln, ausgehend von dem vorliegenden Dimensionierungsbeispiel, zeigt die vielseitige bzw. vielschichtige Interaktion zwischen den Regeln und den zu ihrer Auswertung herangezogenen Informationen (Produktmodell), wobei eine deutliche Trennung **ausgeschlossen** ist. Dies erfordert die Möglichkeit zur Definition der benötigten Information **als** Regel.

Die daraus resultierende Erkenntnis ist ein Prozeßmodell, das sich in verschiedenen Vorgängen (*events*) formulieren läßt. Im vorliegenden Fall kann dann der Dimensionierungsprozeß in folgenden Vorgängen wiederum mit vielseitiger Interaktion ausgeprägt werden:

- **Regelselektion**,
- **komplexe Datenabfrage**,
- **Regelauswertung** und
- **Ergebnisbewertung**.

Die Bewertung der Ergebnisse kann auch als *Regelauswertung* betrachtet werden, in der wiederum bewertende Regeln *ausgewählt* bzw. *ausgewertet* werden. Die komplexe Datenabfrage würde sich auch mit dem Paar *Regelselektion* und *–Auswertung* realisieren lassen, wo zusätzlich mit Anfragen an das Produktmodell und dem Dialog mit dem Benutzer gerechnet werden soll. Letzteres kann auch in eine Regeldefinition eingebettet werden.

[12] STEERBEAR mit der neuen Bezeichnung

Der Dimensionierungsprozeß hat somit einen *rekursiven* Charakter, der mit der **Auswahl** und **Auswertung** der relevanten Regeln, die die Dimensionerungsvorschriften, die Anfragen an das Produktmodell und den Dialog mit dem Anwender beschreiben, ausgeprägt ist. Das Erkennen dieser Eigenschaft ist von **großer** Bedeutung, da zur Problemlösung entsprechende Mittel, d.h. Programmiersprachen und Datenmodelle, herangezogen werden müssen. Das ist das Thema des nächsten Kapitels.

Kapitel 4

Möglichkeiten zur Verarbeitung technischer Regeln

Nachdem im Kapitel 3 die erforderlichen Informationen zur Dimensionierung von Bauteilen nach Regeln dargestellt und die Analyse der charakteristischen Regelstrukturen durchgeführt wurde, werden hier die entsprechenden Organisations– und Implementationsmöglichkeiten eines Regelwerkverwaltungssystems aufgezeigt. Mit dem Begriff Regelwerkverwaltungssystem wird eine Systemkomponente definiert, die in der Lage ist, die Dimensionierungsregeln zu verwalten[1] und die von Anwendungsprogrammen für die Dimensionierung wie z.B. dem *Dimensionierungsprozessor*[2] (s. Kapitel 5) angesprochen werden kann. Daraus folgt, daß das Regelwerkverwaltungssystem eine Schnittstelle bereitzustellen hat, über die Funktionen zur Spezifikation sowie Interpretation der einzelnen Regeln für Anwendungsprogramme verfügbar sind.

Es wird angenommen, daß die erforderlichen Informationen[3] wie *Bauteilgeometrie, Systemparameter* sowie *Material* im sog. Produktmodell zur Verfügung stehen, um die Analyse der Implementationsmöglichkeiten eines beabsichtigten Regelwerkverwaltungssystems durchzuführen. Die dabei gewonnenen Erkenntnisse tragen zur Ermittlung der notwendigen Erweiterungen sowohl des von dem Regelwerkverwaltungssystem zu verwendenden Datenmodells als auch der damit arbeitenden Mechanismen bei.

[1] Speichern, Suchen, Auswerten usw.

[2] Der Dimensionierungsprozessor kann als ein Programm konzipiert werden, das selbst keine Regeln, sondern ausschließlich Funktionen des Regelwerkverwaltungssystems aufruft, z.B. für das Suchen und Auswerten einer Regel für ein zu dimensionierendes Bauteil.

[3] Hauptsächlich sind das Daten, die dem CAD–System abgefragt werden können.

4.1 Analyse des Regelwerkverwaltungssystems

Unter dem Begriff Regelwerkverwaltungssystem sollen zwei Komponenten, nämlich die Gesamtheit aller zur Bemessung von Bauteilen relevanten Regeln (die Regelbasis) und der Kontrollmechanismus, der diese Regeln verarbeitet, verstanden werden. Diese Trennung zwischen Regeln und ihrem bearbeitenden Mechanismus ist zwingend notwendig, weil eine effiziente Programmierung die Trennung von Daten und Algorithmen, die die Informationen und Prozesse für einen Problembereich repräsentieren, voraussetzt [Wirth 83]. Dieser Ansatz kann bzw. muß auf den Problembereich Dimensionierung nach den Vorschriften übertragen werden. Es werden dann die Regeln als Daten und die Mechanismen[4] als Prozesse betrachtet.

Die jeweils getrennte Analyse dieser Komponenten wird zeigen, wie die Regeln aufgrund der Erfordernisse der Mechanismen, die sie verarbeiten, im Rechner dargestellt werden müssen und wie die Mechanismen demgegenüber aufgrund der begrenzten Möglichkeiten für die rechnerinterne Repräsentation und die Interpretation der Regeln implementiert werden sollen.

Der Mechanismus ist auf jeden Fall von der Darstellungsform der Regeln bzw. Regelbasis abhängig: Wenn eine Modifikation der Darstellungsform durchgeführt wird, ist der Mechanismus zu verifizieren. Umgekehrt ist es auch nicht auszuschließen, daß die Form der Regelbasis an die Bedürfnisse des Kontrollmechanismus angepaßt wird.

4.1.1 Datenstrukturen für Dimensionierungsregeln

In diesem Abschnitt erfolgt eine Analyse der für die rechnerinterne Darstellung erforderlichen Datenstrukturen für Dimensionierungsregeln, wobei die Verwendbarkeit der in der jeweils eingesetzten Programmiersprache verfügbaren Datenstrukturen und die Verfügbarkeit der Möglichkeiten zur Definition von neuen Datenstrukturen ermittelt wird. Eine weitere Analyse dieser Datenstrukturen wird im Abschnitt 4.1.2 im Hinblick auf die Mechanismen, die solche Datenstrukturen verarbeiten sollen, durchgeführt.

Zuerst erweist es sich als notwendig, gemäß der Unterscheidung (Abschnitt 3.2) zwischen den Regeln nach Inhalt und Auswirkung auf die Schiffsstruktur einige Bemerkungen über die erforderlichen Datenstrukturen zu machen:

- Aufgrund der Herkunft der Regeln müssen die Datenstrukturen *generische* Eigenschaften haben, d.h. sie müssen auf einem höheren Programmierniveau vom Benutzer neu definierbar bzw. modifizierbar sein.

[4] Gemeint sind die Problemlösungsmethoden, die Regeln zur Lösung des Dimensionierungsproblems verwenden.

- Die Datenstrukturen enthalten häufig abweichend zu interpretierende Teile. Daher ist es notwendig, Datenstrukturen für verschiedene *Interpretationsschichten* zu konzipieren. Beispiele: Interpretationsschichten für Abfragen, Auswertung, Definition.

- Die Datenstrukturen müssen einen Zugang zum Produktmodell haben, um an etwaige topologische, geometrische oder belastungsorientierte Informationen zu gelangen. Daher muß eine *dynamisch definierbare* Verbindung in den Datenstrukturen zu diesen Informationen vorgesehen werden.

- Beim Zugang zum Produktmodell muß eine eindeutige Transparenz bezüglich der verwendeten Datenrepräsentation erreicht werden. So müssen Datenstrukturen, die die erforderlichen Informationen für die Dimensionierungsregeln enthalten sollen, den Datenstrukturen des Produktmodells ähneln, damit bei entsprechendem Datenaustausch zwischen dem CAD– und dem Dimensionierungsprozessor möglichst wenige Modelltransformationen stattfinden.

- *Dynamische* Datenstrukturen[5] sind wegen der Existenz von verweisenden und verschachtelten Regeln erforderlich, deren Verhalten undeterministisch ist.

- Es gibt sowohl *Relationen* als auch *Hierarchien* zwischen Regeln. Möglichkeiten müssen geschaffen werden, um derartige Beziehungen zu definieren bzw. zugänglich zu machen. Relationale Datenmodelle bzw. semantische Netze sind Beispiele dafür [Ullmann 88, Ullmann 89].

- Es gibt Regeln, die andere Regeln als Objekte verwenden. Beispielsweise bestimmen sie die Verwendbarkeit anderer Regeln. Die Datenstrukturen für solche Regeln müssen deshalb die erforderlichen Operationen für die Handhabung von Regeln als Objekte zulassen. (Diese Anforderung entspricht z.B. der Definition von sog. **abstrakten** Datentypen, die nicht nur die Daten sondern auch die dazugehörigen Funktionen festlegen.)

- Rekursion ist für die Datenstrukturen zur Definition der verweisenden bzw. der verschachtelten Regeln erforderlich, da eine verweisende Regel in den von ihr referenzierten Regeln direkt bzw. indirekt aufgerufen werden kann. Um unklare Zyklen zu vermeiden, können dabei einige Restriktionen definiert werden.

  Die **Rekursion** soll nicht nur in den Datenstrukturen, sondern auch in den auswertenden Kontrollfunktionen erlaubt bzw. möglich sein.

Es sind geeignete Datenstrukturen für derartige komplexe Aufgabenstellungen bekannt, die in den Bereichen Wissensverarbeitung, Expertensysteme bzw. Künstliche Intelligenz

[5] Dynamische Datenstrukturen sind solche, die *zur Laufzeit* sowohl vom Anwendungsprogramm als auch vom Benutzer definiert werden. Jedoch sollte dieser Begriff **nicht** mit dem der dynamischen Speicherreservierung bzw. –belegung (*Dynamische Speicherverwaltung*) verwechselt werden, die in den heute üblichen Programmiersprachen wie **C** oder **C++** zur Erzeugung von neuen Objekten genutzt wird. Dies kann man eher dynamische Instanzierung nennen, da die Strukturierung dieser Objekte früher festgelegt ist.

(*Artificial Intelligence*) eingesetzt werden. Einige dieser Datenstrukturen, Wissensrepräsentationen genannt, sind:

Frames: *Frames*[6] sind zuerst von Minsky [Minsky 75] konzipiert worden. Ein *frame* ist ein Wissensrepräsentationsschema, das ein Objekt mit einer Gruppe von Eigenschaften assoziiert (z.B. Fakten, Regeln, *default*–Werte und *aktive* Werte). Jede Eigenschaft wird in einem *slot* (Abteil) gespeichert. Ein *frame* ist die Menge von *slots*, die mit einem spezifischen Objekt in Relation stehen. Ein *frame* entspricht einer Eigenschaftsliste (*property list*) bzw. einem Schema oder einem *record* (Datensatz), wie diese Begriffe in der konventionellen Programmierung genannt werden.

Semantic networks: Die Semantischen Netze sind eine Art der Wissensrepräsentation, bei der Objekte und Werte formal als Knoten (*nodes*) dargestellt und die Knoten durch Verbindungen (*links*) oder Kanten (*arcs*) verbunden werden, um die Relationen zwischen den verschiedenen Knoten anzuzeigen.

Production, if-then rules: Die Wenn/Dann–Regeln sind konditionale Aussagen in zwei Teilen. Der erste Teil besteht aus einem oder mehreren Wenn–Ausdrücken und legt daher die Bedingungen (*constraints*) fest, die eingehalten werden müssen, wenn der zweite Teil, bestehend aus einem oder mehreren Dann–Ausdrücken, ausgeführt werden soll. Die Teilaussagen der Regeln sind normalerweise Attribut/Wert–Paare oder Objekt/Attribut/Wert–Tripel.

Diese Datenstrukturen können teilweise die obigen Anforderungen erfüllen. Für das Regelwerkverwaltungssystem ist ihre Kombination notwendig. Weitere detaillierte Information über diese Datenstrukturen kann aus [Harmon/King 85, Hayes-Roth/Waterman/Lenat 83] entnommen werden. Wie solche Wissensrepräsentationen realisiert werden, hängt von den Datenmodellierungsmöglichkeiten der darunterliegenden Implementationssprache ab. Beispiele sind: Klassen in C++, logische Prädikate in Prolog oder polymorphische Typen in ML.

4.1.2 Analyse des Kontrollmechanismus

Der Mechanismus, der die Regeln verarbeiten soll, ist die zweite wichtige Komponente eines Regelwerkverwaltungssystems. Hier soll zuerst der Ablauf des Dimensionierungsbeispiels (siehe Abschnitt 3.2) betrachtet werden. Die verschiedenen Bearbeitungsphasen können mit unterschiedlichen Auswertungsstrukturen wie Bäumen, Netzen oder Graphen repräsentiert werden. Eine grobe Baumrepräsentation einer solchen Ablaufstruktur für das Beispiel wird in Abbildung 4.1 gezeigt. Es ist hier schon erkennbar, daß eine solche Ablaufstruktur rekursiv (daher zyklisch) werden kann. Für die Lösung

[6] *frame* – Rahmen/Objekt/Einheit

der Dimensionierungsaufgabe werden zuerst Alternativen ermittelt und durch deren Iteration mögliche Lösungen gefunden. Ein Beispiel dafür ist die Auswahl der größeren Plattendicke aus mehreren Werten, die wiederum nach bestimmten Kriterien ermittelt wurden. Zur Auswertung solcher Ablaufstrukturen existieren Verfahren wie z.B. das sog. *backtracking*–Verfahren [Winston 84].

Im konventionellen Sinne ist die Struktur dieses Baumes die Regelbasis; der Mechanismus, der sie bearbeitet, ist die Interpretation bzw. Kontrolle.[7] Da die Dimensionierungsaufgaben nicht von vornherein feststehen, kann keine solide Baumstruktur definiert und implementiert werden. Eine Regel kann durch ihre Auswertung die Struktur des Baumes dynamisch ändern (oder zumindest die weitere Interpretation des Baumes beeinflussen). In Abb. 4.1 sind die eckigen Knoten sogenannte *terminale* Knoten, an die sich keine Subbäume mehr anschließen. Das bedeutet, daß die Berücksichtigung dieser Alternativen aufgrund vorangegangener Entscheidungen (Kriterien) schon ausgeschlossen werden kann oder nach ihren Durchlauf brauchbare Werte vorliegen müssen, um die höherrangigen Werte zu ermitteln. Eine endgültige Auswertung dieser Werte kann aber in einigen Fällen zu einem späteren Zeitpunkt erfolgen (*lazy evaluation*). Beispiel: Wenn das Ergebnis nur noch ein mathematischer Ausdruck ist, in dem einige Variablen zu dem entsprechenden Zeitpunkt noch nicht bekannt sind, wird er *symbolisch* weitergereicht.

In Ausnahmefällen können auch terminale Knoten existieren, die keine verwendbaren Werte enthalten bzw. nicht zu einer Lösung gelangen. Dann müssen entweder Interaktionsprozesse mit dem Benutzer zum Erlangen der erforderlichen Information durchgeführt oder aber *parametrisierte*[8] Werte zurückgeliefert werden. Einige solcher Knoten (mit '[...]' gekennzeichnet) können aufgrund bereits vorhandener Kriterien ignoriert werden, d.h. sie liefern kein Resultat zurück.

4.2 Auswahl der Implementations–Werkzeuge und –Techniken

Zur Realisierung eines Regelwerkverwaltungssystems werden in diesem Abschnitt verschiedene Implementationswerkzeuge, Programmiersprachen bzw. –techniken mit einigen Beispielen, auf deren Verwendbarkeit hin untersucht, um eine Auswahl für eine geeignete Realisierung zu treffen. Die dabei untersuchten Techniken unterscheiden sich in bezug auf die verwendeten Programmierparadigmen bis hin zu betriebssystem-nahen Aspekten. Andere kategorische Unterscheidungen der im folgenden erläuterten Möglichkeiten für Implementation sind durchaus von Interesse, wobei eine vollständige Betrachtung über den Rahmen der vorliegenden Arbeit hinausgeht und daher nicht im Detail diskutiert werden kann.

[7] Somit wird der Ablauf bestimmt: Die Struktur des Regelbaumes bzw. Inhaltes der Regelbasis hat einen eindeutigen Einfluß auf die Auswertung. Dies bedeutet, daß ein klar definiertes Verfahren mit jeglicher Strukturierung der Regelbasis müßte fertig werden können. Damit reduziert sich der Aufwand für die notwendige Realisierung relativ stark.

[8] Damit die Chance besteht, die Werte durch die spätere Bestimmung der einzelnen Parameter zu ermitteln.

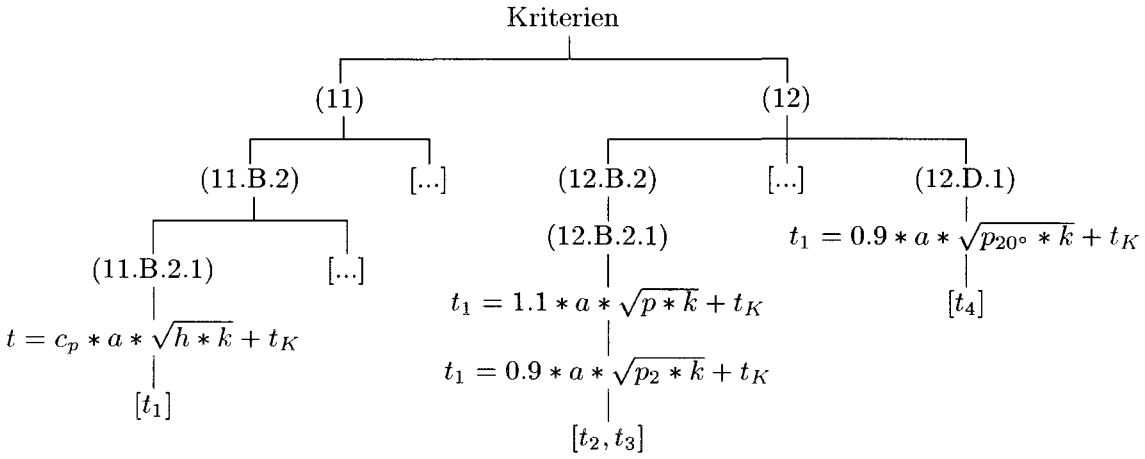


Abb. 4.1: *Baumrepräsentation der Ablaufstruktur eines Dimensionierungsbeispiels*

4.2.1 Programmiersprachen

Prozedurale Sprachen

In diesem Ansatz wird versucht, den Kontrollmechanismus nach dem prozeduralen Konzept

Algorithmen + Datenstrukturen = Programme

als Algorithmus zu realisieren. Das Regelwerkverwaltungssystem muß ein Datenmodell besitzen, das die erforderlichen Informationen beschreibt. Dieser Ansatz ist von vornherein nicht ausreichend, weil die Regeln einerseits selbst prozedurale Informationen enthalten und andererseits von den Algorithmen des Kontrollmechanismus als zu verarbeitende Informationen oder Datenstrukturen verwendet werden müssen. Dieser Ansatz kann nicht auf der Basis imperativer Programmiersprachen wie FORTRAN realisiert werden, weil diese für die Datenabstraktion[9] kaum Möglichkeiten besitzen.

Man kann das *Methodenbank*–Konzept[10] verfolgen, wie in dem Programmsystem GLRULES versucht wurde [Tietgen/+ 84]. Alle den Bauvorschriften entsprechenden Regeln [GL 86] sind hier in FORTRAN programmiert; die Benutzerschnittstelle ist als interaktive Eingabesteuerung konfiguriert. Die Vernetzung der Regeln ist festgelegt. Die Zahl der Regelstrukturen ist eindeutig bestimmt und kann ohne Neuprogrammierung nicht erweitert

[9] Datenabstraktion ist ein informatisches Modell, dem ein Datenmodell, das die komplexe Information für eine Aufgabenstellung als eine strukturierte Einheit repräsentiert, und Algorithmen, die die Elemente (Daten) dieses Datenmodells verarbeiten, zu Grunde liegen.

[10] Eine Methodenbank ist ein System, das schon vorprogrammierte Methoden via Schnittstellen nach außen hin verfügbar macht.

werden. Das System basiert auf vorprogrammierten Methoden, die für die Auswertung der Regeln benutzt werden, einer Datei, in der Systemparameter eingetragen werden müssen. Die Regeln, die durch Programmieren in Algorithmen der Sprache FORTRAN umgewandelt worden sind, müssen bei jeder Änderung des Regelbestands kompiliert und hinterher mit den sie benutzenden Modulen verbunden werden. Eine Modifikation einer Regel erfordert, daß fast sämtliche Module von Hand verifiziert und ggf. angepaßt werden. Jede Modifikation setzt voraus, daß die Kenntnisse über die Methoden zur Auswertung der Regeln, die innere Struktur des Gesamtprogramms sowie die dabei verwendete Programmiersprache verfügbar sind.

Die Möglichkeiten für das direkte Programmieren von Regeln können durch populäre Programmiersprachen wie C oder Pascal nicht weitgehend verbessert werden, obwohl die Datenabstraktion in diesen Sprachen verbessert worden ist und das Modulkonzept zur Verfügung steht. Ein wichtiger Vorteil dabei ist, daß Funktionen, die in der Regel eine abstrakte Form der Methoden für die Regeln sind, durch Zeigerreferenzen dynamisch aufgerufen werden können. Diese müssen aber zur Übersetzungszeit dem Compiler bekannt sein [Sethi 90]. Damit kann eine problemorientierte Repräsentation des Lösungsmodells nur noch mit großem Aufwand erreicht werden.

Objektorientierte Sprachen

Eine neuere Entwicklung im Softwarebereich stellt die sogenannte *objektorientierte* Programmierung dar. Die Datenabstraktion ist hier wesentlich erweitert, da die Einbettung von Methoden (Prozeduren) in den abstrakten Datentyp, der die Basismenge von Informationen für die Problemlösung repräsentieren soll, möglich ist. Es gibt Objekte, die die Informationen und die sie verarbeitenden Methoden enthalten und vor unerlaubten Zugriffen von außen schützen (*Data Encapsulation*), was für die Stabilität und Sicherheit der Programme von erheblicher Bedeutung ist. Durch die Definition von sogenannten *Klassen* kann erreicht werden, Informationen und Methoden von Klassen zu Objekten hin vererben (*inheritance*) zu lassen. Im vorliegenden Fall können z.B. die Klassen Schiffssystemparameter, Schiffsgeometrie, Schotte, deren Subklassen, Schottengeometrie Schottensystemparameter, die Klasse Schotte, deren Beispielobjekt[11] Querschott und deren Methode *Schottbeplattung* definiert werden. Nach den Vererbungsmechanismen wird auf die Nachricht (*message*)

```
querschott->schottbeplattung(parameter, ...)
```

mit der Dimensionierung der Dicke der Schottbeplattung reagiert, wobei die erforderliche Information inklusive Methoden (*member functions*) von Elternklassen vererbt worden sind.

[11] als *Instanz* der Klasse Schotte

Wenn eine Nachricht an Objekte einer Klasse und Objekte derer Elternklassen gesendet werden kann, wird dies als *Polymorphismus* [Wiener/Pinson 88] bezeichnet. Das Sprachkonzept Polymorphismus erlaubt trotz der strengen Typkontrolle hohe Flexibilität beim Umgang mit den benutzerdefinierten Typen. Die Objekte der Eltern– und Subklassen können in Abhängigkeit von Methoden, die in ihnen definiert sind, verschiedene Reaktionen auf eine Nachricht (*message*) zeigen (*virtuelle Funktionen*). Als Beispiel kann man sich vorstellen, daß die *message* `schottbeplattung` automatisch auf alle Objekte der Klasse Schotte angewendet wird. Die Dimensionierung der Schottbeplattung jedes einzelnen Schottes erfolgt jeweils nach eigenen Methoden, die in der jeweiligen Schottinstanz spezialisiert[12] sind. Man kann es sich auch so vorstellen, daß die Dimensionierung eines Schottes nach alternativen Regeln durchgeführt werden kann, wenn verschiedene Schott–Klassen jeweils für alternative Regeln definiert wurden. Objekte dieser Klassen werden sich je nach vorhandenen Kriterien unterschiedlich verhalten, wenn sie eine Nachricht erhalten.

Als Programmiersprache zu diesem Ansatz kann C++ [Stroustrup 94] benutzt werden. Diese Sprache ist ebenso portabel wie C und ist eine Übermenge von C, d.h. auch Standard–C–Programme lassen sich unter C++ übersetzen. Dadurch kann die vorhandene Software in ein solches System ohne Schwierigkeiten integriert werden. Die Kompatibilität von C++ zu C ist nicht nur auf Source–Code–Ebene, sondern auch für die Laufzeit (Interoperabilität) garantiert, was die Wiederverwendbarkeit von C–Laufzeitbibliotheken ermöglicht und so den Kodieraufwand reduziert. Daher kann C++ gegenüber anderen objektorientierten Programmiersprachen wie Smalltalk oder CLOS bevorzugt werden.[13]

Logische Sprachen

Hierfür kommt hauptsächlich die Programmiersprache Prolog (*Programming in Logic*) [Kowalski 88, Clocksin/Mellish 84], die speziell für Probleme geeignet ist, die Objekte — insbesondere Strukturen — und Beziehungen zwischen ihnen beinhalten [Bratko 87]. Auf der Basis dieser Mechanismen ist es möglich zu programmieren, indem man das Problem beschreibt (nicht *wie* das Problem zu lösen ist, sondern *was* das Problem ist). Die Lösungsalternativen werden aus der Wissensbasis extrahiert, um die definierten Ziele zu erreichen. Eine Inferenzmaschine, die in Prolog selbst enthalten ist und auf der Basis von Rückwärtsverkettung arbeitet, basiert auf der Prädikaten–Logik und der logischen Deduktion[14]. Diese Mechanismen sind dem Benutzer auf der Meta–Ebene [Neumann 88] in einer modifizierbaren Form verfügbar (s.u.). Ein Prolog–Programm ist eine Liste von Klauseln, die aus Fakten und Regeln besteht. Daher ist es möglich, die Programme

[12] Das heißt, daß besondere Schottypen als Subklasse die Methode 'Beplattung' überschreiben können, die sie z.B. von der Elternklasse geerbt haben (Generalisierung und Spezialisierung).

[13] Vgl. Abschnitt 'Schlußfolgerungen' Kapitel 1.

[14] Eine Implementation der logischen Deduktion wird auch durch die Mechanismen *backtracking* und *pattern matching* (Unifikation) erreicht.

als Daten zu interpretieren oder zu manipulieren. Wegen dieser Eigenschaft ordnet man Prolog auch den Programmiersprachen zur symbolischen Datenverarbeitung zu.

Die Dimensionierungsregeln enthalten viele Informationen in Form von verbalen Aussagen, die man primitiv symbolisch erfassen kann, und numerischen Aktionen. In den meisten Fällen werden textliche Bedingungen erfragt und die numerischen Aktionen durchgeführt. Oft tritt dabei der Fall auf, daß eine Regel eine andere anspricht, sei es in Form von Hinweisen zur Auswertung dieser Regel oder zur Erlangung einiger Informationen zur Auswertung von sich selbst durch die Auswertung der referenzierten Regeln.

Das Programmieren der Dimensionierungsregeln mit Prolog wird dadurch erleichtert, daß sich der Programmierer nicht mehr darum zu kümmern braucht, wie die Regeln manipuliert werden, sondern darum, wie sie beschrieben werden. Außerdem verfügt Prolog über die Möglichkeit der Meta–Programmierung [Neumann 88], die einen neuen Ansatz in der Programmiertechnik darstellt. Dieser Ansatz basiert darauf, daß die Programme als Daten behandelt werden und durch die Entwicklung eines Programmsystems nicht die Repräsentation des Lösungsmodells, sondern die Interpretation der für dieses Lösungsmodell verwendeten Sprache komplexer wird. Damit ist gemeint, daß immer eine Erweiterung des Interpreters für eine neue Aufgabenstellung vorgesehen werden kann und so eine *problemorientierte* Sprache entsteht, die alle Aufgabenstellungen mit ihren Konstrukten abdecken kann[15]. Darüber hinaus verfügt Prolog über die Möglichkeiten zur Definition einer Grammatik und Implementation beliebiger Operatoren. Dies erlaubt es, Sprachen mit hoher Ausdruckskraft für einen speziellen Problembereich zu definieren.

Ein weiterer Vorteil von Prolog ist, daß die Kontrollmechanismen, die zur Lösung eines Problems geeignet sind, wiederum in den abstrakten Datentypen repräsentiert werden können, mit denen die zur Lösung dieses Problems erforderliche Information modelliert bzw. repräsentiert wird. Ein System läßt sich so bezüglich seines Datenmodells und seiner Kontrollmechanismen erweitern. Damit können nicht nur die Regeln, sondern auch ihre Abarbeitung modifiziert werden.

4.2.2 Datenbanken

Es ist der Ansatz vorstellbar, ein Datenbank–Schema für die Regeln zu definieren und dadurch die Regeln in einer Datenbank zu verwalten. Einige konventionelle Datenbanken verfügen über Schnittstellen zu den externen prozeduralen Programmiersprachen und eine Datendefinitions- und Datenmanipulationssprache (DDL, DML). Die Merkmale, die als erforderliche Information zur Definition der Regeln ausreichend sind, können durch

[15] Das ist z.B. der Fall bei dem Textverarbeitungssystem LaTeX [Lamport 86], in dem auf der Interpretationsebene Makros geschrieben werden. Dem End–Benutzer scheint der Programmieraufwand immer gleich zu bleiben, obwohl durch diese Makros die Interpretation komplexer wird.

die DDL definiert und durch die DML des jeweiligen Datenbankverwaltungssystems verarbeitet werden. Die heute verfügbaren Datenbanksysteme stellen einige Datenmodelle zur Verfügung, die als *hierarchisch, netzwerkartig* oder *relational* bezeichnet werden. Neuere Datenbanksysteme verfügen über *objektorientierte* Modelle. Einige Datenbanken mit relationalem Datenmodell sind erweitert worden, um wissensbasierte Modelle (Regeln, Inferenzmethoden) zu ermöglichen [Stonebraker/Rowe 86].

Daher ist es denkbar, die in Abschnitt 4.1.1 beschriebenen Datenstrukturen durch die Hilfsmittel des jeweiligen DBMS zu repräsentieren bzw. zu interpretieren. Dieser Ansatz kann aber mit dem heute üblichen relationalen DBMS nicht realisiert werden, da es über die oben erwähnten Datenstrukturen nicht verfügt. Deshalb müssen die neueren Datenbanksysteme [Kerschberg 86a, Ullmann 89] untersucht werden.

Die in einer Datenbank modellierten Regelstrukturen haben den Vorteil eines einfachen Zugriffs auf die anderen Datenobjekte (z.B. auf das Produktmodell der CAD–Daten) in der Datenbank. Die Vorteile einer derartigen Anwendung liegen in der einfachen Definition der Zusammenhänge (*relations*) zwischen den Regeln, der *automatischen* Überprüfung der Datenkonsistenz und der Herleitung der fehlenden Information mit Hilfe von Inferenz [Karagiannis 87].

Die Methoden, die die prozeduralen Eigenschaften der Regeln implementieren sollen, können in der DML des jeweiligen DBMS oder auch über Schnittstellen zu prozeduralen Sprachen programmiert werden. Dafür sind vor allem bidirektionale Schnittstellen geeignet, da die Methoden über die Schnittstelle zum DBMS auf die anderen Regeln zugreifen müssen. Das DBMS Postgres [Stonebraker/Rowe 86] ist eine Erweiterung des relationalen DBMS Ingres und verfügt über die Möglichkeiten zur Definition von Programmen als Daten und sogenannten Regeln, mit Hilfe derer mit *backtracking*–Algorithmen Schlußfolgerungen (*Inferenzen*) gezogen werden können.

Für die diesem Ansatz entsprechende Programmierung der Dimensionierungsregeln ist die Kenntnis der Sprachen des Datenbankverwaltungssystems erforderlich, um die Regeln in Datenstrukturen und Algorithmen zu repräsentieren. Mit objektorientierten DBMS läßt sich dieser Ansatz sehr gut verfolgen, weil die Regeln in ihrer Struktur den dort vorhandenen Modellierungskonzepten entsprechen.

Deduktive Datenbanken

Die deduktiven Datenbankverwaltungssysteme sind aus der Verbindung von relationalen Datenmodellen und Prolog entstanden. In [Minker/Nicolas 83] wird ein dDBVS so definiert:

> *"A deductive database is a database in which new facts can be derived from facts that were explicitly introduced."*

Die Eigenschaften eines deduktiven Datenbankverwaltungssystems erweitern die Anwendungs– und Einsatzmöglichkeiten von konventionellen Datenbankverwaltungssystemen, besonders in Richtung wissensbasierter Systeme. Hierzu zählen z.B.:

- *inferenzielle* Eigenschaften (*Deduktion*), die es ermöglichen, durch die Verwendung der Regeln aus vorhandenem Wissen neues zu generieren und somit die Datenbank selbständig zu erweitern,
- Modellierung von Objekten mittels Klassifizierung und Generalisierung,
- Verarbeitung von *unvollständigen* Informationen,
- Strukturierte Antworten anstatt Faktenaufstellungen,
- Meta–Modellierung bzw. Behandlung des Datenbankschemas.

Solche Systeme werden meistens auf der Basis der Kombination

Prolog + relationales Datenbankverwaltungssystem

realisiert [Kerschberg 86b, Kerschberg 86a]. Direktimplementationen derartiger DBVS existieren ebenfalls. Sie besitzen als Datenbanksprache für Definition, Abfrage und Manipulation eine Prolog ähnliche Sprache (z.B. DATALOG). Der Grund dafür ist, daß das verwendete Datenmodell ein *logisches* Datenmodell ist.

4.2.3 Entwicklungswerkzeuge für Expertensysteme

Weitere geeignete Software–Produkte für die regelbasierten Problemstellungen sind die sogenannten Expertensystem–Shells (*expert system shells*), wie sie im Bereich der Künstlichen Intelligenz eingesetzt werden.

Die meisten Shells entstanden aus der *Inferenz–Maschine* der früheren wissensbasierten Systeme (*knowledge based systems*), die eigentlich eine Wissensbasis und eine Inferenz–Maschine besitzen, die entsprechend dieser Wissensbasis Schlußfolgerungen ziehen kann. Die Arbeitsweise einer Inferenzmaschine (*inference engine*) basiert auf bestimmten Wissensrepräsentationen, die Sachverhalte mit Fakten und Regeln modellieren sollen. Wenn man bei solchen wissensbasierten Systemen die Wissensbasis entleert, so erhält man Expertensystem–Shells, die sich mit beliebigen anderen Wissensbasen füllen und damit für unterschiedliche Einsatzbereiche verwenden lassen. So ist Emycin–Shell z.B. aus dem wissensbasierten System Mycin [Buchanan/Shortliffe 84] entstanden. Neuere Shells werden auf der Basis dieser Erfahrung schon von vornherein programmiert. Für ihre Implementation werden Sprachen wie C, LISP oder Prolog eingesetzt. Beispielsweise sind die

Shells Gold Works, Personal Consultant in LISP, Advisor–II, XPERT–der Bär in Prolog [Fischer 88]), TRC [Kary 86] und NEXPERT Object [Lauer 89] in C realisiert. Eine hybride Shell [Brach/Pielmeier 88, Karagiannis 87], die in der Lage ist, verschiedene Wissensrepräsentationsarten wie Produktionsregeln, *frames* und semantische Netze kombinierend zu verarbeiten, kann in Form eines Meta–Programms (d.h. das Gesamtsystem muß sich selbst mit diesen Wissensrepräsentationen interpretieren lassen, z.B. das Regelwerkverwaltungssystem selbst kann ein semantisches Netz sein) sehr gute Unterstützung beim Programmieren von Regeln bieten. Das hybride System KANON [Karagiannis 87] verfügt über eine hybride Shell und ein deduktives Datenbankverwaltungssystem (dDBVS).

Ein für einen Problembereich zu entwickelndes wissensbasiertes System wird später auf der Basis einer Shell so implementiert, daß nur die Wissensbasis, die aus den Regeln und Fakten für diesen Problembereich besteht, eingegeben wird. Auf diese Weise besteht das Programmieren in der Beschreibung oder Modellierung des Problems auf Basis der von dieser Shell angebotenen Wissensrepräsentation. Deshalb kann man die Shells als deklarative Sprachen bezeichnen.

Die für die Entwicklung von Expertensystemen verwendeten Shells verfügen oft über eine Wissensrepräsentationsform, die als Produktionsregeln (*production rules*) bezeichnet wird. Andere Wissensrepräsentationen sind semantische Netze (*semantic networks*), Rahmen (*frames*) usw., die verschiedene Modellierungskonzepte beinhalten. NEXPERT Object verfügt z.B. über die Wissensrepräsentationsformen *frames* und Produktionsregeln und ihre Kombination. TRC verfügt über die Produktionsregeln. Produktionsregeln haben häufig die Form:

wenn *Vorbedingungen*
dann *Folgerungen*

und weisen auf eine Ähnlichkeit mit den Syntaxregeln der Compiler–Generatoren wie `yacc` auf. Die Verarbeitung der Regeln spiegelt dort nochmals ihre Wichtigkeit in relativ komplexen Aufgabenstellungen wider.

Die oben kurz erwähnten Produkte verfügen auch über diese Wissensrepräsentationsform. In den *Vorbedingungen* und *Folgerungen*, die oft durch A/W–Paaren oder O/A/W–Tripeln[16] (siehe Abschnitt 4.1.1) abgebildet werden, können andere Regeln aufgerufen werden. Die Inferenzmaschinen unterscheiden sich meistens in der Reihenfolge der Aufrufe von Regeln. Hier sind z.B. die Inferenzmechanismen für Rückwärts– und Vorwärtsverkettung zu unterscheiden. Bei der Vorwärtsverkettung werden zuerst alle Regeln, die mit ihren "Vorbedingungen" die vorgegebenen Ziele erfüllen, temporär abgespeichert, erst dann werden entsprechend dieser abgelegten Regelmenge "Folgerungen" interpretiert bzw. innerhalb derer evtl. andere Regeln aufgerufen und ähnliches. Bei der Rückwärtsverkettung werden "Folgerungen" gleich interpretiert, sobald die "Vorbedingungen" erfüllt werden. Die beiden Inferenzmethoden sind je nach Aufgabenstellung

[16] Attribut/Wert–Paar bzw. Objekt/Attribut/Wert–Tripel

mehr oder weniger geeignet. Die Inferenzmaschinen können unterschiedliche Techniken zur Verarbeitung der Regelbasis jeweils für verschiedene Zwecke oder für die verschiedenen Wissensrepräsentation verwenden.

Eine Erweiterung dieser Verarbeitungsform, die bisher nur über die Produktionsregeln verfügt, um andere Repräsentationsarten wie Frames eignet sich auch für Dimensionierungsregeln, da die Dimensionierungsregeln auf komplexe Objekte wie das Produktmodell angewandt werden, für deren Definition Hilfsmittel wie Frames geeignet sind.

4.2.4 Definition einer problemorientierten Sprache

Dieser Ansatz macht es erforderlich, die syntaktischen und semantischen Eigenschaften der Regeln genauer zu analysieren. Bei dieser Analyse spielt die erforderliche Information zur Auswertung der Regeln eine große Rolle, weil diese mit einer speziellen Programmiersprache durch einen Interpreter oder Compiler verarbeitet werden muß, wobei die Auswertungsmechanismen in Abhängigkeit von syntaktischen und semantischen Eigenschaften der Regeln in die Programmiersprache eingebettet sein müssen. Nur so wird es möglich, eine Regel in Form einer Beschreibung zu verarbeiten, d.h. das System muß fähig sein, aus Lexemen[17] die Ablaufstrukturen zu generieren. Zum Beispiel könnte eine Operation in Form von

$$t_{\square} = \begin{cases} t_1 & : \quad t_1 > t_2 \leftarrow [12.D.1] \\ t_2 & : \quad t_2 > t_1 \leftarrow [12.B.1] \\ t_1 & : \quad t_1 \cong t_2 \leftarrow [1.J] \end{cases}$$

durchgeführt werden.

Eine vorstellbare Menge von Elementen einer solchen Sprache ist in Tab. 4.1 (vgl. Abschnitt 7.2) gegeben. Die für die Erstellung einer problemorientierten Sprache verwendbaren Software–Hilfsmittel sind Compiler–Generatoren wie `yacc` des UNIX–Betriebssystems [Johnson 75] oder LALR [Mann 87]. Mit solchen Werkzeugen kann die Syntax einer problemorientierten Sprache definiert und daraus ein Parser erzeugt werden. Die semantischen Aktionen sind jeweils zu programmieren.[18]

Schlußfolgerungen

Die Flexibilität beim Programmieren komplexer, stark interaktiver Systeme mit Prolog kombiniert mit Möglichkeiten für die Verarbeitung eingebetteter Sprachen mittels *definite*

[17] die Elemente dieser Sprache
[18] Vgl. [Jensen 92].

APPLIC:

Stichworte zur Beschreibung des Verwendungszwecks einer Formel. Diese Information wird vom Regelwerkverwaltungssystem ausgewertet, um für eine Fragestellung geeignete Gleichungen zu finden. Beispiel: 'Beplattung'

ID:

Kennzahlen oder Texte, die insbesondere Hinweise auf die Quelle einer Gleichung geben sollen. Es sind fest definierte Formate denkbar, um Standards wie z.B. Klassevorschriften auf einfache Weise identifizieren zu können. Beispiel: '11.B.1.2'

EQN:

Eine solche Anweisung dient der Benennung der algebraischen Zielgröße einer Gleichung bzw. eines Gleichungssystems.

- *Arithmetischer Ausdruck:*
  Zur Darstellung der Formeln gibt es alle wichtigen mathematischen Elemente für die Notation von algebraischen Gleichungen wie

Operatoren:	`+ - * / \% ( )`
Variablen:	Namen beliebiger Länge und Typen

- *Funktionen:*
  Alle wesentlichen mathematischen Funktionen stehen zur Verfügung. Darüber hinaus gibt es Funktionen für den Zugriff auf das Produktmodell und die Regelbasis.

- *Allgemeine Anweisungen:*
  Es gibt eine kleine Anzahl von Kontrollanweisungen:

Blockstruktur:	Einrückungen oder explizit durch Zeichen wie `{` und `}`
Sequenzen:	`; , ...: :  | [ ]`
Verzweigung:	`if then else`
Iteration:	`repeat foreach`
Kommentare:	`//` `/* */`

Beispiel: '$t = 1.1 * a * \sqrt{p} + t_K$'

RESULT:

Das Regelwerkverwaltungssystem muß das Ergebnis einer Formel symbolisch im Voraus bestimmen können. Durch geeignete Stichworte kann dieses Ziel beschrieben werden.

Tab. 4.1: *Sprachelemente einer Regelbeschreibungssprache*

clause grammars, welches in letzter Zeit ein eingebautes Werkzeug vieler Prolog–Implementationen geworden ist, läßt keine weiteren Alternativen zu, als für die Realisierung eines Regelwerkverwaltungssystems die Programmiersprache Prolog zu verwenden. Die relevanten Vorteile von Prolog für den vorliegenden Fall nochmals zusammengefaßt wären:

- *Rapid prototyping* – Typenfreies Programmieren gibt dem Programmierer die Möglichkeit, sich nur noch mit Werten der Variablen zu beschäftigen, nicht mit Typen der Variablen.
- Eingebauter Theorembeweiser – Er gibt die Möglichkeit, Regeln direkt mit Prolog zu verarbeiten. Die Übersetzung aus einer problemorientierten Regelsprache wäre viel einfacher als die Übersetzung in eine andere Programmiersprache, in der die Regelverarbeitung zusätzlich programmiert werden muß.
- Interaktive Programmentwicklung – ist eine weitere Unterstützung für den ersten Punkt.
- Datenbankprogrammierung – Die Laufzeitdatenbank von Prolog erlaubt Programmierweisen, die sonst nur noch mit Datenbanken möglich sind, wie komplexe Datenabfragen, automatische Suche etc.

Kapitel 5

Systemarchitektur

Um den Dimensionierungsprozeß zwischen den ingenieurmäßigen Programmsystemen für Entwurf, Strukturanalyse, Konstruktion und Dimensionierung ohne Datenredundanz zu unterstützen, wird hier eine flexible und leistungsfähige Architektur vorgeschlagen, deren Komponenten zum Teil bereits implementiert, in der Entwicklung oder im Entwurf sind. Einige Komponenten wie das STEP–Parser–System und einige Werkzeuge, die auf dem Parser–System basieren, wurden im Rahmen des Forschungsvorhabens *"Datengenerierung für Finite–Elemente–Berechnungen mit Hilfe von CAD–Systemen"* [Toparlak 91a] entwickelt und implementiert oder befinden sich in der Weiterentwicklung.

Ein spezielles Datenmodell für die Stahlstruktur konnte im Rahmen einer Dissertation entworfen werden [Bronsart 90]. Eine Erweiterung dieses Datenmodells wurde in [Koch 91][1] nach STEP durchgeführt. Die vorliegende Arbeit und das Forschungsvorhaben *"Entwicklung eines wissensbasierten Systems zur Spezifikation und Interpretation der Dimensionierungsvorschriften für die schiffbauliche Stahlstruktur"* erweitert dieses Modell, um die Bemessung an einem genormten Produktmodell zu ermöglichen.

Die in Abb. 5.1 vorgeschlagene Architektur stellt einen Dimensionierungsprozeß dar, in dem mit Hilfe eines einheitlichen Datenmodells zwischen allen beteiligten Systemen ein redundanzfreier Datenaustausch ermöglicht wird.

Diese Architektur eignet sich auch für einen normalen Datenaustausch zwischen CAD und FEA, wobei davon ausgegangen wird, daß eine Dimensionierung bereits innerhalb des CAD–Systems erfolgt ist. Im folgenden wird beispielhaft gezeigt, wie mit Hilfe des Parser–Systems eine Konvertierung der CAD–Daten in das Eingabeformat eines FE–Systems erfolgt ([Toparlak 91a]).

[1] Das von KOCH entwickelte Produktmodell für die schiffbauliche Stahlkonstruktion ist inzwischen Bestandteil der ISO–Norm STEP (*ISO Draft Proposal 10303 Part 218*).

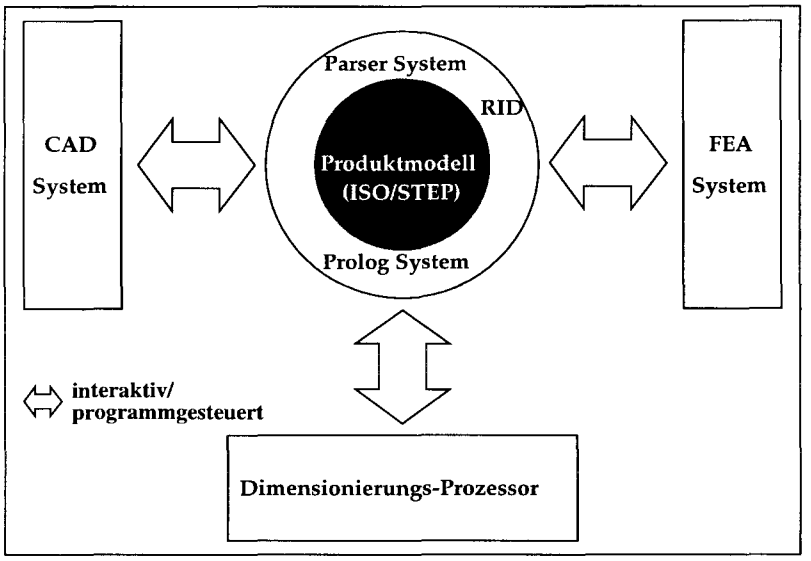


Abb. 5.1: *Systemarchitektur*

Weitere Details dieser Architektur sind dem FDS–Bericht Nr. 249/1993 [Toparlak 93b] zu entnehmen.

5.1 RID der Produktdaten

Die rechnerinterne Darstellung der Produktdaten, die für den Dimensionierungsprozeß erforderlich sind, basiert auf einer hauptspeicherresidenten Abbildung von STEP *physical files*. *Physical files* sind nach STEP die **neutralen**, d.h. systemunabhängigen Dateien für den Datenaustausch.

Das Format[2] der *Physical files* ist eine kontextfreie Grammatik, die für die Abbildung des File–Inhalts auf den Arbeitsspeicher des Rechners von einem Compiler–Generator fast automatisch abgearbeitet werden kann, um einen entsprechenden Parser aufzubauen, mit dessen Hilfe dann der Abbildungsvorgang realisiert wird. Die hierfür verwendete File–Syntax ist eine vereinfachte Form der durch STEP veröffentlichten Syntax, die selbst noch nicht einmal in den Anfängen beschlossen ist. Die Tab. 5.1 zeigt die allgemein verwendete Syntax der **rechnerexternen** Produktdaten für die Stahlschiffsstruktur. Die rechnerinterne Abbildung wird mit Hilfe der dynamischen Speicherverwaltungsfunktionen der Programmiersprache **C** realisiert. Dabei wurde ein kostspieliges Laufzeitverhalten in Kauf genommen, um eine **flexible** Schnittstelle für die Anwenderprogramme zur Verfügung zu stellen. Damit wurden wie erwünscht gute Erfahrungen gemacht, da sich diese einfache Schnittstelle mit relativ geringem Aufwand optimieren ließ.

[2] im folgenden Syntax genannt

```
%{
/* Description: STEP physical file syntax definition for yacc(1)          */
/* Time-stamp: "94/03/27 17:59:36 at"                                     */
%}

%token          REAL            FILE_SCHEMA
                INTEGER         TYPE
                STRING          IDENTIFIER

%%                                                                        10

step_physical_file:     file_schema  entity_sequence
        ;
file_schema:            FILE_SCHEMA '(' string ')'
        ;
entity_sequence:        entity
        |               entity_sequence, entity
        ;
entity:                 IDENTIFIER '=' TYPE '(' attributes ')'
        |               application /* application_specific_entity */     20
        ;
attribute_sequence:     attribute
        |               attribute_sequence, attribute
        ;
attribute:              REAL
        |               INTEGER
        |               STRING
        |               reference
        |               record
        |               '(' attribute_sequence ')'                         30
        ;
reference:              IDENTIFIER
        ;
record:                 TYPE '(' attribute_sequence ')'
        ;
%%
```

Tab. 5.1: *Filesyntax für Produktdaten*

$$\left|\begin{array}{llllll} Id_1 & Type_1 & Attr_{11} & Attr_{12} & \cdots & Attr_{1n_1} \\ Id_2 & Type_2 & Attr_{21} & Attr_{22} & \cdots & Attr_{2n_2} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ Id_m & Type_m & Attr_{m1} & Attr_{m2} & \cdots & Attr_{mn_m} \end{array}\right|$$

Zugriffe auf Daten

$$Id_m$$
$$Id_m.Type_m$$
$$Id_m.Attr_x \quad \{-1 < x < n_m + 1\}$$

$$Id_m \equiv Id_m.Attr_0$$
$$Id_m + x \equiv Id_m.Attr_x$$

Abb. 5.2: *Dynamische Speicherbelegung*

Die rechnerinterne Repräsentation der Daten ist einer (m, n)–Matrix ähnlich (Abb. 5.2), wobei n eine *elastische* Zahl ist, und zwar für jede Entity verschieden[3]. Die Zahl m ist dagegen *plastisch*, da sie ausschließlich anwächst. Der Lese– und Abbildungsaufwand, der durch den Parser geleistet wird, um diese Matrix im dynamischen Speicherplatz ähnlich abzubilden, lohnt sich insofern, als dann der Zugriff auf die Daten mittels eines einfachen Indexes realisiert werden kann, was die Entwicklung von Anwenderprogrammen mit dieser Schnittstelle erheblich vereinfacht.

5.2 STEP–Parser–System

Die Realisierung von Schnittstellen für STEP–Modelle ist auf Grund des großen Umfangs und der internen komplexen Beziehungen der enthaltenen Entities der STEP–Datenmodelle sehr aufwendig. Die Verarbeitung der STEP–Modelle durch ingenieurmäßige Systeme erfordert sowohl Prozessoren zur Erzeugung als auch zum Lesen und Analysieren derselben in Form von Dateien. Darüber hinaus sind Werkzeuge für das Testen der korrekten Funktion der Prozessoren und zur Analyse des Dateiinhalts erforderlich. Das vorliegende Parser–System enthält daher die folgenden Komponenten:

- einen programmierbaren (*generischen*) Parser für die physikalische File–Struktur (STEP *physical file*), (s. Abschnitt 8.2)
- eine Bibliothek zur Erzeugung von STEP–Modellen und deren physikalische Abbildung in ein STEP–File,
- eine Reihe von Werkzeugen zur Analyse, Optimierung u.ä. von STEP–Modellen.

[3] verschiedene Anzahl der Attribute

5.3 STEP–Prolog–System

Prolog ist speziell für Probleme geeignet, die Objekte — insbesondere Strukturen — und Beziehungen zwischen ihnen beinhalten. Diese Eigenschaften sind gerade auch für die Entity–Instanzen der STEP–Modelle geeignet. Die datenorientierte Programmiermethodik, Suchstrategien und Problemlöse–Mechanismen wie *Deduktion, Resolution* usw. in Prolog bieten erhebliche Vorteile gegenüber einer konventionellen Programmiersprache. Die **'deklarative'** Programmiermethodik ist daher ein geeignetes Mittel, um STEP–Modelle mit Prolog zu verarbeiten. Einige der Vorteile von Prolog sind dabei:

- Transparente Abbildung der Entity–Instanzen in der Prolog–Datenbank sowie im Format der verfügbaren Datenrepräsentation in Prolog.
- Die regel– bzw. wissensbasierten Techniken ermöglichen eine leichte Handhabung der Semantikanalyse mit der Abbildung von EXPRESS auf Prolog–Prädikate.

Die EXPRESS–Modelle, welche die Semantik der Entities vorgeben, sind ebenfalls relativ einfach in Prolog abzubilden.

5.4 Prolog Dimensionierungsprozessor

Nachdem in den vorangegangenen Abschnitten die Problematik der Verarbeitung von Daten, die für den Dimensionierungsprozeß erforderlich sind, behandelt und die Möglichkeit ihrer Zusammenfassung in einem Produktmodell gezeigt wurde, können jetzt erste Lösungsansätze entwickelt werden. Weiterhin steht die Unvollständigkeit der erforderlichen Information als ein ungelöstes Problem im Vordergrund. Dieses Problem kann mit zwei Methoden gelöst werden:

- Herleitung der fehlenden Informationen aus bereits vorhandenen unter Verwendung von Regeln, die zum Teil aus den Bauvorschriften gewonnen werden können.
- Automatisierte Interaktion mit dem Anwender, wenn die Gewinnung fehlender Information nicht unter Verwendung von Regeln oder z.T. komplizierten geometrisch topologischen Operationen möglich ist.

Bei der vorliegenden Systemarchitektur wurde davon Abstand genommen, fehlende Informationen wie Geometrie oder Topologie durch komplizierte geometrische Operationen zu ergänzen. Eine Vervollständigung der geometrisch–topologischen Information ist nur dann sinnvoll, wenn sie unter Verwendung von Bemessungsregeln möglich ist, weil dies in das Umfeld der beabsichtigten Arbeitsweise eines Dimensionierungsprozesses fällt.

5.4.1 Repräsentation der Vorschriften

Für die rechnerinterne Repräsentation der Bemessungsregeln wurde als Implementationsbasis einer *problemorientierten* Sprache Prolog ausgewählt (s. Kapitel 7 und Abschnitt 7.2). Prolog ist für diese Thematik besonders geeignet, da sie das Produktmodell direkt mit eigenen Sprachkonstrukten abbilden kann und die erforderlichen Möglichkeiten für die Repräsentation und Ausführung von Bemessungsvorschriften bietet. Die Möglichkeiten der Meta–Programmierung in Prolog sind die Voraussetzung, diese Sprache zu konzipieren und zu realisieren. Prolog ist auch für die Implementierung von *eingebetteten* Sprachen geeignet. Das heißt, eine solche problemorientierte Sprache kann direkt von einem Prolog–Interpreter ausgeführt werden, wenn die Sprachsyntax der von Prolog ähnelt.

Um die Vorgehensweise bei der Programmierung der Bemessungsregeln zu verdeutlichen, wird in diesem Abschnitt ein Referenz–Prototyp vorgestellt, der die im Abschnitt 3.2.1 gezeigte Dimensionierungsaufgabe löst, indem die Bemessungsregeln zuerst auf Prolog–Regeln abgebildet und dann mit Hilfe eines einfachen Inferenzmechanismus, der mittels Resolutionsverfahrens in Prolog eingebaut zur Verfügung steht, ausgewertet werden. Eine Schnittstelle zum Produktmodell ist automatisch vorhanden, weil sie durch Fakten repräsentiert wird und transparent abgefragt werden kann.

Für die Abbildung der Dimensionierungsregeln auf eine für die Datenverarbeitung geeignete Form eignet sich Prolog besonders gut, da die Vorschriften eine verallgemeinerte Form der rekursiven Regelauswahl und Regelauswertung darstellen. Die **Rekursion** in Prolog ist sowohl auf Datenstrukturen als auch auf Regeln, wenn diese als Prädikate definiert sind, möglich. Sehr viele Aspekte, die in Vorschriften vorkommen, sind mit Hilfe der rekursiven Regelauswahl und Regelauswertung abzubilden. Damit kann eine generelle Formalisierung erreicht werden.

5.4.2 Interpretation der Vorschriften

Wenn die Vorschriften als Prolog–Regeln implementiert werden, können diese mit Hilfe eines Prolog–Interpreters ohne weiteres abgearbeitet werden. Die Ausführung der Regeln erfolgt mittels eines Inferenzmechanismus, der auf *logischer Deduktion* basiert und im Inneren eines Prolog–Interpreters mit Hilfe eines Resolutionsverfahrens realisiert wird. Die logische Deduktion ist ein Verfahren, das für die oben erwähnte Form der rekursiven Regelauswahl und –auswertung besonders effektiv ist. Die Lösung eines Dimensionierungsproblems ist dann eine logische Schlußfolgerung, die durch die Anwendung des Deduktionsverfahrens auf Regeln gewonnen wird. Das Programm in Abb. 5.3 veranschaulicht das beispielhafte Programmieren von Bemessungsregeln und ihre Ausführung (Abb. 5.4) mit diesem Ansatz.

```
schottbeplattung(S):-
        schiffstyp(standard),
        kontruktionsgruppe(wasserdicht),
        select(11,S).
schottbeplattung(S):-
        schiffstyp(standard),
        select(1-22,S).
schottbeplattung(S):-
        schiffstyp(tanker),
        select(24,S).                                          10
schottbeplattung(S):-
        schiffstyp(erzschiffe),
        select(23,S).
select(11,S):-
        kontruktionstyp(standardschott),
        select(11:B:2,S).
select(11:B:2,S):-
        S = c_p * a * sqrt(h * k) + t_K.
```

Abb. 5.3: *Wissensbasiertes Programm 'Schottbeplattung'*

```
% Product data
#1 : schiffstyp(standard).
#2 : kontruktionsgruppe(standardschott).
% Interface to product model and user
laderaumnutzung(L):-
        #1 : laderaumnutzung(L)
        ;
        ask_user(laderaumnutzung,[standard,ballast],L),
        % Add info to product model
        insert(laderaumnutzung,L).                             10
% ...
?- schottbeplattung(T).
laderaumnutzung?        [standard,ballast]
@ ballast
konstruktionsgruppe?    wasserdicht:[ja,nein]
@ ja
T = c_p * a * sqrt(h * k) + t_K
yes
?-
```

Abb. 5.4: *Prolog-Sitzung 'Schottbeplattung'*

Weitere Anforderungen an die Systemarchitektur

Schnittstelle zu einer objektorientierten externen Sprache

Die Schnittstelle der Prolog–Programmierumgebung zu einer objektorientierten Sprache ist für die Manipulation von externen Datenobjekten (z.B. CAD–Objekte in der Produktmodelldatenbank) geeignet. Dadurch kann die Verwaltung von Objekten sehr effizient organisiert werden [Koschman/Evens 88]. Die bei den meisten Prolog–Systemen verfügbare Schnittstelle von Prolog zu C kann für diesen Ansatz zu C++ [Wiener/Pinson 88, Koschman/Evens 88] erweitert werden. Durch die Bidirektionalität einer solchen Schnittstelle können zum Beispiel die Methoden, die aufwendig mit Ansätzen auf der Basis prozeduraler Sprachen zu implementieren sind, in Prolog programmiert werden. Umgekehrt können Aktionen, die deklarativ schwerer als prozedural zu beschreiben sind (grafische Ein– und Ausgabe, Numerik, Verwalten von CAD–Objekten), in Prolog durch eine Schnittstelle zu C++ programmiert werden. Dieser Ansatz wird als *multiparadigm programming* bezeichnet [Koschman/Evens 88]. Gemeint ist hiermit, daß die Möglichkeit besteht, innerhalb eines Programmiersystems die jeweils am besten geeigneten Hilfsmittel (deklarative, prozedurale oder objektorientierte Programmierung) zu nutzen. Die Mechanismen wie *backtracking* oder *unification* müssen über diese Schnittstelle zur Verfügung gestellt werden.

Externes Datenbankverwaltungssystem

Die Schnittstelle zu einem objektorientierten Datenbankverwaltungssystem ist notwendig. Die CAD–Objekte des Produktdatenmodells, die in einer externen Datenbank liegen, können über diese Datenbankschnittstelle angesprochen und verarbeitet werden.

Wissensakquisitionssystem

Für die Aktualisierung der Regeln in der Regelbasis können verschiedene Konzepte verfolgt werden. Das System muß über Werkzeuge wie syntaxorientierte Editoren, Compiler o.ä. für das Programmieren von Regeln verfügen, um die Eingabe der Regeln benutzerfreundlicher zu gestalten. Die Programmierung durch eine problemorientierte Sprache vermindert vor allem den Programmieraufwand zur Aktualisierung von Regeln oder Fakten (*knowledge acquisition*).

Meta–Programmierungssystem

Das Meta–Programmierungssystem dient zur Definition einer problemorientierten Sprache und zur Modifikation des gesamten Systems. Beispielsweise muß dieses System die Funktionalitäten für die Definition einer problemorientierten Sprache und auch andere Komponenten innerhalb des Gesamtsystems modifizieren können.

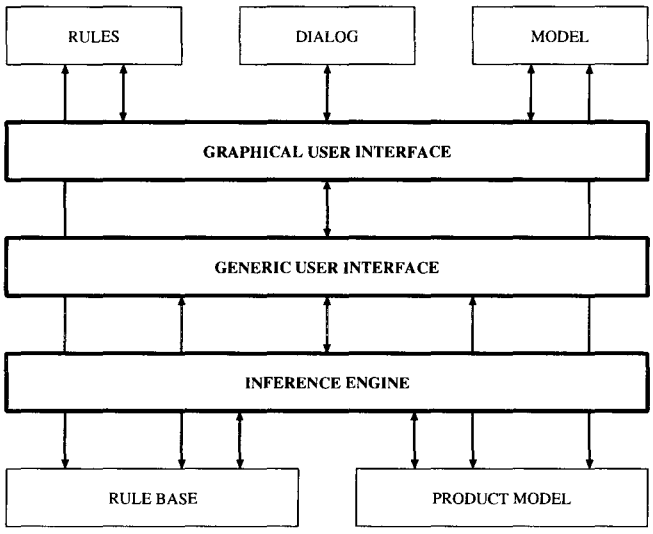


Abb. 5.5: *Architektur eines prototypischen Regelwerkverwaltungssystems*

Schlußfolgerungen

Ein Systemarchitektur–Entwurf für den Dimensionierungsprozessor, der den oben gestellten Forderungen gerecht werden soll, wird in der Abb. 5.5 dargestellt. Dieser erweist sich durchaus zureichend, da er über die besondere Eigenschaft verfügt, daß jede einzelne Komponente unabhängig von allen anderen und in sich abgeschlossen ist. Er unterstützt daher eine schnelle Realisierung des Gesamtsystems, indem dies zuerst nur noch prototypisch implementiert wird.

Kapitel 6

Logische Programmierung und Prolog

Prolog (*Programming in logic*) ist eine im Bereich der "Künstlichen Intelligenz" verbreitet eingesetzte Programmiersprache [Bratko 87]. Sie beschränkt sich auf eine kleine Menge von Mechanismen wie *pattern matching*, baumbasierte Datenstrukturen und automatisches *backtracking*. Diese kleine Menge bildet eine überraschend mächtige und flexible Programmierumgebung. Die Unterstützung der logischen Programmierung macht aus der Programmiersprache Prolog ein hervorragendes Werkzeug für die Programmierung von technischen Regeln jedweder Art, die sich in Form von sog. 'logischen Klauseln' formulieren lassen.

Die logischen Klauseln (das sind sogenannte Horn–Klauseln der Logik erster Stufe) dienen im allgemeinen zur Beschreibung der komplexen Sachverhalte mittels Fakten, Informationen, deren Korrektheit (oder auch umgekehrt) keiner Beweisführung bedarf, und Regeln, mit deren Hilfe neue Fakten aus bestehenden gewonnen werden können.

Ganz allgemein haben die Regeln in Logik erster Stufe (bzw. auch in Prolog) die folgende Form:

$$L \leftarrow R_1, R_2, \ldots, R_n,$$

in welcher der Term der linken Seite gilt, wenn die Terme der rechten Seite beweisbar, d.h. wahr sind. Die Fakten erweisen sich daher als eine einfache Spezialform der Regeln, bei denen die folgende Form:

$L \leftarrow true$ oder
$L \leftarrow fail$

die Tatsache beschreibt. Hierbei entspricht die erste Form der Wahrheit, bei der zweiten Form handelt es sich um ein explizites negatives Wissen (Negation), d.h. Information, die **nicht** wahr ist. Es ist leicht einzusehen, daß bei dieser Form der Regeln die rechte Seite selbstbestimmend ist und daher keinen Beweis erfordert. Da die Negation oft nicht verwendet wird, verzichtet man darauf, die Fakten des rechten Teils, wenn sie den Wert $true$ haben, weiterhin zu spezifizieren und begnügt sich mit der Equivalentform:

$$L \equiv L \leftarrow true.$$

Bei den Termen handelt es sich lediglich um einfache oder komplexe Objekte, die z.B. Zahlen, Zeichenketten (Atome), Listen oder ganz allgemein Bäume sein können. Die letzten Objekte sind zusammengesetzte Objekte, die ausschließlich aus den Atomen bestehen. Ein Term weist die folgende Form auf

$$functor(Subterm_1, Subterm_2, \ldots, Subterm_n),$$

wobei jeder Subterm ein Term oder einfaches Objekt sein kann. Einen Term in dieser Form kann man auch mit $functor/n$ bezeichnen, wobei es sich hier um einen Term $functor$ mit n Subtermen handelt. So ähnelt ein Term in der logischen Programmierung einem Funktionsaufruf mit dem Funktionsnamen $functor$ und ihren Eingabe– oder Ausgabeparametern der Anzahl n. Ein Term kann dementsprechend den Rückgabewert einer solchen Funktion in einer prozeduralen (oder funktionalen) Sprache mit einem zusätzlichen Subterm ($Subterm_{(n+1)}$) simulieren. In der logischen Programmierung kann man (nur informell) von einem Rückgabewert sprechen, der nur noch die Werte $wahr$ oder $nicht\ wahr$[1] haben kann. Er ist aber nicht direkt beeinflußbar durch die Anweisungen eines Programms. Umgekehrt entspricht dieser Wert dem Rückgabewert einer Funktion einer prozeduralen Sprache, die den booleschen (oder logischen) Wert 0 oder 1 zurückliefert.

Eine solche Funktion wird meistens mit der Definition

$$f : (a, b, \ldots n) \rightarrow \{0, 1\}$$

formal angegeben.

Daher spielt die folgende Equivalentform

$$f : (a, b, \ldots n) \rightarrow \{0, 1\} \cong functor(a, b, \ldots n)$$

[1] Der Wert 'falsch' existiert in dem Sinne nicht, weil er nicht bewiesen werden kann; deswegen $fail$ (scheitern, versagen), wenn nicht bewiesen werden kann, daß es wahr ist.

zwischen einem logischen Term und einer booleschen Funktion bei der Konzeption der Schnittstelle zwischen der logischen Programmiersprache Prolog und der prozeduralen Sprache **C** eine wichtige Rolle. Wie bereits erwähnt, wurde die Programmiersprache **C** für die Implementation des STEP–Parser–Systems (SPS) eingesetzt. Daher ist die Betrachtung der **C**/Prolog Schnittstelle (zum Zwecke der Integration des RWVS's mit dem SPS) aus dieser formalen Sicht notwendig.[2]

Da die Terme nicht nur den prozeduralen Funktionsaufrufen ähneln, sondern auch komplexe Daten repräsentieren können, sind sie durchaus in der Lage, die rechnerinterne Darstellung von Produktdaten, die gegebenenfalls komplexe Strukturen wie Bäume, Netze, assoziative Listen usw. aufweisen können, zu übernehmen. In Kapitel 5 wurden die ersten Ansätze für die Verwaltung der Produktdaten in Prolog mit Hilfe von Fakten kurz dargestellt, die hier weiter vertieft werden sollen. Die Fakten sind logische Prädikate der Form:

$$L \leftarrow true$$

wobei die linke Seite L ein beliebiger Term sein kann. So kann z.B. eine komplexe Entity wie `polyline` aus dem STEP–Geometriemodell etwa mit dem logischen Fakt

$$polyline(10, coordinate\_system(\ldots), [2, 7, 89, \ldots]) \leftarrow true$$

definiert werden. Dabei ist die linke Seite ein Term, der aus dem Funktor $polyline$ mit den baumartig eingeordneten Subtermen (wie Parameter) besteht. Diese sind lediglich die Identifikation der Entity, die Zahl 10, das Koordinatensystem $coordinate\_system(\ldots)$, dem das geometrische Objekt `polyline` zugeordnet ist, und die Liste von `point` Entities, die mit ihrer Identifikation referenziert werden, als dritter Subterm. Der Subterm $coordinate\_system$ ist hier nicht genau spezifiziert, da '...' jeweils auf weitere Subterme zur Beschreibung des lokalen Koordinatensystems hinweist.

Es wird deutlich, daß die logischen Terme komplexe Datenstrukturen sehr flexibel beschreiben, weswegen auch in prozeduralen Sprachen ein Entwurf entsprechender Datentypen dringend erforderlich ist. Ohne ihn besteht sonst keine Möglichkeit, solche Daten zu handhaben und zum Beispiel auf deren bestimmte Attribute zuzugreifen. Ein wichtiger Vorteil liegt darin, daß die Terme **rekursiv** und **typenfrei** (*typeless*) aufgebaut werden können und daher jeder Komplexitätsgrad für den Aufbau von strukturierten Daten zu bewerkstelligen ist.

Durch das Weglassen der rechten Seite, d.h. des Terms $true$, wird eine Darstellung der Entities aus dem Produktmodell als eine Menge von logischen Fakten, deren Interpretation einfach vorliegt, erreicht. Diese Interpretation entspricht in der vorliegenden Form ganz

[2] Die Erörterung der technischen Details dazu befindet sich im Abschnitt 8.1.

genau ihrer Repräsentation und somit ihrer physikalischen Abbildung, z.B. in Form einer Datei. Unter der Interpretation eines logischen Programms kann man sich alle Aussagen über die in dem Programm existierende Begriffswelt vorstellen, die aus dem Programm *ableitbar* sind. Darunter versteht man alle neu gewonnenen Fakten, die unter Verwendung der vorhandenen Regeln und Fakten ermittelt werden können. Da in der vorliegenden Begriffswelt noch keine Regeln definiert wurden, ist ihre Interpretation nichts weiter als die Gesamtheit aller Fakten:

$$I_W \equiv F_W.$$

Somit läßt sich das Produktmodell ohne jegliche Interpretation als eine Menge von Fakten wie etwa

$$\begin{array}{l} point(1,\$,0.0,0.0,0.0) \\ point(2,\$,1.0,0.5,2.34^{-2}) \\ \ldots \\ polyline(21,\$,[1,2,\ldots]) \\ polyline(200,coordinate\_system(\ldots),[\ldots]) \end{array}$$

als ein einfaches logisches Programm rechnerintern[3] darstellen und mit einem Interpreter für Logik (z.B. Prolog–Interpreter) ohne weiteres ausführen. Das zeigt einen der wichtigsten Aspekte der wissensbasierten Programmierung, die sich mit Hilfe der Methoden der logischen Programmierung realisieren läßt, nämlich Programme und Daten nicht voneinander zu trennen und diese gleichartig zu verarbeiten. Solch ein Programm kann als eine Art Datenbank betrachtet werden, an die mit Hilfe logischer Variablen Fragen gestellt werden können.

Solche Anfragen werden unter Verwendung der logischen Variablen in Form von Existenzquantoren [Gibbins 90] realisiert wie z.B. in

$$\neg(\exists I)(\exists X)(\exists Y)(\exists Z)point(I,\$,X,Y,Z),$$

deren Beweis etwa durch die Substitution

$$\{\{X=0.0,Y=0.0,Z=0.0\},\{X=1.0,Y=0.5,Z=2.34^{-2}\},\ldots\}$$

durchgeführt werden kann. Die logische Negation ($\neg$)[4] wurde hier als Anfrageform gewählt, wie sie auch in Prolog verwendet wird. Als Beweismechanismus wird dabei das Verfahren der logischen Refutation angewendet, bei der versucht wird, eine Gegenaussage zu finden, mit der die folgende konjunktiv–normale Form gilt, d.h. immer 'wahr' ist:

[3] sowohl im Hauptspeicher des Rechners als auch auf Speichermedien in Form von Dateien

[4] Die vordefinierte Zeichenkette '?-' in Prolog entspricht der Bezeichnung $\neg$ für die logische Negation.

$$\neg P \vee P.$$

Damit reduziert sich der Aufwand der Beweisführung auf Suche und Mustervergleich (hier speziell auf den Nachweis der Existenz der logischen Aussage P), die in Prolog mit dem Suchalgorithmus *deep first Backtracking* und dem Algorithmus für Mustervergleich *pattern matching unification* verwirklicht wird. Daher nennt man die logische Programmiersprache Prolog auch eine symbolverarbeitende Sprache. Ein Nachweis der Existenz der logischen Aussage

$$P \equiv point(I, \$, X, Y, Z)$$

erfüllt die oben angegebene Bedingung mit der Substitution der logischen Variablen X, Y, Z bei entsprechender Unifikation. So würde ein Prolog–Interpreter auf die Anfrage $?-point(I, \$, X, Y, Z)$ nach der Eingabe obiger Fakten (aus dem Produktmodell) wie folgt reagieren:

```
I = 1, X = 0.0, Y = 0.0, Z = 0.0
Yes ;
I = 2
Yes
```

Nach Eingeben weiterer konjunktiver Ziele kann eine Anfrage für besondere Zwecke verwendet werden, wie z.B. das Definieren von Beschränkungen (Bedingungen – *constraints*). So können im allgemeinen komplexere Fragen gestellt werden, mit denen die Möglichkeiten einer Abfragesprache eines Datenbankverwaltungssystems erreicht werden. Folgende Prolog–Anfrage spezifiziert (oder beschränkt) z.B. den Wert der X–Koordinate:

```
?- point(I,$,X,Y,Z), X=0.0.
I = 1, X = 0.0, Y = 0.0, Z = 0.0
Yes
```

Am Anfang dieses Abschnitts wurde die Regelform der Logik erster Stufe als

$$L \leftarrow R_1, R_2, \ldots, R_n$$

definiert, die ohne linke Seite einer logischen Anfrage entspricht. Mit anderen Worten ist die logische Spezifikation einer Datenbankfrage eine Art Regel ohne Ableitung. Als logische Konsequenz ergibt sich die Konstellation, daß Regeln selbst als Anfrageschnittstelle in diesem Sinne definiert werden, nicht etwa nur als Regeln zur Ableitung bestimmter Schlußfolgerungen. Unter Verwendung dieser Eigenschaft besteht die Möglichkeit, z.B. nicht nur nach den Attributen der Entities zu fragen, sondern Regeln zu definieren, die

die Semantik der Entities überprüfen können. Im Abschnitt 8 werden einige Ansätze formuliert, wie mit Hilfe logischer Prädikate die EXPRESS–Semantik der STEP–Entities analysiert werden kann (siehe dazu auch in [Toparlak 91c]).

Es ist zwar nicht zwingend erforderlich, daß in einem Programmsystem wie dem RWVS eine Analyse des Produktmodells (z.B. auf seine Korrektheit) durchgeführt wird, da diese zentral durchgeführt werden kann, jedoch sind anwendungspezifische Überprüfungen des Produktmodells etwa bei einer Modelltransformation oder der Berechnung neuer Attribute[5] während der Laufzeit der Applikation möglich. So können z.B. einige Prädikate zur Überprüfung der Modellsemantik der STEP–Entities nach ihrer EXPRESS–Definition mit Hilfe der logischen Variablen definiert werden, die dann nicht mehr als Existenzquantoren, sondern lediglich zur Übertragung der Werte zwischen den linken und rechten Teilen einer Regel dienen. Das folgende Prädikat zeigt dieses:

```
% is_a_point/5 -- checks the point semantics
%
is_a_point(I,CS,X,Y,Z) :-
      integer(I), uniq_entity_id(I),
      optional_coordinate_system(CS),
      real(X),
      real(Y),
      real(Z).
```

Dieses Prädikat kann dann für die Überprüfung der Semantik der `point` Entities aus dem Produktmodell folgendermaßen verwendet werden:

```
% -- find out each point and check its semantics
%
?- point(I,CS,X,Y,Z), is_a_point(I,CS,X,Y,Z).
I = 1, CS = $, X = 0.0, Y = 0.0, Z = 0.0;
I = 2, CS = $, X = 1.0, Y = 0.5, Z = 2.34
Yes
```

Dabei versucht der eingebaute 'Theorembeweiser' des Prolog–Interpreters zuerst in der Modellierungswelt Aussagen über das Prädikat $point/5$ zu finden; hierbei funktionieren die logischen Variablen als Existenzquantoren, sie werden bei erfolgreicher Suche mit den entsprechenden Werten 'unifiziert'. Nach diesem Vorgang versucht der Beweiser den in der Anfrage als konjunktives Ziel angegebenen Term $is\_a\_point/5$ zu erfüllen, während die logischen Variablen bereits initialisiert worden sind. Die Beweisführung läuft lediglich mit den Termen auf der rechten Seite des als Regel definierten Prädikats $is\_a\_point/5$ weiter. In diesem Falle wurden die logischen Variablen als 'universal' quantifiziert (oder All–Quantoren) funktioniert. Diese Regel kann verbal wie folgt ausgedrückt werden:

[5] sog. abgeleitete Attribute (*derived parameters*)

"Für 'alle' X, Y, Z und I, CS gilt: Ein gültiger kartesischer Punkt ist ein Punkt, dessen Koordinaten reelle Zahlen sind, der mit einer eindeutigen Identifikation in Form einer Integerzahl gekennzeichnet ist und einem lokalen oder globalen Koordinatensystem zugeordnet wurde."

Diese Regel funktioniert als eine Menge von Fakten, die sonst jedesmal definiert werden müßte. Sie beschreibt letztendlich 'alle' gültigen Punkte.

Schlußfolgerungen

Diese erstaunlich kleine, aber mächtige Menge von Möglichkeiten der logischen und zugleich wissensbasierten Programmierung mit Prolog, flexible Datenstrukturen (Terme), Parameter als Eingabe sowie als Ausgabe in testender oder generierender Weise (logische Variablen), Beschreibung eines Problems mit Fakten und Regeln und die Anwendung eines recht einfachen Beweismechanismus bilden einen relativ komfortablen Rahmen für die Realisierung des RWVS, der sich relativ überschaubar modifizieren läßt.

Weitere Einzelheiten der Programmiersprache Prolog, die in der vorliegenden Arbeit bei der Realisierung des RWVS als Implementationssprache eingesetzt wurde, können der Literatur [Clocksin/Mellish 84, Sterling/Shapiro 86, Bratko 87, O'Keefe 90] entnommen werden. Daher wird auf weitergehende Erläuterungen der Programmiersprache Prolog verzichtet.

Kapitel 7

Ein wissensbasiertes System

In diesem Kapitel wird ein wissensbasiertes System vorgestellt, das aufgrund der in den vorangegangenen Abschnitten formulierten Überlegungen bzw. Erkenntnisse entwickelt wurde. Die wissensbasierte Programmiermethodik hat dazu beigetragen, eine Prototypimplementation schon während der Entwurfsphase des vorliegenden Systems zu realisieren. Es ist durchaus in der Lage, als Gegenstand der weiteren Forschung zu dienen und im Laufe der Zeit mehr als ein Prototyp zu werden, so daß es als eine weitere wichtige Komponente innerhalb des schiffbaulichen Konstruktionsprozesses eingesetzt werden kann. Das Beispiel im Anhang B wird diesen Aspekt verdeutlichen, indem darin gezeigt wird, wie mit Hilfe dieses wissensbasierten Systems ein spezifischer Dimensionierungsprozessor programmiert werden kann, der dann als ein unabhängiges Programm aufgerufen werden kann.

Im Abschnitt 3.2 wurde an Hand eines Dimensionierungsbeispiels die erforderliche Information ansatzweise ermittelt, die aber in diesem sehr frühen Stadium nicht festgelegt werden kann. Es ist vielmehr versucht worden, die **allgemeingültigen** Aspekte der erforderlichen Information zu erkennen und damit eine Basis für die Formalisierung der Spezifikation der notwendigen Datenstrukturen zur Speicherung dieser Information zu erhalten. Die Informationsanalyse hat sich dort auf die **reinen** Daten beschränkt, die als Eingangsparameter bei der Auswertung der Dimensionierungsregeln dienen sollten. Da der wissensbasierte Programmieransatz versucht, Daten und Programme gleich zu behandeln, wurde eine entsprechende Analyse für die Regeln zur Dimensionierung im Kapitel 3 angestrebt, in der auch die Regeln als Daten betrachtet worden sind. Das hat den erheblichen Vorteil, daß beim Programmieren der Regeln kein Aufwand daran verschwendet wird, die Regeln in Prozeduren umzuwandeln, was in der konventionellen Programmierweise bis heute ein primäres Beispiel ist bzw. ein Problem darstellt. Programme als Daten zu betrachten, ist keine neue Technologie. Bisher wurde diese Technik jedoch nur in begrenzten Gebieten wie der Compilertechnik oder der theoretischen Informatik angewandt.

Im Kapitel 4 wurden allerdings z.T. basierend auf dieser Betrachtung Voraussetzungen für die Datenstrukturen, die Dimensionierungsregeln abbilden sollen, definiert. Da die Lösung eines Dimensionierungsproblems unter der Ausführung der entsprechenden Regeln erfolgen kann, wurden dort die notwendigen Mechanismen, die diese Ausführung steuern sollen, festgelegt. Gerade dieser Teil des vorliegenden Prototyps ähnelt dem prozeduralen Teil eines konventionellen Programms. Bei der Festlegung wurden die bereits existierenden Mechanismen (oder Methoden) der Informationstechnik (IT), speziell der Künstlichen Intelligenz, analysiert und bewertet.

Ebenso sind die Programmiersprachen, die in diesem interdiziplinären Zweig der Informatik gebraucht werden, darauf hin untersucht worden, inwieweit sie sich für die Realisierung der angestrebten Mechanismen eignen. Zwar ermöglichen die gebrauchsfertigen Werkzeuge der KI wie *expert system shells* relativ kurze Entwicklungszeiten, leiden aber unter dem Mangel an Möglichkeiten, sich den neuen Anwendungsfällen anzupassen oder besitzen oft festgelegte und damit begrenzte Methoden, die sich bei der Bewältigung neuartiger Probleme oft als unzureichend erwiesen.[1]

Nach diesen Überlegungen sieht die Auswahl der Programmiersprache Prolog als Realisierungswerkzeug für die Implementation und Pflege (gemeint sind die Möglichkeiten zur Weiterentwicklung) des vorliegenden System in seiner wissensbasierten Form entsprechend gut aus.

Es ist immer wieder zu beobachten, daß Programmsysteme relativ wenige Überlebenschancen haben, wenn sie keine offene Systemarchitektur besitzen [Koch 89]. Umgekehrt gilt dies auch für die Akzeptanz der Programmsysteme. Programme mit offenen Systemarchitekturen lassen sich mit erheblich geringerem Zeit– und Kostenaufwand einsetzen, da der heutige CAD–Sektor besonders im Schiffbau verhältnismäßig viele als Insellösungen eingesetzte Programmsysteme mit relativ ungünstigen Kommunikationsmöglichkeiten aufweist. Daher wurde im Kapitel 5 eine Systemarchitektur konzipiert, die diese Anforderung erfüllen soll. Es stellte sich anschließend heraus, daß diese Konzeption wegen des relativ günstigen Übergangs in die Implementierungsphase eine gewinnbringende Rolle gespielt und somit die Realisierung des Systems sogar zeitlich positiv beeinflußt hat.

Die vorgestellte offene Systemarchitektur verdankt ihre Herkunft den Arbeiten, die im Rahmen des AiF–Forschungsvorhabens [Toparlak 91a] gemacht worden sind. Es ging dort um eine Untersuchung, welche prinzipiellen Lösungsansätze dauerhaft geeignet sein können, den Datenaustausch zwischen CAD und FEA–Systemen auf einer relativ systemunabhängigen Basis zu realisieren. Aus diesen Überlegungen ist eine rechnerinterne Darstellung, ein sogenannter abstrakter Datentyp, entstanden, für dessen Handha-

[1] Dies ist eine selbstverständliche logische Konsequenz dieser Vorgehensweise. Denn die meisten *expert system shells* sind aus einem Expertensystem 'herausgerissen' worden, das für einen spezifischen Anwendungsfall entwickelt worden ist. Eine *expert system shell* — wie bereits in früheren Abschnitten erwähnt — ist die sogenannte *inference engine*, die die Problemlöse–Komponente des Expertensystems ausmacht.

bung schließlich das STEP–Parser–System (oder STEP Toolkit [Toparlak 91b]) entwickelt wurde. Dieses System spielt eine zentrale Rolle in der hier entwickelten offenen Systemarchitektur. Die Möglichkeit, beliebige Produktmodelle aus den systemunabhängigen Neutralformatdateien (STEP *physical files*) bearbeiten zu können, und das normgerechte Erstellen der STEP–Dateien aus einer durch eine spezielle Applikation [Grafe 93, Koch 91] erzeugten RID haben ebenfalls die Integration des im folgenden beschriebenen wissensbasierten Systems erheblich vereinfacht.

Für diese Integration war lediglich die Implementation einer Schnittstelle zwischen dem STEP–Parser–System und Prolog erforderlich, auf die im Kapitel 8.1 eingegangen wird. Diese Schnittstelle beruht auf zwei getrennten Schnittstellen. Die eine besteht aus einer Verbindung, die durch den File–Transfer realisiert wird. Es handelt sich hierbei um eine Umwandlung der STEP–Dateien in ein Datenformat, das durch einen Prolog–Interpreter relativ einfach gelesen und entsprechend in einer Prolog–eigenen Datenbank gespeichert werden kann. Der zweite Teil der Schnittstelle basiert auf der Implementation einer allgemeinen Prolog/C–Schnittstelle. Speziell hierfür wurde die Installation der Prolog/C–Schnittstelle eines am Arbeitsbereich im Rahmen der vorliegenden Arbeit eingesetzten, kommerziell vertriebenen Prolog–Interpreters bzw. Compilers des belgischen Forschungsinstituts für Management ProLog *by* BIM [BIM 93] übernommen. Die Verfügbarkeit einer bilateralen Verbindung bei dieser Schnittstelle ermöglichte den Aufruf der externen Routinen aus der Prolog–Umgebung heraus und die Auswertung der Prolog–Prädikate durch das STEP–Parser–System (über den umgekehrten Weg, d.h. C/Prolog–Verbindung).

Dabei erprobt ist jedoch lediglich der Modellaustausch der Produktdaten zwischen STEP Toolkit und Regelwerkverwaltungssystem (RWVS) durch den File–Transfer (in Form von STEP–konformen Neutralformatdateien) oder durch den Aufruf der entsprechenden Routinen aus der Funktionsbibliothek des STEP–Parser–Systems auf die im Hauptspeicher abgebildete RID der Produktdaten. Diese Art der Datenhandhabung bewahrt auch die Offenheit der gesamten Systemarchitektur, wobei die zentrale Datenverwaltung mit den einzelnen Applikationen nicht unbedingt etwas zu tun haben muß. Eine Überlagerung der neuartigen Funktionalitäten wie Aufruf der Dimensionierungsregeln von Seiten des Datenverwaltungssystems würde die Allgemeingültigkeit eines so wichtigen, zentralen Elements wesentlich beschränken.

7.1 Systemaufbau und seine Komponenten

Der Systemaufbau des RWVS beruht auf der Grundlage der modular aufgebauten Prolog–Prädikate, die nur noch aus logischen Klauseln bestehen. Die Implementierung des Regelwerkverwaltungssystems basiert hauptsächlich auf den Prädikattypen, die dem 'natürlichen' Auswertungskonzept der Vorschriften möglichst eins-zu-eins entsprechen. Das Konzept ähnelt der Vorgehensweise eines Konstrukteurs, der während der Pro-

blemlösung nicht unbedingt einen statischen Weg zu den richtigen Regeln (und für Auswertung und Bewertung der Ergebnisse) kennen muß, wenn er früher mit einer ähnlichen Dimensionierung konfrontiert wurde. Er sammelt zuerst alle relevanten Regeln, wählt einige Auswahlkriterien zur Auswertung, sammelt Informationen, die bei der Auswertung benötigt werden, und muß ggf. für diese Informationen weitere Regeln auswählen, diese auswerten und deren Ergebnisse bewerten, wenn diese Informationen z.B. im CAD–System nicht vorliegen oder mit Hilfe anderer Programme nicht berechnet werden können. Er kann umgekehrt mit der Auswertung anfangen, ohne alle notwendigen Daten ermittelt zu haben. Er kann die Auswertung ruhen lassen und sich mit dem Ermitteln der Informationen beschäftigen. Anschließend führt er die Auswertung fort. Die Bewertungsphase ist weniger mit dieser Freiheit der Vorgehensweise verbunden. Dieser Vorgang muß nicht unbedingt abgebrochen werden, da die für die Bewertung erforderlichen Informationen schließlich Ergebnisse des Auswertungsvorgangs sind und meistens vollständig vorliegen. Die Bewertung der Ergebnisse ist wiederum dem Auswertungsvorgang ähnlich, wenn Bewertungskriterien dafür vorgegeben sind. Diese Phase ist daher weniger problematisch.

Um diese Vorgehensweise möglichst exakt zu simulieren, wurden im Regelwerkverwaltungssystem entsprechende Prädikate implementiert, die den einzelnen Schritten eines Konstrukteurs bei der Lösung eines Dimensionierungsproblems ähneln. Hilfsprädikate spielen dabei die Rolle der Werkzeuge wie Ingenieurwissen oder Taschenrechner. Hauptglieder dieser Systemprädikate sind:

rule/4 ist ein Prädikat, mit dem die meisten Regeln beschrieben werden. Es kann in jeder Richtung verwendet werden, das heißt, das Prädikat kann als Zugang zu den mit ihm implementierten Vorschriften als Abfrageschnittstelle oder als Lösungskomponente eines Dimensionierungsvorgangs verwendet werden. Nicht vollständig vorhandene Parameter führen dazu, daß mögliche Alternativen für diese Parameter gezeigt werden. Das Prädikat $rule/4$ hat die Form

$$rule(Category, Goal, Input, Output),$$

bei der eine sehr flexible Gestaltung der einzelnen Subterme (Parameter) erreicht werden kann. Jeder dieser Subterme kann in der Pluralform eingegeben werden. Das heißt, die Zielspezifikation $Goal$ kann in Form einer Liste — hier als Listenkonstrukt der Prolog — wie $[Goal_1, Goal_2, \ldots]$ eingegeben werden. So wird beschrieben, daß das Prädikat $rule/4$ alle vorgegebenen Zielangaben zu erreichen (auswerten/lösen) versuchen soll. Gegebenenfalls wird der Subterm in eine Liste umgewandelt, wenn mehrere Ergebnisse aus der mit dem Prädikat $rule/4$ definierten Vorschrift zu erzielen sind oder erhalten werden. Da der Parameter $Output$ vor dem Aufruf eine logische Variable sein darf, kann er jede beliebige Form annehmen, wie etwa Zahlen, Strings, Terme (Bäume, Netze) oder eine Liste. Eine Liste ist eine Spezialform eines logischen Terms, ein sogenannter binärer Baum, der nur noch nach rechts aufwächst. Die kurze Schreibweise für eine Liste $[subt_1, subt_2, \ldots]$ ist in der Tat ein binärer Baum, der wie folgt aussieht:

$.(subt_1, .(subt_2, .(\ldots)))$.

Da der Subterm $Output$ eine logische Variable ist, kann er gleichzeitig als Eingabe ($Input$) funktionieren. Der Verzicht auf den Subterm $Input$ ergibt dann ein neues Prädikat $rule/3$, das wie folgt aussieht:

$rule(Category, Goal, Input/Output)$.

Der dritte Subterm $Input/Output$ hat dann zwei Funktionen: Zum ersten kann er als unifiziert spezifiziert werden, womit ein Testeffekt (Überprüfen) erzielt wird. Gleichzeitig versucht das System herauszufinden, ob diese Eingabe die Bedingungen der Regel erfüllen kann. Zum zweiten kann der Subterm $Input/Output$ als eine nicht unifizierte logische Variable, welche eigentlich eine typenlose Hülle ist, die jedes beliebige Objekt enthalten kann, verwendet werden. Das System versucht, alle möglichen Lösungen zu finden, und liefert seine Antworten mit Hilfe dieser logischen Variable, die dann mit den Werten der Ergebnisse unifiziert wird, zurück. Mit anderen Worten: Das Ergebnis der Auswertung wird auf diese Variable zugewiesen, wie dies der Fall in prozeduralen Sprachen ist. Dieser Vorgang unterscheidet sich von den prozeduralen Sprachen, bei denen ein Scheitern zum Abbruch führen kann, wenn dieser nicht explizit abgefangen bzw. behandelt wird. Im Vergleich dazu werden weitere Klauseln des Prädikats $rule$ im Falle des Scheiterns automatisch ausgewertet, was bei der Verwendung von konventionellen Programmiersprachen extra zu implementieren wäre. Beim Scheitern einer Regel, die mit dem Prädikat spezifiziert ist, bleibt die logische Variable frei[2], so daß diese mit Lösungen anderer Klauseln vorgesehen wird. Mehrere Lösungen sind daher möglich.

Der Subterm $Category$ bezieht sich auf die Quelle der Vorschrift. Diese kann z.B. die Bezeichnung der entsprechenden Kapitel, Abschnitte oder Paragraphen sein. Die Bezeichnung ist frei von irgendwelchen Beschränkungen wie Format, Inhalt etc. So kann sie zum Beispiel die Überschrift eines Abschnitts oder dessen Kennzeichen sein. Da die Behandlung der Bezeichnung in der Regel dem Prädikat, in dem sie spezifiziert wird, überlassen ist, kann diese die Freiheit nutzen und ermöglicht ein großes Anpassungsvermögen und eine leichte Pflege. Wie im folgenden gezeigt wird, ist die Bezeichnung einer Vorschrift nicht der einzige Zugang zu ihr. Dieser Subterm dient nicht nur zur Bezeichnung der Quelle, aus der die Regel stammt, sondern auch dazu, Regeln, welche z.B. nicht unbedingt auf eine konkrete Vorschrift zurückzuführen sind, beliebig in Kategorien einzuordnen. So können technische Formeln, die nicht in der Bauvorschrift enthalten sind, als eine Regel in die Regelbasis eingebracht werden. Mehrere Kategorien von Regeln sind damit verfügbar und können programmiertechnisch realisiert werden.

Das Weglassen des Subterms $Category$ innerhalb eines $rule/3$ Prädikats ergibt dann ein neues Prädikat $rule/2$ mit der Form

[2] $free$, nicht unifiziert

$rule(Goal, Input/Output)$.

Aus dieser ergeben sich zwei Bedeutungen bei der Verwendung der Regeln, die mit diesem Prädikat spezifiziert werden: Zum einen kann das Prädikat eine lokale Verwendung finden, deren Bedeutung zum Teil allgemeingültig erscheint. Zum anderen kann es globale Bedeutung haben, was eine allgemeingültige Verwendung ermöglicht. Die erste Form ist lediglich eine kürzere Schreibweise für Regeln, die implizit einer Kategorie angehören. Diese Betrachtungsweise gilt aber nicht für die zweite Form, da eine Kategorie eine Unterteilung aufweist, wenn man z.B. sagt, der Subterm $Category$ enthält etwa die Bezeichnung '$global$', wenn er nicht angegeben ist.

Der Subterm $Goal$ ist unter den Subtermen des allgemeinen Prädikats $rule/n$ $(2 < n < 4)$ der wichtigste. Er beschreibt die Spezifikation des Ziels, was einer Problembeschreibung vergleichbar formuliert ist. Dieser Subterm spielt daher bei der Formgebung der Benutzerschnittstelle der Regeln eine besonders wichtige Rolle, da ein Anwender (z.B. Konstrukteur) zuerst mit der Problembeschreibung konfrontiert ist, etwa mit der 'Beplattung des Querschottes QS-03'. Trotz dieser besonderen Wichtigkeit bleibt die Möglichkeit erhalten, der Spezifikation des Subterms $Goal$ eine beliebig flexible Form zu geben, die dann die Gestalt der Schnittstelle bestimmt. Es wird beispielhaft gezeigt, wie der Subterm $Goal$ nach den Überschriften der Klassevorschriften oder nach den Bezeichnungen der zu erwartenden Ergebnisse spezifiziert wird.

Da die Realisierung eines Dimensionierungsprozessors von der Erstellung eines Regelnetzes, das aus einer Menge von Klauseln der Prädikate $rule/4$, $rule/3$ und $rule/2$ besteht, abhängig ist, wird ein Entwickler (oder Programmierer) zwangsweise für die Konsistenz der Regeln sorgen müssen, da sowohl bei den aufrufenden als auch bei den aufzurufenden Klauseln der Regel die Spezifikationen der Subterme wie $Category$ und $Goal$ übereinstimmen müssen. Der Aufbau der Regeln in einem Dimensionierungsprozessor ähnelt dem Konzept der semantischen Netze (*semantic networks*). Um ein semantisches Netz zu vervollständigen, fügt man in der Regel Zwischenknoten ein, die in dem vorliegenden Fall als Hilfsprädikate angesehen werden können. Zum Beispiel ist das Prädikat $rule/2$ im folgenden Programmsegment ein solcher Zwischenknoten (eine Art Abkürzung) innerhalb des Regelnetzes und bildet zugleich eine simple Schnittstelle zu seinem Partner, dem Prädikat $rule/4$.

```
% rule/2 -- module name: [] (global)
rule(Schottbeplattung,_t_best) :-
      rule([11,B,2],Schottbeplattung,Plattendicke,_t_best).
      % -- calls rule/4


% rule/4 -- module name: [11, B.2]
rule([11,B,2],Schottbeplattung,Plattendicke,_t_max) :-
      solve(_t,    c_p*a*sqrt(p)+t_K),
```

```
        solve(_t_min,6.0*sqrt(f)),
        ...
        max([_t,_t_min],_t_max).
```

Dabei wurden die beiden Prädikate jeweils mit einer Klausel spezifiziert. Eine Gestaltung des Prädikats $rule/4$ mit mehreren Klauseln ist gegebenenfalls möglich, wenn weitere Alternativen zu berücksichtigen sind. Dies würde dann wie folgt aussehen:

```
% rule/4 -- module name: [11, B.2]
rule([11,B,2],Schottbeplattung,Plattendicke,_t_best) :-
        rule([11,B,2,1],Schottbeplattung,t,_t1),
        rule([11,B,2,1],Schottbeplattung,t_min,_t2),
        rule([11,B,2,2],Schottbeplattung,Plattendicke,_t3),
        var(_t3) ->
                max([_t1,_t2],_t_best)
        ;
                _t_best = _t3.


% rule/4 -- module name: [11, B.2.1]
rule([11,B,2,1],Schottbeplattung,t,_t) :-
        solve(_t = c_p*a*sqrt(p)+t_K).

% rule/4 -- module name: [11, B.2.1] (second clause)
rule([11,B,2,1],Schottbeplattung,t_min,_t_min) :-
        solve(_t_min = 6.00*sqrt(f)).


% rule/4 -- module name: [11, B.2.2]
rule([11,B,2,2],Schottbeplattung,Plattendicke,_t) :-
        ask('kleines Schiff'),
        ask('Spantabstand entsprechender Steifenabstand'),
        ask('Plattendicke der Außenhaut',_t).
```

So entsteht ein Regelnetz, das immer wieder erweitert bzw. vervollständigt werden kann. Die Abbildung 7.1 vermittelt eine Vorstellung von einem solchen Netz. Der Aufbau dieser Struktur, die während ihrer Entstehung aufwächst und im Gegensatz zu den Bäumen in keiner Richtung behindert ist, erlaubt eine flexiblere Gestaltung der Regelbasis. Für den Entwickler (gemeint ist der Programmierer) eines Dimensionierungsprozessors besteht die Möglichkeit, sich entsprechend abwechselnd mit den unteren oder oberen Konzeptionsschichten der jeweiligen Implementation eines solchen Programms zu beschäftigen. Während der Entwicklung eines Programms ist er nicht wie bei der herkömmlichen Programmierung auf eine konkrete Richtung der Fortführung, z.B. *bottom-up* oder *top-down*, beschränkt.

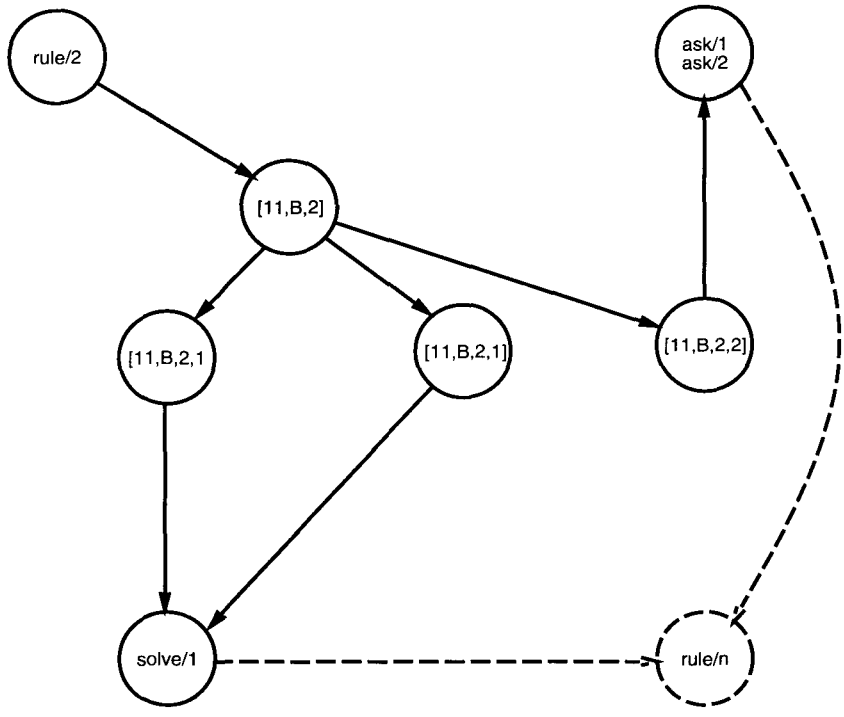


Abb. 7.1: *Teil eines Regelnetzes als gerichteter Graph*

ask/2 ist ein sehr allgemeines Prädikat, das innerhalb des Regelwerkverwaltungssystems eine wichtige Rolle spielt, da es eine recht einfache und flexible Schnittstelle für die Datenabfrage bildet. In seiner derzeitigen Implementation hat das Prädikat $ask/2$ folgende Form:

$$ask(Query, Answer).$$

Dabei sind die Subterme $Query$ und $Answer$ als logische Variablen konzipiert, wobei es sich um ein Frage/Antwort–Paar handelt, mit dem alle relevanten Fragetypen ohne Beschränkung auf den Datentyp der Antwort gestellt werden können. Wie in den vorherigen Beispielen gezeigt, handelt es sich bei dem Subterm $Query$ um den Datentyp $atom$, der in Form eines einfachen Symbols zur Beschreibung selbst komplizierter Fragen dienen kann. Ganz generell besteht die Möglichkeit, innerhalb dieses Symbols höhere Datenabfrageformen unterzubringen, etwa mit strukturierten Abfragesprachen oder natürlicher Sprache. Da der Subterm $Query$ als logische Variable andere Formen aufnehmen kann, ist es möglich, die Abfrageschnittstelle entsprechend den Anforderungen der jeweiligen Anwendungen zu erweitern. Ganz allgemein gilt bei dieser Konzeption, daß eine Anfrage an Hand ihrer Form ausgewertet und eine entsprechende Klausel des Prädikats $ask/2$ ausgeführt wird.

Das Prädikat $ask/2$ wurde ebenfalls in Anlehnung an die Vorgehensweise eines Konstrukteurs, wie dies schon bei der Konzeption des Prädikats $rule/4$ auch der Fall war, konzipiert. Das heißt, an Hand der Anfrage wird zuerst nach Regeln gesucht, die in der Lage sind, diese Frage zu beantworten. Diese Suche basiert auf einem sehr einfachen Prinzip: Es wird nach den Regeln gesucht, deren Subterm $Goal$ mit der Anfrage (dem Subterm $Query$) übereinstimmt. Die Übereinstimmung der beiden Terme braucht nicht exakt zu sein, da mit Hilfe von Tabellen (einer Art Lexikon) der Zugang zu der richtigen Regel gesteuert werden kann.

Wenn keine Regel zur Beantwortung der gestellten Anfrage gefunden werden kann, wird diese Anfrage an die Datenbank für Produktdaten weitergeleitet und nach den entsprechenden Daten aus dem Produktmodell gesucht. Die Suche erfolgt wiederum an Hand des Subterms $Query$, der in diesem Falle mit dem entsprechenden Attribut einer Entity übereinstimmen muß. Zur Zeit ist das Attribut, das aus der Menge aller Entities des Produktmodells gesucht wird, auf den Typ der Entity beschränkt. Das bedeutet, wenn eine Entity in der Datenbank mit Hilfe eines logischen Terms definiert ist, besteht die Antwort aus den Subtermen oder Parametern dieser Entity, deren Typ der Funktor des logischen Terms ist. Da die Antwort alle Attribute dieser Entity enthält, kann eine Spezialisierung der Anfrage durch das Auswählen bestimmter Attribute (oder durch die Vorgabe der Beschränkungen (*constraints*) von Werten) realisiert werden. Folgende Anfragen an die Prolog–Datenbank verdeutlichen dies beispielhaft:

```
...
Schiffstyp(Tanker).
point(#2, $, 1.0, 0.5, 2.34).
...
?- ask(Schiffstyp, _a).
_a = Tanker
Yes

?- ask(point, _a).
_a = [#2, $, 1.0, 0.5, 2.34]
Yes
?- ask(point, _a), _a = [# 2, _, X, _, _].
X = 1.0
Yes
```

Die zweite Anfrage zeigt die Vorgehensweise bei der Spezialisierung der Attribute. Wie zu erkennen ist, wird die Anwendung, d.h. die aufrufende Seite des Prädikats $ask/2$, mit der Struktur und deren Verarbeitung des Rückgabedatentyps konfrontiert, aber nicht das Prädikat $ask/2$ selbst. Bei der zweiten Anfrage ist die Antwort eine Liste von Attributen der Entity `point`, wogegen die erste einen einfachen Datentyp, ein sogenanntes $atom$, zurückliefert. Dies zeigt die Unabhängigkeit des ask Prädikats von der jeweiligen Applikation, die für Frage und Antwort (bzw. deren Behandlung) selber verantwortlich

ist. Die anwendungsunabhängige bzw. neutrale Gestaltung der Systemkomponenten erleichtert notwendige Veränderungen des Regelwerkverwaltungssystems bzw. die Pflege der realisierten Dimensionierungsprozessoren. Dieses Beispiel zeigt die praktischen Möglichkeiten der zweckmäßigen (*problemorientierten*) Programmierung auf der Basis von wissensbasierten Techniken. Prolog erfüllt in den ersten Ansätzen die Voraussetzungen in dieser Hinsicht.

Das Prädikat $ask/2$ versucht als letztes, durch einen Dialog mit dem Benutzer die Frage zu beantworten, wenn die Suche nach einem verwertbaren Ergebnis in der Datenbank bzw. vorher in der Regelbasis scheitert. Der Benutzer hat während des interaktiven Dialoges drei Möglichkeiten, die an ihn weitergeleitete Frage zu beantworten:

- durch direktes Antworten, d.h. Eingeben von Attributwerten z.B. Zahlen, Zeichenketten, Symbole,
- durch Definition einer neuen Regel, die zuerst in die Regelbasis eingefügt und dann zur Beantwortung der Frage herangezogen wird oder
- durch Einfügen einer Entity–Instanz in die Produktmodell–Datenbank.

Für den letzten Fall, das Einfügen einer neuen Entity, wird die Antwort dahingehend überprüft, ob sie grundsätzlich zur Beantwortung der Frage herangezogen werden kann, bevor sie in die Datenbank eingefügt wird. Sonst wird der Dialog erneut durchgeführt. Bei der zweiten Alternative sieht es etwas anders aus: Auch hier wird die Definition der neuen Regel auf eine entsprechende Antwort hin überprüft, aber ein verwertbares Ergebnis kann nur nach der Ausführung dieser Regel abgeleitet werden. Zu diesem Zweck muß die eingegebene Regel in die Regelbasis eingeführt werden, was einen weiteren Effekt zur Folge hat: Nach dem Einfügen der neuen Regel in die Regelbasis ändert sich die Struktur derselben so, daß ein neues Verhaltensmuster des Regelnetzes zu erkennen ist. So können z.B. Regeln, die bis dahin bei ihrer Ausführung gescheitert sind, neu ausgewertet werden, wenn sie zufällig auf die neu hinzugefügten Regel Bezug nehmen. Dieser Dialog hat somit eine ergänzende Funktion für den gesamten Dimensionierungsprozessor. Diese Funktion ist daher von besonderer Bedeutung.[3]

Eine Vereinfachung des Prädikats $ask/2$ liegt mit dem Prädikat $ask/1$ in der Form

$$ask(Query/Answer)$$

vor. Dieses Prädikat besitzt einen einzigen Subterm, der das Frage/Antwort–Paar auf eine *selbstbeschreibende* (*self-contained*) Weise spezifiziert. Es unterstützt Fragen, die nur mit 'Ja' oder 'Nein' beantwortet werden können. Ähnlich wie bei dem Prädikat $ask/2$,

[3]Das ist der Grund dafür, warum nicht nur der Programmierer, sondern auch der Benutzer (z.B. Konstrukteur) in der Lage sein muß, neue Regeln einzugeben.

wird zuerst die Regelbasis konsultiert und nach Regeln gesucht, deren Auswertung die logischen Werte $true$ oder $fail$ zurückliefert. Im Falle des Scheiterns wird ebenfalls in der Produktmodell–Datenbank nach der Entity gesucht, die dem Subterm in der Anfrage entspricht. Beispielsweise hätte man in dem vorherigen Programmsegment die Anfrage über den Schiffstyp mit dem Prädikat $ask/1$ wie folgt formulieren können:

```
?- ask(Schiffstyp(Tanker)).
@ Schiffstyp(Tanker) [y(es)/n(o)] ?
@ y
Yes
?- ask(Schiffstyp(_t)).
_t = Tanker
Yes
```

Das Prädikat ask/n wird in der Abbildung 7.2 als ein Netz mit gerichteten Graphen dargestellt.

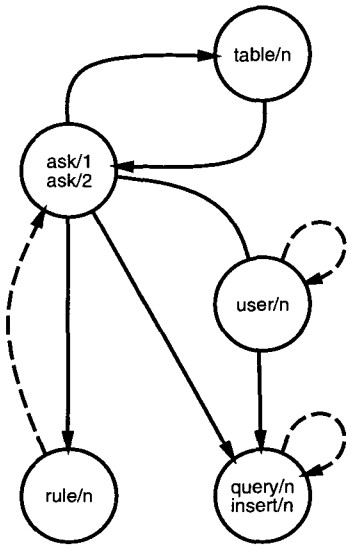


Abb. 7.2: *Netzstruktur des Prädikats* ask/n

Das ***table/n*** als Hilfsprädikat spielt dabei im allgemeinen die Rolle eines Lexikons, um Anfragen (falls möglich) eine eindeutige Bedeutung zu verleihen, um somit eine klare Begriffswelt herzustellen, da nur bei Übereinstimmung zwischen den Systemkomponenten (Regelbasis, Produktmodell–Datenbank, Hilfsprädikate und Lexikon [*data dictionary*]) die beabsichtigte Kommunikation[4] entstehen kann [Hansen/Mühlbacher/Neumann 93].

solve/1 ist ein Prädikat zur Lösung mathematischer Gleichungen, wie sie z.B. in den Klassifikationsvorschriften des Germanischen Lloyd vorkommen. Das Argument des

[4] Datenfluß, Signalaustausch usw.

Prädikats ist ein einziger Subterm, der die Gleichung (oder die Formel) in einer Form beschreibt, wie sie in der Vorschrift vorgegeben ist. Die derzeitige Implementation des Prädikats $solve/1$ ist von prototypischer Natur. Einige Besonderheiten, die in den Formeln des GL–Regelwerkes vorkommen, sind bei der Auswertung der Formeln oder zur Lösung der algebraischen Gleichungen in diesen Formeln in dem entsprechenden Sinne simuliert.

Das Prädikat besteht aus einer Menge von gleich aussehenden logischen Klauseln. Jede einzelne von ihnen wird danach angesprochen, wie die Formel formuliert ist. Bestimmte Muster in den Gleichungen bewirken die Auswahl derjenigen Klausel, die sich für die Lösung der jeweiligen Gleichung am besten eignet. Da die Möglichkeit besteht, neue Klauseln mit verschiedenen Gleichungsmustern in die Menge einzufügen, kann man jederzeit das Prädikat $solve/1$ so erweitern, daß neue Gleichungstypen in den zu programmierenden Vorschriften unterstützt werden. Diese Vorgehensweise erlaubt die gleitende Umsetzung einer Vorschrift in ein Programm, wobei eine Transparenz zwischen den Formeln auf der Codierungsebene und im Text der Vorschrift entsteht. Eine solche Transparenz zwischen der Vorschrift und dem Programm ist aus Wartungsgründen von großem Nutzen.

Auch bei dem Entwurf dieses Prädikats hat die Vorgehensweise eines Konstrukteurs eine große Rolle gespielt. Bei der Auswertung einer algebraischen Formel unterbricht der Konstrukteur z.B. die Berechnung, solange einige Parameter noch unbekannt sind, die in den Formeln auftauchen und deren Ermittlung etwa die Ausführung anderer Vorschriften sowie Interaktion mit dem CAD–System und ähnliches voraussetzt, solange, bis er mit der Berechnung fortfahren kann. Oft braucht er einen Katalog, in dem mathematische Symbole mit ihren Bezeichnungen untergebracht sind. Beispielsweise werden Symbole dieser Art in einem Abschnitt 'Begriffsbestimmungen' abgehandelt, wenn man mit den Bauvorschriften des Germanischen Lloyd arbeitet.

Mit Hilfe eines solchen Kataloges kann sich der Konstrukteur ein Bild über die mögliche Bedeutung der entsprechenden Begriffe machen und erst dann zu den Informationen gelangen, die letztendlich benötigt werden. So ähnlich verhält sich auch das Prädikat $solve/1$ für die Auswertung der Formeln: Es unterbricht die Auswertung einer Formel, wenn innerhalb derer ein Symbol vorkommt, das nicht direkt arithmetisch ausgewertet werden kann und sucht in der Regelbasis nach einer Regel, die für dieses Symbol eine Antwort liefert, oder in der Produktmodell–Datenbank nach einer Entity, die diesem Symbol entspricht. Es konsultiert das Lexikon (*data dictionary*) und, falls vorhanden, einen Katalog für Begriffsbestimmungen, welcher als eine Regel (mit Hilfe des Prädikats $rule/n$) implementiert worden ist. Wenn ein arithmetisch verwendbarer Wert (Ausprägung, Instanz) für das betrachtete Symbol auf diese Weise ermittelt worden ist, wird mit der Auswertung der Formel fortgefahren.

Das Prädikat ruft sich jedesmal rekursiv, wenn nach der Ermittlung der Datenwerte einzelner Symbole eine symbolische Umsetzung der Formel in dem logischen Subterm erfolgt ist. Dem rekursiven Aufruf wird der auf diese Weise manipulierte Subterm als Eingabeparameter übergeben. Eine Rückkehr findet statt, wenn keine unbekannten

Symbole mehr vorhanden sind und eine entsprechende arithmetische Auswertung des Subterms durchgeführt werden kann.

Da die Ermittlung der Datenwerte dieser Symbole durch eine Schnittstelle zu dem oben vorgestellten Prädikat $ask/2$ erfolgt, wird ein automatischer Dialog mit dem Benutzer durchgeführt, wenn keine Instanz für das jeweilige Symbol aus der Datenbank oder Regelbasis gefunden wird. Der Konstrukteur braucht analog dazu einen Dialog mit Kollegen, falls er die Antwort nicht kennt. Selbst wenn er die Antwort kennen würde, kann dies als ein Dialog (mit ihm selbst) betrachtet werden, dessen er sich oft nicht bewußt ist.

Das Prädikat $solve/1$ ist ein Beispiel für Meta–Programmierung, da die Interpretation des arithmetischen Ausdrucks nicht mehr nach dem Auswertungsmechanismus der darunterliegenden Sprache, hier Prolog, sondern den Erfordernissen der Applikation entsprechend erfolgt. Eine letztendliche Berechnung läuft zwar durch den Prolog–Interpreter, aber die Interpretationsschicht wird problemorientiert umgeformt, ohne dafür einen neuen Interpreter (für mathematische Ausdrücke) zu implementieren.

Allen vorhandenen Operatoren, die in der darunterliegenden Sprache verfügbar sind, wurde eine spezielle Semantik verliehen, um mit den neuen Aufgabenstellungen fertig zu werden. Zum Beispiel werden in GL–Vorschriften Winkelwerte innerhalb trigonometrischer Funktionen im Gradmaß angegeben, wogegen trigonometrische Funktionen in den meisten Programmiersprachen Winkelwerte im Bogenmaß erwarten. Diese Situation zwingt jeden Programmierer, erforderliche Umsetzungen von Grad zu Radiant[5] durchzuführen, was einen erhöhten Implementationsaufwand verursacht. Da das $solve/1$ den Subterm, den es als Eingabeparameter bekommt, so teilt, daß jedes einzelne Objekt an sich analysiert werden kann, war es nicht sehr kompliziert, die Parameter, die an die trigonometrischen Funktionen weitergegeben werden, zu untersuchen und evtl. umzusetzen.

In der derzeitigen Version wandelt das Prädikat $solve/1$ Integerwerte ins Bogenmaß um, da es sich sehr wahrscheinlich um Winkelwerte in Grad handelt, wenn mit GL–Vorschriften gearbeitet wird. Sonstige Werte (Gleitkomma–Zahlen, arithmetische Ausdrücke) werden als Radiant zur endgültigen Auswertung an die Funktion übergeben.

Diese Eigenschaften verleihen dem Einsatz eines solchen Gleichungslösers eine besondere Bedeutung, da der Entwickler oder Programmierer eines Dimensionierungsprozessors sich nicht mehr mit den Einzelheiten der arithmetischen Ausdrücke in Formeln beschäftigen muß und sich somit mehr auf das Wesentliche konzentrieren kann. Eine weitere wichtige Eigenschaft dieses Gleichungslösers ist die Möglichkeit, Formeln nicht nur zur Berechnung, sondern auch zur Kontrolle der evtl. vorhandenen oder zuvor ermittelten Werte anzuwenden. Dies öffnet den Weg, vorhandene Konstruktionen an Hand der Bauvorschriften mit dem gleichen Dimensionierungsprozessor zu überprüfen, der auch zur Bemessung der Konstruktion verwendet wird.

[5] bei der Parameterübergabe an die jeweilige Funktion

Der folgende Prolog–Dialog veranschaulicht die Verwendung des Prädikats $solve/1$ in der vorher aufgeführten Hinsicht:

```
rule(Material,Materialfaktor,_k) :-
      solve(_k = 295.0/(R_eH+60)).
rule(Material,Fließgrenze,_r_eh) :-
      ask(St,_st),
      R_eH(_st,_r_eh).


?- solve(_t >= c_p*a*sqrt(h*k)+t_K).
Bemessungsfaktor/c_p? "...".
@ 1.2.
Steifenabstand/a? "...".
@ 1.0.


Druckhöhe/h? "...".
@ 1.0.
Stahlsorte/St? [normalfest,hfest27,hfest32,hfest36]
@ normalfest
Abrostungszuschlag/t_K "...".
@ 1.1.
_t = 2.39
Yes
```

Nach diesem Dialog befinden sich in der Produktdatenbank die durch den Benutzer gelieferten Informationen, welche wie die Produktdaten gespeichert werden. Die Prolog–interne Datenbank, die im allgemeinen zur Speicherung von Fakten und Regeln dient, enthält nach dem obigen Dialog folgende Fakten:

```
Bemessungsfaktor(#1001, 1.2).
Steifenabstand(#1002, 1.0).
...
Druckhöhe(#1003, 1.0).
Stahlsorte(#1004, normalfest).
```

Die so gewonnene Information wird bei den folgenden Konsultationen nicht mehr durch den Dialog mit dem Benutzer abgefragt; sie wird bei der Auswertung der weiteren Regeln aus der Datenbank gelesen, wie etwa:

```
?- solve(_t  = 1.1*a*sqrt(p*k)+t_K).
Entwurfsdruck/p? "...".
@ ...
_t = ...
Yes
```

query/2, insert/2 sind Datenbankprädikate für die Verwaltung der Produktdaten. Die Abspeicherung der Produktdaten ist allgemeingültig konzipiert, bei der jeweils die für die Verarbeitung der in dieser Form abgelegten Information abfragende bzw. einfügende Anwendung verantwortlich ist. Die Datenbank stellt daher eine Menge von Fakten dar, die die folgende Form besitzen:

$$Id : Attribute(Value).$$

Diese Form ähnelt der Wissensrepräsentation mit Attribut/Wert-Paaren bei eindeutiger Trennung der Identifikation (id) der Objekte einer sogenannten Begriffswelt (UoD – *Universe of Discourse*). Diese Form zur Wissensrepräsentation reicht für die Darstellung der erforderlichen Informationen im prototypischen Dimensionierungsprozessor. Da man unter dem Subterm $Value$ jeden beliebigen Wert darstellen kann, ist die Datenbeschreibung für Objekte komplexeren Typs keine große Schwierigkeit. Die einfachsten Instanzen des Subterms $Value$ sind die booleschen Datentypen oder Wahrheitswerte ($true$ oder $fail$), die die Form der Fakten in der Datenbank wie folgt vereinfachen:

$$Id : Attribute(true) \rightarrow Id : Attribute$$
$$Id : Attribute(fail) \rightarrow \emptyset$$

Die Abbildung der negativen Information wird, wie oben angegeben, einfach ignoriert, da sie meistens nicht erforderlich und nur unter bestimmten Annahmen herleitbar ist (*negation as failure*). Das heißt, ein Sachverhalt wird als 'falsch' beurteilt werden, wenn er nicht nachzuweisen ist.

Da diese beiden Prädikate nicht für den direkten Einsatz in der Implementierung der Dimensionierungsprozessoren gedacht sind, gehören sie nicht in die Anwendungsschnittstelle des RWVS. Sie werden automatisch über das Prädikat ask/n aufgerufen. Dies geschieht auch mit dem Prädikat $solve/1$. Programmierschnittstellen klein zu halten, ist besonders wichtig, wenn Regelwerke größer werden. Detailinformationen über diese Datenbankprädikate sind für die Pflege und Weiterentwicklung des RWVS selbst von Interesse. Sie sind sozusagen Mitglieder des Systemkerns.

Wenn das Produktmodell durch die Verwendung des Prädikats $insert/n$ erstellt wird, kann eine eindeutige Identifizierung der in der Datenbank verwalteten Objekte gewährleistet werden. Nur eine direkte Übernahme des Produktmodells aus externen Quellen kann diese Integritätsbedingung verletzen, wobei jedoch auf eine entsprechende Überprüfung verzichtet werden kann. Außerdem muß eine solche Überprüfung nicht unbedingt die Aufgabe des RWVS sein. Die Verwendung eines zentralen Systems (Beispiel: STEP–Parser–System) kann für solche grundlegenden Problemstellungen eine Lösung liefern. Das RWVS verfügt über entsprechende Prädikate ($dbload/1$, $dbsave/1$) für den **direkten** Import und Export der externen Produktmodelle in Form von STEP–Dateien.

text/2, range/2 sind Prädikate, die als sogenannte *data dictionary* funktionieren. Ihre Verwendung finden sie in anderen Prädikaten, welche die allgemeine Anwenderschnittstelle ausmachen. Sie sind vorläufig der Platzhalter des noch nicht festgelegten Modells für Daten primitiver Art, die in Regelwerken vorkommen. Unter diesen Datentyp fallen alle gängigen Informationen, die man nicht direkt innerhalb des Produktmodells oder mit Hilfe von Regeln beschreiben kann. Mit Hilfe dieser beiden Prädikate können Tabellen, Abkürzungen, Begriffsbestimmungen und ähnliches erfaßt und evtl. deren Wertebereich prototypisch vorgegeben werden. Diese Vorgaben lassen dann die anderen Prädikate etwas genauer durchlaufen. So benutzt z.B. das Prädikat ask/n während eines interaktiven Dialoges mit dem Konstrukteur diese beiden Prädikate für die Gestaltung der Abfragen. Abzufragende Objekte (Variablen, Zustimmung u.ä.) werden mit ihrer Beschreibung und, falls vorhanden, mit ihrem Wertebereich vorgestellt, so daß eine Orientierung der Antwort auf einfache Weise erzielt werden kann. Vorteilhaft ist es daher, daß eine Anfrage automatisch mit einer Orientierungshilfe ausgestattet wird, ohne daß sie dafür zusätzlich programmiert werden muß. Nach und nach formt sich die Abfrageschnittstelle des interaktiven Dialoges gemäß dem Datentyp und Gültigkeitsbereich des gefragten Gegenstandes. Somit wird die Wahrscheinlichkeit einer Fehleingabe verringert, da bei Typ– und Wertebereichsverletzungen während der manuellen Eingabe die Abfrage automatisch wiederholt wird. Dagegen erlaubt das Nichtvorhandensein der Typ– und Bereichsdefinitionen unbeschränkte Antworttypen, deren Interpretation zum Teil der Anwendung überlassen wird.

Die Prädikate $text/2$ und $range/2$ besitzen die folgende einfache Form:

$$text(Object, Description)$$
$$range(Object, Type/Value)$$

Aufgrund der hohen Flexibilität der logischen Variablen können die zweiten Subterme ($Description$ und $Type/Value$) ohne Einschränkungen definiert werden, um somit alle möglichen Eigenschaften (*properties*) beliebig komplexer Objekte zu handhaben. Beispielsweise können unter dem Subterm $Description$ komplette Texte, unter dem Subterm $Type/Value$ baumartige bzw. mehrfache Definitionen erlaubt sein.

Das folgende Programmsegment verdeutlicht die Anwendung dieser Prädikate:

```
text(st,Schiffstyp).
range(Schiffstyp, [Tanker,Erzschiff,Standard,Massengutschiff]).

?- ask(st,_x).
Schiffstyp? [1) Tanker,
             2) Erzschiff,
             3) Standard oder
             4) Massengutschiff]
@ 4
```

```
_x = Massengutschiff
Yes
```

Die automatische Gestaltung der Abfrageform, die durch das Prädikat $ask/2$ generiert wird, kann evtl. neu konzipiert und den höheren Anforderungen entsprechend weiterentwickelt werden. Hierfür kommen z.B. graphische Schnittstellen in Frage. Die Abbildung 7.3 zeigt eine mögliche graphische Benutzungsoberfläche (GUI – *Graphical User Interface*). Eine solche graphische Erweiterung des Benutzerdialoges oder sogar des Regelwerkverwaltungssystems erhöht die Benutzerfreundlichkeit des Gesamtsystems und ist unter Anwendung der vorgestellten Werkzeuge relativ schnell zu realisieren. Im Abschnitt 7.3 wird das Thema ausführlicher behandelt.

7.2 Problemorientierte Sprache und ihre Integration

In diesem Abschnitt soll der Entwurf einer problemorientierten Sprache vorgestellt werden, von deren Bedeutung in den vorangegangenen Abschnitten schon öfteres die Rede war. Dabei muß erwähnt werden, daß ein solcher Entwurf in dieser Phase der Forschung nicht im Vordergrund bei der Entwicklung eines Regelwerkverwaltungssystems stehen darf. Eine problemorientierte Programmiersprache für die Bauvorschriften ist nicht, wie zu vermuten wäre, der Kern eines solchen Systems; sie ist vielmehr die Schnittstelle zwischen dem mit ihrer Hilfe zu erzeugenden Programmcode und dem Programmierer, der mittels eines beschränkten und somit überschaubaren Befehlsatzes (*instructions set*) den Code schnell und einfach implementieren kann.

Das Kleinhalten von problemorientierten Sprachen hat besondere Vorteile wie:

- gute Erlernbarkeit,
- Überschaubarkeit,
- relativ schnelle Entwicklung der Übersetzerwerkzeuge, die die eingegebene Spezifikation in ausführbare Programme umsetzen,
- einfache Fehlersuche und geringer Wartungsaufwand,
- bessere Chancen bei der Realisierung einer Norm,
- größere Systemunabhängigkeit und verbesserte Portabilität.

Da für den Übersetzer, welcher die in der problemorientierten Sprache formulierten Spezifikationen der Vorschriften auf den ausführbaren Code (Systemprädikate des RWVS) abbilden soll, die Zielsprache letztendlich Prolog ist, muß die Möglichkeit, auch auf der

Ebene der problemorientierten Sprache mit Prolog programmieren zu können, bis zur endgültigen Definition einer auf Vorschriften abgestimmte Programmiersprache gefordert werden. Denn gerade diese in die problemorientierte Sprache eingebettete Schnittstelle zu Prolog soll die vorhandenen Möglichkeiten der vorteilhaften Programmierung mit Systemprädikaten des RWVS oder sogar mit systeminternen Prädikaten der dabei verwendeten Prolog–Installation nicht unbrauchbar machen[6]. Dies sollte allerdings nach Möglichkeit nur noch vorläufig und nur in Extremfällen verwendet werden, um mit den vorher aufgeführten vorteilhaften Aspekten der Kleinhaltung der DRL (*Dimensioning Rule Language*) nicht zu kollidieren.

Der vorliegende Entwurf ist eine einfache Schnittstelle zu den Systemprädikaten des RWVS. Die Spezifikation der Regeln erfolgt in einer Textdatei, die dann mit einem entsprechenden Übersetzer in die Prolog–Version des RWVS umgewandelt wird. Für die Festlegung der Sprachsyntax dienen die Systemprädikate als Vorlage, auf deren einfache benutzerfreundliche Schreibweise ein großer Wert gelegt werden muß.

Im folgenden werden der globale Aufbau der Sprachsyntax und die einzelnen Sprachelemente (Konstrukte) beschrieben. Für eine formale Beschreibung der Syntax reicht die Backus–Naur–Form für die Spezifikation von kontextfreien Sprachen vollkommen aus. Die Semantik wird einfach aus der Semantik der darunterliegenden Programmiersprache Prolog abgeleitet, in der letztendlich die Interpretation der Vorschriften durchgeführt werden soll.

Syntax der DRL

Hauptkonstrukt der DRL ist der modulare Aufbau der File–Struktur, in der die einzelnen Regeln jeweils in getrennten Blöcken untergebracht sind. Diese Trennung erlaubt eine fehlertolerante Umsetzung der Regeln, indem mit der Übersetzung fortgefahren werden kann, auch wenn in den vorangegangenen Blöcken auf Fehler gestoßen wurde. Nach der Korrektur der fehlerhaften Spezifikation kann ein inkrimentelles Laden der Regeln durchgeführt werden, da ein gezieltes Übersetzen der Blöcke in den Dateien möglich ist.[7]

Jedem File, in welchem Regeln definiert sind, wird ein Modul–Name mit folgender Syntax zugeordnet:

MODULE *module name.*

Ein Modulname dient der zusätzlichen Trennung von Regeln, die zufällig den gleichen Namen haben. Modulname und die sequenzielle Zuordnung im File gewährleistet eine

[6] Dieses erinnert an Schnittstellen wie z.B. zwischen C und Assembler für die jeweilige Hardware.
[7] Sonst kann das ganze File evtl. nochmals geladen werden.

eindeutige Trennung der Regeln. Das Weglassen der Spezifikation des Modulnamens ist möglich, wodurch die Regeln im File für andere Module allgemein zugänglich werden.

Der Modul–Spezifikation folgt die Definition der Regeln, die für den vorliegenden Prototyp (als Sprachentwurf für Dimensionierungsregeln) von relativ einfacher Art sind. Sie ähnelt der Definition von Produktionsregeln (*production rules*), die in Systemen für Regelverarbeitung wie OPS5[8] vorkommen. Ähnlich der OPS–Syntax sieht eine DRL–Regel aus (s. Tab. 7.1).

```
RULE rule name
        | second name
        ...
        | n-th name
IF
        premise
        second premise
        | alternate second premise
        ...
        n-th premise
THEN
        conclusion
        second conclusion
        | alternate second conclusion
        ...
        n-th conclusion
END rule name.
```

Tab. 7.1: *DRL–Regelsyntax*

In dem ersten Abschnitt der Regeldefinition (RULE) werden ein oder mehrere Namen für die Regel spezifiziert. Sie dienen einerseits als Regelreferenzen, andererseits als Akronym, wenn Regeln als Objekt in einer der Voraussetzungen (*premise* [IF]) oder in einer der Folgerungen (*conclusion* [THEN]) verwendet werden.

Für die Spezifikation von Voraussetzungen und Folgerungen stehen, wie auch im RWVS (auf der Programmierungsebene mit Prolog), logische Terme zur Verfügung. Einige nützliche Operatoren sind zu definieren, um die Schreibweise einiger Systemprädikate wie ask/n, $solve/n$ usw. zu vereinfachen. Für die Behandlung der Werte (auch komplexer Art) stehen die logischen Variablen zur Verfügung. Da eine Erkennung der Variablen als Eingabe (IF) oder Ausgabe (THEN) (oder lokal zu der Regel) nicht unbedingt erforderlich

[8] OPS5 [Browston/Farrell/Kant/Martin 88] gehört der OPS Klasse, die bei der Implementierung des Expertensystems XCON für die Konfiguration von VAX Rechnern eingesetzt wurde (OPS – *Official Production System*).

ist, konnte auch auf die Spezifikation der Aufrufparameter (RULE) verzichtet werden, was zu einer Erleichterung bei der Programmierung der Regeln führt und eine Annäherung zu der in gesprochener Sprache vorkommenden natürlichen Form darstellt. Beispiele in den folgenden Abschnitten zeigen, wie Prolog–Terme in die Regeln integriert werden. Die Definition von Voraussetzung und Folgerung als Konstrukte der Regelbeschreibungssprache DRL ermöglicht zugleich, daß auch Systemprädikate des Prolog–Systems ohne besondere Schnittstellen innerhalb DRL aufgerufen werden. Diese einfache Art ist für die Implementation eines Übersetzers von besonderem Vorteil. Ein Beispiel–Übersetzer befindet sich in [Toparlak 93a].

7.3 Benutzungsoberfläche

Die Gestaltung der Benutzungsoberfläche eines komplexen Programmsystems spielt eine große Rolle sowohl für den Endnutzer als auch für den Programmierer. Sie hilft in allen Phasen des Lebenszyklus eines Programms. Die Einbeziehung der Computer–Graphik in die Benutzerschnittstelle ist heute in der modernen Informationstechnik für den - ständig wachsenden und in kürzeren Zeiten zu bewältigenden Datenfluß[9] unentbehrlich. Die Entwicklung von graphischen Benutzerschnittstellen erfordert jedoch längere Entwicklungszeiten und erhöhte Kosten. Da solche Schnittstellen von komplexer Natur sind, ist eine leistungsfähigere Soft– und Hardware für die Anwendung sowie für die Entwicklung besonders erforderlich.

Die Erkenntnisse über den vielfältigen Entwicklungs– bzw. Nutzungsaufwand der graphischen Benutzerschnittstellen in komplexen Programmsystemen führte in der Informationstechnik zu einer radikalen Änderung. Einer der wichtigsten Wendepunkte war die Freigabe des X–Window–Systems des *Massachusetts Institut of Technology* (MIT) [Scheifler/ Gettys 86]. Mittlerweile hat sich dieses System, eine gigantische Softwarebibliothek zur Entwicklung der graphischen Oberflächen sowie zur graphischen Programmierung, als de facto Standard in der Computer–Industrie durchgesetzt. Dabei spielten auch die breiten Einsatzmöglichleiten des Betriebssystems UNIX auf einer weit gefächerten Hardware–Palette, von Supercomputern bis zu Handrechnern, als Basisplattform für das X–Window–System eine wichtige Rolle.

Für die vorliegende Forschungsarbeit steht die Realisierung einer graphischen Benutzungsoberfläche von ihrer Wichtigkeit her nicht an der ersten Stelle. Ihre Konzeption ist jedoch frühzeitig durchzuführen und zukunftsorientiert festzulegen. Dabei ist die Auswahl des X–Window–Systems als Ausgangspunkt vorteilhaft. Im Rahmen der Arbeit ist diese Wahl zum Teil zwangsläufig, da für das Betriebssystem UNIX kaum anderes als das X–Window–System verfügbar ist. Das Betriebssystem UNIX wurde gewählt, wegen

[9] zum Teil wegen der durch die Entwicklung verursachten immer wieder höhergestellten Ansprüche in allen Schichten der EDV–Gesellschaft

ausgebreiteter Verfügbarkeit auf allen Rechnerplattformen, auf denen das Implementationswerkzeug des RWVS ProLog *by* BIM, eine kommerziell verfügbare Prolog–Entwicklungsumgebung, läuft. Das STEP–Parser–System läuft ebenfalls auf dem Betriebssystem UNIX.

Die Gestaltung der graphischen Benutzerschnittstelle des RWVS besitzt für die weitere Entwicklung prototypischen Charakter. Damit ist gemeint, daß starre Festlegungen oder Anbindungen an spezielle Schnittstellen (Subsysteme des X–Window–Systems) möglichst vermieden werden. Da selbst für das X–Window–System sehr verschiedene konzeptionelle Zweige existieren, muß hier auch eine entsprechende Auswahl getroffen werden, um die Systemunabhängigkeit des RWVS zu bewahren. Die Programmierbibliothek XView[10] für die Entwicklung von interaktiven Anwendungen mit graphischen Schnittstellen wurde zu diesem Zweck ausgewählt, weil sie frei erhältlich ist [Heller 91, Miller 90]. Die Programmieransätze dieser Bibliothek mit objektorientierter Fensterverwaltung, *pull-down, pop-up* Menues, Ein– und Ausgabefelder, Emulationen von Texteditoren und Bildschirmterminalen, Unterstützung der graphischen Ein/Ausgabe (*drawing canvas*) und Mechanismen für die Kommunikation dieser Elemente untereinander (*drag and drop*) usw. entsprechen den Bedürfnissen des RWVS vollkommen.

Die graphische Benutzungsoberfläche (GUI – *Graphical User Interface*) des Regelwerkverwaltungssystems besteht aus Komponenten, die ihre entsprechende Equivalenz im Kern des Systems wiederfinden. Das heißt, sie basieren vollständig auf den Systemprädikaten, oder sind deren Abbildung in einer graphischen Form. Die GUI sollte nicht mehr oder weniger Möglichkeiten als der Systemkern haben, so daß eine parallele Entwicklung des Kerns und seiner Benutzungsoberfläche möglich ist.

Der Gesamtumfang der GUI umfaßt mehrere Elemente, die unter sich Objekte austauschen können, was das Arbeiten mit GUI erheblich vereinfacht. Zum Beispiel kann eine Regel, die während des Arbeitens mit dem Rule Browser gewählt wird, unmittelbar mit dem *Rule Editor* modifiziert und anschließend aufgerufen werden. Eine grobe Zusammenfassung der Elemente der GUI des RWVS wird unten angegeben:

- *Dialog Panel* – führt den Dialog mit dem Benutzer. Hinter ihm verbirgt sich das Systemprädikat ask/n. Es enthält Eingabefelder für Datentypen wie `integer`, `real`, `text` und `list`. Das Feld `list` ist für die Auswahl einer oder mehrerer Antworten zwischen Alternativen bestimmt. Felder für die logische Verknüpfung, wie z.B. 'nur eine von vielen', 'eine oder mehrere', beschränken die Eigenschaften einer Liste (s. Abbildung 7.3). Ähnlich dem Systemprädikat ask/n erlaubt das *Dialog Panel* die Eingabe einer Regel als Antwort, die nach Beendigung der Eingabe ausgeführt wird, wenn der Dialog von dem Systemprädikat $solve/n$ generiert wurde. Das heißt, die eingegebene Regel wird ausgeführt, um einen arithmetischen Wert auszurechnen, falls er in einem Ausdruck benötigt wird. Das *Dialog Panel* ruft den

[10] *X Window System-based Visual/Integrated Environment for Workstations*

Rule Editor, wenn diese Art der Eingabe verlangt wird. Dieser Vorgang hat für die Regelbasis einen aufbauenden Charakter. Die Nutzung dieser Möglichkeit ist besonders für den Programmierer von Regelwerken von Interesse.

- *Rule Editor* – ist eine ergänzende, zum Teil selbständige Komponente für das *Dialog Panel*. Er besteht aus einem Texteditor[11], der für das Editieren und das Verifizieren von DRL–Texten geeignet ist. Das Aufrufen des DRL–Übersetzers geschieht entweder interaktiv oder automatisch nach Beendigung der Eingabe (s. Abbildung 7.4, die Knöpfe VERIFY und DONE).

- *Rule Browser* – ist ein Werkzeug, das die Struktur der Regeln innerhalb eines Moduls graphisch darstellt. Die graphische Darstellung der Netzstruktur der Regeln vermittelt wertvolle Informationen wie Querverweise zwischen den Regeln und Beziehungen zueinander. Einzelne Regeln können ausgewählt und durch den Aufruf des *Rule Editor* verarbeitet werden (s. Abbildung 7.5).

- *Model Editor* – ist eine Komponente für die Modellierung der Objektdatentypen sowie deren Attribute. Das Modell besteht aus einer der Modellierungssprache EXPRESS ähnlichen Ausprägung, einer dem RWVS angepaßten Form. Hierfür steht die RWVS/SPS–Schnittstelle zur Verfügung (s. Abschnitt 8.1)

- *Model Browser* – ist eine Komponente zur Verwaltung der Modellstruktur sowie ihrer Instanzen während einer Interaktion mit RWVS bzw. *Dialog Panel*. Mit ihr können bereits vorgegebene Eingaben untersucht und ggf. variiert werden. Hierfür wurde eine tabellarische Form zur Darstellung der Instanzen vorgenommen, da sonst eine graphische Darstellung dieses Sachverhalts schnell unüberschaubar werden kann. Eine graphische Darstellung ist nur noch auf der Modell–Ebene gewährleistet. Objektkommunikation zwischen *Model Browser* und *Model Editor* ist auch möglich.

Denkbare Erweiterungen der graphischen Schnittstelle sind z.B. Anschlüsse zu einem konventionellen Datenbankverwaltungssystem und zu dem Datenverwaltungssystem des CAD–Systems, wo die Kommunikation der Objekte zwischen diesen und dem RWVS visualisiert bzw. graphisch verwaltet werden kann.

Schlußfolgerungen

Das vorliegende Regelwerkverwaltungssystem mit seinem wissensbasierten Aufbau eignet sich nicht nur für die Programmierung von Regelwerken, sondern auch für den weiteren Ausbau des Systems selbst. Die wissensbasierte Programmiermethodik beschleunigt die Realisierung des jeweiligen Prototyps als Baustein zur Bewältigung der immer wieder neu anfallenden Aufgabenstellungen. Gerade die letzten Abschnitte 7.2 und 7.3 können

[11] Ein kontextsensibler Editor wäre auch denkbar, der auf die **richtige** Eingabe von DRL beschränkt ist.

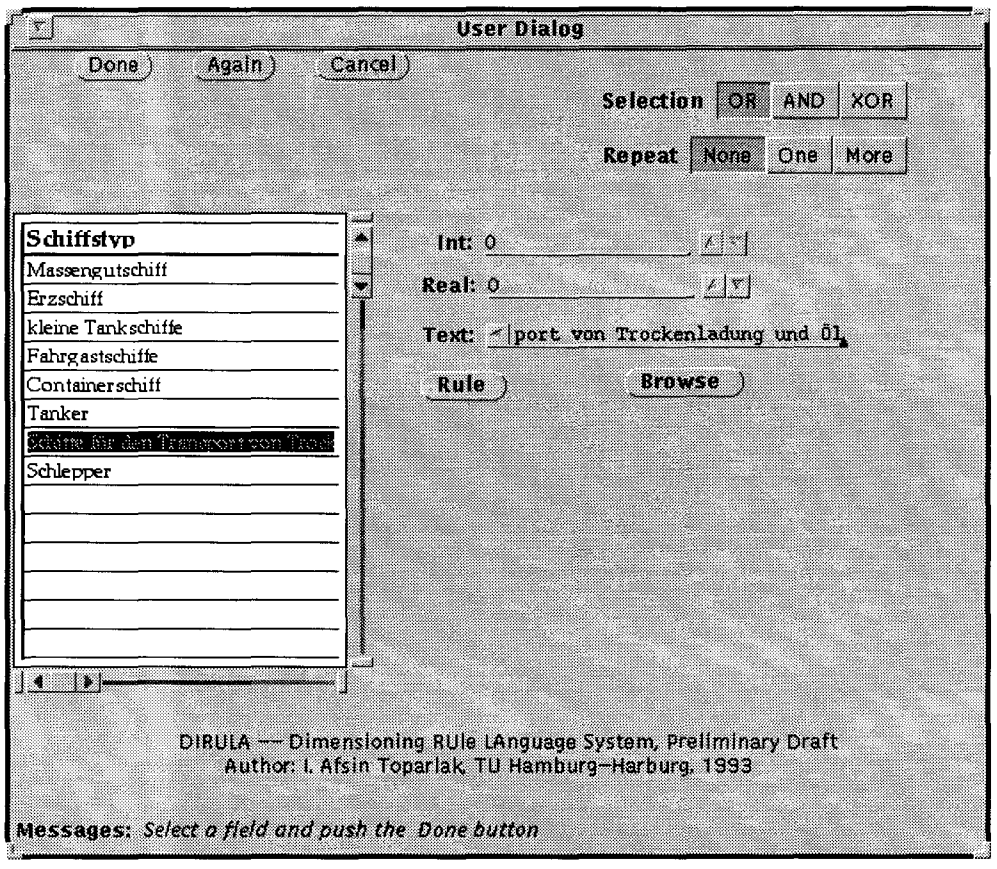


Abb. 7.3: *Beispielhafte GUI für den Abfragedialog mit dem Benutzer*

zur Veranschaulichung der Anpassungsfähigkeit des RWVS[12] als Beispiele dienen, wobei die Funktionalität der dort, wenn auch prototypisch entwickelten Werkzeuge (DRL und GUI) weitestgehend in das System eingeflossen ist, ohne die Komplexität des Gesamtsystems zu erhöhen.[13]

[12] in Verbindung mit dem SPS

[13] Vgl. Abschnitt 'Schlußfolgerungen', Kapitel 5.

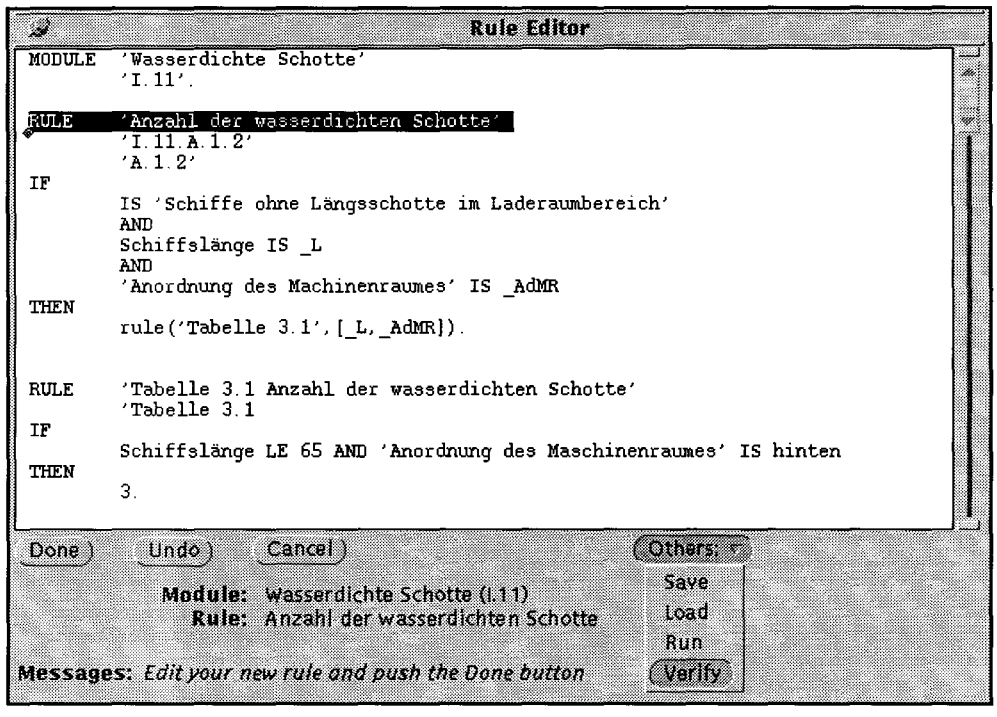


Abb. 7.4: Rule Editor *GUI–Prototyp*

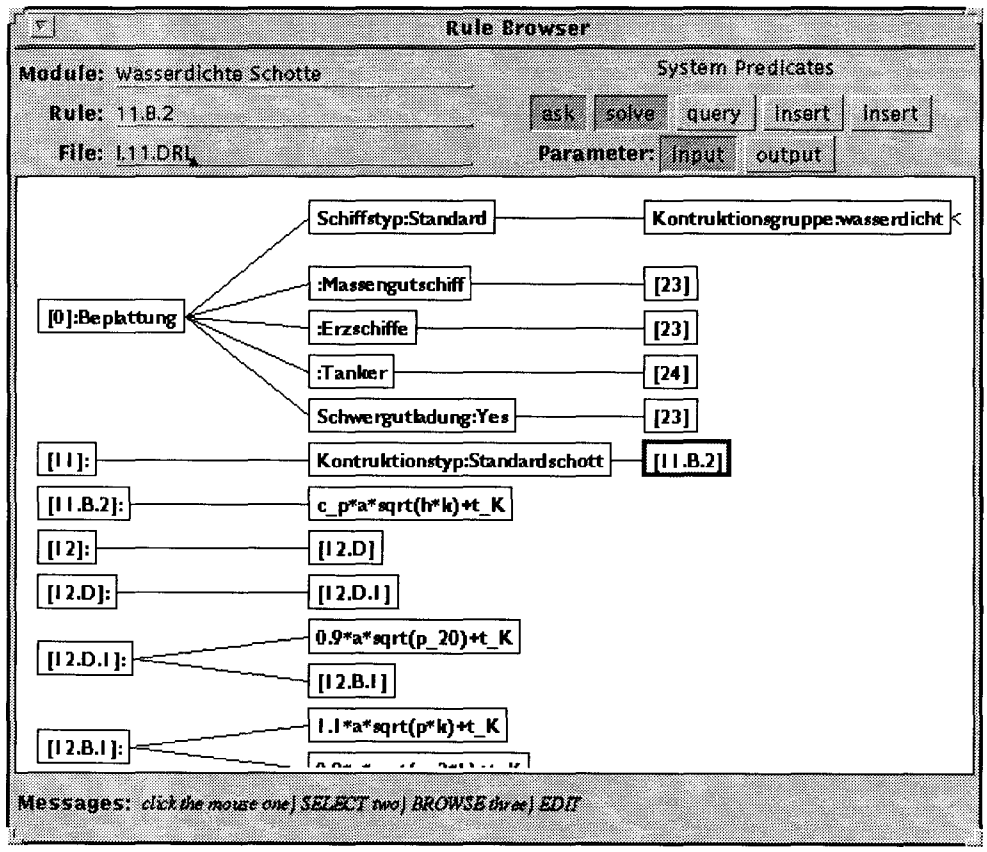


Abb. 7.5: Rule Browser *GUI–Prototyp*

Kapitel 8

Implementation

Von besonderer Wichtigkeit ist es, externe Produktmodelle bzw. ihre physikalische Instanz mit dem RWVS zu verarbeiten. Den Anwendern der CAD–Systeme ist dessen Aufbau bzw. Anordnung in der Regel vorbehalten, so daß eine Modifikation des Produktmodells oft nicht möglich ist. Eine Be– bzw. Verarbeitung eines solchen Produktmodells findet nur in den Modulen des Programmsystems statt. Der direkte Zugriff von außen ist nur in den seltensten Fällen möglich. Das Modell ist auch nicht ohne weiteres aus dessen Instanzen herleitbar, auch wenn sie ggf. in Form von Dateien vorliegen. So bleibt einem Anwender noch übrig, nur über eine vom Hersteller festgelegte Abfrageschnittstelle mit einzelnen Komponenten (voneinander getrennten Datensätzen) eines **abstrakten** Datenmodells, aber nicht mit dessen Aufbau oder Strukturierung auf einer höheren Ebene, zu kommunizieren.

Es ist daher sinnvoll, die Kommunikation zwischen dem Regelwerkverwaltungssystem und dem in Frage kommenden CAD–System auf einer höheren Ebene herzustellen. Eine ähnliche Vorgehensweise wurde während des Forschungsvorhabens, AiF–Nr. 7555, *Datengenerierung für Finite–Elemente–Berechnungen mit Hilfe von CAD–Systemen*, bei der Forschungsstelle erfolgreich erprobt, wo zwischen dem schiffbaulichen CAD–System STEERBEAR und dem FEA–System MARC/MENTAT zur Datenmodellierung der ISO–Standard STEP herangezogen und auf eine 1-1–Verbindung zwischen den beiden Systemen verzichtet wurde. Für die Repräsentation der dadurch auszutauschenden Daten wurde ebenfalls die von ISO vorgeschlagene rechnerexterne Speicherung (STEP *physical files*) in Form von Dateien[1] verwendet.

Die Möglichkeit der Verarbeitung der CAD–Daten auf der Modellierungsebene führt zu

[1] Da das CAD–System STEERBEAR unter dem Betriebssystem VMS und das FE–Paket MARC/MENTAT unter UNIX arbeiten, war es notwendig, Transferdienste wie FTP (*File Transfer Protocol*) aufgrund der Unterschiede zwischen den File–Systemen der beiden Betriebssysteme für die Kommunikation einzusetzen. Neuerdings existieren Netzwerkverbindungen wie *VMS/ULTRIX Connection*, die solche Probleme grundsätzlich aufheben.

frühzeitigen Ergebnissen, die z.T. eine Allgemeingültigkeit besitzen. Die dabei gewonnenen Erkenntnisse lassen sich ohne große Schwierigkeit in andere Gebiete übertragen. Das obige Beispiel zeigt dies: Die aus dem CAD–System extrahierten Daten werden in Form von STEP–Dateien extern abgespeichert. Der Aufbau der Instanzen der einzelnen Entities, die den Datensätzen in der CAD–Datenbank entsprechen, ist bekannt. Das Datenmodell liegt in einer Form vor, die vom Rechner verarbeitet werden kann. Die Modellierungssprache EXPRESS, die ebenfalls ein Bestandteil des STEP–Standards (*ISO Draft Proposal 10303*) ist, beschreibt denselben. Eine Aufgabenstellung, die sich mit Modelltransformationen beschäftigt, wie sie bei CAD/FE–Verbindungen oft vorkommen, ist einfacher zu bewerkstelligen, wenn das zugrundeliegende Informationsmodell in einer leicht zugänglichen Form vorliegt. So war es im Rahmen der oben erwähnten Forschungsarbeiten möglich, innerhalb kürzester Zeit Regeln für die Modelltransformationen zu erstellen, sie auf jede einzelne Entity der Modellinstanz der CAD–Daten anzuwenden und dem FE–Modell entsprechend umzusetzen. Dies bewahrt den Programmierer davor, sich mit jeder einzelnen Informationseinheit (*Entity*) getrennt zu beschäftigen, er befaßt sich nur noch mit dem Modell und kann den vorliegenden Problemfall auf einer relativ übersichtlichen Ebene zu einer allgemein vertretbaren Lösung führen.

8.1 Schnittstelle zum SPS

In bezug auf die Erkenntnisse aus diesem typischen Vorfall ist es sinnvoll, das Regelwerkverwaltungssystem nicht direkt mit dem CAD–System zu verbinden, solange dieses die notwendigen, auf das Datenmodell bezogenen Schnittstellen nicht zur Verfügung stellt, sondern eine Zwischenschicht mit Möglichkeiten der Modellierung von CAD–Daten und ihrer Verarbeitung durch das STEP–Datenmodell, ähnlich wie bei der CAD/FEA–Kopplung, zwischen die beiden Systemen zu legen. Zweckmäßig wurde auch hierfür das STEP–Parser–System eingesetzt, von dem in vorangegangenen Kapiteln berichtet wurde.

Eine der wichtigen Eigenschaften von Prolog ist, daß die verfügbaren Datentypen meistens sich selbst repräsentieren (einfache Symbole oder deren einfache oder komplexe Komposition), was zur Folge hat, daß der Inhalt eines STEP–Files, da auch dieses nur Daten enthält, relativ leicht in eine sich selbst darstellende Form, die Prolog versteht, umgewandelt werden kann. In konventionellen Programmiersprachen hingegen werden Daten nur mit Hilfe von sie vertretenden 'Variablen' verarbeitet. Oft sind auch neuere Typvereinbarungen notwendig. Dies ist in Prolog nicht zwingend erforderlich. Die folgenden Programmsegmente verdeutlichen diesen Vorteil der Programmiersprache Prolog.

```
/* C */
typedef enum {TANKER, ORE_CARRIER,
  /* ... */
} ShipType;
```

```
typedef struct {char *name; ShipType type; char *yard;
  float length;
  float width;
  float depth;
  int   deplacement;
  /* ... */
} Ship;

Ship myShip;
myShip.name   = "Carmen";
myShip.type   = TANKER;
myShip.length = 250.0;
calculate_design_displacement(myShip);


/* Prolog */
?- assert(Ship(Carmen, Tanker, 250.0, ...)).
Yes
?-  Ship(Carmen, _t, _, _, _, _, _).
_t = Tanker
Yes
```

Wie aus diesem Programmsegment zu erkennen ist, ist die Vereinbarung einer Variablen und die Belegung mit den Daten, welche später mit ihrer Hilfe referenziert, verändert oder verwendet werden, in Prolog nicht notwendig. Da die Daten sich selbst repräsentieren und so abgespeichert werden können, daß man auf sie mit Hilfe von effizienten Suchalgorithmen, die von Prolog zur Verfügung gestellt werden, ohne große Mühe zugreifen bzw. sie abfragen kann, ist der Verzicht auf Variablen[2] dieser Art nicht unbedingt mit Nachteilen verbunden. In prozeduralen Sprachen erfolgt die Unterscheidung der Daten über die unterschiedlichen Namen ihrer Variablen, nicht über deren Inhalt[3]. In **deklarativen** Sprachen wie Prolog ist die Trennung der Daten durch ihre Ausprägung gewährleistet, d.h., der Inhalt der Daten ermöglicht es, die im Programm verwendeten Objekte ohne zusätzliche Benennung voneinander zu trennen und sie als selbständige Objekte anzufassen.

Unsere physikalische Welt ist ein Beispiel dafür:

> In einem Raum stehen ein Stuhl, ein Bücherregal und ein Arbeitstisch. Andere Gegebenheiten sind auch da. Objekte in diesem Raum sind sofort erkennbar: Man kann sich ein Buch aus dem Regal holen, sich auf den Stuhl setzen und das Buch lesen. Dabei muß man diese Objekte nicht unbedingt benennen. Selbst wenn ein zweiter Tisch dort anwesend wäre, hätte man vielleicht den Tisch **vor** dem Regal

[2] Im Sinne der Variablen, die in prozeduralen Sprachen vorkommen, die einen Speicherbereich aufweisen, der die von diesen Variablen vertretenen Daten beinhaltet.

[3] Sofern dieser nicht ausdrücklich verglichen wird

oder statt dessen den **am** Fenster gewählt. Eine Unterscheidung der Tische bzw. ihre Auswahl ist problemlos möglich, da sie nicht am selben Ort stehen. Weitere Unterscheidungen sind nur notwendig, wenn bestimmte Ähnlichkeiten eine getrennte Erkennung erschweren, etwa 'der dunkle Tisch', wenn beide am Fenster stehen. Eine deutliche Trennung durch Bezeichnungen wie 'T1', 'T2' würde als etwas Ungewöhnliches erscheinen. Eine Identifikation ist nur dann notwendig, wenn keine deutlich erkennbaren Unterschiede vorhanden sind, wie im Kino: Es gibt dort nur noch lange Reihen gleichartiger Stühle und die Leinwand. Die Identifizierung der Sitzplätze ist die einzige Orientierungshilfe. □

Die Entities, die die Instanz eines Produktmodells ausmachen und in einer systemunabhängigen Datei vorliegen, werden, nach dem ISO–Standard STEP, an Variablen gebunden. Diese Variablen dienen gleichzeitig der Definition und der Referenzierung von Entities. Sie sind sozusagen die Identifikation der Entities. So ist z.B. die Identifikation einer Entity der Ausgangspunkt für die Verarbeitung eines Produktmodells mit dem STEP–Parser–System oder ganz allgemein mit konventionellen Systemen; z.B. bei der Suche eines bestimmten Datensatzes innerhalb der Modellinstanz. Nach STEP bekommt jede Entity eine eindeutige Identifikation, die entsprechend der Definition aus einer Integerzahl[4] besteht. Nachteilig erweist sich diese Art der Datenmodellierung, wenn z.B. nicht nach den Identifikationen sondern nach den Attributen innerhalb des Produktmodells gesucht wird. Beispielsweise verwendet das STEP–Parser–System als Suchtabelle einen balancierten Binärbaum, der nach den Identifikationen der Entities während der Definition erneut geordnet wird, um die nachfolgende Suche zu kürzen. Die somit vereinfachte Suche nützt auch nichts, wenn für sie keine Identifikation als Ausgangspunkt zur Verfügung steht, wie z.B. bei der Suche nach geometrischen Entities, die gleichen oder 'ähnlichen' Inhalt aufweisen.

An Hand dieser Überlegungen wurde bei der Schnittstelle zwischen RWVS und SPS die Identifikation in ein Integer–Attribut umgewandelt, zu deren Erkennung nicht mehr nach dem Zeichen # gesucht, sondern die Entity–Semantik herangezogen (im Falle einer Entity–Referenz) wird. Die Erkennung der Id einer Entity–Definition ist aus dem Kontext möglich: Ihr folgt nämlich ein Gleichheitszeichen bzw. Doppelpunkt, wenn das erste nicht überschrieben werden soll. Die Eigenschaften der Schnittstelle sind im folgenden kurz dargestellt:

- Direkte Übernahme der Basis–Datentypen, `integer`, `real`, `string`.
- Umwandlung der Entity–Id in ein Attribut der Entity. Sie ist dann der erste Parameter der Entity und damit ein logischer Term.

[4]Sie wird kombiniert mit dem Zeichen #, das eine klare Trennung zwischen den Integerwerten als Identifikation oder Attribut darstellt. Eine Integerzahl ist die Identifikation einer Entity, wenn sie dem Zeichen # folgt. Sonst ist es ein einfaches Attribut einer Entity. Beispiel: `#1001`. Allerdings ist diese Ausprägung meistens unbedeutend, da die Erkennung einer Integerzahl als Identifikation oder Attribut aus dem jeweiligen Kontext oder nach der Semantik der Entity möglich ist.

- Abbildung der Aggregat–Typen, `list`, `set`, `bag`, in den Prolog–eigenen Datenkonstrukt `list`[5]. Listen in Prolog sind Binärbaume, auf die direkt zugegriffen werden kann. Der eingebaute Listenoperator '|' schafft dafür Abhilfe, eine Liste zu zerlegen und deren Elemente anzufassen.

- Abbildung der Instanz eines Produktmodells in ein Prolog–File, das später direkt gelesen werden kann.

- Integrierte Überprüfung der Modellsemantik, entsprechend Modellierungssprache EXPRESS des ISO/STEP–Standards. Dabei werden die Prolog–Klauseln erzeugt, die für die Überprüfung der STEP–Entities über einen Meta–Aufruf automatisch interpretiert werden, d.h, die Gestaltung der Entity im STEP–File verrät, welche logische Klausel (oder ein Prädikat) aufgerufen werden muß.

- Dynamische Generierung der Klauseln zur semantischen Analyse nur noch nach Anforderung und somit inkrimenteller Aufbau eines auf die Anwendung abgestimmten Systems [Toparlak 91c].

Die Abbildung 8.1 zeigt die Abläufe während des Arbeitens mit der RWVS/SPS–Schnittstelle.

8.2 STEP–Parser–System

Der programmierbare Parser ist als eine Funktion realisiert, die über eine generische Schnittstelle mit Anwendungsprogrammen verbunden werden kann. Diese Konfiguration wurde gewählt, um den Einsatz des Parsers in Kombination mit verschiedenen und sich ändernden Datenmodellen zu ermöglichen. Alle diese Werkzeuge basieren auf einer generischen RID. Der Parser ist somit ein Umsetzungswerkzeug von der physikalischen Repräsentation der STEP–Entities in die Hauptspeicherdatenbank.

Die Implementation dieser Schnittstellen basiert auf dem Analyse–Synthese–Modell der Compilertechnik [Aho/Sethi/Ullman 88]. Bei der Analyse wird das File in seine Bestandteile zerlegt. Die Analyse besteht aus drei Teilen:

- Lineare/Lexikalische Analyse,

- Hierarchische/Syntaktische Analyse,

- Semantische/Bedeutungsanalyse.

[5] Mit dem Unterschied zu STEP, daß anstelle '(...)' '[...]' verwendet wird.

Bei der Synthese wird versucht, den Inhalt des Files in der Zielsprache neu zusammenzufassen bzw. auszudrücken (Compilierung/Codeerzeugung). Bei der Realisierung der oben erwähnten Schnittstellen wurde jedoch die RID der Produktdaten als Zielsprache definiert, die dann zur Abbildung der Dateninhalte der STEP–Dateien bzw. eines Produktmodells im Hauptspeicher des Rechners dient. Somit wird das Verarbeiten der Produktmodelle durch die Anwendungsprogramme auf eine effektive Weise ermöglicht.

Analog zu der Zielsetzung der Compilertechnik,

"ein Programm, das in einer dem Menschen gerechten Quellsprache formuliert ist, in eine der Maschine gerechten Zielsprache zu übersetzen",

erfolgt hier das 'Übersetzen' letztendlich durch die Anweisungen des Anwendungsprogramms, z.B. Modelltransformationen [Toparlak 92b], [Grafe 93], Visualisierung [Koch 90], Analyse [Toparlak 92a], usw.

8.2.1 Lexikalische Analyse von STEP

Eines der zentralen Elemente des STEP–Parser–Systems ist die Komponente zur lexikalischen Analyse von STEP–Dateien. Aufgabe dieser Komponente bzw. des Scanners ist das Lesen der Zeichen und ihre Zusammenfassung zu Symbolen (*tokens*) sowie deren anschließende Übergabe an die Komponente zur syntaktischen Analyse (dem Parser).

Die Implementation des lexikalischen Teils des STEP–Parser–Systems wurde mit Hilfe des Scanner–Generators `lex` [Paxson 89] realisiert. Der Scanner–Generator bietet die Möglichkeit, Scanner in der Programmiersprache **C** durch eine Definitionsdatei, in der die sogenannten lexikalischen Regeln (*regular expressions*) zur Zusammenfassung von Symbolen (*tokens*) definiert werden, automatisch zu erzeugen. *Regular expressions* beschreiben einfache Regeln, die Symbole aus den Zeichen vorschreibend zusammenfassen. Die Tab. 8.1 zeigt die vereinbarten Tokendefinitionen entsprechend dem STEP–Standard [STEP 89]. Die einfache Vereinbarung reicht nicht immer aus. Wenn komplizierte Tokenwerte benötigt werden, z.B. bei der Erkennung einer Gleitkommazahl, muß diese Zahl, die als Zeichenkette gespeichert ist, in die interne Darstellung dem auf dem Rechner verfügbaren arithmetischen Prozessor entsprechend übersetzt werden.[6]

Anwendungsprogramme oder Teile der STEP–Parser–Bibliothek sind damit wiederum unabhängig von etwaigen Änderungen der lexikalischen Elemente der STEP–Dateien.

[6] Die Standard ANSI **C**–Bibliothek enthält z.B. entsprechende Umwandlungsroutinen wie `double atof(char*), long atol(char*)` etc.

```
/* Description: STEP physical file token definition for lex(1)                    */
/* Date: Tue Wed 30 15:35:17 MET DST 1992                                         */
/* Author: Isa Afsin Toparlak                                                     */

KEYWORD             "!"?[A-Z_]([A-Z_]|[0-9])*
COMMENT             "/"\*([^*/]|[^*]"/"|\*+[^/])*\*\**"/"
STRING              "'"([^'\n]|"''")*"'"
INTEGER             [+-]?[0-9]+
EXPONENT            [Ee]{INTEGER}
REAL                [+-]?[0-9]+(("."[0-9]*{EXPONENT}?)|{EXPONENT})                 10
ENTITY              [#@][0-9]{1,9}

%%
"STEP"                      {RETURN(STEP);}
"HEADER"                    {RETURN(HEAD);}
"ENDSEC"                    {RETURN(ENDSEC);}
"DATA"                      {RETURN(DATA);}
"ENDSTEP"                   {RETURN(ENDSTEP);}

{ENTITY}                    {RETURN(ENTITY);}                                      20
{STRING}                    {RETURN(STRING);}
{REAL}                 {yylval.SP->val.real= atof(yytext); RETURN(REAL);}
{INTEGER}         {yylval.SP->val.integer= atol(yytext); RETURN(INTEGER);}

"AXIS2_PLACEMENT"|"AXIS2_PL" {RETURN(AXIS2_PLACEMENT);}
"CIRCLE"|"CIRC"             {RETURN(CIRCLE);}
"COORDINATE_SYSTEM"         {RETURN(COORDINATE_SYSTEM);}
"DIRECTION"|"DIR"           {RETURN(DIRECTION);}
"FACE"                      {RETURN(FACE);}
"LINE"                      {RETURN(LINE);}                                        30
"PLANE"                     {RETURN(PLANE);}
"PLATE"                     {RETURN(PLATE);}
"POINT"|"PT3"               {RETURN(POINT);}
"POLYLINE"|"PLN"            {RETURN(POLYLINE);}
"POLY_LOOP"|"POLYLOOP"      {RETURN(POLYLOOP);}
"VERTEX"|"VX"               {RETURN(VERTEX);}
/* application specific entity types come here ...
 */

{KEYWORD}                   {RETURN(GENERIC);}                                     40

"$"                         {RETURN(OPTIONAL);}
[\n]|{COMMENT}              {lineno += strnchr(yytext,'\n');}
"/N/"|"/F/"[ \t]+           ;

.                           {return yytext[0];}
%%
```

Tab. 8.1: *STEP physical file–Tokendefinition*

8.2.2 Syntaktische Analyse von STEP

Zentrales Element des STEP–Parser–Systems ist der mit Hilfe des Compiler–Generators `yacc` [Rubin 86] erzeugte Parser, dessen Definitionsdatei der in der Abb. 5.1 vereinfacht dargestellten File–Syntax entspricht. Abgesehen von den prinzipiellen Vorteilen, die der Einsatz eines Compiler–Generators wie `yacc` bietet, erlaubt diese Vorgehensweise auch die relativ schnelle Anpassung der Parser–Bibliothek an die sich nach wie vor ändernde Syntax der physikalischen File–Struktur. Die Möglichkeit der Verbindung eines mit Hilfe des Scanner–Generators `lex` [Paxson 89] generierten Scanners mit dem ebenso automatisch generierten `yacc`–Parser ist in den vorliegenden Versionen der *software engineering tools* `yacc` und `lex` auf dem UNIX–Betriebssystem gegeben, welche der Übertragung der Attribute dienen.

Wenn der Parser vom Anwendungsprogramm aktiviert wird, liest er das entsprechende STEP–File und wandelt dabei nacheinander jede einzelne Entity in ihre generische RID um. Die Elemente der RID erlauben die Darstellung beliebiger Entity–Inhalte und Konfigurationen in einer Weise, die detaillierte Kenntnisse über den semantischen Aufbau der einzelnen Entity–Typen nicht erfordert und somit ohne Einsatz eines EXPRESS–Compilers auskommt.

Nach diesem Verarbeitungsschritt liegen die einzelnen Entity–Instanzen der Datei in ihrer Hauptspeicher–Repräsentation vor. Das Resultat des Lesevorgangs einer STEP–Datei wird somit automatisch innerhalb der hauptspeicherresidenten STEP–Datenbank gespeichert. Durch die gewählte Methode wird der überwiegende Teil der notwendigen Analyseschritte für eine Datei aus dem Anwendungsprogramm in die Bibliothek verlagert. Die Instanzierung der hauptspeicherresidenten RID wird mit Hilfe entsprechender abstrakter Datentypen in C durchgeführt, wobei die entsprechenden Werte durch den Parser nach Konvertierungen der Zeichenketten in Gleitkommazahlen, Reservierung des entsprechenden Speicherplatzes für den Zugriff auf die Attribute usw. erzeugt werden.

Das Auflösen von Referenzen auf andere Entities wird dabei auch durchgeführt, so daß vom Anwenderprogramm kein Suchaufwand geleistet werden muß. Weiterhin wird die File–Abbildung entsprechend im Hauptspeicher **linearisiert**, d.h. die Identifikations– und Referenz–Werte werden entsprechend der Stellung der Entities im File abgeglichen. Beispiel: Kommt Entity #201 als dritte Entity im File vor, wird ihr *identifier* zu #3. Entities, die auf #201 referenzieren, verlieren den Anschluß nicht. Diese Linearisierung bringt erhebliche Ersparnisse an Arbeitsspeicher, und der sonst übliche Suchaufwand entfällt.

8.2.3 Semantische Analyse

Ein Compiler muß überprüfen, ob die Quellprogramme sowohl den syntaktischen als auch den semantischen Konventionen der Quellsprache genügen. Die Überprüfung der

Semantik wird oft in die *statische* und *dynamische* Überprüfung unterteilt. Statische Überprüfungen beinhalten:

- *Typüberprüfungen,*
- *Überprüfungen auf Eindeutigkeit,*
- *auf Namen bezogene Überprüfungen.*

Die semantischen Fehler, die während der Laufzeit auftreten und durch statische Analyse nicht entdeckt werden können, beinhalten z.B.:

- *unzulässige Wertebereiche,*
- *unzulässige Benutzung parametrisierter Variablen* u.a.

Die zu überprüfende Semantik der STEP–Modelle beinhaltet auch ähnliche Fälle, wobei sich die STEP–Semantik mehr oder weniger auf EXPRESS beschränkt, d.h. die Semantik der STEP–Modelle bzw. Entities, Schemata und ihrer Instanzen in der physikalischen Datei wird durch die Modellierungssprache EXPRESS definiert bzw. festgelegt.

Da sich in der derzeitigen Gestaltung der STEP–Parser–Bibliothek die Codeerzeugung mehr oder weniger auf die Erstellung einer RID der Entity–Instanzen in der Datei beschränkt und die eigentliche 'Übersetzung' dem Anwendungsprogramm überlassen wird, können die folgenden Ansätze als Teillösungen der oben dargestellten Probleme der Modellsemantik in STEP angesehen werden:

- Abbildung der EXPRESS–Modellsemantik auf C ([Koch 91]),
- auf eine kontextfreie Grammatik [Toparlak 91a, Grafe 93] oder
- auf Prolog–Prädikate [Toparlak 91c]. (s. Abschnitt 8.1

Beispiel: Semantik geometrischer Primitiva

Das (vereinfachte) Informationsmodell für diese Primitiva, das mittels der Modellierungssprache EXPRESS spezifiziert nach dem STEP–Geometriemodell [Wilson/Kennicott 88] wird, wird in Abb. 8.2 gezeigt. Dabei werden die Entities `cartesian_point` und `polyline` so spezifiziert, daß ein Punkt mit drei Attributen des Realtyps (wobei das letztere optional anzugeben ist und daher bei der Bestimmung der Dimension (2D/3D)

berücksichtigt wird) und eine Polyline mit dem Attribut 'Liste von mindestens zwei Punkten' definiert werden.

Die formale Syntax für das Format der STEP–File–Struktur definiert nur die 'generischen' Eigenschaften der Entity–Instanzen und kann daher zur Analyse der Modellsemantik nicht verwendet werden. Für das vorliegende Beispiel wäre die Erweiterung der generischen Syntax um die neuen Syntaxregeln für die geometrischen Primitiva eine mögliche Lösung. Diese Syntaxregeln lassen sich aus der EXPRESS–Definition ableiten (Abb. 8.3). So ist der erweiterte generische Parser in der Lage, die Entity–Instanzen `cartesian_point` und `polyline` auf ihre Semantik genauer zu untersuchen. Mit dieser Methode wurden einige spezifische Parser im Rahmen des Forschungsvorhabens *"Generierung von FE–Modellen aus CAD–Struktur–Beschreibungen"* [Grafe 93] entwickelt.

8.3 STEP–Prolog–System

8.3.1 Semantische Analyse mittels Prolog

Eine der Möglichkeiten zur Überprüfung der Modellsemantik, wie sie durch die Modellierungssprache EXPRESS der STEP–Norm vorgegeben wird, ist die Abbildung der Semantik auf Prolog–Prädikate. Mit Prolog–Prädikaten können sowohl Fakten als auch Regeln definiert werden. Fakten sind i.d.R. auch Regeln, für die allerdings keine Bedingung erfüllt werden muß. Im Rahmen der vorliegenden Forschungsarbeit wurde ein System entwickelt, das in der Lage ist, aus EXPRESS–Modellen automatisch ausführbare Programme für die Semantiküberprüfung zu generieren [Toparlak 91c]. Weiterhin ermöglicht das System eine unmittelbare Ausführung der generierten Programme für die Analyse von STEP–Dateien. Die Verbindung zwischen einem STEP–File und dem dafür geeigneten EXPRESS–Modell wird mit einer Spezifikation eines Schema–Files (Entity `file_schema`) vorgegeben.

Unter Verwendung dieses Systems sehen z.B. die Prädikate für die Überprüfung der Modellsemantik der vorher erwähnten geometrischen Primitiva wie in Abb. 8.4 aus. Die Auswertung dieser Semantikregeln erfolgt dann mit einem Meta–Aufruf einer Entity als Funktion, die aus einem File gelesen wird. Die Abb.8.4 zeigt einen solchen Meta–Aufruf.

Beispiel: Modelltransformation geometrischer Primitiva

Das in den vorangegangenen Abschnitten mehrfach untersuchte Beispiel soll hier zur Verdeutlichung der oben genannten Vorteile von Prolog dienen. Nach einer geringfügigen Veränderung des Dateiformates ist es möglich,

- die Basistypen wie `integer, real, string` ohne weiteres,
- die Attributlisten in Form von Prolog–Listen sowie
- die Entity–Instanzen in Form von Prolog–Prädikaten

abzubilden. Das STEP–Parser–System stellt eine solche Konvertierung zur Verfügung, um das File analysieren, in Prolog übersetzen und als Prolog–Programm ($Prolog \equiv Programme = Daten$) ohne weiteres zu interpretieren.

Hier soll ein beispielhafter Modelltransformation der geometrischen Primitiva gezeigt werden. Das Ziel ist es, eine vereinfachte Geometrie in Form von Neutralformatdateien, s. Abb. 8.5) zuerst zu erzeugen und dann die notwendige Modelltransformation durchzuführen, um die aus dem CAD–System extrahierte Geometrie einfach an das FE–System weiterreichen zu können. Dabei wird die folgende Strategie verfolgt [Toparlak 92b]:

- Das Parser–System wird verwendet, um aus dem STEP–File ein interpretierbares Prolog–Programm zu generieren.
- Semantische Analyse mittels generierter Prolog–Prädikate.
- Ein Prolog–Interpreter führt dieses File mit Hilfe von *Transformationsregeln* aus, um ein Eingabefile für z.B. FE–Preprozessor zu erzeugen.

Die generierte Ausgabe (Abb. 8.6) wird vom Prolog–Interpreter eingelesen, und das Programm ProSM (Abb. 8.7) erzeugt die in der Abb. 8.8 dargestellte Eingabe. Die mit Hilfe des Prolog–Programms ProSM erzeugte Eingabedatei kann anschließend mit dem eigenen Preprozessor weiterverarbeitet werden, um entsprechende Verfeinerungen am Modell durchzuführen.

8.4 Regelsprache und ihre Übersetzung

Die Übersetzung der im Rahmen der vorliegenden Arbeit entworfenen problemorientierten Regelsprache DRL (*Dimensioning Rule Language*), erfolgt mittels *definite clause grammars* (DCG). Der DCG–Formalismus ist ein ziemlich viel verbreitetes Werkzeug in der Prolog Community, das meistens bei der Analyse sowie Erkennung gesprochener Sprache im KI–Bereich erfolgreich eingesetzt wird. Dieses eignet sich auch für die Implementierung von Parsern bzw. Compilern im üblichen Sinne. So wird es auch in dem vorliegenden Fall zur Realisierung eines Compilers für DRL verwendet.

Ein besonderer Vorteil ist, daß sich ein mittels DCG realisierter Compiler auch als Meta–Interpreter einsetzen läßt. Das heißt, der DRL–Compiler kann mit einem Prolog Interpreter kombiniert werden, um auf die Übersetzung vollständig zu verzichten. Nach dem

Laden des DCG–Compilers ist Prolog einfach in der Lage, DRL direkt zu interpretieren. Die jeweilige DCG–Implementation sorgt dafür, daß DRL–Code auf Anforderung intern übersetzt und gleich interpretiert wird, ohne dabei bemerkt zu werden. Die Abbildungen 8.9, 8.10 und 8.11 schildern diese effektive Form der Realisierung einer problemorientierten Programmier–Umgebung (Language, Compiler, Interpreter). Siehe dazu auch im Anhang A und B.

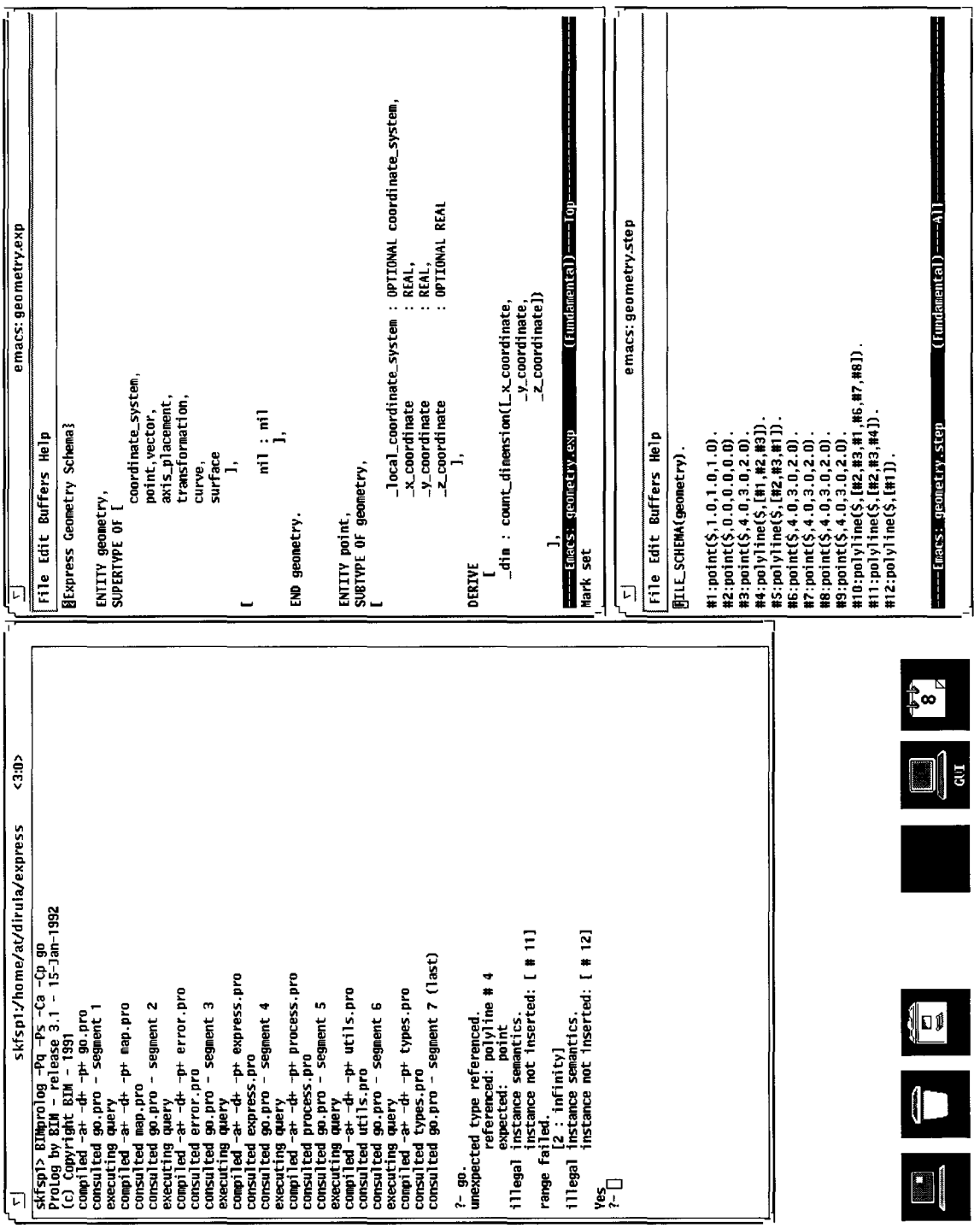
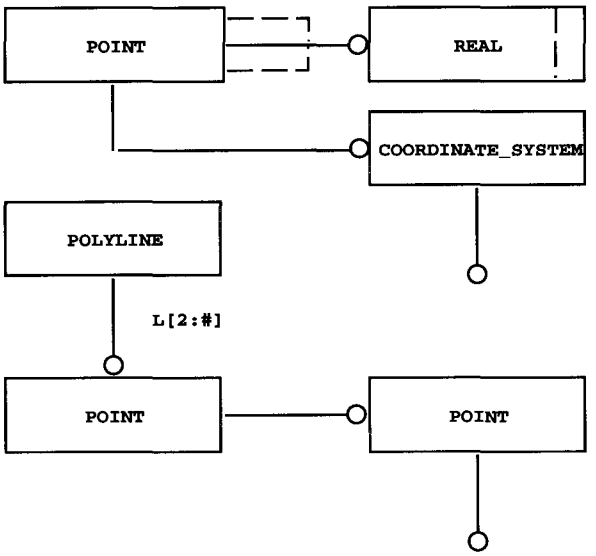


Abb. 8.1: *Arbeiten mit der RWVS/SPS–Schnittstelle*



a) EXPRESS–G–Repräsentation

```
ENTITY cartesian_point
        SUBTYPE OF (point);
        x_coordinate        :REAL;
        y_coordinate        :REAL;
        z_coordinate        :OPTIONAL REAL;
DERIVE
        space     :INTEGER
                  := coordinate_system(z_coordinate);
END_ENTITY;

ENTITY polyline
        SUBTYPE OF (bounded_curve);
        points LIST [2:#] OF cartesian_point;
END_ENTITY;
```

b) EXPRESS–Repräsentation

Abb. 8.2: *EXPRESS–Modell für Polygonzüge*

```
/* Description: adding application oriented syntax in to yacc(1)                */
/* Time-stamp: <94/03/27 18:16:55 at> */

%token                  POLYLINE        REAL

%%

application:            polyline
        |                 point
        ;                         /* and other application spec. entities */
polyline:               POLYLINE '(' optional_coordinate_system ','
                                   '(' list_of_points ')'
                        ')' ;
list_of_points:       point, point     /* minimum number of points is 2 */
        |               list_of_points, point
        ;
point:                  cartesian_point
        ;               /* and other point types such as point_on_surface */
cartesian_point:        POINT '(' optional_coordinate_system ','
                                  REAL ','
                                  REAL ','
                                  optional_REAL
                              ')' ;
optional_REAL:          '$'
        |               REAL
        ;
                                                /* optional_coordinate_system:
        ... */

%%
```

Abb. 8.3: *Abbildung der Semantik auf kontextfreie Grammatik*

```
% Predicates for the semantic check of geometrical primitiva:
% CARTESIAN_POINT and POLYLINE.
cartesian_point($,X,Y,Z):-
        real(X),
        real(Y),
        real(Z).
% The following one checks a polyline: 2 is the minimum number of
% the elements in the list of points.
polyline($,[F,S]):-
        !,
        cartesian_point(F),
        cartesian_point(S).
polyline($,[H|T]):-
        cartesian_point(H),
        polyline($,T).
% Example of a meta call as semantic check
% Last argument is not a real number
?- read(Term), Term.
@ cartesian_point($,0.00,0.00,0).
No
?-
```

Abb. 8.4: *Prolog-Prädikate für Modellsemantik*

```
...
FILE_SCHEMA('geometry.express');
#1 = PT3($, 1647.0, 0.0,   0.0);
#2 = PT3($, 1647.0, 0.0, 400.0);
#3 = PT3($,  750.0, 0.0, 400.0);
#4 = PT3($,  750.0, 0.0,   0.0);
#10= PLN($, (#1, #2, #3, #4));
...
```

Abb. 8.5: *STEP File*

```
/* Description: Operator declaration/overloading to simply read STEP,      */
/* [...] are built-in lists.                                               */
/* Time-stamp: <94/03/27 18:30:59 at>                                      */

...
op(fx,  750, '#').
op(xfx, 700, ':').
...
FILE_SCHEMA('geometry.express').
#1 : PT3($, 1647, 0,   0).                                                  10
#2 : PT3($, 1647, 0, 400).
#3 : PT3($,  750, 0, 400).
#4 : PT3($,  750, 0,   0).
#10: PLN($, [#1, #2, #3, #4]).
...
```

Abb. 8.6: *Prolog File*

```
?- printf("TITLE 'ProSM'\nELEMENTS, 75\nEND\n").
yes
?- printf("COORDINATES\n").
yes
?- #U : PT3($,X,Y,Z),
   printf("%d, %f, %f, %f, \n", X,Y,Z), fail.
no
?- printf("CONNECTIVITY\n").
yes
?- #U : PLN($,[#P1,#P2,#P3]),                                               10
   printf("%d, 75, %f, %f, %f, \n",U,P1,P2,P3),
   fail.
no
?- #U : PLN($,[#P1,#P2,#P3,#P4,#P5,#P6]),
   printf("%d, 75, %d, %d, %d, %d \n",U,P1,P2,P3,P4),
   printf("%d, 75, %d, %d, %d, %d \n",55000-U,P4,P5,P6,P1),
   fail.
no
?- printf("ENDOPTION\n").
yes                                                                         20
```

Abb. 8.7: *Die Prolog-Sitzung 'ProSM'*

```
TITLE 'ProSM'
ELEMENTS, 75,
END
COORDINATES

1, 1647., 0., 0.,
2, 1647., 0., 400.,
3, 750., 0., 400.,
4, 750., 0., 0..
$ ...                                                          10
CONNECTIVITY

1, 75, 1, 2, 3, 4,
$ ...
ENDOPTION
```

Abb. 8.8: *Die generierte MARC-Eingabe*

```
[
        RULE,
                [schottbeplattung,beplattung],
        IF,
                schiffstyp(tanker),
        THEN,
                solve(t=a*b*c*sqrt(t_k))
].
```

Abb. 8.9: *Beispiel DRL–Eingabe*

```
:- compatibility.

drl -->
        ['RULE'],          drl_name(N),
        ['IF'],            drl_head(H),
        ['THEN'],          drl_body(B),
        {
                drl_compile(N,H,B)
                /* call(H),
                assert(B) */                                    10
        }.

drl_name(R) -->
        [N], {R=N}.

drl_head(R) -->
        [H], {R=H}.

drl_body(R) -->
        [B], {R=B}.                                             20


drl_compile([],H,B) :-
        ! /*,, assert((B :- H)) */
        .
drl_compile([N|T],H,B) :-
        !,
        assert((rule(N) :- H, B)),
        drl_compile(T,H,B).
drl_compile(Any,H,B) :-                                         30
        assert((rule(Any) :- H, B)).
```

Abb. 8.10: *Definite clause grammars für DRL*

```
rule(schottbeplattung, _t) :-
        rule(beplattung, _t).
rule(beplattung, _t) :-
        schiffstyp(tanker),
        solve(_t = a*b*c*sqrt(t_k)).
```

Abb. 8.11: *Generiertes DRL–Programm*

Kapitel 9

Zusammenfassung und Ausblick

Die vorliegende Arbeit hatte das Ziel, die Bauvorschriften wie zum Beispiel zur Dimensionierung der Stahlschiffskonstruktionen auf einfache Weise zu definieren sowie zu interpretieren, und dadurch die Wartungsfähigkeit eines sogenannten Regelwerks zu erhöhen, indem eine problemorientierte und direkt ausführbare Sprache das aufwendige Programmieren mittels Programmiersprachen wie FORTRAN und **C** für den Anwender überflüssig macht. Der vorliegende Systemprototyp soll den Programmierer (Entwickler) sowie den Endnutzer (Anwender) beim Programmieren eines neuen oder dem Modifizieren (*update*) eines vorhandenen Regelwerks unterstützen.

Es wurde gezeigt, wie ein Dimensionierungsprozessor implementiert werden kann, der die erforderliche Information durch geeignete und flexible Schnittstellen aus dem Produktmodell abfragen und weiterverarbeiten kann. Die Verwendung der wissensbasierten Methoden erleichtert die Lösung des Problems der aufwendigen Programmierung von Vorschriften und ihre Pflege.

So ist es gelungen, einen akzeptablen Prototyp zu entwickeln, der die Produktdaten (*Stahlstruktur* als Beispiel) verwendet, um Abfragen zu stellen, und somit die Dimensionierungsregeln interpretiert. Dieser wird mit Hilfe eines Prolog–Interpreters ausgeführt. Dabei wird zwischen Daten und Programmen **nicht** unterschieden. Sie sind Objekte gleicher Natur, zu deren Definition entweder Produktdaten (*Fakten*) oder Vorschriften (*Regeln*) herangezogen werden. Gerade diese Eigenschaft macht den **wissensbasierten** Charakter des Prototyps aus. Die Lösung eines mit Hilfe der Regeln beschriebenen Dimensionierungsproblems ist nichts weiter als die Auswertung der für dieses Problem geeigneten Regeln mit dem dafür erforderlichen Datensatz. Fehlt dieser erforderliche Datensatz, so wird ein Dialog ausgeführt, in dem der Benutzer nach Fakten oder Regeln gefragt wird, ohne den Ablauf des Dimensionierungsprozesses abzubrechen. Der Prototyp unterscheidet sich vor allem hinsichtlich dieser Gesichtspunkte wesentlich von den herkömmlichen Systemen.

Da die Dimensionierungsregeln letztlich als direkt aufrufbare Prolog–Prädikate generiert werden, wird eine Verbindung zwischen dem Prolog–System und dem Produktmodell benötigt, die mit Hilfe des STEP–Parser–Systems zur Verfügung gestellt wird. Bei dem derzeitigen Prolog–Prototyp besteht die Möglichkeit, das Produktmodell mit Hilfe des SPS auf Prolog–Prädikate abzubilden, um eine in sich geschlossene Arbeitsplattform für das Regelwerk zu bilden.

Das alt–bekannte Thema, daß Prolog–basierende Methoden hauptspeicherintensiv sind und daher mit ernsthaften Situationen nicht klar kommen, ist kein primäres Problem mehr, da moderne Prolog–Implementationen es möglich machen, direkt Maschinencode zu erzeugen, dessen Effizienz durchaus vergleichbar mit den der maschinennahen Programmiersprachen wie **C** bzw. **C++** ist [Gabber 90]. Einige Referenz–Implementationen für Prolog wären z.B. ProLog *by* BIM [BIM 93] oder Aquarius Prolog[1] mit nennenswerten Performance–Werten von ca. 2 bis 10 MLIPS[2].

Das Produktmodell *Stahlstruktur* kann in der realen Anwendung (mehrere 10000(0) Bauteile) für seine Datenverarbeitung mehrere 100MB Speicherplatz anfordern, was den Einsatz hauptspeicherbasierender Methoden erschwert. Die heute verfügbaren Datenbanksysteme mit relationalem Datenmodell sind aufgrund der hohen Komplexität der notwendigen Modellierung ungeeignet [Bronsart 90, Koch 91].

Datenbanken basierend auf dem objektorientierten [Kim/Lochovsky 89] oder dem erweiterten relationalen Modell [Stonebraker/Rowe 86] befinden sich gerade in der aktuellen Entwicklung.

Auf der anderen Seite ist zu erwarten, daß derzeitige und zukünftige Entwicklungen in der Hardware–Branche die genannten Nachteile des entwickelten Prototyps teilweise reduzieren und seine Verwendbarkeit für reale Anwendungsfälle verbessern. Beispiele dafür sind Rechner der fünften Generation, die Prolog als Maschinensprache enthalten.

Mit der Verwendung von wissensbasierten Systemen auf der Basis von Prolog eröffnet sich im übrigen für die weitere Gestaltung und den Ausbau der Bemessungsregeln ein neues, weites Feld. Es ist davon auszugehen, daß ganz allgemein die Übertragung von wissenschaftlich technischen Forschungserkenntnissen in verwertbare, für den Konstrukteur einsetzbare Bauvorschriften in Zukunft mit den realisierten Werkzeugen besonders wirkungsvoll erfolgen kann.

[1] University of California at Berkeley. Das Projekt 'Aquarius Prolog'.

[2] Millionen logische Inferenzen pro Sekunde, was für die Interpretation einiger tausenden Regeln durchaus ausreichen wird.

Literaturverzeichnis

[Aho/Sethi/Ullman 88] A. V. Aho, R. Sethi, J. D. Ullman. *Compilerbau*, Band I+II. Addison–Wesley, Inc., Bonn, 1988.

[BIM 93] BIM Information Technology, Everberg. *ProLog by BIM Version 4.0 Manual*, Juli 1993.

[Brach/Pielmeier 88] U. Brach, E. Pielmeier. Ein Werkzeugsystem zur Entwicklung von Expertensystemen in Prolog. *unix/mail*, 6(1):55–63, Januar 1988.

[Bratko 87] I. Bratko. *Prolog for Artificial Intelligence Programming*. Addison–Wesley, Inc., Reading, 1987.

[Bronsart 90] R. Bronsart. *Ein Datenmodell für schiffbauliche Stahlstrukturen*. Dissertation, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, *Fortschritt–Berichte*, Nr. 21/20, VDI–Verlag, Düsseldorf, 1990.

[Browston/Farrell/Kant/Martin 88] Lee Browston, R. Farrell, Elaine Kant, Nancy Martin. *Programming Expert Systems in OPS5*. Addison–Wesley, Inc., 1988.

[Buchanan/Shortliffe 84] B. G. Buchanan, E. H. Shortliffe. *Rule–Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison–Wesley, Inc., Massachusetts, 1984.

[Clocksin/Mellish 84] W. F. Clocksin, C. S. Mellish. *Programming in Prolog*. Springer–Verlag, Berlin, 2. Ausgabe, 1984.

[Fischer 88] R. Fischer. *PC–Expertensysteme*. Markt & Technik, München, 1988.

[Gabber 90] Eran Gabber. Developing a Portable Parallelizing Pascal Compiler in Prolog. In Leon Sterling (Hrsg.), *The Practice of Prolog*, Logic Programming Series. MIT Press, Cambridge, Massachusetts, 1990.

[Gibbins 90] Peter Gibbins. *Logic with Prolog*. Oxford Press, 1990.

[GL 86] Vorschriften für Klassifikation und Bau von stählernen Seeschiffen. Kapitel 2 – Schiffskörper, Germanischer Lloyd, Hamburg, 1986.

[GL 92] Klassifikations– und Bauvorschriften. Code I – Schiffstechnik, Teil 1 – Seeschiffe, Germanischer Lloyd, Hamburg, 1992.

[Grafe 93] W. Grafe. Generierung von FE–Modellen aus CAD–Strukturbeschreibungen, Teil 1. Bericht zum BMFT–Vorhaben MTK 0442C4, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, Mai 1993.

[Hansen/Mühlbacher/Neumann 93] H. R. Hansen, R. Mühlbacher, G. Neumann. *Begriffsbasierte Integration von Systemanalysemethoden*, Band 53, *Betriebs– und Wirtschaftsinformatik*. Physica–Verlag, Heidelberg, 1993.

[Harmon/King 85] P. Harmon, D. King. *Expertsystems – Artificial Intelligence in Business*. Addison–Wesley, Inc., New York, 1985.

[Hayes-Roth/Waterman/Lenat 83] F. Hayes-Roth, D. A. Waterman, D. B. Lenat (Hrsg.). *Building Expert Systems*, Band 1, *Teknowledge Series in Knowledge Engineering*. Addison–Wesley, Inc., Reading, Massachusetts, 1983.

[Heller 91] D. Heller. *XView Programming Manual*. Addison–Wesley, Inc., 1991.

[Jensen 92] H. Jensen. *Überprüfung der Anwendbarkeit und Einhaltung technischer Regelwerke sowie Dimensionierung an Hand von Regelwerken mit Hilfe von Rechnern*. Dissertation, Bericht Nr. 528, Institut für Schiffbau der Universität Hamburg, Dezember 1992.

[Johnson 75] S. C. Johnson. `yacc` – yet another compiler compiler. Comp. Science Techn. Report 32, AT&T, Murray Hill, 1975.

[Karagiannis 87] Dimitris Karagiannis. *Repräsentation und Verarbeitung von Wissensstrukturen: Das hybride System Kanon*. Dissertation, Technische Universität Berlin, 1987.

[Kary 86] Danial D. Kary. *The TRC Reference Manual*. North Dakota State University, Computer Science Department, Fargo, 1986.

[Kerschberg 86a] Larry Kerschberg (Hrsg.). *Expert Database Systems: Proceedings from the First Int. Workshop*. The Benjamin/Cummings Publishing Company Inc., Reading, 1986.

[Kerschberg 86b] Larry Kerschberg (Hrsg.). *Proceedings from the First Int. Conf. on Expert Database Systems*. The Benjamin/Cummings Publishing Company Inc., Reading, 1986.

[Kim/Lochovsky 89] W. Kim, F. H. Lochovsky (Hrsg.). *Object–Oriented Concepts, Databases, and Applications*. Addison–Wesley, Inc., Reading, Massachusetts, 1989.

[Koch 89] T. Koch. *SwStd – Standards und Richtlinien für die Software–Entwicklung, Handbuch, Version 1.1.* Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, Januar 1989.

[Koch 90] T. Koch. Viewer – Polygonal Object Viewer. Bericht skb–90–tk–1, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, 1990.

[Koch 91] T. Koch. *Austausch von Produktdaten zwischen CAD–Modelliersystemen am Beispiel schiffbaulicher Stahlstrukturbeschreibungen.* Dissertation, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, *Fortschritt–Berichte*, Nr. 54/20, VDI–Verlag, Düsseldorf, 1991.

[Koschman/Evens 88] T. Koschman, M. W. Evens. Bridging the Gap between Object–Oriented and Logic Programming. *IEEE Software*, 1988.

[Kowalski 88] R. A. Kowalski (Hrsg.). *Logical Programming*, Band I+II. The MIT Press, Reading, 1988.

[Lamport 86] Leslie Lamport. LaTeX: *A Document Preparation System User's Guide & Reference Manual.* Addison–Wesley, Inc., Reading, 1986.

[Lauer 89] M. Lauer. *NEXPERT Object.* Addison–Wesley, Inc., Bonn, 1989.

[Mann 87] P. Mann. *LALR User's Manual.* LALR Research, Knoxville, November 1987.

[Miller 90] J. D. Miller. *An OPEN LOOK at UNIX – A Developer's Guide to X.* Prentice–Hall, Inc., Redwood City, 1990.

[Minker/Nicolas 83] J. Minker, J. M. Nicolas. On Recursive Axioms in Deductive Databases. *Information Science*, 3(1):1–13, 1983.

[Minsky 75] M. Minsky. A Framework for Representing Knowledge. In Patrick H. Winston (Hrsg.), *The Psychology of Computer Vision.* McGraw–Hill, Inc., New York, 1975.

[Neumann 88] G. Neumann. *Meta–Programmierung und Prolog.* Addison–Wesley, Inc., Bonn, 1988.

[O'Keefe 90] Richard A. O'Keefe. *The Craft of Prolog.* Logic Programming. The MIT Press, Cambrodge, 1990.

[Paxson 89] V. Paxson. `flex` – *Fast Lexical Analyzer Generator (*`lex` *replacement)*, 1989. Version 2.1.

[Rubin 86] P. Rubin. `bison` – *GNU Project Parser Generator* `yacc` *replacement).* Free Software Foundation, Waterloo, Juli 1986.

[Scheifler/Gettys 86] R. W. Scheifler, Jim Gettys. The X Window System. MIT Project Athena, MIT, Cambridge, Oktober 1986.

[Sethi 90] R. Sethi. *Programming Languages – Concepts and Constructs.* Addison–Wesley, Inc., 1990.

[STE 89] STEP Physical File Structure, Mapping to Physical, First Draft Proposal. Annex A, Annex C, ISO TC184/SC4/WG1, Februar 1989.

[Sterling/Shapiro 86] Leon Sterling, Ehud Shapiro. *The Art of Prolog – Advanced Programming Techniques.* Logic Programming. The MIT Press, Cambridge, 1986.

[Stonebraker/Rowe 86] M. Stonebraker, L. A. Rowe. The Design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 340–355, 1986.

[Stroustrup 94] Bjarne Stroustrup. *Die Programmiersprache C++.* Addison–Wesley, Inc., Bonn, 1994.

[Tietgen/$^+$ 84] B. Tietgen, et al. GLRULES – Dialog Programmpaket für die Vorschriften für Klassifikation und Bau von stählernen Seeschiffen des Germanischen Lloyd. (unveröffentliche Programmdokumentation), Oktober 1984.

[Toparlak 91a] İ. A. Toparlak. Datengenerierung für Finite–Elemente–Berechnungen mit Hilfe von CAD Systemen. Bericht Nr. 230/1991, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, FORSCHUNGSZENTRUM DES DEUTSCHEN SCHIFFBAUS, April 1991.

[Toparlak 91b] İ. A. Toparlak. Parser Toolkit for Processing ISO/STEP Product Models. Bericht Version 1.5, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, März 1991.

[Toparlak 91c] İ. A. Toparlak. A Prolog Meta Program for the Semantic Check of STEP/EXPRESS Product Models. Bericht, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, Hamburg, November 1991.

[Toparlak 92a] İ. A. Toparlak. An Enhanceable Software Development Library for User Specific Application Frameworks with ISO/STEP Models. Bericht Level 1, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, Hamburg, Januar 1992.

[Toparlak 92b] İ. A. Toparlak. Logic Programming Techniques to Transform Geometric Models into Finite Element Models. Bericht, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, Hamburg, Januar 1992.

[Toparlak 93a] İ. A. Toparlak. DRLi – Dimensioning Rule Language to Prolog Compiler. Bericht, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, April 1993.

[Toparlak 93b] İ. A. Toparlak. Entwicklung eines wissensbasierten Systems zur Spezifikation und Interpretation der Dimensionierungsvorschriften für schiffbauliche Stahlstruktur. Bericht Nr. 249/1993, Technische Universität Hamburg–Harburg, Arbeitsbereich Schiffstechnische Konstruktionen und Berechnungen, FORSCHUNGSZENTRUM DES DEUTSCHEN SCHIFFBAUS, Juni 1993.

[Ullmann 88] Jeffrey D. Ullmann. *Database and Knowledge – Base Systems*, Band 1. Computer Science Press, Inc., Rockville, 1988.

[Ullmann 89] Jeffrey D. Ullmann. *Database and Knowledge – Base Systems*, Band 2. Computer Science Press, Inc., Rockville, 1989.

[Wiener/Pinson 88] Richard S. Wiener, Lewis J. Pinson. *An Introduction to Object–Oriented Programming and C++*. Addison–Wesley, Inc., Reading, 1988.

[Wilson/Kennicott 88] P. R. Wilson, P. R. Kennicott. STEP/PDES, Testing Draft: PART1 of IPIM(Denver). Document N241, ISO TC184/SC4/WG1, Mai 1988.

[Winston 84] Patrick H. Winston. *Artificial Intelligence*. Addison–Wesley, Inc., Reading, 2. Ausgabe, 1984.

[Wirth 83] N. Wirth. *Algorithmen und Datenstrukturen*. Teubner–Verlag, Stuttgart, 1983.

Anhang A

Wasserdichte Schotte in DRL

Auszüge aus dem GL als Ausgangspunkt

DRL Input

B. Abmessungen

1. Allgemeines, Begriffsbestimmung

1.1 Sollen Laderäume auch zur Aufnahme von Ballastwasser benutzt werden, so müssen ihre Schotte auch den Anforderungen des Abschnitts 12, C. entsprechen.

1.2 Die Festigkeit von Schotten, die Erzladeräume begrenzen, muß auch den Anforderungen des Abschnitts 23 entsprechen.

1.3 Begriffsbestimmung

t_K = Korrosionszuschlag gemäß Abschnitt 3, K.1.

a = Steifenabstand in [m]

p = 9,81 h [kN/m^2]

f = $235/R_{eH}$

...

2. Schottbeplattung

2.1 Die Dicke der Schottbeplattung darf nicht kleiner sein als:

$$t = c_p \cdot a \cdot \sqrt{p} + t_K [mm]$$

$$t_{min} = 6,0\sqrt{f} [mm]$$

...

Tab. A.1: *Abschnitt 11 – Wasserdichte Schotte*

C. Tanks mit großen Längen oder Breiten
Breitensuche mit dem Ziel 'Beplattung' im Abschnitt 12: *B. Abmessungen* *und* *2. Beplattung*

...

B. Abmessungen

1. Begriffsbestimmung

k = Werkstoffkennziffer gemäß Abschnitt 2, B.2.

p = Druck in [kN/m^2] gemäß Abschnitt 4, D.1.

...

2. Beplattung

2.1 Die Plattendicke darf nicht kleiner sein als:

$$t_1 = 1,1 \cdot a \cdot \sqrt{p \cdot k} + t_K[mm]$$

$$t_2 = 0,9 \cdot a \cdot \sqrt{p_2 \cdot k} + t_K[mm]$$

...

Tab. A.2: *Abschnitt 12 – Tankverbände*

C. Erzschiffe

3. Quer- und Längstschotte

3.2 Laderaumschotte, die dem Erzdruck ausgesetzt sind, sind nach B.8. zu bestimmen. Die Seitenlängstschotte müssen mindestens die für Tankschiffe vorgeschriebenen Abmessungen erhalten.

...

B. Massengutschiffe

8. Laderaumschotte

8.1 Die Abmessungen der Laderaumschotte sind wie folgt zu bestimmen:

.1 Die Plattendicke darf nicht kleiner sein als:

$$t_1 = 1,1 \cdot a \cdot \sqrt{p_{Lh} \cdot k} + t_K [mm]$$

a = Steifenabstand in [m]

Die Plattendicke darf jedoch nicht kleiner sein als für ein wasserdichtes Schott nach Abschnitt 11 erforderlich ist bzw. 9 mm nicht unterschreiten.

...

Tab. A.3: *Abschnitt 23 – Verstärkungen für Schwergutladung, Massengut und Erzschiffe*

```
RULE    11.B.1.1 | Schottbeplattung WITH Dicke
IF
        Laderaumnutzung IS Ballastwasser
THEN
        RULE 12.C
END

RULE    11.B.1.2 | Schottbeplattung WITH Dicke
IF
        Schiffstyp IS Erzschiffe
THEN
        RULE 23
END

RULE    11.B.2.1 | Schottbeplattung WITH Dicke
        SOLVE t = c_p * a * sqrt(p) + tK
OR
        SOLVE t_min = 6.0 * sqrt(f)
END

RULE    12.C | Schottbeplattung WITH Dicke
        RULE 12.B.2
END

RULE    12.B.2 | Schottbeplattung WITH Dicke
        RULE 12.B.2.1
END

RULE    12.B.2.1 | Schottbeplattung WITH Dicke
        SOLVE t_1 = 1.1 * a * sqrt(p * k) + tK
OR
        SOLVE t_2 = 0.9 * a * sqrt(p_2 * k) + tK
END

RULE    23 | Schottbeplattung WITH Dicke
        RULE 23.B.8.1
END

RULE    23.B.8.1 | Laderaumschotte  WITH Plattendicke
        SOLVE t_1 = 1.1 * a * sqrt(p_Lh) + t_K
END
```

Tab. A.4: *DRL Input — Abschnitte 11, 12, 23 aus dem Regelwerk GL*

Anhang B

Ablauf Wasserdichte Schotte

Übersetzung der DRL–Eingabe mittels DRL–Meta–Interpreter

Bemerkung: Mittels DCG realisierter DRL–Compiler bzw. –Interpreter wurde im Kapitel 8 kurz dargestellt. Die Einzelheiten des Meta–Interpreters sind [Toparlak 93a] zu entnehmen.

Direkte Ausnutzung der generierten Prädikate als ausführbares Programm

Bemerkung: Die nach automatischer Anforderung eingegebenen Werte während des Ablaufs des in der Tabelle B.1 gegebenen Programms haben mit der Realität nichts zu tun und sind nur zufällig erfunden worden, um die Interaktion mit dem Programm rein prinzipiell schildern zu können.

Das generierte Modell nach dem Programmablauf

```
rule('11.B.1.1', Schottbeplattung, _r) :-
        ask(Laderaumnutzung, Ballastwasser),
        rule('12.C', Schottbeplattung, _r).

rule('11.B.1.2', Schottbeplattung, _r) :-
        ask(Schiffstyp, Erzschiffe),
        rule('23', Schottbeplattung, _r).

rule('11.B.2.1', Schottbeplattung, _r) :-
        solve(_t = c_p * a * sqrt(p) + tK),
        _t = _r.

rule('11.B.2.1', Schottbeplattung, _r) :-
        solve(_t_min = 6.0 * sqrt(f)),
        _t_min = _r.

rule('12.C', _g, _r) :-          /* _g -- Goal, _r -- Result */
        rule('12.B.2', _g, _r).

rule('12.B.2', _g, _r) :-
        rule('12.B.2.1', _g, _r).

rule('12.B.2.1', Schottbeplattung, _r) :-
        solve(_t_1 = 1.1 * a * sqrt(p * k) + tK),
        _t_1 = _r.
rule('12.B.2.1', Schottbeplattung, _r) :-
        solve(_t_2 = 0.9 * a * sqrt(p_2 * k) + tK),
        _t_2 = _r.

rule('23', Schottbeplattung, _r) :-
        rule('23.B.8.1', Laderaumschotte, _r1),
        _r1 = _r.

rule('23.B.8.1', Laderaumschotte, _r) :-
        solve(_t_1 = 1.1 * a * sqrt(p_Lh) + t_K),
        _t_1 = _r.
```

Tab. B.1: *GL Abschnitte 11, 12 und 23 in Prolog*

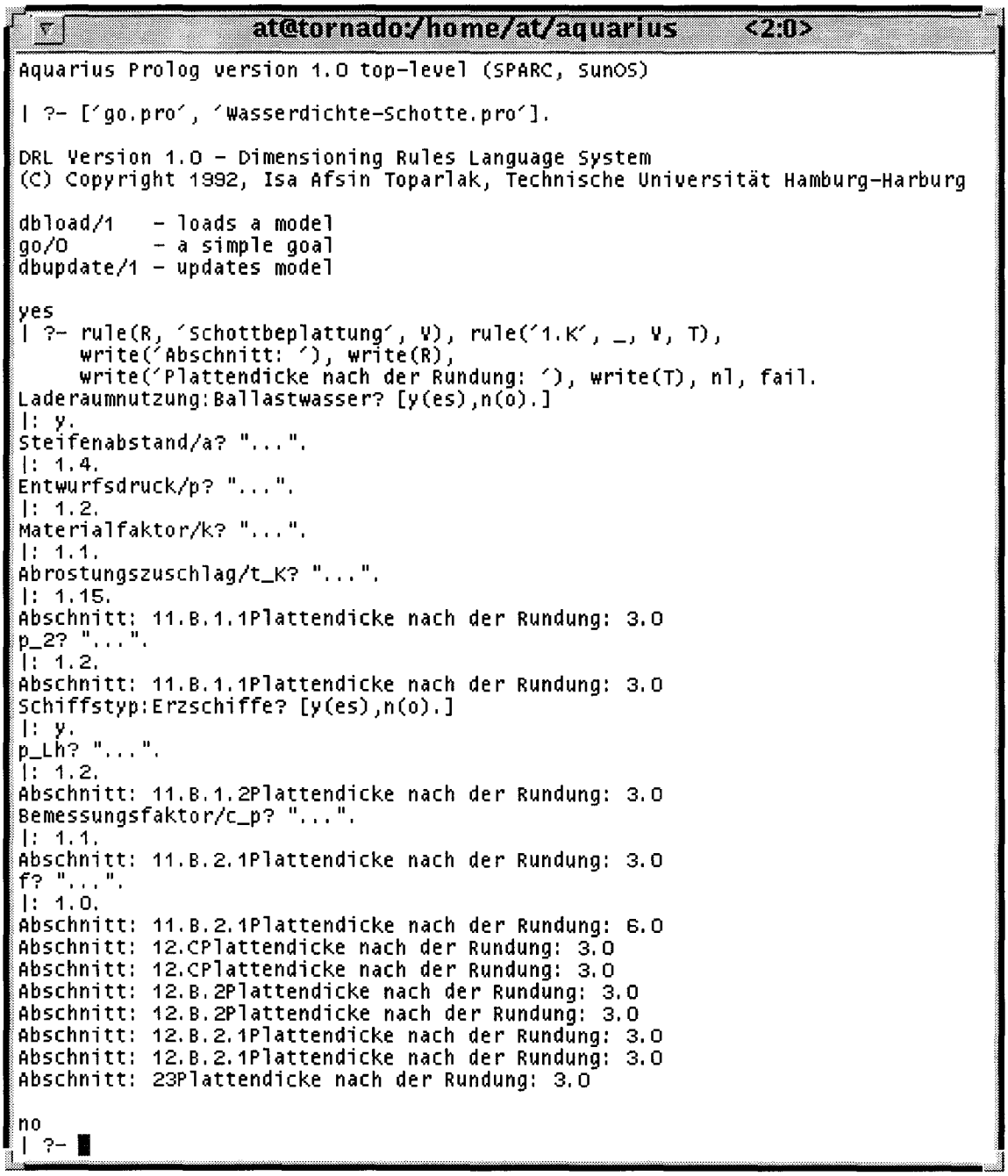


Abb. B.1: *Interaktiver Ablauf des generierten Programms*

```
#1  : Schiffstyp(Standard).
#5  : Bemessungsfaktor(1.2).
#6  : Steifenabstand(1.2).
#7  : Druckhöhe(1.2).
#8  : Materialfaktor(1.5).
#9  : Abrostungszuschlag(1.2) .
#10 : Laderaumnutzung(Ballastwasser) .
#11 : Entwurfsdruck(1.2).
#12 : p_2(1.2).
#13 : Druckhöhe_20(1.2).
#15 : p_Lh(3.4).
#17 : Schiffstyp(Erzschiffe) .
```

Abb. B.2: *Das generierte Eingabemodell für die angesprochenen Regeln*

Anhang C

Tabelle Systemprädikate

Predicate/Arity	Argument:Type	Summary
rule/4	*arg1*: Atom	Category
	arg2: Atom	Goal
	arg3: Any	Input
	arg3: Any	Output
rule/3		as rule/4 but arg1
rule/2	*arg1*: Atom	Goal (local)
	arg2: Any	Input/Output (Accumulator)
ask/2	*arg1*: Term	Query (partially unified)
	arg2: Any	Answer
ask/1	*arg1*: Any	Query/Answer
table/2	*arg1*: Atom	Thing
	arg2: Free	Description (to Atom unified)
solve/1	*arg1*: Non-ground	Equation
insert/2	*arg1*: Term	Attribute of Id
	arg2: Any	Value
query/2		Counterpart to insert/2
range/2	*arg1*: Term	Entity
	arg2: Term	Type/Value Domain
text/2	*arg1*: Term	Entity
	arg2: Atom	Description