

BABYLON

Spezifikation und Konstruktion

Forschungsgruppe Expertensysteme ¹
Gesellschaft für Mathematik und Datenverarbeitung mbH
Institut für Angewandte Informationstechnik
Forschungsgruppe Expertensysteme
Postfach 1240
D-5205 Sankt Augustin 1
Tel. 02241-14-2679

März 1988

¹Dieser Bericht entstand im Rahmen des Verbundvorhabens WEREX, das zum Teil mit Mitteln des Bundesministers für Forschung und Technologie unter dem Förderkennzeichen ITW8505A2 finanziert wird.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Zielsetzung und Überblick	1
1.2	Anforderungen	1
1.2.1	Allgemeine vs. spezielle Werkzeugsysteme	1
1.2.2	Geschlossene vs. offene Systeme	2
1.2.3	Anforderungen an das Architekturkonzept	3
1.2.4	Probleme bei der Portierung von Expertensystemen	4
1.3	Verwendete Hilfsmittel und Notationskonventionen	4
2	Systemspezifikation	6
2.1	Einordnung	8
2.2	Funktionsbeschreibung	9
2.3	Systemschnittstelle	9
2.3.1	Systemschnittstelle zum Metaprozessor	10
3	Systemkonstruktion	11
3.1	Konfiguration einer Wissensbasis	11
3.2	Die Inter-Prozessor-Kommunikation	12
3.3	Prozessor-Mixin Konventionen	17
3.4	Prozessor-Flavor Konventionen	17
3.5	Sprachbehandlung	19
3.6	Arbeitsmodi von Prozessoren	20
3.7	Checkliste	21
4	Spezifikation des Regel-Prozessors	23
4.1	Anforderungen	23
4.2	Einordnung	23
4.3	Funktionsbeschreibung	24
4.3.1	Regeln und Regelpakete	24
4.3.2	Auswertungsstrategien	24
4.3.3	Der IF-Teil von Regeln	26
4.3.4	Der THEN-Teil von Regeln	28
4.4	Schnittstellenbeschreibung	29
4.4.1	basic-rule-mixin	29
4.4.2	mini-rule-mixin	30
4.4.3	normal-rule-mixin	30

5	Konstruktion des Regel-Prozessors	31
5.1	Übersicht	31
5.2	Zerlegung	31
5.3	Integration ins Gesamtsystem	33
5.4	Der Basic-Rule-Processor	33
5.4.1	rule-base	33
5.4.2	data-base	35
5.4.3	rule-interpreter	36
5.5	Der Mini-Rule-Processor	39
5.5.1	Der Normal-Rule-Processor	40
5.6	Importierte Leistungen	41
5.7	Exportierte Leistungen	41
6	Spezifikation des Frame-Prozessors	43
6.1	Anforderungen	43
6.2	Einordnung	43
6.3	Funktionsbeschreibung	44
6.3.1	Frames	44
6.3.2	Behaviors	45
6.3.3	Instanzen	46
6.3.4	Standard-Properties für Frameslots	46
6.3.5	Aktive Werte	48
6.4	Schnittstellenbeschreibung	50
6.4.1	Funktionen für Objekte	50
6.4.2	Nachrichten für Objekte	51
6.4.3	Frame-Referenzen	54
6.4.4	Nachrichten für den Frame-Prozessor	54
7	Konstruktion des Frame-Prozessors	55
7.1	Übersicht	55
7.2	Zerlegung	55
7.3	Integration ins Gesamtsystem	57
7.4	Der Basic-Frame-Processor	57
7.4.1	frame-base	57
7.4.2	frames	58
7.4.3	frame-interpreter	59
7.5	Mini-Frame-Processors	59
7.5.1	poss-val-mixin	59
7.6	Normal-Frame-Processors	60
7.6.1	active-value-mixin	60
7.7	Importierte Leistungen	61
7.8	Exportierte Leistungen	61
8	Spezifikation des Prolog-Prozessors	62
8.1	Anforderungen	62
8.2	Einordnung	62
8.3	Funktionsbeschreibung	63

8.3.1	Klauseln und Klauselmengen	63
8.3.2	System-Prädikate (Build-In Predicates)	64
8.3.3	Externe Prämissen (Frame-Referenzen)	67
8.3.4	Beweisprozeß	68
8.3.5	Trace	69
8.4	Schnittstellenbeschreibung	69
8.4.1	basic-prolog-mixin	69
8.4.2	mini-prolog-mixin	71
8.4.3	normal-prolog-mixin	72
9	Konstruktion des Prolog-Prozessors	73
9.1	Übersicht	73
9.2	Zerlegung	73
9.3	Integration ins Gesamtsystem	75
9.4	Der Basic-Prolog-Processor	75
9.4.1	Klauselverwaltung	76
9.4.2	Klausel-Transformation und Unifikation	76
9.4.3	Der Beweisprozeß	77
9.5	Der Mini-Prolog-Processor	78
9.6	Importierte Leistungen	79
9.7	Exportierte Leistungen	79
10	Spezifikation der Benutzerschnittstelle	80
10.1	Anforderungen	80
10.2	Einordnung	80
10.3	Funktionsbeschreibung	81
10.4	Schnittstellenbeschreibung	82
10.4.1	Kommunikation über den Dialogstream	82
10.4.2	Spezielle Kommunikationsformen	82
10.4.3	Auswahl aus einem Menü	83
10.4.4	Kommunikation über spezielle Fenster	83
10.4.5	Schnittstelle zum Benutzer	83
11	Konstruktion der Benutzerschnittstelle	84
11.1	Übersicht	84
11.2	Zerlegung	84
11.3	Das Basisinterface	85
11.4	Das Minimalinterface	86

Abbildungsverzeichnis

2.1	Definition einer Konfiguration	7
2.2	Nutzung einer Konfiguration in einer Wissensbasis	7
2.3	Integration der Formalismen in BABYLON	9
3.1	Architektur einer Konfiguration	13
3.2	Kommunikation der Prozessoren	14
3.3	Definitionen zur Typerkennung beim Frame-Prozessor	15
3.4	Integration des Regel-Prozessors ins Gesamtsystem	18
3.5	Direkter Zugang zum Regel-Prozessor	18
3.6	Zusätzliche Definition von Prozessorleistungen als Funktion	19
3.7	Definition sprachabhängiger Teile im Regel-Prozessors	20
3.8	Makros zur Sprachbehandlung	21
4.1	Regelpaketsyntax	24
4.2	Regelsyntax	24
5.1	Architektur des Regel-Prozessors	32
5.2	Struktur von Regelpaketen	34
5.3	Vorwärtsauswertung mit DO-ALL-Kontrollstrategie	36
7.1	Architektur des Frame-Prozessors	56
7.2	Basiskomponenten der Frames	57
9.1	Architektur des Prolog-Prozessors	74
9.2	Goal-Repräsentation	75
11.1	Architektur des Minimalinterfaces	85

Kapitel 1

Einleitung

1.1 Zielsetzung und Überblick

Dieser Bericht enthält die funktionale Spezifikation und Konstruktionsbeschreibung von **BABYLON**. **BABYLON** stellt eine Entwicklungs- und Ablaufumgebung für Expertensysteme dar [PriBre85]. Die vorliegende Beschreibung basiert auf einer Überarbeitung des ersten Prototyps von **BABYLON** im Hinblick auf leichtere Portierbarkeit und größere Freiheiten bei der Konfiguration des Systems. Ziel dieses Berichtes ist es, einerseits die Portierung von **BABYLON**, das auf Lispmaschinen der Familie Symbolics 36xx in ZetaLisp entwickelt wurde, nach CommonLisp in einer UNIX- und MS-DOS-Umgebung zu unterstützen und andererseits eine Basis für die koordinierte Weiterentwicklung von **BABYLON** in verschiedenen Projekten zu liefern.

Dieses einführende Kapitel enthält noch einmal die Anforderungen, die wir an ein Expertensystemwerkzeug stellen. Am Ende bringen wir einige Hinweise zu Hilfsmitteln und Konventionen, die wir in den weiteren Kapiteln verwenden. Dabei wird zuerst die Spezifikation einer **BABYLON**-Komponente in einem Kapitel dargestellt, gefolgt von einem Kapitel, das die Konstruktion der entsprechenden Komponente enthält. Wir beginnen mit der funktionalen Spezifikation des Metaprozessors von **BABYLON**. Dieses Kapitel stellt u.a. dar, wie die Forderung nach hybrider Wissensrepräsentation durch die Architektur in **BABYLON** erfüllt wird. Die Spezifikationen aller anderen Subsysteme setzen es für ihr Verständnis voraus. Danach folgen die Kapitel zu den drei bisher in **BABYLON** realisierten Interpretern von Wissensrepräsentationsformalismen: Produktionsregeln, Frames und Prolog. Den Abschluß bilden die Kapitel zur Benutzerschnittstelle.

1.2 Anforderungen

1.2.1 Allgemeine vs. spezielle Werkzeugsysteme

Bei Knowledge-Engineering-Werkzeugen kann man nach [Sys85] zwischen zwei Arten von Systemen unterscheiden. Die einen sind für den Einsatz in engen Problemfeldern gedacht. Ein in diesem Sinne *spezielles* System ist z.B. **MED1** [Pup83], das auf diagnostische Aufgaben im medizinischen Bereich abgestimmt ist. Die zweite Art sind Systeme, die vom methodischen Funktionsumfang her den Anspruch erheben, unabhängig von einzelnen Anwendungsfeldern einsetzbar zu sein, z.B. Systeme wie **BABYLON** [PriBre85], **KEE**

[KEE83] und **LOOPS** [BobSte83].

Die zuletzt genannten Systeme sind außerdem *hybride* Systeme [Kun84], da sie zum Aufbau von Wissensbasen die Möglichkeit bieten, verschiedene Wissensrepräsentationsformalismen alternativ bzw. komplementär zueinander zu gebrauchen. Das bedeutet, daß zur Wissensrepräsentation z.B. neben Produktionsregeln, auch objektorientierte Darstellungsmöglichkeiten (Frames) oder ein logischer Formalismus zur Verfügung stehen. Dafür gibt es keine zwingende theoretische oder technische Notwendigkeit, denn alle Formalismen sind gleichmächtig, d.h. jedes repräsentierbare Wissen läßt sich in jedem Formalismus darstellen. So hat z.B. Hayes [Hay80] nachgewiesen, daß objektorientierte Konstrukte sich leicht in eine prädikatenlogische Darstellung überführen lassen, und Frames damit keine qualitative Steigerung der Wissensrepräsentationsmächtigkeit bedeuten.

Der Grund für die Bereitstellung verschiedener Formalismen liegt vielmehr in der Absicht, die unterschiedlichen Wissenstypen einer Anwendung jeweils in der *natürlichsten* Form darstellen zu können, z.B. Erfahrungsregeln des Experten in Form von Produktionsregeln (**WENN** dies vorliegt **DANN** ist folgendes zu tun), definitorische Zusammenhänge des Gegenstandsbereiches in Prädikatenlogik und die Zustandsbeschreibungen der Objekte in Form von Frames. Bezweckt wird damit, daß sowohl Benutzer als auch Systementwickler ihre Problemlösungen bzw. das Verhalten des Expertensystems nicht in einen uniformen Formalismus zwingen bzw. interpretieren müssen. Dadurch soll erreicht werden, daß auch komplexe Wissensbasen transparent bleiben und ihr Entwicklungs- und Wartungsaufwand reduziert wird.

1.2.2 Geschlossene vs. offene Systeme

Allgemeine Werkzeugsysteme werden wie die speziellen letztlich immer für die Realisierung konkreter Anwendungen gebraucht und müssen daher eine problemorientierte Darstellung des betreffenden Gegenstandsbereichs erlauben. Entscheidend sind in dem Zusammenhang die Mittel, die vom allgemeinen Werkzeugsystem bereitgestellt werden, um es an das anvisierte Problemfeld anzupassen. Diese Mittel können nicht allein die Basis-Wissensrepräsentationsformalismen sein, denn das würde bedeuten, daß jeder einzelne Formalismus ein *spezielles* System wäre und damit das *allgemeine* Werkzeugsystem nichts Anderes als ein Nebeneinander von spezialisierten Systemen. Aber auch die Integration der gegebenen Formalismen, wie auch immer gestaltet, reicht im allgemeinen nicht aus, um die Anforderungen einer beliebigen speziellen Anwendung zu erfüllen.

Das wichtigste Kriterium, um zu beurteilen, ob ein allgemeines System anpaßbar und damit praktisch eingesetzt werden kann, ist der *Grad der Offenheit* des Systems. Darunter verstehen wir die Spezialisierungsmöglichkeiten, die das System bietet (Offenheit), und der dazu notwendige Realisierungsaufwand (Grad). Die Offenheit alleine ist ja nicht entscheidend, sonst wäre jede andere Programmiersprache auch ein geeignetes Expertensystemwerkzeug, weil sie sich für jede Anwendung gebrauchen läßt.

Entscheidend ist beides, die Ausdrucksmächtigkeit der vorhandenen Formalismen und der Aufwand, mit dem sie sich für den jeweiligen Bereich adäquaten Darstellungsformen anpassen lassen. Wenn man sich den Weg, der von einer Programmiersprache wie **LISP** zu einer konkreten Expertensystemanwendung führt, bildlich vorstellt, so sollte ein *gutes* allgemeines Werkzeugsystem irgendwo in der Mitte sein, nicht zu nah der Basissprache und nicht zu weit von den Anwendungsfeldern. Je näher ein Werkzeugsystem einer konkreten Anwendung ist, um so spezieller und geschlossener ist es.

BABYLON ist deshalb als ein allgemeines und offenes System entworfen und realisiert werden. Das Funktionsangebot von **BABYLON** umfaßt folgendes:

- die heute einschlägigen Formalismen zur Wissensrepräsentation
- Hilfen zur Wissensbasisverwaltung und
- Erklärungen für die ablaufenden Problemlösungsprozesse.

All dies wird dem Wissensingenieur über eine benutzerfreundliche Oberfläche angeboten, die auf modernen Rasterdisplays (grafische Visualisierungen, maussensitiven Menüs und Fenstertechnik) basiert.

1.2.3 Anforderungen an das Architekturkonzept

Die meisten allgemeinen Werkzeugsysteme haben sich aus früheren speziellen Expertensystem-Shells entwickelt, die ihrerseits aus einzelnen Expertensystementwicklungen hervorgegangen waren, und denen zunächst nur ein Repräsentationsformalismus zugrunde lag. Das gilt z.B. für **S.1** [S1], auf dem produktionsregelbasiertem **EMYCIN** [vanMel80] basiert, das die im Expertensystem **MYCIN** [Sho76] benutzten Techniken verallgemeinert.

Besser ist es, wenn die Forderung nach multipler Wissensrepräsentation bereits dem Architekturkonzept von vornherein zugrundegelegt werden kann. Das ist dann möglich, wenn kein spezielles Werkzeugsystem existiert, zu dem ein aufwärtskompatibles allgemeines System entwickelt werden muß. Dann kann die gewünschte offene Architektur ohne Abstriche entwickelt werden.

Das System verarbeitet eine Wissensbasis, die aus unterschiedlich repräsentiertem Wissen (z.B. in Form von Produktionsregeln, Frames oder Horn-Klauseln) besteht. Für jeden der Formalismen gibt es einen eigenständigen Interpreter, den wir auch Prozessor nennen. Der *Frame-Prozessor* interpretiert die Objektkonstrukte, der *Prolog-Prozessor* die Horn-Klauseln und der Regelprozessor die Produktionsregeln. Die Interaktionen zwischen den verschiedenen Prozessoren sollen an zentraler Stelle in einem sogenannten Meta-Prozessor verwaltet werden.

Um die Offenheit der Architektur zu gewährleisten, dürfen die Basis-Prozessoren nichts voneinander wissen. Sie sind lediglich in der Lage zu erkennen, ob im Rahmen einer ihnen gestellten Aufgabe eine Teilaufgabe entsteht, die nicht in ihre Kompetenz fällt. In einem solchen Fall schalten sie den Meta-Prozessor ein, der den Typ der Teilaufgabe bestimmt und sie an einen der anderen Basis-Prozessoren bzw. an seine eigene Meta-Instanz (den Benutzer) weiterleitet.

Da jeder Basis-Prozessor nur mit dem Meta-Prozessor direkt kommuniziert, ist es relativ einfach, einen Basis-Prozessor auszutauschen bzw. einen neuen hinzuzufügen. Dies bezeichnen wir als Vorteil der *Offenheit in der Breite*. Nicht zufällig sprechen wir von *Basis-Prozessoren*, wenn wir die Interpreter für die jeweiligen Wissensrepräsentationsformalismen meinen. So sind Rechnerarchitekturen vorstellbar (und teilweise schon realisiert), die auf mehreren Hardware-Prozessoren basieren und auf denen dieses Architekturkonzept viel effizienter implementiert werden kann.

Falls die technischen Voraussetzungen vorliegen, sollen also die Prozessoren von der jetzigen Einbettung in einer Einprozessor-**LISP**-Welt in andere spezialisierte Software-/Hardware-Umgebungen verlagert werden können. Das nennen wir den Vorteil der *Offenheit in der Tiefe* (Richtung Hardware).

Schließlich soll das System auch *offen in der Höhe* sein. D.h., man kann das System selbst durch Bootstrapping-Techniken erweitern.

1.2.4 Probleme bei der Portierung von Expertensystemen

Die Entwicklung allgemeiner Werkzeugsysteme dient dazu, unterschiedliche Expertensysteme mit möglichst geringem Aufwand zu entwickeln. Das ist allerdings nur ein erster notwendiger Schritt, damit mehr Expertensysteme entstehen können. Solange dasselbe Werkzeugsystem und dieselbe Hardware-/Software-Basis verwendet wird, mag er genügen. Es ist allerdings absehbar, daß, vergleichbar zu anderen Leistungssystemen wie Datenbankverwaltungssysteme, auch Expertensysteme portierbar sein müssen. Dies ist natürlich die Systemprogrammierersicht. Der Anwender fordert nicht die Portabilität des Werkzeuges sondern die Verfügbarkeit auf unterschiedlicher Hardware-/Softwarebasis. Dies kann der Systemprogrammierer durch Portierung erreichen aber auch durch eine vollständige Reimplementierung. Den Hinweis verdanken wir B.Bartsch-Spörl.

Es gibt zwei verschiedene Gründe, die dazu zwingen sowohl das Werkzeugsystem als auch die damit erstellten Wissensbasen portierbar zu machen. Da ist zum einen zwischen Entwicklungs- und Einsatzumgebung eines Expertensystems zu unterscheiden. Im Augenblick ist es gängige Praxis, die Strukturierung des Anwendungsbereiches und den ersten Wissenserwerb auf leistungsfähigen aber teuren Spezialrechnern, z.B. **LISP**-Maschinen, durchzuführen.

Ein Grund für die ökonomische Relevanz von Expertensystemen ist, daß mit ihnen Wissen leicht vervielfältigt werden kann, das bisher nur wenige Experten besitzen. Damit ein Expertensystem einsetzbar wird, muß es in entsprechend vielen Kopien verfügbar sein. Erfordert jede Kopie die Anschaffung eines solchen Spezialrechners, wird der geplante Einsatz unrentabel. Es kann aber auch möglich sein, daß auf Grund der vorhandenen Infrastruktur beim Einsatz andere Rechnersysteme verwendet werden müssen.

Die Portierbarkeit des Werkzeugsystems garantiert auch die Portierbarkeit der damit erstellten Wissensbasen und ist mit bekannten Mitteln des Software-Engineering realisierbar. Die Definition funktionaler Schnittstellen zum Graphiksystem und Betriebssystem der fraglichen Gastrechner gehört z.B. dazu.

Noch vollkommen ungelöst ist das Problem, Wissensbasen unabhängig von ihren Werkzeugsystemen zu portieren. Für die dort verwendeten Repräsentationsformalismen ist keine Standardisierung in Aussicht und bisher gibt es auch noch keine Übersetzungsprogramme, um z.B. **LOOPS**-Ausdrücke in äquivalente **BABYLON**-Ausdrücke abzubilden. Die Konsequenz ist, daß ein einmal gewähltes Werkzeugsystem solange unterstützt werden muß, wie man damit erstellte Wissensbasen verwenden will.

1.3 Verwendete Hilfsmittel und Notationskonventionen

BABYLON ist in ZetaLisp geschrieben und als Forschungsgegenstand nicht von vornherein unter dem Gesichtspunkt leichter Portierbarkeit auf andere Hard- und Softwareumgebungen entworfen worden. Dies betrifft insbesondere die Verwendung des Fen-

stersystems und der Graphik für die Benutzerschnittstelle, als auch die Verwendung des Flavor-Systems, also von objektorientierten Konzepten, zur Konstruktion und Implementierung der Software.

Dieser Bericht und insbesondere die Kapitel zur funktionalen Spezifikation sind nicht Grundlage zur Entwicklung des ersten Prototyps von **BABYLON** gewesen. Sie sind, wie schon zu Anfang des Berichtes gesagt, im Zuge einer Überarbeitung dieses Prototyps entstanden. Dabei wurden u.a. die Aspekte der Benutzerschnittstelle und der Entwicklungsumgebung aus den einzelnen Interpretern abstrahiert und an geeigneten Stellen zentralisiert.

Die Spezifikation ist weitgehend an objektorientierten Denkweisen [ObjHSArt] ausgerichtet. Die Modularisierung und Leistungsbeschreibung von **BABYLON** geschieht mit Hilfe von Objekttypdefinitionen und deren Protokollen. Das erlaubt softwaretechnisch eine direkte Umsetzung in die Konstruktion und Implementierung der spezifizierten Teile. Im Hinblick auf die Portierbarkeit haben wir von dem in ZetaLisp vorhandenen Flavor-System nur soviel benutzt, wie wir zur Zeit in dem leicht portierbaren Mikro-Flavor-System an Leistung anbieten [Kifs85]. Falls in den nächsten Versionen des Mikro-Flavor-Systems Leistungen hinzukommen, die die Konstruktion erleichtern, werden wir davon Gebrauch machen.

Für Syntaxbeschreibungen wird die übliche Backus-Naur Form verwendet. Dabei werden nichtterminale Symbole durch spitze Klammern gekennzeichnet, z.B. $\langle \text{Wert} \rangle$. Optionale Ausdrücke werden mit geschweiften Klammern versehen, z.B. $\{ \langle \text{Symbol} \rangle \}$.

Zusätzlich wird die in Beschreibungen von Lisp-Syntax übliche Punktnotation übernommen:

$$(A\ B\ C\ .\ (D\ E))$$

steht dabei für

$$(A\ B\ C\ D\ E)$$

In einigen Fällen verwenden wir auch die intuitiv klare Darstellungsform:

$$(\langle \text{irgendetwas} \rangle\ \langle \text{Symbol}_1 \rangle\ \dots\ \langle \text{Symbol}_N \rangle)$$

alternativ zu

$$(\langle \text{irgendetwas} \rangle\ .\ \langle \text{Liste von Symbolen} \rangle).$$

Wir setzen voraus, daß klar ist, was mit $\langle \text{Liste von etwas} \rangle$ gemeint ist, und erklären gewöhnlich nur $\langle \text{etwas} \rangle$ weiter. Strenggenommen müßte stets ergänzt werden

$$\langle \text{Liste von etwas} \rangle ::= () \mid (\langle \text{etwas} \rangle\ .\ \langle \text{Liste von etwas} \rangle)$$

Parameter von Kommandos und Methoden werden in *Schrägschrift* notiert. Optionale Parameter werden, wie allgemein üblich, durch ein vorangestelltes *&optional* gekennzeichnet.

Kapitel 2

Systemspezifikation

BABYLON ist eine Umgebung zur Entwicklung und zum Betrieb von Expertensystemen. Das problemspezifische Wissen des Systems wird in einer Wissensbasis verwaltet. Die Wissensbasis besteht aus unterschiedlichen Teilen, die jeweils von einem speziellen Interpreter oder Prozessor bearbeitet werden.

Die aktuell verfügbaren Prozessoren sind

1. Frame-Prozessor
2. Regel-Prozessor
3. Prolog-Prozessor und
4. Constraint-Prozessor

in jeweils unterschiedlichen Ausbaustufen.

Aus der Sicht der Softwarearchitektur entspricht jeder Wissensbasis ein Objekt eines speziellen Typs (Flavor), einer sogenannten Konfiguration. Instanzen dieses Typs mit den zugehörigen speziellen Prozessoren sind ablauffähige Expertensysteme, die diese Wissensbasis enthalten. Name und Bestandteile einer Konfiguration werden durch das **def-kb-configuration**-Macro festgelegt. Zu den Bestandteilen zählen die jeweiligen Ausprägungen (Ausbaustufen) der Prozessoren und eine Ausprägung eines Interfaces. Als mögliche Bestandteile stehen die in **BABYLON** vordefinierten Ausprägungen der Prozessoren und Interfaces zur Verfügung. Ausprägungen von Interfaces sind mit Ausnahme des **basic-interface-mixin** und **mini-interface-mixin** allerdings hardware-spezifisch (s. Kapitel 10 und 11). Darüberhinaus kann man selbstverständlich beliebige systemdefinierte oder benutzerdefinierte Flavors hinzunehmen. Das folgende Beispiel 2.1 zeigt eine solche Typdefinition einer Wissensbasis unter Nutzung von Babylon-definierten, Benutzer-definierten und System-definierten Bestandteilen.

Welche Ausbaustufen der Prozessoren von **BABYLON** zur Verfügung gestellt werden, ist bei den jeweiligen Prozessoren beschrieben. Generell bezeichnet der Namensbestandteil **basic** den jeweiligen Kern eines Prozessors, **mini** und **normal** sind jeweils um Aspekte wie Tracing und Entwicklungsunterstützung oder ähnliches erweiterte Prozessoren.

In der Datei, die die Wissensbasis selbst enthält, wird das **def-kb-instance**-Macro benutzt, um eine Instantiierung dieser Konfiguration zu machen, und diese an den Namen der Wissensbasis zu binden.

```

; defining and compiling my sample knowledge base configuration

(def-kb-configuration sample-config
  (:procs sys:process
    mini-frame-mixin          ; system defined
    my-special-rule-mixin     ; babylon defined
    lisp-mixin                ; user defined
    free-text-mixin           ; babylon defined
    (:interface basic-interface-mixin)) ; babylon defined

  (compile-flavor-methods sample-config))

```

Abbildung 2.1: Definition einer Konfiguration

```

; my sample knowledge base
(def-kb-instance sample-kb sample-config :language 'german)

; the frame part
(...)

; the rule part
(...)

```

Abbildung 2.2: Nutzung einer Konfiguration in einer Wissensbasis

Diese einzelnen Ausprägungen von Prozessoren stehen nicht isoliert nebeneinander, sondern erlauben die Bezugnahme auf Konstrukte der jeweils anderen Prozessoren. Dieses Zusammenspiel muß kontrolliert werden. Darüberhinaus verwaltet das Konfigurationsobjekt auch alle Aspekte der Wissensbasis und des Expertensystems als Gesamtheit. Einer der Aspekte koordiniert die verschiedenen Leistungen der speziellen Prozessoren für die einzelnen Repräsentationsformalismen. Dieser sogenannte Metaprozessorkern benutzt die von den einzelnen Prozessoren exportierten Leistungen. Einem speziellen Bestandteil der Wissensbasis entspricht dann jeweils ein Objekt des entsprechenden Prozessortyps, z.B. entspricht dem Produktionsregelanteil der Wissensbasis eine Instanz eines Regelprozessors. Diese Prozessorobjekte stehen dann mit dem Konfigurationsobjekt, dieses wird auch häufig als der Metaprozessor der Wissensbasis bezeichnet, in Beziehung.

2.1 Einordnung

Eine Wissensbasis hat drei verschiedene Erscheinungsformen. Die externe Repräsentation (konkrete Syntax) ist die Darstellung einer Wissensbasis in einer Datei. Einzelne Teile dieser externen Repräsentation werden durch einen jeweils zuständigen **Konstruktor** in eine abstrakte Form (abstrakte Syntax) überführt, die den einzelnen Bestandteilen der Konfiguration übergeben werden. Diese abstrakte Form wird dann von den entsprechenden Prozessoren in ihre eigene interne Form gebracht.

Die externe Repräsentation einer Wissensbasis besteht aus bis zu sechs verschiedenen Bereichen:

1. Wissensbasisdeklaration
2. Objektbereich
3. Produktionsregelbereich
4. Relationaler Bereich
5. Constraints-Bereich
6. Instruktionsbereich

Wissensbasisdeklaration und Instruktionsbereich werden vom Metaprozessor verwaltet. Alle anderen Bereiche sind optional und werden von dem jeweils zuständigen Prozessor verwaltet. Der Instruktionsbereich besteht aus einer Sequenz von Aufträgen (in Form von Nachrichten, Funktionsaufrufen oder Makros) an das ablauffähige Expertensystem. Aufträge können an das Expertensystem als Gesamtheit oder an einen der Prozessoren gerichtet sein. Das Expertensystem verteilt und koordiniert die Zustellung und Erledigung der Aufträge und eventuell daraus resultierender Unteraufträge im Gesamtsystem.

Solche Aufträge werden von den Prozessoren erteilt, wenn sie bei der Bearbeitung auf Konstrukte treffen, die nicht von ihnen selbst behandelt werden. In der Abbildung 2.3 werden die Möglichkeiten der Bezugnahme zwischen Konstrukten verschiedener Formalismen gezeigt.

	<i>Lisp</i>	<i>Objekte</i>	<i>Regeln</i>	<i>Relationen</i>
<i>Lisp</i>	—	Kreiere Objekt Sende Nachricht an Objekt	Starte Regel-Paket- Evaluation	Starte Horn-Klausel- Prozessor
<i>Objekte</i>	Behaviors :before daemons :after daemons Aktive Werte	—		
<i>Regeln</i>	Lisp-Ausdruck als Prämisse oder Aktion einer Regel	Frame-Referenz als Prämisse oder Aktion einer Regel	—	Relationaler Ausdruck als Prämisse einer Regel
<i>Relationen</i>	Lisp-Ausdruck als Unterziel einer Klausel	Frame-Referenz als Unterziel einer Klausel		—

(Benutzung von Formalismus X erlaubt Zugriff auf Formalismus Y, wie in Zeile X, Spalte Y dargestellt)

Abbildung 2.3: Integration der Formalismen in BABYLON

2.2 Funktionsbeschreibung

Der Metaprozessorkern definiert hauptsächlich die Kommunikation zwischen allen anderen **BABYLON**-Modulen. Er stellt sicher, daß **BABYLON** auf der Prozessebene beliebig verändert werden kann. Die einzelnen Prozessoren benötigen lediglich eine Schnittstelle zu ihm. Sie übergeben die Kontrolle genau dann an diese, wenn sie bei der Auswertung auf Ausdrücke stoßen, die für den jeweiligen Prozessor nicht interpretierbar sind. Diese werden dann als Argument in einer Nachricht an den Metaprozessor gesendet. Dieser entscheidet auf Grund von syntaktischen Kriterien, welcher (andere) Prozessor in der Lage sein sollte, diesen Ausdruck zu evaluieren. Der Benutzer kommuniziert im wesentlichen mit dem Metaprozessor, genauer mit dem **interface-mixin** des Metaprozessors, der z.B. Benutzerkommandos auf Nachrichten an die jeweiligen Prozessoren abbildet.

2.3 Systemschnittstelle

Die Systemschnittstelle umfaßt alle diejenigen Methoden, Funktionen oder Makros, die von einem Metaprozessor einer Wissensbasis angeboten werden. Sie sind durch Evaluieren in einem Lisp-Listener unmittelbar ausführbar. Bei Methoden sollte man die Funktion **send-kb** *selector &rest arguments* benutzen, um die entsprechende Methode der *aktuellen* Wissensbasis auszuführen. Die Schnittstelle setzt sich zusammen aus solchen Leistungen, die von den einzelnen Prozessoren ins Gesamtsystem exportiert werden und den originären Methoden des Metaprozessorkerns und der Wissensbasisverwaltung. Da die exportierten Leistungen eines Metaprozessors von seiner jeweiligen Konfiguration abhängen, kann hier nur auf die generell verfügbaren Leistungen eingegangen werden. Mit *Wissensbasis* wird in den folgenden Abschnitten die externe als auch die verschiedenen internen Repräsentationsformen bezeichnet. Unter Berücksichtigung des Kontexts sollten jedoch keine Pro-

bleme deshalb entstehen.

2.3.1 Systemschnittstelle zum Metaprozessor

Der Metaprozessorkern und die Wissensbasisverwaltung stellen folgende originären Methoden, Funktionen oder Makros zur Verfügung.

- :eval** *request mode processor &rest args* **Methode**
 Diese Methode wird von den einzelnen Prozessoren benutzt, um Ausdrücke, die sie nicht selbst auswerten können, dem Metaprozessor zur Auswertung zu übergeben. *request* ist der Ausdruck, *mode* ist der Auswertungsmodus und *processor* ein Symbol zur Kennzeichnung des auftraggebenden Prozessors, das im System Trace benutzt wird.
- :select-kb** **Methode**
 Macht die Wissensbasis zur aktuellen Wissensbasis, propagiert alle Änderungen, die mit Hilfe eines Editors an der Wissensbasis vorgenommen wurden, an alle Prozessoren und startet die Verarbeitungsschleife. Die Verarbeitungsschleife (die Methode *:run*) liefert per default unmittelbar *t* und wird normalerweise im Interface entsprechend redefiniert.
- :reset-kb** **Methode**
 Setzt die Wissensbasis inklusive aller vorhandenen speziellen Prozessoren auf den Ausgangszustand zurück.
- :reset-kb-confirmed** **Methode**
 Setzt die Wissensbasis nach Bestätigung durch den Benutzer inklusive aller vorhandenen speziellen Prozessoren auf den Ausgangszustand zurück.
- :kill-kb** **Methode**
 Entfernt die Wissensbasis aus der Menge der bekannten Wissensbasen.
- :describe-kb** **Methode**
 Schreibt statistische Angaben über die Größe der Wissensbasis und ihrer einzelnen Bestandteile auf den *dialog-stream* der Wissensbasis.
- :kb-inform** *&optional (stream *standard-output*)* **Methode**
 Schreibt statistische Angaben über die Größe der Wissensbasis und ihrer einzelnen Bestandteile auf den *stream*.
- :start-kb** *&optional (list-of-instructions nil)* **Methode**
 Startet die Abarbeitung der Instruktionen der Wissensbasis oder der als Parameter mitgegebenen Instruktionsliste.
- :start-kb-confirmed** *&optional (list-of-instructions nil)* **Methode**
 Startet die Abarbeitung nach Bestätigung durch den Benutzer der Instruktionen der Wissensbasis oder der als Parameter mitgegebenen Instruktionsliste.
- send-kb** *selector &rest arguments* **Funktion**
 Sendet eine Nachricht mit dem namen *selector* mit den Parametern *arguments* an die aktuelle Wissensbasis.

Kapitel 3

Systemkonstruktion

3.1 Konfiguration einer Wissensbasis

Ein ablauffähiges Expertensystem kann jeweils unterschiedliche Leistungen verwenden. Die Leistungen werden von den verschiedenen Prozessoren bzw. deren Ausbaustufen zur Verfügung gestellt. Welche Prozessoren in einer Wissensbasis verwendet werden, muß in der **:procs** Option des **def-kb-configuration**-Macros angegeben werden. Die **:interface** Option dagegen bestimmt, welche Ausprägung von Interface für diese Konfiguration verwendet werden soll.

Ein solche Konfiguration wird als ein Flavor definiert. Die **:procs** Option des **def-kb-configuration**-Macros gibt eine Liste der in die Konfiguration einzubringenden **mixins** an. Ein solches Mixin beschreibt die Leistungen eines Prozessors, die im Gesamtsystem verfügbar gemacht werden sollen. Für die Standardprozessoren sind dies die folgenden Mixins:

- **basic-frame-mixin** oder
- **mini-frame-mixin** oder
- **normal-frame-mixin** für den Frame-Formalismus,
- **basic-rule-mixin** oder
- **mini-rule-mixin** oder
- **normal-rule-mixin** für den Regel-Formalismus,
- **basic-prolog-mixin** oder
- **mini-prolog-mixin** für den Prolog-Formalismus,
- **constraint-mixin** für den Constraint-Processor,
- **free-text-mixin** für den Free-Text-Processor und
- **lisp-mixin** für den Lisp-Prozessor.

Welche Leistungen die einzelnen Ausprägungen der Prozessoren beinhalten, ist jeweils der Beschreibung der einzelnen Formalismen zu entnehmen.

Das **lisp-mixin** ist insofern untypisch, als es den Prozessor selbst schon enthält. Es wurde aber aus Gründen der Einheitlichkeit hier mit eingeführt.

Ein neuer oder modifizierter Prozessor wird durch Angabe eines entsprechenden Mixin in das Gesamtsystem eingebunden. Soll etwa ein modifizierter Regelprozessor für eine Wissensbasis benutzt werden, so gibt man in der **:procs** Option z.B. an:

```
(def-kb-configuration my-config
  (:procs normal-frame-mixin my-rule-mixin lisp-mixin free-text-mixin)
  (:interface my-interface-mixin)
  :language 'german)

(compile-flavor-methods my-config)
```

Die **:procs**-Angaben bilden die Grundlage für die Definition eines Flavors mit dem Namen der Konfiguration, von dem eine Instanz zusammen mit den Instanzen der entsprechenden Prozessoren als ablauffähiges Expertensystem die Wissensbasis im Lisp-System repräsentiert. Zur Konstruktion dieses Flavors werden zusätzlich zu den im **def-kb-configuration**-Macro angegebenen Prozessor-Mixins die folgenden Flavors hinzugezogen:

- **knowledge-base** für die Behandlung der Wissensbasis als Gesamtheit,
- **meta-processor-core** für die Behandlung der Kommunikation zwischen Prozessoren.

Man kann auch auf Flavors, die auf dem jeweiligen Hard- und Software-System definiert sind, zurückgreifen. So kann man z.B. durch Angabe von **sys:process** in der **:procs** Option auf Lispmaschinen eine Wissensbasis mit Prozesseigenschaften versehen.

Damit ergibt sich die in der Abbildung 3.1 dargestellte Struktur des Flavors für eine Wissensbasis.

3.2 Die Inter-Prozessor-Kommunikation

Die Kommunikation zwischen den Prozessoren einer Wissensbasis wird durch das Flavor **meta-processor-core** bewerkstelligt, das automatisch Bestandteil eines Prozessorobjektes für eine Wissensbasis ist. Stößt ein Prozessor auf einen Ausdruck, den er nicht selbst auswerten kann, schickt er diesem Prozessorobjekt die Nachricht:

```
:eval <expression> <mode> <processor> <arguments>
```

<Mode> kennzeichnet die gewünschte Auswertungstrategie für <expression>, z.B. :recall oder :remember. <Processor> ist der Name des auftraggebenden Prozessors.

Die Methode **:eval** ermittelt den *Typ des Ausdrucks* und den *Selektor* für die Methode, die die gewünschte Auswertungsstrategie für diesen Typ realisiert, um sie sodann aufzurufen. Das bedeutet, daß der Prozessor der Wissensbasis an sich selbst eine entsprechende Nachricht schickt, in der <expression>, <mode> und <arguments> als Parameter weitergereicht werden. Dazu müssen die aufgerufenen Methoden im zuständigen Prozessor-Mixin

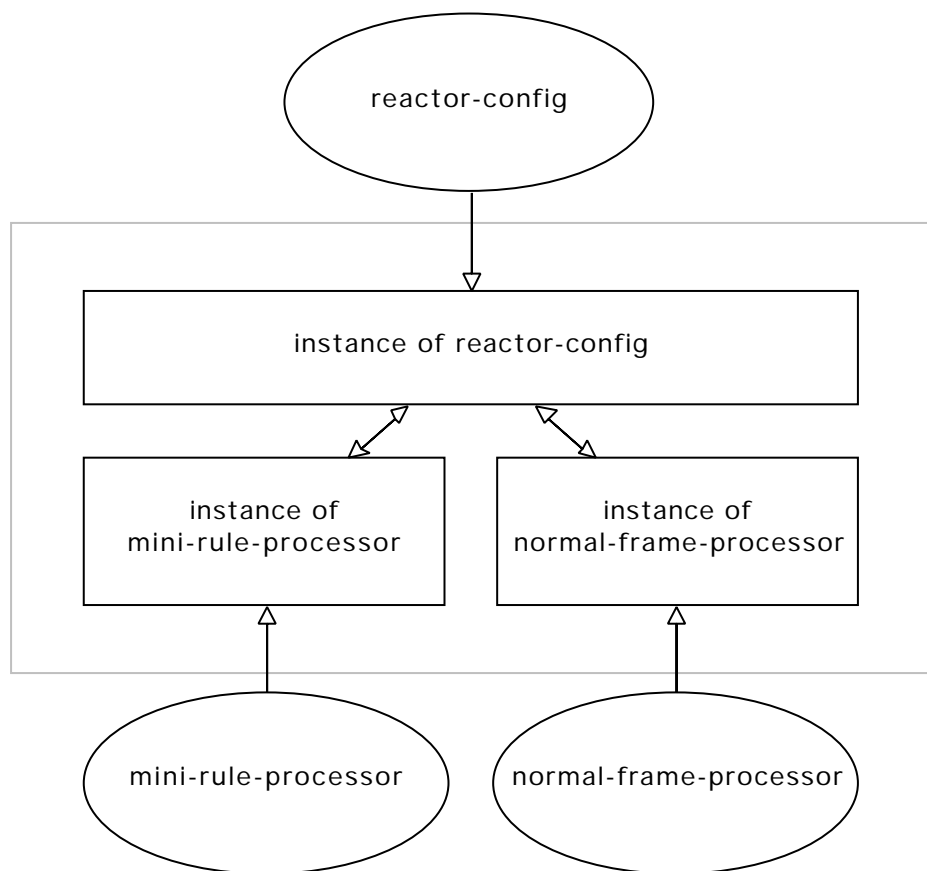


Abbildung 3.1: Architektur einer Konfiguration

bereitgestellt werden. In der Regel senden diese Methoden dem zugeordneten Prozessor eine Nachricht, in der die verlangte Interpretationsleistung spezifiziert wird. Abbildung 3.2 stellt den Nachrichtentyp-Graphen dieser Interprozessorkommunikation dar.

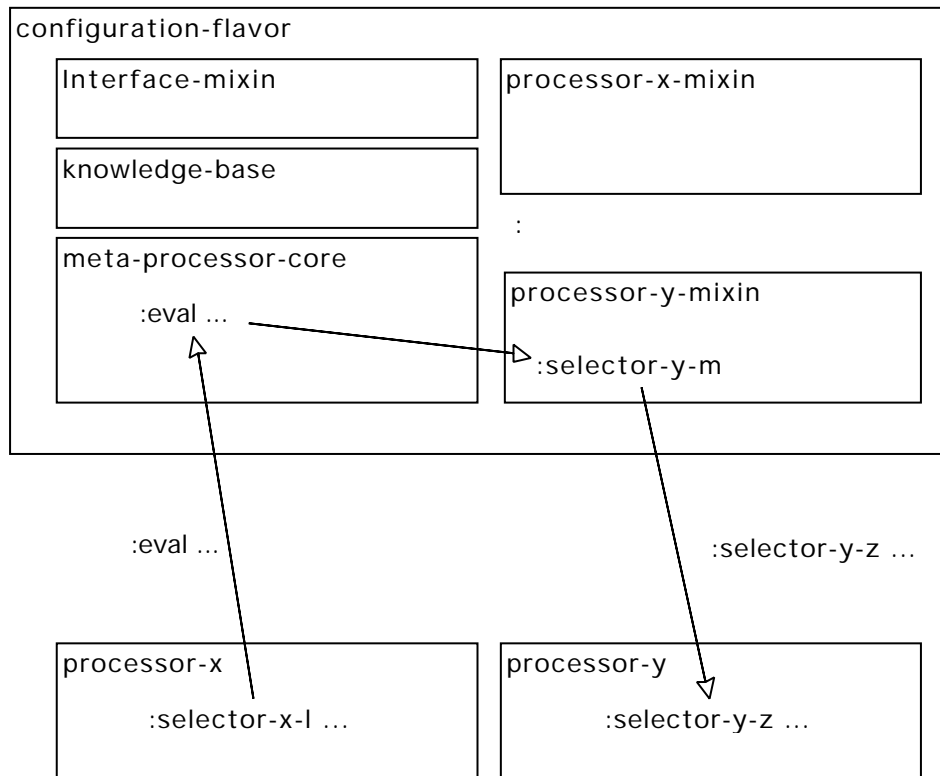


Abbildung 3.2: Kommunikation der Prozessoren

Zur Ermittlung des *Typs eines Ausdrucks* dient eine Methode **:get-type**, die aus Makros generiert wird. (Wer die nachfolgend beschriebene etwas *barocke* Konstruktion betrachtet, wird leicht erkennen, daß sie die im **flavor**-System anzutreffenden elaborierten **:OR** und **:PROGN** Methodenkombinationen nachvollzieht. Unter Nutzung dieser Methodenkombinationen wäre zwar eine sehr viel elegantere Lösung möglich, die aber hier zu Gunsten leichter Portierbarkeit nicht gewählt wird.) Sie setzt sich zusammen aus allen Typerkennungsmakros der Prozessoren, die in der Wissensbasisdeklaration angegeben sind. Die Makros sind im jeweiligen Prozessor-Mixin bereitzustellen. Sie liefern für einen Ausdruck den Typ, falls dieser dem Prozessor bekannt ist, sonst NIL, und dürfen auf alle Instanzvariablen des jeweiligen Prozessor-Mixins zugreifen. Sie sind mittels **assign-typefkt** Funktionsaufruf dem Prozessor-Mixin zuzuordnen:

```
(assign-typefkt <type-predicate-macro> <processor-mixin>)
z.B. (assign-typefkt 'frame-type 'basic-frame-mixin).
```

Die Reihenfolge, in der die Prozessor-Mixins in der **:procs** Option erscheinen, bestimmt auch die Reihenfolge der Typerkennung. Dies ist von Bedeutung, falls ein Ausdruck

```

(defmacro frame-type (request)
  '(if (is-true-list ,request)
      (cond ((%is-instance-name (first ,request) pkg)
             'frame-reference)
            ((is-frame-meta-predicate (first ,request))
             'frame-meta-predicate-reference)
            ((%is-frame-name (first ,request) pkg)
             'frame-class-reference)
            ((and (%is-instance-name (second ,request) pkg)
                  (%is-slot (first ,request) (second ,request) pkg))
             'frame-predicate-reference)
            ((and (%is-instance-name (third ,request) pkg)
                  (%is-slot (second ,request) (third ,request) pkg))
             'frame-predicate-reference)
            ((and (%is-instance-name (second ,request) pkg)
                  (%is-behavior (first ,request) (second ,request) pkg))
             'behavior-reference))))

(assign-typefkt 'frame-type 'basic-frame-mixin)

(defrequest frame-reference
  :recall          :eval-frame-reference
  :recall-immediate :eval-frame-reference
  :remember        :eval-frame-reference
  :store           :eval-frame-reference
  :ask             :ask-eval-frame-reference
  :prolog          :eval-prolog-frame-reference)

```

Abbildung 3.3: Definitionen zur Typerkennung beim Frame-Prozessor

mehreren Typen zuzurechnen ist. Insbesondere hält das Typerkennungsmakro des **free-text-mixins** jeden Ausdruck für *free-text*, so daß das **free-text-mixin** stets an **letzter** Stelle genannt werden muß.

Ferner ist für jeden Typ festzulegen, *welche Methode welche Auswertungsstrategie realisiert*. Dazu dient das Makro **defrequest**. Abbildung 3.3 zeigt einen Auszug aus **basic-frame-mixin** mit dem für die Typerkennung notwendigen Code für den Frame-Prozessor:

Die Methode **:eval-frame-reference** wertet also Frame-Referenzen aus, falls die gewünschte Strategie **:recall**, **:recall-immediate**, **:remember** oder **:store** ist. Für die Strategien **:ask** und **:prolog** sind jeweils spezielle Methoden definiert.

Das Makro **frame-type** stellt fest, ob ein gegebener Ausdruck vom Frame-Processor evaluiert werden kann. In der aktuellen Version der Frame-Sprache können dies sogenannte Frame-Referenzen sein (genauer gesagt sind dies Referenzen auf Attribute von Instanzen), die in Produktionsregeln vorkommen können. Oder es sind vergleichbare Referenzen in Horn-Clausen, die lediglich syntaktisch anders notiert werden. Darüberhinaus werden hier die Meta-Prädikate für den Frame-Formalismus erkannt. Es sind dies z.B. (**FRAME Name**) oder (**HAS-SUPER Frame-Name Super**).

Die Verbindung zwischen dem Typprädikat, z.B. **frame-type**, und dem entsprechenden Prozessor-Mixin erfolgt über den Indikator **:typefkt** auf der Eigenschaftsliste des

Mixin-Namens, z.B. für **basic-frame-mixin**:

```
(assign-typefkt 'frame-type 'basic-frame-mixin)
=
(setf (get 'basic-frame-mixin :typefkt) 'frame-type)
```

Der Konstruktor **DEF-KB-CONFIGURATION** benutzt diese Information, um eine Methode **:get-type** zu generieren, die die Typerkennung leistet. Dazu werden die Typenrerkennungsmakros aller Prozessor-Mixins mittels **or** verbunden. Hier wird also eine **:OR**-Methodenkombination nachvollzogen, für die auch die Kenntnis aller Mixins vonnöten ist.

Nach diesen etwas abstrakten Beschreibungen soll das Zusammenspiel der einzelnen Methoden bei der Kommunikation von Prozessoren noch einmal an einem konkreten Beispiel erläutert werden. Angenommen ein Regel-Prozessor soll nachfolgende Regel **RULE1** der Regelmenge **:CHECK** vorwärts auswerten.

```
(DEFRULE-SET :CHECK
  (RULE1 ($AND (PRIMARY-COOLING-SYSTEM PRESSURE = DECREASING)
    (HPIS STATUS = ON))
    ($CONCLUDE (PRIMARY-COOLING-SYSTEM INTEGRITY = CHALLENGED)))
  ... )
```

Die in der Regel auftretenden Prämissen müssen dazu evaluiert werden. Der Junktor **\$and** bewirkt, daß dies ohne eventuelle Nachfrage an den Benutzer geschieht, also in dem Modus **:recall** nicht **:ask**, der bei dem Junktor **?and** benutzt würde. Der Regel-Prozessor sendet also an den Meta-Prozessor folgende Nachricht:

```
(send meta-processor :eval
  '(PRIMARY-COOLING-SYSTEM PRESSURE = DECREASING) :recall 'rule)
```

Der dritte Parameter gibt einen für das Tracing auf Systemebene benutzten Kennzeichner des auftraggebenden Prozessors an, der hier nicht weiter interessiert. Die **:eval**-Methode des Meta-Prozessors benutzt die **:get-type**-Methode, um den Typ des Ausdrucks **'(PRIMARY-COOLING-SYSTEM PRESSURE = DECREASING)** zu bestimmen. In diesem Falle wird der vom **basic-frame-mixin** stammende Anteil der **:get-type**-Methode — das **frame-type**-Makro — den Ausdruck als **frame-reference** erkennen. Auf der Eigenschaftsliste von **frame-reference** wird unter **:recall** die Methode **:eval-frame-reference** gefunden, die für die Bearbeitung dieses Typs von Ausdruck unter diesem Modus zuständig ist. Wenn der Typ nicht erkannt wurde, oder keine Methode zu dem Typ/Modus-Paar existiert, wird ein Fehler signalisiert. Anderenfalls sendet der Meta-Prozessor eine Nachricht

```
(lexpr-send-message self :eval-frame-reference
  '(PRIMARY-COOLING-SYSTEM PRESSURE = DECREASING) :recall)
```

an sich selbst. Die dazugehörige Methode ist im **basic-frame-mixin** definiert, d.h. in dem Bestandteil des Meta-Prozessors, der die ins Gesamtsystem exportierten Leistungen des Basic-Frame-Prozessors beschreibt. Die Methode **:eval-frame-reference** des **basic-frame-mixin** schickt nun letztlich eine Nachricht an die **:eval-reference** Methode des Basic-Frame-Prozessors,

```
(send-message frame-processor :eval-reference
  '(PRIMARY-COOLING-SYSTEM PRESSURE = DECREASING) :recall)
```

mit gleichem Ausdruck und Modus. Der Bearbeiter des Ausdrucks ist damit gefunden und das Resultat der Auswertung gelangt auf umgekehrtem Weg zum Auftraggeber zurück.

3.3 Prozessor-Mixin Konventionen

Neben den im vorigen Abschnitt beschriebenen für die Kommunikation zwischen den einzelnen Prozessoren einzuhaltenden Konventionen werden an ein Prozessor-Mixin weitere Anforderungen gestellt, um die einfache Integration neuer oder modifizierter Prozessoren zu erleichtern.

Dies betrifft die Erzeugung und Einbindung einer Instanz der jeweiligen Ausprägung eines Prozessors. Dies wird in einer **:after :init** Methode des jeweiligen Prozessor-Mixins erledigt. Nachdem eine Instanz des Meta-Prozessors erzeugt wurde, wird dieser Dämon aktiviert. In seiner Definition muß dafür gesorgt werden, daß eine Instanz des entsprechenden Prozessor-Flavors erzeugt wird. Sie wird einer Instanzenvariablen zugewiesen und auch dem Gesamtsystem bekannt gemacht. Das letzte geschieht dadurch, daß die Prozessorinstanz dem aktuellen Wert der Instanzenvariable **procs**, die in **knowledge-base** definiert ist, hinzugefügt wird.

Die Abbildung 3.4 zeigt an einem Ausschnitt aus dem **basic-rule-mixin**, der die Integration des Regel-Prozessors ins Gesamtsystem leistet, die oben angesprochenen Konventionen exemplarisch auf.

Durch Nutzung von **basic-rule-mixin** als Komponente eines benutzerdefinierten Mixin und durch Redefinition von **:generate-rule-processor** kann leicht statt des Flavors **basic-rule-processor** ein anderer Regel-Prozessor eingebunden werden.

Zu den Konventionen gehört auch, eine Funktion zur Verfügung zu stellen, um eine Nachricht unter Umgehung des Meta-Prozessors der Wissensbasis **direkt** an den jeweiligen Prozessor senden zu können. Für den Regel-Prozessor ist dies die Funktion **send-rule**, die wie in Abbildung 3.5 gezeigt definiert ist.

Für alle sonstigen Prozessorleistungen, die zur Verfügung gestellt werden sollen, sind passende Methoden im Prozessor-Mixin bereitzustellen, die die jeweiligen entsprechende Methoden des Prozessor-Flavors aufrufen. Die Namen der Methoden in allen Mixins einer Konfiguration müssen, falls nicht anders gewollt, verschieden sein, da sonst Überschreibungen stattfinden. Für einige der Methoden ist zusätzlich eine Funktion oder ein Makro definiert, wie Abbildung 3.6 zeigt.

3.4 Prozessor-Flavor Konventionen

Auch die Prozessor-Flavors selbst müssen einige Konventionen beachten, um in das Gesamtsystem integrierbar zu sein. Sie müssen folgende Methoden exportieren:

:reset-proc setzt den Prozessor in den Ausgangszustand zurück,

:kb-inform liefert statistische Angaben über den prozessor-spezifischen Teil einer Wissensbasis,

```

(defflavor basic-rule-mixin
  (rule-processor
    (rules nil)
    (hypotheses nil))
  ()
  :settable-instance-variables
  (:required-instance-variables
    kb-name procs system-trace system-trace-window language)
  (:documentation "basic-rule-mixin makes the basic-rule-processor
available to kb."))

(defaction (basic-rule-mixin :after :init) (&rest ignore)
  "Create rule processor and make it known."
  (send-message self :generate-rule-processor)
  (setf procs (cons rule-processor procs)))

(defaction (basic-rule-mixin :generate-rule-processor) ()
  "Make an instance of basic-rule-processor and associate it with kb."
  (setf rule-processor (make-instance 'basic-rule-processor
    :meta-processor self
    :alternate-meta-processor
    (make-instance 'kb-stub :meta-processor self))))

```

Abbildung 3.4: Integration des Regel-Prozessors ins Gesamtsystem

```

(defun send-rule (selector &rest args)
  "Send to current rule-processor."
  (lexpr-send-message (send-kb :rule-processor) selector args))

```

Abbildung 3.5: Direkter Zugang zum Regel-Prozessor

```

(defaction (basic-rule-mixin :find-implications)
  (&optional (rule-set-name nil)
    (control-structure :DO-ALL)
    (condition T)
    (bindings nil))
  "Find implications (forward evaluation)."
```

```

  (send-message rule-processor :start-forward
    rule-set-name control-structure condition bindings))

(defun find-implications (&optional
  (rule-set-name nil)
  (control-structure :DO-ALL)
  (condition T)
  (bindings nil))
  (send-kb :find-implications
    rule-set-name control-structure condition bindings))

```

Abbildung 3.6: Zusätzliche Definition von Prozessorleistungen als Funktion

:reset-pointer wird benutzt, um den Prozessor mit externen Änderungen zu synchronisieren. Eine solche Synchronisierung wird z.B. nötig nach Änderungen am Aufbau von *Frames* mit Hilfe eines Editors, die auf interner Ebene der Repräsentation wirksam werden sollen. Es ist Sache des Interfaces, mit welchen Mitteln dies erreicht werden kann, oder bei welcher Gelegenheit dies als Seiteneffekt vorgenommen wird.

Die angeführten Methoden werden bei Ausführung der zugehörigen Methoden auf Meta-Prozessor-Ebene benutzt, damit **alle** Prozessoren ihren jeweiligen prozessor-spezifischen Anteil am Kommando durchführen. Dies könnte eleganter durch eine **:PROGN**-Methodenkombination erreicht werden, ist aber wegen Portierungsüberlegungen nicht gemacht worden. Stattdessen wird mittels **mapc** die entsprechende Nachricht an alle in **procs** verzeichneten Prozessoren gesandt.

Weiter muß jeder Prozessor seinen zugehörigen Meta-Prozessor in einer Instanzenvariablen festhalten. Diese Instanzenvariable ist z.B. Bestandteil des Flavors **processor-core**, der als genereller fundamentaler Flavor in jedem Prozessor einsetzbar ist. Er beinhaltet die Instanzenvariable **meta-processor**, die als Rückverweis auf den zugeordneten Meta-Prozessor benutzt wird, um z.B. als Empfänger der **:eval** Methode zu dienen. Diese Variable wird sinnvollerweise in der **:after :init** Methode des Mixins beim Instantiieren des Prozessors gesetzt.

3.5 Sprachbehandlung

Bei der Implementierung wurde darauf geachtet, alle sprachspezifischen Teile der Benutzerschnittstelle leicht auf eine andere Sprache umstellen zu können. Bisher realisiert ist eine deutsche und eine englische Version des Systems. Sprachabhängige Teile sind jeweils herausfaktorisiert und in getrennten Dateien untergebracht. Sie beinhalten die Definition

```

(defhash-table rule-io-table english :size 200)

(defentry rule-set-name-error-fstr rule-io-table english
  "~S: wrong name for a rule set of knowledge base ~S.~@
  The name must be a symbol.")

(defentry rule-trace-menu-items rule-io-table english
  '(("Toggle Rule Trace" :funcall toggle-rule-trace
    :documentation "Toggle Rule Tracing.")
    (" " :no-select nil)))

(defhash-table rule-io-table german :size 200)

(defentry rule-set-name-error-fstr rule-io-table german
  "~S: Nicht zulaessiger Bezeichner fuer ein RegelPaket der Wissensbasis ~S.
  ~\"Der Bezeichner muss ein Symbol sein.\")

(defentry rule-trace-menu-items rule-io-table german
  '(("Umschalten Regel Tracing" :funcall toggle-rule-trace
    :documentation "Umschalten des Regel Tracing.")
    (" " :no-select nil)))

```

Abbildung 3.7: Definition sprachabhängiger Teile im Regel-Prozessors

einer Hash-Table mit Einträgen, deren Werte sprachabhängig sind. Die Einträge der Hash-Table können nicht nur Zeichenketten, sondern beliebige Lisp-Objekte sein. Abbildung 3.7 zeigt eine Gegenüberstellung von englischen und deutschen Teilen der **rule-io-table** des Regel-Prozessors. Jeder Prozessor hat prinzipiell seine eigenen io-tables. Für gemeinsame prozessorübergreifende Tabellen kann auch die **io-table** als gemeinsame Resource verwendet werden.

Bei der Definition einer Konfiguration **def-kb-configuration** oder bei der Instantiierung einer Konfiguration **def-kb-instance** kann in der **init-plist** angegeben werden, ob die deutsche oder englische Version benutzt werden soll. Bei der Auswahl einer Wissensbasis (**:make-yourself-current** und alle darauf aufbauenden Methoden) wird die Sprache der Wissensbasis in die globale Variable ***language*** geschrieben mit den möglichen Werten **german** und **english**. Diese Werte entsprechen den in der Hash-Table benutzten Bezeichnern. Die Makros **defhash-table** und **defentry** dienen der Definition der Hash-Table und von Einträgen darin. Das Makro **getentry** wird benutzt, um auf Werte dieser Tabellen zuzugreifen. Beim Zugriff wird die Tabelle benutzt, die dem aktuellen Wert der Variablen ***language*** entspricht. Abbildung 3.8 zeigt die Definition der Makros zur Sprachbehandlung.

3.6 Arbeitsmodi von Prozessoren

Ein Prozessor, der das Flavor **processor-core** benutzt, kann mit Hilfe der Methode **:switch-mode** zwischen zwei verschiedenen Arbeitsweisen umgeschaltet werden. Die normale integrierte Arbeitsweise besteht darin, daß Ausdrücke, die einem Prozessor unbekannt

```
(defmacro defhash-table (table-name language &rest options)
  '(putprop ',table-name (make-hash-table ,@options) ',language))

(defmacro defentry (key table-name language value)
  '(let ((*language* ',language))
    (puthash ',key ,value (get ',table-name ',language))))

(defmacro getentry (key table-name)
  '(gethash ',key (get ',table-name *language*)))
```

Abbildung 3.8: Makros zur Sprachbehandlung

sind, ggf. von einem der anderen vorhandenen Prozessoren evaluiert werden. Daneben wird eine Arbeitsweise geboten, in der der Benutzer selbst an die Stelle der restlichen Wissensbasis tritt. Genauer gesagt werden alle auszuwertenden Ausdrücke nur noch als **free-text** interpretiert und einem Free-Text-Prozessor zur Auswertung übergeben. Der Free-Text-Prozessor ersetzt gewissermaßen das Gedächtnis des Benutzers und übernimmt das Fragen.

Man kann diesen Arbeitsmodus auch auffassen als eine Redefinition der Typen von Ausdrücken. Zur Zeit sind die Möglichkeiten der Redefinition darauf beschränkt, daß alle Ausdrücke immer als Free-Text interpretiert werden. Die Implementation geschieht deshalb im Moment nicht durch gezielte Redefinition der Typerkennungsmethode des Meta-Prozessors, sondern durch Ersetzen des Meta-Prozessors für den Prozessor durch eine Instanz von **kb-stub**. Diese Instanz ersetzt die gesamte **:eval** Methode durch eine, die nur den Typ **Free-Text** kennt.

Das Flavor **processor-core** besitzt zur Implementierung dieser verschiedenen Arbeitsmodi neben der Instanzvariablen **meta-processor** auch **alternate-meta-processor**, die bei der Initialisierung des Prozessors mit einer Instanz von **kb-stub** belegt wird. Die Methode **:switch-mode** vertauscht dann lediglich die Werte dieser beiden Variablen.

Man beachte bei der Nutzung dieses Modus, daß nicht identisches Verhalten der Wissensbasis erwartet werden kann. Insbesondere sind sämtliche Seiteneffekte, die bei der Standardinterpretation geschehen, hier nicht zu verzeichnen. Dies gilt schon für Lisp-Funktionen, die z.B. **nicht** ausgeführt werden.

Dieser Modus ist deshalb auch nicht als Methode in den Standard-Prozessor-Mixins verfügbar, sondern kann nur durch Senden einer Nachricht **direkt** an den Prozessor erreicht werden. Für den Regel-Prozessor also durch

```
(send-rule :switch-mode).
```

3.7 Checkliste

Nachfolgende Checkliste soll dem Implementierer von Prozessoren ein Hilfsmittel an die Hand geben, um die Vollständigkeit der Erfüllung aller notwendigen Voraussetzungen zur Einbindung eines Prozessors in das Gesamtsystem zu überprüfen.

Prinzipiell müssen ein Prozessor-FLAVOR und ein Prozessor-MIXIN vorhanden sein. Die Aufnahme eines Prozessors in ein ablauffähiges Expertensystem erfolgt über die Angabe des Prozessor-Mixin-Namens in der Deklaration einer Konfiguration.

Das Prozessor-MIXIN muß folgendes enthalten:

1. eine Instanzenvariable, deren Wert die Prozessorinstanz ist.
2. einen **:after :init** Dämon, in dem die Prozessorinstanz erzeugt, der entsprechenden Instanzenvariablen zugewiesen und der Instanzenvariablen **procs** hinzugefügt wird.
3. Konstruktoren, um die prozessorspezifischen Anteile der Wissensbasis zu erzeugen. Die Konstruktorenbezeichner sollten alle mit DEF anfangen.
4. Instanzenvariablen, deren Werte die für den Prozessor relevanten Anteile in der Wissensbasis sind.
5. Definition der Ausdrücke, die vom Prozessor bearbeitet werden sollen, und ein entsprechendes Typerkennungsprädikat. Das Typprädikat muß als Makro definiert werden. Der Zugriff erfolgt datengesteuert von der :EVAL-Methode des Flavours **meta-processor-core** über den Indikator TYPFKT auf die Eigenschaftsliste des Prozessor-Mixins. Die notwendige Zuordnung wird mit Hilfe von ASSIGN-TYPEFKT erreicht.
6. Methoden, in denen Evaluierungsanfragen von außen in Nachrichten an die Prozessorinstanz umgesetzt werden und die Zuordnung dieser zu den entsprechenden Typen von Ausdrücken bzw. Evaluierungsstrategien. Die Evaluierungsstrategien und deren Abbildung auf entsprechende Selektoren geschieht mit Hilfe von DEFREQUEST.
7. Methoden und/oder Funktionen für Nachrichten, die im Instruktionsteil für den Prozessor verwendet werden können.
8. Methoden und/oder Funktionen für prozessorspezifische Leistungen in der Programmierungsumgebung.

Die Definition des Prozessor-FLAVORS muß folgendes enthalten:

1. eine Instanzenvariable, deren Wert der Meta-Prozessor ist.
2. eine **:reset-proc** Methode, mit der das prozessorspezifische Arbeitsgedächtnis zurückgesetzt werden kann.
3. eine **:kb-inform** Methode
4. eine **:reset-pointer** Methode.
5. Instanzenvariablen, deren Werte die verarbeitbare Form der prozessorspezifischen Anteile der Wissensbasis enthalten
6. unbekannte Ausdrücke müssen mit einer :EVAL-Nachricht an den Meta-Prozessor gesendet werden

Kapitel 4

Spezifikation des Regel-Prozessors

4.1 Anforderungen

Wissensrepräsentation mit Hilfe von Produktionsregeln ist einer der ältesten Formalismen, die beim Bau von Expertensystemen benutzt werden. Regelerorientierte Darstellung von Wissen ist innerhalb von **BABYLON** nur eine der möglichen Formen neben z.B. objektorientierter oder logikorientierter Darstellung. Es war deshalb bei der Festlegung der Anforderungen an den regelerorientierten Formalismus hier nicht so sehr die Forderung nach möglichst großer Mächtigkeit, sondern eher die Forderung nach einer sinnvollen Abgrenzung der Eigenschaften gegenüber denen anderer Formalismen zu beachten.

Die wichtigsten Forderungen waren, die Regelmenge strukturieren zu können, verschiedene Auswertungsstrategien verfügbar zu machen und offen gegenüber Erweiterungen zu sein.

Die erste Forderung wird dadurch erfüllt, daß die Regelmenge in unabhängige Teilbereiche gliederbar ist, die unabhängig voneinander auswertbar sind. Einem solchen Teilbereich wird erst dynamisch beim Aufruf der Auswertung die Auswertungsstrategie zugeordnet. Vorwärts- und Rückwärtsauswertung mit unterschiedlichen Kontrollstrategien werden unterstützt. In Regeln wird sowohl auf Konstrukte anderer Formalismen referiert, als auch auf Konstrukte, die in der zugrundeliegenden Programmiersprache realisiert sind.

4.2 Einordnung

Der Regel-Prozessor realisiert die regelerorientierte Form der Wissensrepräsentation im Gesamtsystem und bietet dem Benutzer eine entsprechende Programmierumgebung, um regelerorientiertes Wissen zu formulieren, auszuwerten und dessen Auswertung nachzuvollziehen.

Innerhalb von Regeln wird auf Konstrukte anderer Formalismen Bezug genommen. Referenzen auf solche Konstrukte werden dem Meta-Prozessor zur Auswertung übergeben. Dieser entscheidet, welcher der verfügbaren spezialisierten Prozessoren, den jeweiligen Typ von Anfragen behandeln soll. Auf diese Weise wird die Integration der einzelnen Formalismen erreicht (vgl. dazu Kapitel 2 und 3). Der Regel-Prozessor selbst kennt keine eigene *Sprache*, um Fakten und Operanden in Regeln darzustellen.

4.3 Funktionsbeschreibung

4.3.1 Regeln und Regelpakete

Die regelorientierte Form der Wissensrepräsentation ist ein optionaler Bestandteil einer Wissensbasis. Dieser Regelbereich besteht aus einem oder mehreren Regelpaketen. Ein **Regelpaket** (RuleSet) faßt unter einem bestimmten anwendungsspezifischen Aspekt zusammengehörige Regeln unter einem eigenen Namen zusammen.

```
(DEFRULE-SET <RuleSetName>
  <Rule 1>
  <Rule 2>
  ...
  <Rule n>)
```

Abbildung 4.1: Regelpaketsyntax

Eine einzelne **Regel** besteht aus einem Namen, einem IF- und einem THEN-Teil. Der IF-Teil besteht aus einer Verknüpfung von Prämissen mittels eines **Junktors**, der THEN-Teil aus einer Verknüpfung von Aktionen mittels eines **Aktionstyps**.

```
(<Regelname> (<Junktor>      <Praemisse 1>
              ...
              <Praemisse n>)
  (<Aktionstyp> <Aktion 1>
              ...
              <Aktion n>))
```

Abbildung 4.2: Regelsyntax

4.3.2 Auswertungsstrategien

Regeln können vorwärts oder rückwärts ausgewertet werden. Die Regelpakete selbst enthalten keine Information darüber, mit welcher Strategie sie abgearbeitet werden sollen. Das wird erst bei der Ausführung eines Regelpaketes festgelegt. Diese Herausfaktorisierung der Kontrolle hat den Vorteil, daß dasselbe Regelpaket in verschiedenen Situationen mit unterschiedlicher Strategie ausgewertet werden kann.

Vorwärtsauswertung

Bei der Vorwärtsauswertung von Regeln wird der Aktionsteil von Regeln, deren Bedingung erfüllt ist, ausgeführt. Die Auswertung eines Aktionsteils kann erfolgreich oder nicht erfolgreich sein. Das hängt von dem jeweiligen Aktionstyp und den Aktionen ab und wird später im Einzelnen beschrieben.

Es sind vier verschiedene Vorwärtsstrategien realisiert:

:DO-ONE

die Auswertung des Regelpaketes wird beendet, sobald die erste Regel erfolgreich ausgeführt worden ist.

:DO-ALL

alle Regeln werden der Reihe nach einmal auf ihre Anwendbarkeit hin untersucht und gegebenenfalls ausgeführt.

:WHILE-ONE <while-Bedingung>

die Kontrollstruktur :DO-ONE wird sukzessive solange angewendet bis entweder keine Regel mehr ausgeführt werden kann oder die Aktion STOP-EXECUTION von einer Regel ausgeführt wird oder die <while-Bedingung> nicht mehr gültig ist. Die <while-Bedingung> ist ein Lisp-Ausdruck. Sie ist genau dann gültig, wenn der Lisp-Ausdruck zu einem Wert ungleich NIL ausgewertet wird.

:WHILE-ALL <while-Bedingung>

die Kontrollstruktur :DO-ALL wird sukzessive solange angewendet bis entweder keine Regel mehr ausführbar ist oder die Aktion STOP-EXECUTION von einer Regel ausgeführt wird oder die <while-Bedingung> nicht mehr gültig ist. Zur Gültigkeit der <while-Bedingung> s. :WHILE-ONE.

Falls der Aktionstyp \$CONCLUDE (s.u.) ist und es keine Aktion gibt, deren Ausführung einen Wert ungleich NIL liefert, so gilt die Regel insgesamt als nicht ausgeführt (im Sinne der vier möglichen Arten der Vorwärtsverkettung).

Rückwärtsauswertung

Bei der Rückwärtsauswertung wird versucht, eine Regel zu finden, die es gestattet, eine zu überprüfende Hypothese abzuleiten. Die Vorbedingung dieser Regel muß zu **wahr** ausgewertet werden. Die Auswertung einer Prämisse der Regelbedingung kann ihrerseits die (Rückwärts-) Anwendung weiterer Regeln erforderlich machen, falls ihr Wert **unbekannt** („-“) ist. Bleibt der Wert einer Prämisse **unbekannt**, so wird sie wie **falsch** behandelt.

Die Verifikation einer Hypothese verläuft nach folgender Strategie:

1. überprüfe, ob die Hypothese bereits bestimmt ist. Ist sie bestimmt, so liefere **wahr**, falls sie gilt, **falsch**, falls sie nicht gilt. Ist sie unbestimmt, dann prüfe, ob sie bereits als **nicht ableitbar** markiert wurde. Ist das der Fall, so liefere **falsch**.
2. Suche alle Regeln, die die Hypothese als Aktion enthalten. Falls keine Regeln gefunden werden, bestimme die Hypothese durch Befragen des Benutzers und gehe nach Schritt 1.
3. Versuche eine Einschnitt-Vorwärtsauswertung der Regeln, die die Hypothese als Aktion enthalten. Kann eine Regel ausgeführt werden, so liefere **wahr**.
4. Versuche der Reihe nach die Bedingungsteile der Regeln zu verifizieren (Rückwärtsauswertung). Falls es eine Regel gibt, für die entsprechend dem Junktortyp der Bedingungsteil erfüllt ist, so markiere die Hypothese als **ableitbar** und liefere **wahr**. Falls das nicht der Fall ist, so markiere die Hypothese als **nicht ableitbar** und liefere **falsch**.

4.3.3 Der IF-Teil von Regeln

Der IF-Teil einer Regel besteht aus der Verknüpfung von Prämissen durch einen Junktor. Alle Prämissen werden mit einem dem Junktor entsprechenden Modus an den Meta-Prozessor zur Auswertung weitergereicht. Die Interpretation der als Prämissen auftretenden Ausdrücke hängt von der aktuellen Konfiguration der Wissensbasis ab und wird durch die verfügbaren speziellen Prozessoren geleistet. Zu den Ausdrücken, die von Standard-Prozessoren behandelt werden können, gehören:

1. Lisp-Ausdrücke

Ein vom **lisp-mixin** bearbeitbarer Ausdruck muß folgende Form besitzen:

$$(<\text{Lisp-Funktionsname}> . <\text{Argumente}>)$$

Es gibt einige Funktionen und Makros, die in **BABYLON** vordefiniert sind, und hier benutzt werden können. Dies sind:

STOP-EXECUTION

beendet die Abarbeitung eines Regelpaketes.

SAY

verhält sich wie die Funktion **FORMAT** [LMM], mit der Ausnahme, daß das erste Argument (der output stream) nicht angegeben wird. Alle anderen Argumente entsprechen denen von **FORMAT**. Als output stream wird der aktuelle **dialog stream** von **BABYLON** benutzt.

Die Prämisse gilt als **falsch**, wenn ihre Auswertung den Wert **NIL** ergibt, in allen anderen Fällen ist sie **wahr**.

2. Framereferenzen

Bezüglich der Form von Framereferenzen s. Kapitel 6.

3. Prologgoals

Bezüglich der Form von Prologgoals s. Kapitel 8.

4. freier Text

Falls die Konfiguration einer Wissensbasis den **free-text-mixin** enthält, wird alles, was nicht anderweitig interpretiert werden kann, als freier Text behandelt. Anderenfalls führt der Ausdruck als von unbekanntem Typ zu einer Fehlermeldung. In diesem Sinne bildet freier Text eine Restkategorie. Die Interpretation von freiem Text wird von einem Free-Text-Prozessor geleistet. Freier Text ist **wahr**, wenn er in der Datenbasis des Free-Text-Prozessors als verifiziert enthalten ist, **falsch**, wenn er als falsifiziert enthalten ist. Ansonsten wird sein Wahrheitswert je nach Strategie und in der Regel vorkommendem Junktor durch Anwendung weiterer Regeln oder durch Befragen des Benutzers bestimmt. Konnte der Wahrheitswert eines freien Textes auf diese Weise ermittelt werden, so wird er mittels des Free-Text-Prozessors entsprechend markiert in dessen dynamischer Datenbasis eingetragen.

Alle eben beschriebenen Prämissen können auch in negierter Form auftreten. Sie haben dann die Form:

(NOT <nicht-negierte-Prämisse>).

Prämissen einer Regel können durch einen der folgenden Junktoren miteinander verknüpft werden:

\$TRUE

nullstelliger Junktor, d.h. es sind keine Prämissen anzugeben. In diesem Fall ist die Regel immer ausführbar.

\$FALSE

nullstelliger Junktor, d.h. es sind keine Prämissen anzugeben. Die Regel wird nie ausgeführt.

\$AND

die Prämissen werden der Reihe nach überprüft. Wird eine Prämisse zu **falsch** ausgewertet, so wird eine Überprüfung weiterer Prämissen abgebrochen, und die Regel ist nicht anwendbar. Dasselbe gilt bei Vorwärtsauswertung der Regeln, wenn eine Prämisse zu **unbekannt** ausgewertet wird. Wird eine Prämisse bei Rückwärtsauswertung zu **unbekannt** ausgewertet, und gibt es keine weiteren Regeln, mit deren Hilfe ihre Gültigkeit abgeleitet werden könnte, so wird der Benutzer nach ihrer Gültigkeit gefragt. Sind alle Prämissen **wahr**, so ist die Regel ausführbar.

?AND

unterscheidet sich von \$AND nur darin, daß

- auch bei Vorwärtsauswertung von Regeln der Benutzer gefragt wird, falls die Auswertung einer Prämisse **unbekannt** ergibt. Es wird jedoch nur dann eine Frage an den Benutzer gestellt, wenn keine der (nachfolgenden) Prämissen zu **falsch** ausgewertet wird,
- bei Rückwärtsauswertung der Regeln die Gültigkeit einer **unbestimmten** Prämisse auch dann sofort vom Benutzer erfragt wird, wenn ihre Gültigkeit auch durch Anwendung weiterer Regeln abgeleitet werden könnte.

AND

der Bedingungsteil (d.h. der Junktor mit allen Prämissen) wird geschlossen an den Prolog-Prozessor zur Auswertung übergeben. Falls der Bedingungsteil keine Variablen enthält, so liefert der Prolog-Prozessor **wahr** oder **falsch** zurück. Falls Variablen enthalten sind, so werden alle Variablenbindungen geliefert, die den Bedingungsteil wahr machen. Für jede dieser Variablenbindungen wird eine Instanziierung der Regelaktion(en) vorgenommen und diese ausgeführt. Die Regelaktionen müssen dabei voll instanziiert werden.

\$OR

Die Prämissen werden der Reihe nach überprüft. Wird eine Prämisse zu **wahr** ausgewertet, so wird eine Überprüfung weiterer Prämissen abgebrochen, und die

Regel ist anwendbar. Wird eine Regel bei Rückwärtsauswertung zu **unbekannt** ausgewertet, so wird der Benutzer nach ihrer Gültigkeit gefragt, wenn es keine weitere Regel gibt, mit deren Hilfe ihre Gültigkeit abgeleitet werden könnte. Ist keine der Prämissen **wahr**, so ist die Regel nicht ausführbar.

?OR

unterscheidet sich von \$OR nur darin, daß

- auch bei Vorwärtsauswertung von Regeln der Benutzer gefragt wird, falls die Auswertung einer Prämisse **unbekannt** ergibt. Es wird jedoch nur dann eine Frage an den Benutzer gestellt, wenn keine der (nachfolgenden) Prämissen zu **wahr** ausgewertet wird,
- bei Rückwärtsauswertung der Regeln die Gültigkeit einer **unbestimmten** Prämisse selbst dann vom Benutzer erfragt wird, wenn ihre Gültigkeit durch Anwendung weiterer Regeln abgeleitet werden könnte.

OR

analog zu **AND**

WICHTIG: Regeln mit den Junktoren **AND** und **OR** können nur vorwärts ausgewertet werden!

4.3.4 Der THEN-Teil von Regeln

Der THEN-Teil einer Regel besteht aus der Verknüpfung von Aktionen mittels eines Aktionstyps. Alle Aktionen werden mit einem dem Aktionstyp entsprechendem Modus der Wissensbasis zur Auswertung übergeben.

Die folgenden Aktionstypen sind im THEN-Teil einer Regel zulässig:

\$CONCLUDE

die Aktionen werden der Reihe nach ausgeführt. Das heißt jeder der Ausdrücke wird mit dem Mode `:remember` zur Auswertung an die Wissensbasis weitergereicht.

Der Aktionsteil insgesamt gilt nur dann als erfolgreich ausgeführt, wenn mindestens eine der Aktionen ein positives Ergebnis (ungleich NIL) liefert. Insbesondere liefert jeder Versuch, schon bekannte Information erneut abzuspeichern, den Wert NIL.

\$EXECUTE

unterscheidet sich von \$CONCLUDE darin, daß der Aktionsteil in jedem Fall als erfolgreich ausgeführt gilt, ungeachtet der bei der Auswertung der einzelnen Aktionen zurückgelieferten Werte.

\$ASK

die Aktionen werden als Fragen an den Benutzer interpretiert. Das heißt jeder der Ausdrücke wird mit dem Mode `:ask` zur Auswertung an den Metaprozessor weitergereicht.

Der Aktionsteil gilt in jedem Fall als erfolgreich ausgeführt.

4.4 Schnittstellenbeschreibung

Unter der Schnittstelle wollen wir hier alle die Leistungen — Methoden, Funktionen oder Macros — verstehen, die von einer Ausprägung eines Regelprozessors über das entsprechende Prozessor-Mixin in einer Wissensbasiskonfiguration verfügbar gemacht werden. Dabei werden diejenigen Leistungen nicht berücksichtigt, die hauptsächlich internen Zwecken dienen, und normalerweise nicht direkt genutzt werden.

4.4.1 basic-rule-mixin

Dieses Mixin enthält die in eine Wissensbasis zu exportierenden Leistungen des **basic-rule-processor**. Es bildet analog zu **basic-rule-processor** die Basis aller weiteren darauf aufbauenden Ausprägungen von Regel-Prozessor-Mixins.

Die Methoden zur Auswertung von Regelpaketen bestehen aus der Methode zur Vorwärtsauswertung und zwei Versionen einer Methode zur Rückwärtsauswertung. Alternativ zu den folgenden Nachrichten können auch die Funktionen **test-hypotheses**, **obtain** und **find-implications** verwendet werden. Sie haben die gleichen Parameter und die gleiche Wirkung wie die entsprechenden Nachrichten.

:find-implications &optional (*rule-set-name nil*)

(*control-structure :DO-ALL*)

(*condition T*)

Methode

Wertet das Regelpaket *rule-set-name* mittels Vorwärtsauswertung mit der Kontrollstrategie *control-structure* aus. *control-structure* ist eine der Optionen :DO-ONE, :DO-ALL, :WHILE-ONE oder :WHILE-ALL. Im Falle einer :WHILE-Option kann ein Lisp-Ausdruck als *condition* angegeben werden. Er wird vor jeder Iteration ausgewertet und führt bei NIL zum Abbruch der Auswertung des Regelpaketes. Default für *condition* ist T.

:test-hypotheses &optional (*number-of-hypotheses-to-verify 1.*)

(*list-of-hypotheses nil*)

(*rule-set-name nil*)

Methode

Überprüft die in *list-of-hypotheses* aufgeführten Hypothesen durch Auswerten des Regelpaketes *rule-set-name* mit Rückwärtsauswertung. *number-of-hypotheses-to-verify* als Zahl oder das Schlüsselwort :ALL begrenzt die Auswertung der Hypothesen auf die angegebene Anzahl.

:obtain *number-of-hypotheses-to-verify*

goal-specification

&optional (*rule-set-name nil*)

Methode

Analog :test-hypotheses, jedoch können die Hypothesen teilspezifiziert sein. Eine solche *goal-specification* wird vor der Auswertung zunächst expandiert zu Hypothesen. Eine *goal-specification* kann aus einem Atom oder einer Liste aus zwei Elementen bestehen. Die Expansion besteht aus der Menge aller Aktionen des Regelpaketes, die das Atom als erstes Element oder die beiden Elemente der Liste als erste Elemente enthalten.

:print-true-facts

Methode

Zeigt alle im Verlauf der Regelauswertung zu **wahr** ausgewerteten Facts.

:print-hypotheses-verified **Methode**
 Zeigt alle im Verlauf der Regelauswertung als bestätigt verifizierten Hypothesen.

:list-rules **Methode**
 Zeigt eine Regel nach Auswahl des Namens des Regelpaketes und der Regel aus einem Menü.

send-rule *selector &rest args* **Funktion**
 Sendet eine Nachricht mit dem Selector *selector* an den Regelprozessor der aktuellen Wissensbasis.

4.4.2 mini-rule-mixin

Das Mixin **mini-rule-mixin** erweitert **basic-rule-mixin** um die Aspekte des Tracings, die der zugehörige **mini-rule-processor** zur Verfügung stellt. Für die Ausgabe des Regeltrace ist im **mini-rule-mixin** die Instanzvariable **rule-trace-window** vorhanden, die in dem jeweiligen **interface-mixin** an ein Fenster gebunden wird, das die Methode **:format** verstehen muß.

:send-rule-trace-window *selector &rest args* **Methode**
 Leitet alle Nachrichten für das Regeltrace-Fenster an **rule-trace-window** weiter.

:rule-trace **Methode**
 Liefert den aktuellen Zustand des Regeltrace. T bedeutet den eingeschalteten Zustand, NIL den ausgeschalteten.

:toggle-rule-trace **Methode**
 Schaltet das Tracing des Regel-Prozessors um.

4.4.3 normal-rule-mixin

Das Mixin **normal-rule-mixin** erweitert **mini-rule-mixin** um die Aspekte der Erklärung und Entwicklungsunterstützung. Für die Ausgabe von Erklärungen ist im **normal-rule-mixin** die Instanzvariable **explanation-window** vorhanden, die in dem jeweiligen **interface-mixin** an ein Fenster gebunden wird, das die Methode **:format** verstehen muß.

:send-explanation-window *selector &rest args* **Methode**
 Leitet alle Nachrichten für das Erklärungs-Fenster an **explanation-window** weiter.

:print-rule **Methode**
 Anzeige einer Regel eines Regelpaketes.

:inspect-terms **Methode**
 Erlaubt, das Vorkommen von Termen in Regeln festzustellen.

:explain-results **Methode**
 Bietet verschiedene Möglichkeiten, sich die Auswertungsversuche verifizierter oder nicht verifizierbarer Fakten erklären zu lassen.

Kapitel 5

Konstruktion des Regel-Prozessors

5.1 Übersicht

Die Konstruktion des Regel-Prozessors verfolgt das Ziel, eine ausgewogene Modularisierung der im Kapitel 4 beschriebenen Leistungen durch Trennung der Aspekte Tracing, Entwicklungsunterstützung und Erklärung vom reinen Interpreter zu erreichen. Der Interpreter selbst greift auf die Leistungen von Komponenten zur Verwaltung von Regeln und von Daten zurück. Die in das Gesamtsystem integrierbaren Ausprägungen eines Regel-Prozessors bestehen dann aus dem Regel-Interpreter und unterschiedlichen Zusammenstellungen der optionalen Bestandteilen für die vorgenannten Aspekte. Die Einbindung ins Gesamtsystem (s. Kapitel 3, Systemkonstruktion) geschieht mittels eigener Komponenten, die die im Gesamtsystem verfügbaren Leistungen einer Ausprägung eines Regel-Prozessors beschreiben und die Integrierbarkeit sicherstellen.

5.2 Zerlegung

Der Regel-Prozessor kann aus folgenden Teilen bestehen:

1. **processor-core**

Die Basiskomponente aller Prozessoren, hier des Regel-Interpreters, die die Schnittstelle zum Meta-Prozessor realisiert und die *Instanzenvariablen* definiert, die benötigt werden, damit der Prozessor ins Gesamtsystem integrierbar ist.

2. **rule-base**

Verwaltet die Regelpakete und Regeln des Regel-Interpreters.

3. **data-base**

Verwaltet die virtuelle dynamische Wissensbasis des Regel-Interpreters.

4. **rule-interpreter**

Realisiert die Basisleistungen des Regel-Prozessors unter Nutzung der Leistungen von **rule-base** und **data-base**.

5. **rule-trace-mixin**
Erweitert **rule-interpretier** um den Aspekt des Tracing.
6. **rule-develop-mixin**
Erweitert **rule-base** um den Aspekt der Entwicklungsunterstützung für Regeln.
7. **rule-explain-mixin**
Erweitert **data-base** um den Aspekt der Erklärung.
8. **basic-rule-processor**
Fügt dem **rule-interpretier** die *Methoden* hinzu, die benötigt werden, damit der Prozessor ins Gesamtsystem integrierbar ist.
9. **mini-rule-processor**
Fügt dem **basic-rule-processor** das **rule-trace-mixin** als Komponente hinzu.
10. **normal-rule-processor**
Fügt dem **mini-rule-processor** das **rule-explain-mixin** und das **rule-develop-mixin** als Komponenten hinzu.

Abbildung 5.1 zeigt den Zusammenhang der einzelnen im Aufbau des Regel-Prozessors verwendeten Flavors.

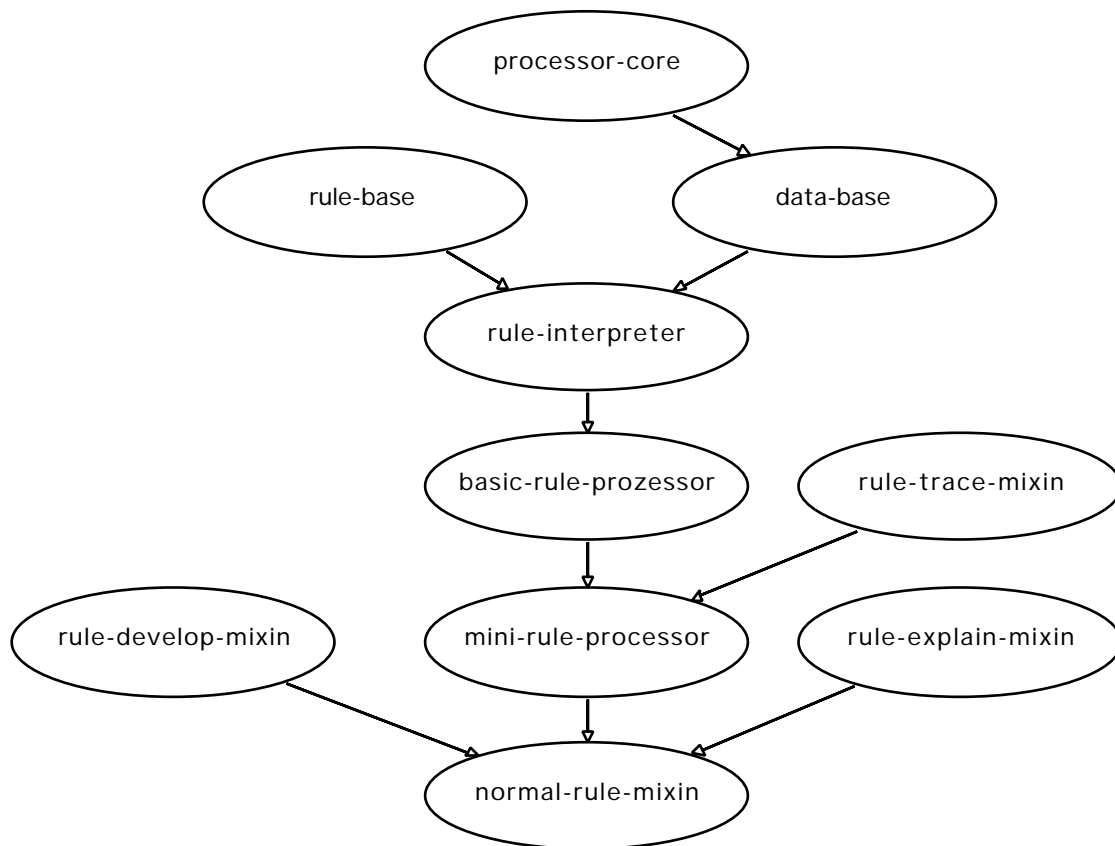


Abbildung 5.1: Architektur des Regel-Prozessors

Die Leistungen der unterschiedlichen Ausprägungen des Regel-Prozessors können durch die entsprechenden Flavors **basic-rule-mixin**, **mini-rule-mixin** bzw. **normal-rule-mixin** in einer Wissensbasiskonfiguration verfügbar gemacht werden.

5.3 Integration ins Gesamtsystem

Die Integrationsfähigkeit des Regel-Prozessors wird prozessorseitig in den beiden Flavors **processor-core** und **basic-rule-processor** sichergestellt. Zur Verwendung innerhalb des Meta-Prozessors selbst definieren die Flavors **basic-rule-mixin**, **mini-rule-mixin** bzw. **normal-rule-mixin** die Leistungen der Ausprägung des Regel-Prozessors, die im Gesamtsystem verfügbar gemacht werden sollen.

Das Flavor **processor-core** ist Bestandteil von **data-base** und realisiert das Interface – die verlangten **Instanzenvariablen** – zum Meta-Prozessor. Eine Beschreibung der Rolle dieses Flavors für die Einbindung ins Gesamtsystem findet sich in Kapitel 3.

Das Flavor **basic-rule-processor** auf der anderen Seite definiert die verlangten **Methoden** und bildet sie auf die in den Komponenten verfügbaren Leistungen ab. Es legt darüberhinaus die Basiskonfiguration des eigentlichen Interpreters fest.

5.4 Der Basic-Rule-Processor

Die eigentlichen Leistungen des Regelinterpreters sind im Flavor **rule-interpreter** realisiert. Zur Verwaltung von Regelpaketen und Regeln werden dabei die Leistungen des Flavors **rule-base** und zur Verwaltung von Fakten die des Flavors **data-base** benutzt.

5.4.1 rule-base

Das Flavor **rule-base** verwaltet die Regelpakete für den Regel-Interpreter und stellt die grundlegenden Methoden für die Entwicklungsunterstützung zur Verfügung. Darüberhinaus werden hier z.B. die von jedem Prozessor erwarteten Methoden zur statistischen Information sowie einige Hilfsmethoden für den Interpreter definiert.

Regelpakete und Regeln werden als Listen implementiert. Die Instanzenvariable **rules** zeigt auf die Liste aller Regelpakete, **current-rule-set** auf die Liste des sich in der Evaluierung befindlichen aktuellen Regelpaketes.

Die Liste aller Regelpakete hat den in Abbildung 5.2 gezeigten Aufbau:

Für Zugriff und Änderung der Menge der Regelpakete und Regeln sind folgende Methoden definiert, deren detaillierte Beschreibung im Anhang A zu finden ist:

:get-rule-set-names ()

:get-current-rule-set-name ()

:get-rule-names (*rule-set-name*)

:get-rule-set (*rule-set-name*)

:get-rule (*rule-set-name rule-name*)

:add-rule (*rule rule-set*)

```

(((<rule-set-name1> ...)           ; Liste aller Regelpakete
 (<rule-set-name2> ...)
 ...))

(<rule-set-name>                   ; Liste eines Regelpaketes
 . <rule1> <rule2> ... <ruleN>))

(<rule-name>                       ; Liste einer Regel
 (<junctor> <condition1>
  ...
  <conditionN>))
 (<action-type>
  <action1>
  ...
  <actionN>))

```

Abbildung 5.2: Struktur von Regelpaketen

:kill-rule (*rule rule-set*)

:modify-rule (*rule-set-name new-rule*)

Die folgenden Methoden gehören zu denen, die von jedem Prozessor erwartet werden, und sind hier wie folgt benannt (s. Checkliste im Kapitel Systemkonstruktion):

:reset-pointer ()

Synchronisiert die interne Repräsentation der Regeln mit der auf Wissensbasisebene benutzten abstrakten Form. (Zur Zeit sind interne und abstrakte Repräsentation identisch, so daß die Methode entbehrlich wäre.)

:rule-statistics ()

Liefert statistische Information über die Regeln. Realisiert den regelspezifischen Anteil der auf Gesamtsystemebene benutzten Methode **kb-inform**.

:unparse-rules ()

Erzeugt eine externe wiedereinlesbare Repräsentation der Regeln auf dem Standard-Ausgabemedium ***standard-output***.

Speziell für den Interpreter bzw. auch für die Entwicklungsumgebung werden folgende Methoden angeboten:

:find-matching-conclusions (*goal-specification*
&optional (*rule-set current-rule-set*))

Liefert eine Liste aller Regeln des Regelpaketes *rule-set*, deren Aktionsteile *goal-specification* entsprechen, d.h. die eine Konklusion enthalten, deren erster oder deren beide erste Bestandteile mit der *goal-specification* im Falle eines Atoms oder einer zweielementigen Liste übereinstimmen.

:inif (*term* &optional (*rule-set current-rule-set*))

Liefert eine Liste aller Regeln, die *term* in ihrem IF-Teil enthalten.

:inthen (*term* &optional (*rule-set current-rule-set*))

Liefert eine Liste aller Regeln, die *term* in ihrem THEN-Teil enthalten.

5.4.2 data-base

Das Flavor **data-base** verwaltet eine virtuelle dynamische Datenbasis für den Regel-Interpreter und stellt die grundlegenden Methoden für die Erklärungskomponente zur Verfügung. Es bedient sich bei der Datenverwaltung über das Flavor **processor-core** der Dienste des Meta-Prozessors, und sammelt im Verlauf der Auswertung Informationen, die für Erklärungszwecke verwendet werden können.

Die Instanzvariable **rules-tried** hält die im Verlauf der Auswertung überprüften Regeln, die Variable **rules-used**, die tatsächlich benutzten Regeln fest. Die Variable **hypotheses-verified** speichert die verifizierten Hypothesen und die Variable **justification-list** ist eine Liste, die zu jedem Ausdruck, dessen Wert ermittelt wurde, eine Begründung für diesen Wert enthält. Diese Begründung bildet die Grundlage für die Erklärung.

Als Typen der Begründung eines Ausdrucks — also einer Prämisse oder Aktion — werden von **data-base** festgehalten:

:USER-YES

der Ausdruck ist aufgrund einer Frage an den Benutzer bejaht worden.

:USER-NO

der Ausdruck ist aufgrund einer Frage an den Benutzer verneint worden.

:UNKNOWN

der Ausdruck ist nach einer Frage an den Benutzer unbekannt geblieben.

(:UNPROVABLE *rule-set*)

der Ausdruck ist mit Hilfe des Regelpaketes *rule-set* nicht beweisbar.

(:RULE-ACTION (*rule-set rule*))

der Ausdruck wurde durch die Aktion der Regel *rule* des Regelpaketes *rule-set* ermittelt.

Die Methoden, die für die Evaluierung von Prämissen und Aktionen im Interpreter benutzt werden, also **:recall**, **:remember** und **:ask-user**, senden jeweils eine **:eval** Nachricht mit dem entsprechenden Modus **:recall**, **:remember** und **:ask** an den Meta-Prozessor. Für eine Übersicht über alle Methoden von **data-base** s. Anhang A.

5.4.3 rule-interpreter

Das Flavor **rule-interpreter** realisiert die Vorwärts- und Rückwärtsauswertung von Regelpaketen. Dazu dienen die Methoden **:start-forward** und **:test-hypotheses**. Die Implementierung dieser Auswertungsstrategien erfolgt datengetrieben. Dies soll am Beispiel der Vorwärtsauswertung beispielhaft näher erläutert werden.

Die Methode **:start-forward** hat als (optionale) Parameter den Regelpaketnamen, den Bezeichner für eine Kontrollstrategie und eine Bedingung, jeweils mit einem Standardwert vorbesetzt. Die Methode sendet eine Nachricht an den Regelinterpreter, deren Selektor gerade der Bezeichner der Kontrollstrategie ist, d.h. für jede realisierte Kontrollstrategie gibt es eine entsprechend benannte Methode. So kann eine neue Kontrollstrategie einfach dadurch eingeführt werden, daß eine neue Methode mit entsprechendem Namen definiert wird.

```
(defmethod (rule-interpreter :start-forward)
  (&optional (rule-set-name current-rule-set)
    (control-structure :do-all)
    (condition t)
    (bindings nil))
  "Start rule evaluation by forward chaining."
  (catch 'RULE-INTERPRETER-STOP-TAG
    (send-message self control-structure rule-set-name condition bindings)))

(defmethod (rule-interpreter :do-all) (rule-set-name
  &optional ignore (bindings nil))
  "Internal method. Forward evaluation with DO-ALL control strategy."
  (prog ((rule-set (send-message self :get-rule-set rule-set-name nil bindings))
    trules next-rule result last-result)
    (setq trules (rule-set-rules rule-set))
    A (if (null trules) (return result))
    (setq next-rule (pop trules))
    (setq last-result
      (send-message self :try-rule next-rule rule-set 'FORWARD))
    (if last-result (setq result last-result))
    (go A)))
```

Abbildung 5.3: Vorwärtsauswertung mit DO-ALL-Kontrollstrategie

Für jede der vier Vorwärtsstrategien **:do-one**, **:do-all**, **:while-one** und **:while-all** existiert eine gleichnamige Methode. Die eigentliche Arbeit in diesen Methoden wird von der Methode **:try-rule** geleistet. Sie versucht eine einzelne Regel eines Regelpaketes vorwärts oder rückwärts entsprechend dem Wert des Parameters *mode* auszuführen. Neben den Verwaltungsarbeiten – Methoden **:add-rule-tried** und **:add-rule-used** definiert in **database** – wird hier die Anwendbarkeit der Regel durch Auswertung der Prämissen geprüft und gegebenenfalls der Aktionsteil ausgeführt. Diese Aufgaben werden in den Makros **test-if** und **usethe**n wieder jeweils datengetrieben implementiert. Dazu ist Voraussetzung, das auf der Eigenschaftsliste der als Junktoren oder Aktionstypen verwendeten Symbole unter dem Indikator FORWARD oder BACKWARD der Selektor der Methode abgelegt ist, die die entsprechen Auswertung implementiert. Das Makro **defjunctor** wird dazu

benutzt, diese Zuordnung für eine Methode zu treffen. Die folgenden Ausschnitte aus dem Kode zeigen Beispiele dieser datengetriebenen Implementierung.

```
(defmacro defjuncter (name rule-interpreter-method mode)
  "Establish the association of operators in rules with mode and method
for execution."
  `(progn (setf (get ',name ',mode) ',rule-interpreter-method)
    ',name))

(defun get-op-def (symbol index)
  "Retrieves selector-names attached with operators which may appear in rules."
  (let ((fn (get symbol index)))
    (if (null fn)
      (error (getentry method-property-error-fstr rule-io-table)
        symbol index)
      fn)))

(defmacro testif (rule left-hand-side rule-set mode)
  `(send-message self (get-op-def (get-juncter ,left-hand-side) ,mode)
    ,rule
    (get-rule-conditions ,left-hand-side)
    ,rule-set))

(defmacro usethen (rule right-hand-side rule-set mode)
  `(send-message self (get-op-def (get-action-type ,right-hand-side) ,mode)
    ,rule
    (get-rule-actions ,right-hand-side)
    ,rule-set))

(defmacro instantiate-pattern (alist pattern)
  `(sublis ,alist ,pattern))

(defaction (rule-interpreter :try-rule) (rule rule-set mode)
  "Internal method."
  (send-message self :add-rule-tried `((,rule-set-name rule-set) ,rule))
  (let ((lhs-evaluation-result
    (testif rule (rule-left-hand-side rule) rule-set mode)))
    (cond ((null lhs-evaluation-result) nil)
      ((eq lhs-evaluation-result T)
        (progn (usethen rule (rule-right-hand-side rule) rule-set mode)
          (send-message self :add-rule-used
            `((,rule-set-name rule-set) ,rule))))))
  (t ;; the result is a stream of environments:
    ;; environment-stream ::= (<env1> ... <envN>)
    ;; <env> ::= ((<var1> . <value1>) ... (<varN> . <valueN>))
    (do ((envs lhs-evaluation-result (rest envs))
      (result nil))
      ((null envs) result)
      (let* ((rule-instance (instantiate-pattern (first envs) rule))
        (use-result (usethen rule-instance
          (rule-right-hand-side rule-instance)
          rule-set
```

```

                                mode)))
    (when use-result
      (setq result use-result)
      (send-message self :add-rule-used
        '(', (rule-set-name rule-set) ,rule-instance)))))))))

(defjunctor $and :and-forward FORWARD)

(defaction (rule-interpreter :and-forward) (rule conditions rule-set)
  "AND forward control strategy for rule evaluation."
  (declare (ignore rule rule-set))
  (dolist (condition conditions t)
    (if (is-undetermined-or-null (send-message self :recall condition))
        (return nil))))

(defjunctor $or :or-forward FORWARD)

(defmethod (rule-interpreter :or-forward) (rule conditions rule-set)
  "OR forward control strategy for rule evaluation."
  (declare (ignore rule rule-set))
  (dolist (condition conditions nil)
    (if (not (is-undetermined-or-null (send-message self :recall condition)))
        (return t))))

(defjunctor $conclude :conclude FORWARD)
(defjunctor $conclude :conclude BACKWARD)

(defaction (rule-interpreter :conclude) (rule actions rule-set)
  "Performs rule conclusions of action type $CONCLUDE.
The value of :conclude is the value of the last action executed successfully."
  (let (result)
    (dolist (action actions result)
      (setq result (or (send-message self :remember action rule-set rule)
                       result)))))

```

Die beschriebene datengetriebene Implementierungstechnik ist ebenso für die Rückwärtsauswertung gewählt, so daß der Regel-Interpreter in allen seinen Teilen allein durch Hinzufügen neuer Methoden ohne Modifikation bestehenden Codes beliebig erweiterbar ist. Das gilt für neue Kontrollstrategien ebenso wie für neue Junktoren in Prämissen oder für neue Aktionstypen.

Neben der Methode **:test-hypotheses** gibt es eine zweite Methode zur Rückwärtsauswertung von Regel mit dem Namen **:obtain**. Diese Methode erlaubt eine Menge von Hypothesen in Form einer *goal-specification* anzugeben. Sie kann bestehen aus

1. einem Atom.
Diese Form der goal-specification wird expandiert zu einer Liste aller Aktionen von Regeln der Regelmenge, die das Atom als erstes Element enthalten.
2. einer zweielementigen Liste.
Diese Form der goal-specification wird expandiert zu einer Liste aller Aktionen von Regeln der Regelmenge, die mit den ersten zwei Elementen dieser Liste beginnen.

Benutzt kann dies werden, um z.B. in der ersten Form alle Hypothesen, die sich auf eine Instanz beziehen, oder in der zweiten Form, alle Hypothesen bzgl. eines Slots einer Instanz zu bearbeiten.

5.5 Der Mini-Rule-Processor

Der **mini-rule-processor** fügt der reinen Funktion des **basic-rule-processor** Möglichkeiten des Tracing hinzu. Dies wird realisiert durch das Flavor **rule-trace-mixin**. Dazu werden zu einigen der wichtigen Methoden des Flavor **rule-interpreter** `:before` Dämonen definiert. Die Ausgabe des Trace wird mittels der Nachricht **:send-rule-trace-window :format <args>** an den Metaprozessor weitergereicht. Die Methode **:send-rule-trace-window**, die in **mini-rule-mixin** definiert ist, schickt lediglich alle Nachrichten an das an die Instanzenvariable **rule-trace-window** gebundene Objekt weiter. Dieses Objekt, das die Nachricht **:format &rest <args>** verstehen muß, wird vom jeweiligen **<interface-mixin>** zugeordnet. Die **<args>** sind in der Form von **items** für ein **mouse-sensitive-text-scroll-window** aufgebaut (s. [LMM]). Ein solches Fenster ist rollbar und enthält maussensitive Stellen. Maussensitiv sind Regelnamen, deren Definition bei Auswahl gezeigt wird, sowie Hypothesen, für die bei Auswahl angezeigt wird, in welchem Zusammenhang sie zu verifizieren versucht werden. Steht in dem jeweiligen **<interface-mixin>** kein solcher Fenstertyp zur Verfügung, müssen nur die Stringanteile aus den **items** herausgefiltert werden, die dann mit Standard-Ausgabefunktionen behandelt werden können.

Das Flavor **rule-trace-mixin** enthält eine Instanzenvariable **rule-trace**, deren Wert anzeigt, ob das Tracing eingeschaltet ist oder nicht. Die Methode **:toggle-rule-trace** schaltet zwischen diesen Zuständen um.

Für folgende Methoden wird ein `:before` Dämon definiert:

- :before :verify-hypothesis** Erzeugt für jeden Versuch, eine Hypothese mittels Rückwärtsverkettung auf oberster Ebene zu beweisen, einen Eintrag. Der Eintrag ist durch Leerzeilen abgesetzt und enthält keine maussensitiven Stellen.
- :before :try-rule** Erzeugt für jeden Versuch, eine Regel anzuwenden, einen Eintrag. Der Regelname ist maussensitiv und es wird festgehalten, ob die Regel vorwärts oder rückwärts ausgewertet werden soll.
- :before :in-then-part** Erzeugt für jeden Versuch, einen Term zu verifizieren, einen Eintrag. Der Term ist maussensitiv. Zeigt bei Rückwärtsverkettung einen Versuch an, durch Regelanwendung bisher fehlende Voraussetzungen für eine Regelanwendung zu bestimmen.
- :before :ask-user** Erzeugt für jeden Versuch, einen Term durch Befragen des Benutzers zu verifizieren, einen Eintrag. Die Regel, in der der Term auftritt, ist wieder maussensitiv, der Term selbst nicht.
- :before :remember** Erzeugt einen Eintrag, der den Erfolg festhält, einen Term zu verifizieren. Die Regel, in der der Term auftritt, ist wieder maussensitiv, der Term selbst nicht.

5.5.1 Der Normal-Rule-Processor

Der **normal-rule-processor** erweitert **mini-rule-processor** um Aspekte der Erklärung und Unterstützung bei der Entwicklung von Regelpaketen. Die durch das Flavor **normal-rule-mixin** an der Benutzerschnittstelle verfügbaren Methoden benutzen stark die Möglichkeiten von Menüs nach festen Strategien. Will man hier Änderungen vornehmen, so muß man auf die grundlegenden, nicht an der Benutzerschnittstelle verfügbaren Methoden zurückgreifen. In diesem Bereich hat das Experimentieren erst begonnen und es sind gerade hier manche Änderungen zu erwarten. Angestrebt wird, im Regelprozessor nur die grundlegenden unterstützenden Methoden zu belassen, und alle, die Strategie und Ausprägung von Dialogen betreffenden Aspekte, erst auf der Ebene der **<processor-mixins>** bzw. der **<interface-mixins>** festzulegen. Die nachfolgenden Beschreibungen beschränken sich auf die in **normal-rule-mixin** exportierten Methoden.

rule-explain-mixin

Das Flavor **rule-explain-mixin** stellt die Erklärungskomponente des Regel-Prozessors dar. Es wertet die im Flavor **rule-base** aufgezeichneten Informationen über die Auswertung der Regeln aus. Die primäre Methode für Erklärungen ist **:explain-results**. Sie benutzt das **explanation-window** als Ausgabemedium, listet dort alle als wahr ermittelten Fakten und stellt ein Auswahlmenü mit folgenden Einträgen zur Verfügung:

no

Verlassen des Erklärungsmodus.

How?

Erzeugt ein Auswahlmenü aller im Auswertungsprozeß als **wahr** abgeleiteten Fakten, für die man sich erklären lassen kann, wie sie abgeleitet wurden.

How All?

Erzeugt ein Auswahlmenü aller im Auswertungsprozeß *untersuchten* Fakten, für die man sich erklären lassen kann, ob und gegebenenfalls wie sie verifiziert wurden.

Print Rule

Im Gegensatz zu der Anzeige von Regeln in **basic-rule-mixin** werden hier die Regeln in einem Menü angezeigt. Dabei sind solche Bestandteile einer Regel maussensitiv, die in anderen Regeln im IF- bzw. THEN-Teil benutzt werden.

rule-develop-mixin

Das **rule-develop-mixin** implementiert die Leistungen, die in einer Entwicklungsumgebung für Regeln benötigt werden. Dazu gehören Methoden, um Regeln zu listen, oder nach bestimmten Kriterien Regeln zu suchen. Bei allen diesen Methoden wird starker Gebrauch von Auswahlmenüs gemacht, einmal z.B. bei der Bestimmung, welche Regeln in welchem Regelpaket man bearbeiten will, zum anderen werden z.B. die Regeln in einem Menü angezeigt, wobei ihre Bestandteile teilweise wieder maussensitiv sind und zu weiteren Auswahlaktionen benutzt werden können.

:print-rule

Zum Zeigen einer ausgewählten Regel eines ausgewählten Regelpaketes.

:inspect-terms

Zum Zeigen von Regeln, die gewissen Auswahlkriterien entsprechen.

5.6 Importierte Leistungen

Neben den Standard-IO-Leistungen der Benutzerschnittstelle benutzt der Regel-Prozessor die **:eval** Methode des Meta-Prozessors, um alle im IF- und THEN-Teil von Regeln auftretenden Ausdrücke auswerten zu lassen. Dabei werden folgende Auswertungsmodi benutzt:

:recall Ermittle die Gültigkeit eines Ausdrucks ohne Befragen des Benutzers. Mögliche Werte dieser Nachricht sind **unbestimmt**, **nein** (nil) oder **ja** (nonnil).

Dieser Modus wird für alle Ausdrücke im IF-Teil von Regeln verwendet, für die ein Befragen des Benutzers bei noch unbestimmtem Wert **nicht** vorgesehen ist. Dies trifft für die Junktoren \$AND und \$OR zu.

:ask Ermittle die Gültigkeit eines Ausdrucks u.U. durch Befragen des Benutzers, falls er noch unbestimmt ist. Mögliche Werte dieser Nachricht sind **unbestimmt**, **nein** (nil), **ja** (nonnil) oder **why**, falls sich der Benutzer den Zweck der Frage nach dem Wert dieses Ausdrucks erklären lassen will.

Dieser Modus wird für alle Ausdrücke im IF-Teil von Regeln verwendet, für die ein Befragen des Benutzers bei noch unbestimmtem Wert vorgenommen werden soll. Dies trifft für die Junktoren ?AND und ?OR zu. Ebenso wird er benutzt für Ausdrücke im THEN-Teil einer Regel mit dem Aktionstyp \$ASK.

:recall-immediate Ermittle die Gültigkeit eines Ausdrucks, jedoch nur, falls keine aufwendigen Berechnungen dazu nötig sind. Mögliche Werte auf diese Nachricht sind **unbestimmt**, **nil** oder **nonnil**.

Dieser Modus wird nur innerhalb des Erklärungsteils benutzt und soll dort aufwendige Neuberechnungen z.B. bei Prologausdrücken vermeiden helfen. Sind solche nicht notwendig, so sollte sich dieser Modus wie :recall verhalten.

:remember Halte den Wert dieses Ausdrucks fest. Mögliche Werte auf diese Nachricht sind **nil**, falls der Wert bereits bekannt ist, oder **nonnil** sonst.

Dieser Modus wird für Ausdrücke im THEN-Teil von Regeln für die Aktionstypen \$conclude und \$execute angewendet.

5.7 Exportierte Leistungen

Die Flavors **basic-rule-mixin**, **mini-rule-mixin** und **normal-rule-mixin** definieren die Leistungen der jeweiligen Ausprägungen des Regel-Prozessors, die im Gesamtsystem verfügbar sein sollen. Der Aufbau folgt den in Kapitel 3 beschriebenen Prinzipien zur Einbindung eines Prozessors in das Gesamtsystem. Einige der exportierten Methoden (s.

Anhang A) sind auch als Funktionen bzw. Makros verfügbar, so daß sie im Instruktionsteil einer Wissensbasis einfacher aufrufbar sind.

Da der Regel-Prozessor selbst keinen Wissensrepräsentationsformalismus anbietet, werden in **basic-rule-mixin** keine Typerkennungsmakros definiert. Nur die Konstruktoren für Regelpakete **DEFRULE-SET** bzw. **RULE-SET** und **HYPOTHESES** werden hier definiert. Sie führen einen Syntaxcheck durch und speichern die Werte auf den Instanzvariablen **rules** und **hypotheses** ab. Die Methoden sind derart einfach, daß sie hier keiner näheren Erklärung bedürfen.

Kapitel 6

Spezifikation des Frame-Prozessors

6.1 Anforderungen

Die objektorientierte Form der Wissensrepräsentation in **BABYLON** übernimmt wesentliche Züge des Flavorsystems von ZetaLisp [LMM]. Objekte sind **Instanzen** oder Ausprägungen von Objekttypen, die in **BABYLON Frames** genannt werden. Ein Frame legt die **Slots** oder Attribute seiner Instanzen fest und sein Verhalten. Das Verhalten wird in **BABYLON** durch **Behaviors** charakterisiert. Ein Behavior beschreibt die Aktionen, die ablaufen, wenn einer Instanz eine entsprechende Nachricht zukommt.

Frames können als Komponenten Frames enthalten, deren Eigenschaften sie erben. Diese Frames werden als **Super-Frames** oder übergeordnete Frames bezeichnet. Da ein Frame mehrere Super-Frames besitzen kann, wird von multipler Vererbung gesprochen. Der Algorithmus, nach dem Slots und Behaviors vererbt werden, die in mehr als einem Super-Frame vorkommen, entspricht dem des Flavorsystems.

Ererbte Behaviors können durch **:after-** bzw. **:before-**Dämonen modifiziert werden. Andere Formen der Kombination werden nicht unterstützt.

Das Framekonzept von **BABYLON** geht in folgenden Punkten über das Flavorkonzept hinaus:

Mit Slots lassen sich Meta-Informationen verknüpfen, die ggf. automatisch ausgewertet werden. Diese Meta-Informationen werden in **Properties** gespeichert. Der Benutzer hat die Möglichkeit, zusätzlich zu den systemseitig angebotenen Properties eigene zu definieren.

Weiter läßt **BABYLON** als Slotwerte **aktive Werte** zu. Sie sind dadurch ausgezeichnet, daß bei einem lesenden oder schreibenden Zugriff ein für den aktiven Wert spezifisches GET- bzw. PUT-Behavior ausgelöst wird.

6.2 Einordnung

Der Frame-Prozessor realisiert die objektorientierte Form der Wissensrepräsentation im Gesamtsystem und bietet dem Benutzer die bei der Entwicklung und der Konsultation eines Expertensystems benötigten Hilfen für diese Repräsentationsform von Wissen. Häufig wird mit dem objektorientierten Teil die Grundlage gebildet, auf der Teile

der Wissensbasis in regel- oder logikorientierter Darstellung aufbauen. Trotzdem hat der Frame-Prozessor keine übergeordnete Stellung gegenüber dem Regel- oder Logikprozessor s. Kapitel 2.

Der Frame-Prozessor wird in drei Varianten mit unterschiedlichem Leistungsumfang angeboten. Der **Basic-Frame-Processor**, die einfachste Variante, bietet die unabdingbaren Leistungen des Frameformalismus, der **Mini-Frame-Processor** sorgt für die automatische Auswertung der Standard-Properties **:possible-values** und **:explain-answers**, beim **Normal-Frame-Processor** kommt die Unterstützung aktiver Werte hinzu.

6.3 Funktionsbeschreibung

Die folgende Beschreibung gibt eine parallele Darstellung der drei Varianten. Soweit nicht ausdrücklich anders vermerkt, stehen die beschriebenen Leistungen bereits in der einfachsten Variante zur Verfügung.

6.3.1 Frames

Eine Frame-Definition faßt eine Klasse von zu beschreibenden Objekten unter einem eigenen Namen zusammen. Sie enthält Angaben zu Slots, Default-Werten für Slots und Super-Frames.

```
(DEFFRAME <Frame-Name>
  {(SUPERS . <Liste von Frames>)}
  (SLOTS . <Liste von Slot-Spezifikationen>))

<Slot-Spezifikation> ::=
  <Slot-Name> |
  (<Slot-Name N> <Default-Wert> . <Liste von Property-Wert-Paaren>)

<Property-Wert-Paar> ::=
  <Property-Schluesselwort> <Default-Wert>
```

VERERBUNG

In der Frame-Definition können die Frames angegeben werden, die dem Frame ihre Slots und Behaviors vererben. Der Benutzer hat auf die Art der Vererbung wenig Einfluß, da lediglich Slots mit Default-Werten und Properties, sowie Behaviors übernommen werden. Er kann jedoch durch Spezifikation einer Rangfolge unter den Frames festlegen, welcher Frame für die Vererbung herangezogen werden soll. Im Konfliktfall wird der Frame mit höherem Rang zur Vererbung herangezogen. Der Vererbungsmechanismus entspricht dem des Flavor-Vererbungsmechanismus [LMM].

SLOT-SPEZIFIKATION

Eine Slot-Spezifikation besteht aus einem Slot-Namen und Property-Default-Wert-Paaren. Der Default-Wert definiert einen Wert, der als Wert der Property benutzt wird, wenn der

Property kein anderer Wert in den Instanzen zugeordnet wird. Die Properties sind bis auf die **:value**-Property optional. Die Property **:value** ist ausgezeichnet, ihr Name kann wegfallen. Die optionalen Properties lassen sich wie folgt unterteilen: Properties, die sich auf die Property **:value** beziehen; (sie werden als Standard-Properties bezeichnet und sind vom System vorgegeben) und selbstdefinierte Properties (beliebig viele). Den Standard-Properties ist der Abschnitt 6.3.4 gewidmet.

DEFAULT-WERTE

Als Default-Wert ist eine beliebige Lisp-Form, ein multipler Wert, ein aktiver Wert oder einer der speziellen Werte „-“ (unbestimmt) bzw. **unknown** (unbekannt) zulässig.

Lisp-Form läßt einen beliebigen Ausdruck in Lisp zu. Er wird **nicht** ausgewertet.

Multiple Werte können ebenfalls jeder Property zugewiesen werden. Sie geben einen mehrwertigen Zustand wieder. Sie werden folgendermaßen definiert:

```
(:MULTIPLE-VALUE <Wert 1> . . . <Wert N>)
```

Aktive Werte können jeder Property zugewiesen werden, falls ein **Normal-Frame-Processor** verwendet wird. Sie werden im Abschnitt 6.3.5 beschrieben. Ihre Spezifikation lautet:

```
(ACTIVE-VALUE <lokaler Wert> <GET-Behavior> <PUT-Behavior>)
```

„-“ ist das Zeichen für einen noch nicht bestimmten Wert.

unknown bedeutet, daß der Wert der Property unbekannt ist.

6.3.2 Behaviors

Ein Behavior beschreibt eine Funktion, die ausgeführt wird, wenn eine Instanz eine entsprechende Nachricht empfängt. Behaviors werden definiert durch:

```
(DEFBEHAVIOR (<Frame-Name> <Typ> <Behavior-Name>)
              <Lambda-Liste>
              . <Koerper> )
```

Als <Lambda-Liste> und <Koerper> ist alles zulässig, was in einer Lisp-Funktionsdefinition an entsprechender Stelle verwendet werden kann. Wird ein <Typ> angegeben, so muß ein Grund-Behavior (Behavior ohne <Typ>) mit demselben Namen definiert werden. Als <Typ> sind **:before** und **:after** zulässig. Im ersten Fall wird das Behavior als Dämon vor dem Grund-Behavior ausgeführt, im zweiten danach. Die <Lambda-Liste> eines Dämons muß mit der des Grund-Behaviors übereinstimmen.

Innerhalb von <Koerper> kann der Zugriff auf den Wert (einer Property) eines Slots derjenigen Instanz, die das Behavior ausführt, folgendermaßen erfolgen:

```
($VALUE <Slot-Name>) bzw. ($VALUE <Slot-Name> <Prop-Name>)
```

Modifikation von Werten erfolgt durch den allgemeinen Zuweisungsoperator **SETF**

```
(SETF ($VALUE <Slot-Name>) <neuer Wert>)
bzw.
(SETF ($VALUE <Slot-Name> <Prop-Name>) <neuer Wert>)
```

Zum Zugriff auf Slots und Properties und zur Fragegenerierung sind Standard-Behaviors definiert (s. 2.3 Schnittstellenbeschreibung).

6.3.3 Instanzen

Instanzen beschreiben Individuen, deren Merkmale in Frames festgelegt wurden. Instanzen eines Frames werden definiert durch:

```
(DEFINSTANCE <Instanz-Name> OF <Frame-Name>
  WITH
    <Slot-Name 1> = <Slot-Initialisierung 1>
    . . .
    <Slot-Name N> = <Slot-Initialisierung N>)

<Slot-Initialisierung> ::= <Atom> | <multipler Wert> | <aktiver Wert> |
  ({<Wert>} . <Liste von Property-Wert-Paaren>)

<Wert> ::= <Atom> | <multipler Wert> | <aktiver Wert> | <Liste>
```

Soweit keine Properties genannt sind werden die angegebenen Werte dem jeweiligen Slot (der **:value**-Property dieses Slots) zugewiesen. Die Werte werden dabei **nicht** evaluiert. Soll keine Initialisierung erfolgen, so können WITH und die nachfolgenden Spezifikationen entfallen.

Default-Werte, die aktive Werte darstellen, werden bei der Initialisierung **nur** überschrieben, falls der entsprechende Initialisierungswert selbst ein aktiver Wert ist. Andernfalls wird das PUT-Behavior aktiviert.

Falls für ein Slot **:possible-values** spezifiziert wurden, wird überprüft, ob Default- und Initialisierungswerte zulässig sind, sofern es sich nicht um aktive Werte handelt. Diese Überprüfung kann dadurch ausgeschaltet werden, daß die Instanzvariable Check des Frame-Prozessors auf NIL gesetzt wird.

Die beiden letzten Anmerkungen setzen natürlich voraus, daß es sich um einen **Normal-** bzw. **Mini-Frame-Processor** handelt.

Nach der Generierung einer Instanz wird dieser eine **:initialize**-Nachricht geschickt mit der WITH-Spezifikation als Argument. Sie ist als Basis-Methode für **:before-** bzw. **:after-**Behaviors gedacht.

6.3.4 Standard-Properties für Frameslots

Die Standard-Properties beziehen sich auf die **:value**-Property. Sie lassen sich in zwei Klassen unterteilen: Standard-Properties zur Wertebeschreibung und Fragegestaltung.

STANDARD-PROPERTIES ZUR WERTEBESCHREIBUNG

Sie beschreiben die Werte, welche die **:value**-Property annehmen darf. Es existiert eine Property dieser Art, die **:possible-values**-Property. Diese Property wird **nicht** von **Basic-Frame-Processor** ausgewertet. Sie kann Wertespezifikationen annehmen, die sich in verschiedene Klassen einteilen lassen:

beliebiger Wert

:any - der Slot kann jeden beliebigen Wert besitzen, entspricht dem Fehlen der **:possible-values**-Property.

spezieller Wert

Es existieren die folgenden Typen **:boolean**, **:symbol**, **:number**, **:list**, **:string** und **:instance-of**.

:boolean - der Slot kann nur die Werte T (true) und NIL (false) besitzen,

:symbol - der Wert des Slots kann ein beliebiges Symbol sein,

:number - es ist eine beliebige (reelle oder natürliche) darstellbare Zahl zulässig,

:list - der Wert muß eine Liste sein,

:string - der Wert muß ein String sein,

(**:instance-of** <Frame-Name>) - als Wert des Slots kann nur der Name einer Instanz von <Frame-Name> vorkommen.

Mengenoperationen

Es stehen **:interval**, **:one-of** und **:some-of** zur Verfügung.

(**:interval** <untere Grenze> <obere Grenze>) - der Wert muß eine Zahl sein, die in dem Intervall zwischen der Zahl <untere Grenze> und der Zahl <obere Grenze> einschließlich liegt,

(**:one-of** . <Liste von Werten>) - die Elemente der <Liste von Werten> sind beliebig, der Wert des Slots muß einer der Werte dieser Liste sein.

(**:some-of** . <Liste von Werten>) - der Slot kann mehrere Werte besitzen (mehrwertiger Slot), die alle in der <Liste von Werten> enthalten sein müssen.

Der Benutzer kann für die **:possible-values**-Property neue Wertespezifikationen mithilfe eines Behaviors definieren:

```
(DEFINE-POSSIBLE-VALUES-BEHAVIOR (<Frame-Name> <Schluesselwort>)
  (<Parameter>)
  . <Koerper>)
```

Für alle Instanzen des Frames <Frame-Name> kann dann <Schluesselwort> als Wert von **:possible-values** angegeben werden. In <Koerper> wird spezifiziert, wie die Zugehörigkeit zur gewünschten Wertemenge getestet werden soll. Der zu testende Wert wird an <Parameter> gebunden.

STANDARD-PROPERTIES ZUR FRAGEGESTALTUNG

Sie nehmen Einfluß auf den Fragetext, wenn das System versucht, den Wert der **:value**-Property durch Fragen an den Benutzer zu bestimmen.

:ask

Falls der Wert des Slots vom Benutzer erfragt wird, wird anstelle des Standard-Fragetextes „Was gilt für: <Instanz-Name> <Slot-Name> ?“ ein vom Benutzer festgelegter Text benutzt. Der Wert der Property **:ask** hat die Form:

(<Format-String> . {<O-S-Liste>})

Der <Format-String> ist ein String für den Lisp-Formatierer [LMM]. Die <O-S-Liste> gibt an, in welcher Reihenfolge bei der Generierung der Frage für das Vorkommen von z.B. ' S' im <Format-String> der Name des Objektes (O) und der Name des Slots (S) einzusetzen ist.

:explain-answers

Wenn der Benutzer nach einem Wert gefragt wird, dann kann er sich die zulässigen Antworten auf Fragen des Systems erläutern lassen. Diese Möglichkeit besteht **erst** beim **Mini-Frame-Processor**. Dazu muß **:explain-answers** einen Wert der Form:

(<Format-String> . {<O-S-Liste>})

besitzen. Daraus wird auf Anfrage des Benutzers ein erläuternder Text generiert (auf dieselbe Weise wie bei **:ask**). Falls die Property **:possible-values** des Slots einen der Werte **:one-of** oder **:some-of** besitzt, so kann **:explain-answers** auch eine Liste von Erklärungen zu den einzelnen möglichen Werten enthalten:

(<Wert> <Format-String> . {<O-S-Liste>})

6.3.5 Aktive Werte

Aktive Werte werden nur vom **Normal-Frame-Processor** unterstützt. Die beiden anderen Varianten behandeln sie wie gewöhnliche Listen.

DEFINITION AKTIVER WERTE

Aktive Werte sind Werte von Slots oder von Slot-Properties, die aus einem lokalen Wert, einem GET-Behavior und einem PUT-Behavior bestehen. Das GET-Behavior wird bei jedem lesenden Zugriff auf den Wert mit **:get**, das PUT-Behavior bei jedem schreibenden Zugriff auf den Wert mit **:set** oder **:put** ausgeführt. Geliefert bzw. als lokaler Wert eingetragen wird jeweils der Wert des GET-Behavior bzw. des PUT-Behavior. Aktive Werte werden folgendermaßen notiert:

(**ACTIVE-VALUE** <lokaler Wert> <GET-Behavior> <PUT-Behavior>)

Für die Argumente der beiden Funktionen gelten folgende Festlegungen:

GET-Behavior	<ol style="list-style-type: none"> 1. Argument: alter-lokaler-Wert 2. Argument: aktiver-Wert 3. Argument: Property-Name 4. Argument: Slot-Name
PUT-Behavior	<ol style="list-style-type: none"> 1. Argument: neuer-lokaler-Wert 2. Argument: aktiver-Wert 3. Argument: Property-Name 4. Argument: Slot-Name

Bei einem Zugriff auf einen aktiven Wert werden die aktuellen Parameterwerte vom System automatisch an die entsprechenden Parameter des GET- bzw. PUT-Behaviors gebunden.

Für eine oder beide Funktionen kann auch NIL angegeben werden. In diesem Fall verhält der aktive Wert sich bei dem entsprechenden Zugriff wie ein gewöhnlicher Wert.

Aktive Werte können verschachtelt werden, d.h. <lokaler Wert> kann selbst wieder ein aktiver Wert sein. Die Funktionen werden dann hintereinander ausgeführt, und zwar die GET-Behaviors in der Reihenfolge von innen nach außen (innerste zuerst, dann die nächstinnere usw.), die PUT-Behaviors in umgekehrter Reihenfolge, also von außen nach innen. Es wird jeweils der Wert der zuletzt ausgeführten Funktion an den ersten Parameter der als nächste auszuführenden Funktion gebunden.

STANDARDBEHAVIORS FÜR AKTIVE WERTE

BABYLON kennt folgende Standard-GET-Behaviors:

:first-fetch

<lokaler Wert> ist ein Lisp-Ausdruck, der beim ersten Lesezugriff evaluiert wird. Der Wert des Lisp-Ausdrucks wird als Wert (nicht als <lokaler Wert> !) des Slots bzw. der Property eingetragen, d.h. der Wert ist dann möglicherweise nicht mehr aktiv.

:get-indirect

<lokaler Wert> muß eine Liste der Form

$$(\text{<Instanz-Name> } \text{<Slot-Name> } \{\text{<Property-Name>}\})$$

sein. Beim Lesezugriff wird auf den Wert des Slots <Slot-Name> der Instanz <Instanz-Name> bzw. auf den Wert der Property <Property-Name> dieses Slots zugegriffen.

BABYLON kennt folgende Standard-PUT-Behaviors:

:put-indirect

<lokaler Wert> muß eine Liste der Form

$$(\text{<Instanz-Name> } \text{<Slot-Name> } \{\text{<Property-Name>}\})$$

sein. Beim Schreibzugriff wird der Wert des Slots $\langle \text{Slot-Name} \rangle$ der Instanz $\langle \text{Instanz-Name} \rangle$ bzw. der Wert der Property $\langle \text{Property-Name} \rangle$ dieses Slots überschrieben.

:no-update-permitted

Der Wert darf nicht überschrieben werden. Jeder Versuch, das zu tun, wird mit einer Fehlermeldung quittiert.

6.4 Schnittstellenbeschreibung

Die Schnittstelle zum Gesamtsystem besteht aus Methoden, die durch Versenden von Nachrichten an Objekte bzw. an den Frame-Prozessor aufgerufen werden, oder aus Funktionen.

6.4.1 Funktionen für Objekte

is-frame <i>Argument</i>	Funktion
liefert T, wenn das Argument der Name eines Frames ist, sonst NIL.	
is-instance <i>Argument</i>	Funktion
liefert T, wenn das Argument der Name einer Instanz ist, sonst NIL.	
get-instance <i>Instanz-Name</i>	Funktion
liefert die Instanz mit dem Namen Instanz-Name.	
get-instance-list <i>Frame-Name</i>	Funktion
liefert die Liste der Namen aller Instanzen von Frame-Name.	
get-all-instances <i>Frame-Name</i>	Funktion
liefert die Liste der Namen aller Instanzen von Frame-Name selbst und aller Instanzen seiner Subframes.	
set-instance-pointer <i>Symbol Instanz-Name</i>	Funktion
bewirkt, daß Symbol auf dieselbe Instanz zeigt wie Instanz-Name. Symbol wird dabei nicht evaluiert.	
get-supers <i>Frame-Name</i>	Funktion
liefert alle als SUPERS deklarierten (direkten) Oberklassen von Frame-Name.	
get-all-supers <i>Frame-Name</i>	Funktion
liefert alle direkten und indirekten Oberklassen von Frame-Name.	
get-subframes <i>Frame-Name</i>	Funktion
liefert alle Frames, für die Frame-Name direkte Oberklasse ist.	
get-all-subframes <i>Frame-Name</i>	Funktion
liefert alle Frames, für die Frame-Name direkte oder indirekte Oberklasse ist.	

6.4.2 Nachrichten für Objekte

Die Standard-Nachrichten können an alle Instanzen gesendet, von ihnen interpretiert und beantwortet werden. Anstelle von **send-message** kann „<-“ benutzt werden:

(<- <Instanz-Name> <Behavior-Name> . <Argumente>)

<Instanz-Name> muß zum Namen einer Instanz evaluieren, nicht zur Instanz selbst, im Gegensatz zu **send-message**. <Behavior-Name> wird nicht evaluiert, <Argumente> werden evaluiert, sofern <Behavior-Name> nicht der Name eines Slots ist (s.u.).

ZUGRIFF AUF SLOTS UND PROPERTIES

Auf die Properties kann sowohl lesend, als auch schreibend zugegriffen werden. Die Art des Schreibzugriffs kann unterschiedlich sein:

:get-value-only *SlotName* *Optional PropertyName* **Methode**
liefert den Wert bzw. den Wert der angegebenen Property.

:get *SlotName* *Optional PropertyName* **Methode**
wie **:get-value-only**, jedoch wird bei einem aktiven Wert das GET-Behavior aufgerufen.

:replace *SlotName* *NeuerWert* *Optional PropertyName* **Methode**
ersetzt den Wert des Slots durch einen neuen Wert. Dabei findet keine Kontrolle der Zulässigkeit von *NeuerWert* (entsprechend **:possible-values**) statt und aktive Werte werden überschrieben.

:set *SlotName* *NeuerWert* *Optional PropertyName* **Methode**
wie **:replace**, jedoch wird bei einem aktiven Wert das PUT-Behavior aufgerufen.

:put *Slot-Name* *NeuerWert* *Optional PropertyName* **Methode**
wie **:set**, jedoch wird der Wert von **:possible-values** berücksichtigt.

Die Methoden **:get** und **:set** sind dazu ausersehen, nach Bedarf spezialisiert zu werden. Dazu können die Methoden **:get-value-only** und **:replace** benutzt werden, die daher nicht spezialisiert werden sollten.

:delete-property *SlotName* *PropertyName* **Methode**
löscht die Property des Slots. Die Property **:value** kann nicht gelöscht werden.

NACHRICHTEN ZUR FRAGEGENERIERUNG

Es werden Fragen generiert, dem Benutzer gestellt und die Antworten als Werte von Properties eingetragen.

:ask *SlotName* *Optional Argument* **Methode**
Mit *Argument* := <ErwarteterWert> | <Prop-Name> | (<Prop-Name>
<ErwarteterWert>)

Es wird eine Frage nach dem Wert von *SlotName* bzw. nach dem Wert der Property von *SlotName* an den Benutzer gestellt. *ErwarteterWert* hat die Form ({<Relation>} <Wert>). Er gibt an, welcher Wert als Antwort erwartet wird, bzw. in welcher Relation der eingegebene Wert zu <Wert> erwartungsgemäß steht.

SONSTIGE NACHRICHTEN FÜR OBJEKTE

- :initialize** *With-Specification* **Methode**
 Dummy-Behavior, das nach Generierung einer Instanz aufrufen wird. Der Benutzer hat die Möglichkeit durch Spezialisierung dieses Behaviors oder durch Definition von **:before-** und **:after-**Dämonen für diese das Initialisierungsverhalten des Systems zu beeinflussen.
- :objekt-name** **Methode**
 liefert den Namen der Instanz, die die Nachricht erhält.
- :slots** **Methode**
 liefert die Liste aller (d.h. auch der ererbten) Slots der Instanz, die die Nachricht erhält.
- :type** *Optional FrameName* **Methode**
 Falls *FrameName* nicht angegeben ist, so wird der Name des Frames geliefert, dessen Instanz das Objekt ist. Ansonsten wird T oder NIL geliefert, je nachdem ob das Objekt Instanz von *FrameName* ist oder nicht.

SLOTS ALS NACHRICHTEN

Im Makro „<-“ können Namen von Slots anstelle von Behavior-Namen verwendet werden, und zwar:

Zum Lesen eines Slot-Wertes:

```
(<- <Instanz-Name> <Slot-name> {<Property-Name>})
```

gleichbedeutend mit

```
(<- <Instanz-Name> :get <Slot-name> {<Property-Name>})
```

Zum Überprüfen bzw. Modifizieren von Slot-Werten:

```
(<- <Instanz-Name> <Slot-Name> {<Property-Name>}  

   <Relation> <Ausdr> {<Modus>})
```

Die Wirkung dieser Nachricht für <Instanz-Name> hängt von <Modus> ab. Es gibt drei Möglichkeiten:

:recall

Der Default-Modus. Es wird getestet, ob der Wert des Slots, bzw. wenn angegeben, der Wert der Property <Property-Name> des Slots in Relation <Relation> zu <Ausdr> steht. Ist das der Fall, so wird T (wahr) geantwortet, sonst NIL (falsch).

:remember

Als <Relation> ist nur „=“ zulässig. Dem Slot bzw. der Property des Slots wird der Wert <Ausdr> zugewiesen, wenn <Ausdr> nicht gleich dem alten Wert ist. In diesem Fall wird NIL zurückgegeben.

:store

Als <Relation> ist nur „=“ zulässig. Dem Slot bzw. der Property des Slots wird <Ausdr> zugewiesen, auch wenn <Ausdr> gleich dem alten Wert ist.

Als $\langle \text{Relation} \rangle$ ist möglich:

- = ist gültig, wenn $\langle \text{Ausdr} \rangle$ gleich dem Slot- bzw. Property-Wert ist, oder, falls es sich um einen Slot mit mehreren Werten handelt, wenn $\langle \text{Ausdr} \rangle$ einer dieser Werte ist.
- > der Slot bzw. die Property darf nur *einen* Wert besitzen, der vom Typ **:number** sein muß. $\langle \text{Ausdr} \rangle$ muß zu einer Zahl ausgewertet werden. Die Relation gilt, wenn der Wert größer ist als $\langle \text{Ausdr} \rangle$, sonst nicht.
- >= dieselben Bedingungen wie bei >. Die Relation gilt, wenn der Wert größer oder gleich $\langle \text{Ausdr} \rangle$ ist.
- < dieselben Bedingungen wie bei >. Die Relation gilt, wenn der Wert kleiner ist als $\langle \text{Ausdr} \rangle$.
- <= dieselben Bedingungen wie bei >. Die Relation gilt, wenn der Wert kleiner oder gleich $\langle \text{Ausdr} \rangle$.
- one-of** $\langle \text{Ausdr} \rangle$ muß eine Liste sein. Die Relation gilt, wenn der Wert oder (bei mehreren Werten) einer der Werte des Slots bzw. der Property in der Liste enthalten ist.
- all-of** $\langle \text{Ausdr} \rangle$ muß eine Liste sein. Die Relation gilt, wenn der Wert von $\langle \text{Slot-Name} \rangle$ ein multipler Wert ist und alle Werte in $\langle \text{Ausdr} \rangle$ im multiplen Wert enthalten sind.
- between** der Wert des Slots bzw. der Property muß vom Typ **:number** sein, $\langle \text{Ausdr} \rangle$ eine Intervallspezifikation der Form ($\langle \text{untere Grenze} \rangle$ $\langle \text{obere Grenze} \rangle$). Die Relation ist gültig, wenn der Wert innerhalb des spezifizierten Bereiches (einschließlich der Grenzen) liegt.

Der Benutzer kann zusätzliche Relationen definieren durch:

```
(DEFINE-RELATION-BEHAVIOR (<Frame-Name> <Relations-Name>)
  (<Parameter1> <Parameter2>)
  . <Koerper> )
```

An die Parameter werden die auf Gültigkeit der Relation zu prüfenden Werte gebunden, $\langle \text{Koerper} \rangle$ enthält die Spezifikation des dabei auszuführenden Tests.

Für den Ausdruck $\langle \text{Ausdr} \rangle$ gibt es die folgenden Möglichkeiten:

(\$EVAL <Lisp-form>)

die $\langle \text{Lisp-Form} \rangle$ wird evaluiert und der Wert verwendet.

(<Instanz-Name> <Slot-Name> {<Property-Name>})

es wird der Wert des Slots $\langle \text{Slot-Name} \rangle$ von $\langle \text{Instanz-Name} \rangle$ verwendet.

(SELF <Slot-Name> {<Property-Name>})

der Wert des Slots $\langle \text{Slot-Name} \rangle$ bzw. der angegebenen Property dieses Slots derselben Instanz wird verwendet.

beliebiger anderer Ausdruck

der Ausdruck wird **unausgewertet** verwendet.

6.4.3 Frame-Referenzen

Frame-Referenzen sind Ausdrücke, die in Prämissen und Konklusionen von Produktionsregeln sowie in Prämissen von Klauseln auftreten können und sich auf Objektzustände beziehen. Es gibt folgende Arten von Frame-Referenzen:

Standard-Frame-Referenzen

($\langle \text{Instanz-Name} \rangle \langle \text{Slot-Name} \rangle \{ \langle \text{Property-Name} \rangle \} \langle \text{Relation} \rangle \langle \text{Ausdr} \rangle$)

Sie werden bei der Auswertung einer Regel bzw. Klausel übersetzt in Nachrichten (an die Instanz) der Form:

($\langle - \langle \text{Instanz-Name} \rangle \langle \text{Slot-Name} \rangle \{ \langle \text{Property-Name} \rangle \} \langle \text{Relation} \rangle \langle \text{Ausdr} \rangle \langle \text{Modus} \rangle$)

Dabei wird für $\langle \text{Modus} \rangle$ **:recall** eingesetzt, wenn die Frame-Referenz als Prämisse vorkommt, **:remember** wenn sie als Konklusion auftritt.

Für $\langle \text{Relation} \rangle$ und $\langle \text{Ausdr} \rangle$ bestehen dieselben Möglichkeiten wie in der entsprechenden Nachricht.

Standard-Behavior-Referenzen

($\langle \text{Instanz-Name} \rangle \langle \text{Behavior-Name} \rangle . \langle \text{Argumente} \rangle$)

Bei der Auswertung einer Regel oder einer Klausel wird eine Nachricht an $\langle \text{Instanz-Name} \rangle$ gesendet, das Behavior $\langle \text{Behavior-Name} \rangle$ mit den Argumenten $\langle \text{Argumente} \rangle$ auszuführen. Die Definitionen von Behaviors, die in Standard-Behavior-Referenzen verwendet werden sollen, müssen als letzten Parameter *Modus* besitzen. An diesen Parameter wird vom System **:recall** gebunden, wenn die Behavior-Referenz als Prämisse auftritt, **:remember** wenn sie als Konklusion auftritt.

6.4.4 Nachrichten für den Frame-Prozessor

Für die Nachrichten, mit denen auf Slots zugegriffen werden, gibt es gleichlautende Nachrichten an den Frame-Prozessor. Erstes Argument dieser Nachrichten ist der Name einer Instanz. An sie wird die fragliche Nachricht weitergeleitet.

Weitere Nachrichten erlauben die Anzeige von Frames bzw. Instanzen:

:inspect-frames	Methode
beschreibt ausgewählte Frames.	
:inspect-instances	Methode
beschreibt ausgewählte Instanzen.	

Kapitel 7

Konstruktion des Frame-Prozessors

7.1 Übersicht

Die Konstruktion des Frame-Prozessors verfolgt das Ziel, eine ausgewogene Modularisierung durch Abtrennung der Basisleistungen von verschiedenen Zusatzleistungen zu erreichen. Dies eröffnet die Möglichkeit, Frame-Prozessor-Varianten mit unterschiedlichem Leistungsspektrum zu definieren. Drei Varianten stehen zur Verfügung. Der **Basic-Frame-Processor**, die einfachste Variante, stellt nur die Basisleistungen zur Verfügung, der **Mini-Frame-Processor** erlaubt zusätzlich die Verwendung der Standard-Properties **:possible-values** und **:explain-answers**, der **Normal-Frame-Processor** bietet den vollen Leistungsumfang.

7.2 Zerlegung

Der **Normal-Frame-Processor** ist eine Spezialisierung des **Mini-Frame-Processor**, dieser wiederum eine Spezialisierung des **Basic-Frame-Processor**. Der Zerlegung des **Normal-Frame-Processors** ist daher diejenige der beiden anderen Varianten zu entnehmen (siehe Abbildung 7.1):

1. **processor-core**

Die Basiskomponente aller Prozessoren, hier des Frame-Prozessors, die die Schnittstelle zum Meta-Prozessor realisiert und die Instanzenvariablen definiert, die benötigt werden, damit sich der Prozessor ins Gesamtsystem integrieren läßt.

2. **frame-base**

Verwaltet Frames, Behaviors und Instanzen.

3. **frame-interpreter**

Realisiert die Zugriffe auf Slots und Properties der Instanzen sowie die Nachrichten zur Fragegenerierung.

4. **basic-frame-processor**

Faßt **frame-interpreter** und **frame-base** zusammen.

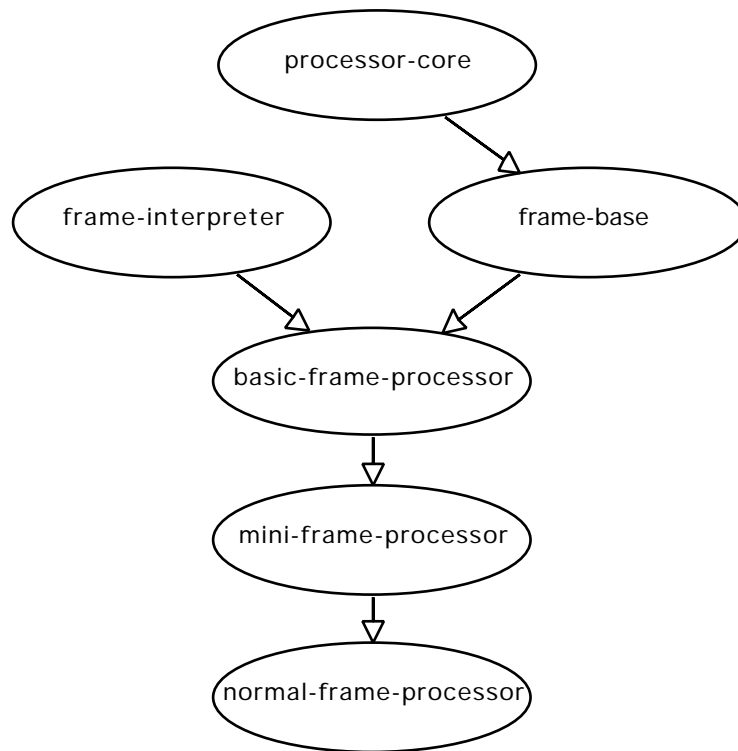


Abbildung 7.1: Architektur des Frame-Prozessors

5. **mini-frame-processor**

Spezialisierung von **basic-frame-processor**. Veranlaßt, daß Frames angelegt werden, für die possible values spezifiziert werden können.

6. **normal-frame-processor**

Spezialisierung von **mini-frame-processor**. Veranlaßt, daß Frames angelegt werden, die active values unterstützen.

Die vom Frame-Prozessor verwalteten Frames werden vom Benutzer definiert. Sie werden als Flavor realisiert, zu denen standardmäßig folgende Komponenten hinzugemixt werden (siehe Abbildung 7.2):

1. **frame-core**

Basis-Flavor für alle Frames. Realisiert die Basis-Zugriffsmethoden auf Slots und ihre Properties.

2. **poss-val-mixin**

Realisiert Methoden zur Wertebeschreibung und Überprüfung.

3. **poss-val-frame-core**

Faßt **poss-val-mixin** und **frame-core** zusammen.

4. **active-value-mixin**

Realisiert Methoden zur Aktivierung von active values.

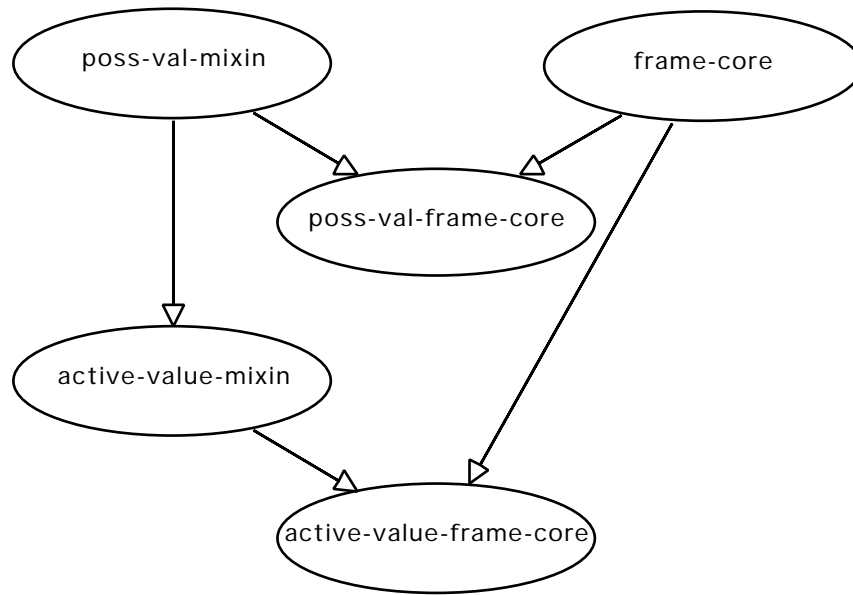


Abbildung 7.2: Basiskomponenten der Frames

5. active-value-frame-core

Faßt **active-value-mixin** und **frame-core** zusammen.

7.3 Integration ins Gesamtsystem

Zur Integration des Frame-Prozessors wird prozessorseitig auf das Standard-Flavor **processor-core** zurückgegriffen. Eine Beschreibung der Rolle dieses Flavors für die Einbindung ins Gesamtsystem findet sich in Kapitel 3.

Als Komponente des Prozessors einer Wissensbasis macht das Flavor **basic-frame-mixin** die Leistungen des **Basic-Frame-Processors** im Gesamtsystem verfügbar. Entsprechendes gilt für das Flavor **mini-frame-mixin** bzw. **normal-frame-mixin** und den **Mini-Frame-Processor** bzw. **Normal-Frame-Processor**.

7.4 Der Basic-Frame-Processor

Die folgenden Abschnitte beschreiben, wie die für einen Frame-Prozessor unabdingbaren Leistungen implementiert sind.

7.4.1 frame-base

Das Flavor **frame-base** verwaltet Frames, Behaviors und Instanzen und stellt Methoden zu ihrer Generierung und Verwaltung bereit.

Frames werden als Flavors realisiert, Behaviors als Methoden. Aufrufe der Konstruktoren **defframe** bzw. **defbehavior** werden dementsprechend umgesetzt in **defflavor-** bzw. **defmethod-**Aufrufe. Diese Umsetzung wird von den Methoden **:new-frame-form** bzw. **:new-behavior-form** bewerkstelligt, die zusätzlich Syntaxchecks vornehmen. Instanzen

eines Frames werden als Instanzen des zugehörigen Flavors realisiert. Die Methode **:new-instance** legt mittels **make-instance** neue Instanzen an, wobei zuvor wieder ein Syntax-check vorgenommen wird. **:new-instance** wird vom Konstruktor **definstance** aufgerufen.

Flavors, die Frames verkörpern, werden in einem eigenen Package abgelegt. Das Package wird von der Methode **:set-up-prefix** generiert. Diese nimmt als Package-Name den Wert der Instanzvariablen **pkg** von **basic-frame-mixin** bzw. den Namen der jeweiligen Wissensbasis, falls **pkg** bei Initialisierung der Wissensbasis kein Wert zugewiesen wurde. Durch die Verwendung von Packages wird erreicht, daß Frames aus verschiedenen Wissensbasen denselben Namen tragen dürfen. Gleiches gilt für die Instanzen der Frames. In der Wissensbasis werden Instanzen üblicherweise mit ihrem externen Namen, unter dem sie in der Wissensbasis definiert wurden, angesprochen. Die Makros **get-instance** bzw. **get-instance-with-check** liefern zu einem Instanznamen die zugehörige Instanz, wobei im zweiten Fall überprüft wird, ob ein Instanzname vorliegt.

Mit dem internen Flavor-Namen werden eine Reihe zusätzlicher Informationen via Property-Liste verknüpft: die Definition des Frames, die Definition der zugehörigen Behaviors, die Supers des Frames, seine Instanzen. Entsprechend wird bei Instanzen mit dem internen Namen die Definition der Instanz verknüpft. Auf diese Definition wird von **reset-instances** wie von **print-instances** zurückgegriffen. Diese Funktionen werden verwendet, wenn eine Wissensbasis die Nachricht **:reset-kb** erhält.

7.4.2 frames

Wie bereits ausgeführt werden Frames als Flavor realisiert. Die zugehörigen Flavor enthalten als Komponente sämtlich das Flavor **frame-core** bzw. dessen Spezialisierungen. Welches Flavor benutzt wird, wird in der Instanzvariablen **frame-type** des Flavors **frame-base** festgehalten. Die allen Frames gemeinsamen Basisleistungen sind als Methoden von **frame-core** realisiert.

Frame-Slots werden zu Instanzvariablen. Sie haben Listen als Werte. Das erste Element der Liste stellt den Slotwert dar, der Rest der Liste ist als Property-Liste organisiert. Außerdem enthält jeder Frame die Instanzvariablen **slots** und **object-name**.

Das Flavor **frame-core** stellt die folgenden Basis-Zugriffsmethoden für Frames bereit:

Die Methode **:get-value-only** bzw. **:set-value-only** liefert bzw. setzt den Wert (einer Property) eines Slots. **:replace** ist synonym zu **:set-value-only**. Die Methoden **:get** bzw. **:set** und **:put** haben für **frame-core** die gleiche Bedeutung wie **:get-value-only** bzw. **:set-value-only**. Sie sind dazu bestimmt, nach Bedarf spezialisiert zu werden. Es wird davon ausgegangen, daß dabei **:get-value-only** bzw. **:set-value-only** benutzt werden, die darum nicht spezialisiert werden sollten.

Für einige Aufrufe der Basis-Zugriffsmethoden gibt es abkürzende Schreibweisen. Sie erlauben es insbesondere, Frame-Instanzen über ihren externen Namen anzusprechen: Anstelle der Methoden **:get** bzw. **:put** können die Makros **get-value** bzw. **put-value** verwendet werden. Sie erwarten als erstes Argument den (externen) Namen einer Instanz, an die die jeweilige Nachricht zu schicken ist. Statt **put-value** kann auch das **setf**-Makro zusammen mit **get-value** benutzt werden. Im Körper von Behaviors kann anstelle von (send-message self :get ...) (\$value ...) geschrieben werden und für (send-message self :put ...) (setf (\$value ...) ...).

Als Basis-Methode, um den Wert (einer Property) eines Slots vom Benutzer zu erfragen, dient **:ask**. Sie ruft **:ask-for-slot-value** bzw. **:ask-for-slot-property** auf. Diese

Unterscheidung wird gemacht, weil nur beim Wert des Slots die Standard-Properties zu berücksichtigen sind. Enthält ein Slot eine **:ask**-Property, wird der dort gespeicherte Text benutzt, um den Benutzer aufzufordern, einen Wert einzugeben, sonst wird ein Standard-Text verwendet. Gibt der Benutzer die an die globale Variable ***c-help-key*** gebundene Taste ein, wird ihm mitgeteilt, welcher Wert erwartet wird, sofern ein entsprechender Wert beim Aufruf der Methode als Parameter übergeben wurde. Gibt der Benutzer **help** oder die an ***help-key*** gebundene Taste ein, wird **HELP** als Wert der Methode weitergereicht, aber nicht als Slot-Wert gespeichert. Dieser Mechanismus dient dazu, Kontext-Informationen anzufordern. Kontext-Informationen geben darüber Auskunft, weshalb die Frage gestellt wurde. Sie stammen von den anderen Prozessoren.

Wird im Makro „<-“ anstelle eines Behavior-Namens ein Slot-Name verwendet, wird hieraus ein Aufruf der Methode **:slot-message** generiert. Diese Methode prüft, ob Relation und Modus verträglich sind, und ruft eine Methode auf, die den jeweiligen Modus realisiert. Jede zulässige Relation, die in einem derartigen Ausdruck auftritt, ist als Methode von **basic-frame** realisiert. Um den Wert (einer Property) des Slots mit dem Wert des übergebenen Ausdrucks zu vergleichen, schickt die Instanz die Relation als Nachricht an sich selbst mit dem Wert und Ausdruck als Parameter.

7.4.3 frame-interpreter

Die im letzten Abschnitt aufgeführten Methoden sind an Frame-Instanzen zu richten. Das Flavor **frame-interpreter** enthält entsprechende Methoden, die als zusätzlichen Parameter einen Instanz-Namen erwarten. An diese wird die zugehörige Methode geschickt.

7.5 Mini-Frame-Processors

Der Mini-Frame-Processor generiert Frames, die anstelle von **frame-core** das Flavor **poss-val-frame-core** enthalten.

7.5.1 poss-val-mixin

Das Flavor **poss-val-mixin** realisiert Methoden zur Wertebeschreibung und Überprüfung.

Jeder Wertetyp wird durch ein Key-Word mit oder ohne Argumenten beschrieben. **:string** bzw. **(:instance-of <Frame-Name>)** sind Beispiele hierfür. Zu jedem dieser Key-Words gibt es in **poss-val-mixin** eine gleichlautende Methode, die die entsprechende Werteüberprüfung realisiert. Im Falle von **:instance-of** wird also geprüft, ob eine Instanz von **<Frame-Name>** vorliegt. Die speziellen Werte „-“ (unbestimmt) und **unknown** werden dabei stets als zulässig behandelt. Ist ein Wert zulässig, liefert die Methode eine Liste mit dem Wert als einzigem Element, andernfalls NIL. (Diese Konstruktion erlaubt eine Modifikation des Wertes im Verlauf des Tests. Hiervon wird bei den Standard-Typen allerdings kein Gebrauch gemacht.).

Mit dem Makro **define-possible-values-behavior** kann der Benutzer zusätzliche Wertetypen einführen. Im Rumpf der Methode, die der Benutzer anzugeben hat, müssen die Sonderfälle „-“ und **unknown** nicht mehr berücksichtigt werden. Der entsprechende Test wird vom Makro automatisch hinzugefügt.

Zur Überprüfung, ob ein Wert für ein Slot dem spezifizierten Wertetyp entspricht, dienen die Methoden **:check-value** bzw. **:check-correct-value** von **poss-val-mixin**. Die

zweite Methode erlaubt eine interaktive Korrektur des Wertes durch den Benutzer.

Die Slotwerte der Instanzen von Frames mit **poss-val-mixin** als Komponente werden bei Generierung auf Zulässigkeit überprüft. In der Frame-Definition gesetzte Default-Werte, werden ebenfalls erst bei Generierung von Instanzen getestet. Diese Überprüfung findet nur statt, falls die Instanzvariable **check** auf T steht.

poss-val-mixin enthält Spezialisierungen folgender Basis-Methoden:

:put überprüft den Wert, der gesetzt werden soll, wobei die Methode **:check-correct-value** benutzt wird.

:ask überprüft den Wert, der vom Benutzer eingegeben wurde, wobei die Methode **:check-value** benutzt wird. Außerdem wird der Benutzer bei der Eingabe von Werten unterstützt. Dies geschieht automatisch, falls ein falscher Wert eingegeben wurde, oder auf Wunsch des Benutzers bei Eingabe der an ***c-help-key*** gebundenen Taste. Die Form der Unterstützung ist datengetrieben organisiert. Es wird eine Methode benutzt, die unter dem Indikator **:supp-method** auf der Property-List des Possible-Value-Key-Words abzulegen ist. Im Falle von **:one-of** könnte diese Methode ein Menü präsentieren, aus dem der Benutzer den gewünschten Wert auszuwählen hat. Es wird erwartet, daß die Methode einen korrekten Wert liefert. Wurde keine Methode spezifiziert, wird die Default-Methode **:default-read-method** verwendet. Diese informiert den Benutzer darüber, welche Restriktionen bestehen und welcher Wert ggf. erwartet wird, bevor ein Wert angefordert wird.

7.6 Normal-Frame-Processors

Der Normal-Frame-Processor generiert Frames, die anstelle von **frame-core** das Flavor **active-value-frame-core** enthalten.

7.6.1 active-value-mixin

Das Flavor **active-value-mixin** sorgt dafür, daß bei lesendem oder schreibendem Zugriff auf einen aktiven Wert die entsprechenden GET- bzw. PUT-Behavior aktiviert werden. Dazu stehen die Methode **:active-value-get** bzw. **:active-value-set** bereit. Da aktive Werte genestet werden können, sind die beiden Methoden rekursiv implementiert.

:active-value-get evaluiert einen einfachen oder aktiven Wert. Ein einfacher Wert wird unverändert übergeben, bei einem aktiven Wert wird das entsprechende GET-Behavior aufgerufen, wobei als erstes Argument der mittels **:active-value-get** ausgewertete lokale Wert übergeben wird. Bei genesteten aktiven Werten wird beim lesenden Zugriff somit der **innerste** aktive Wert **zuerst** ausgeführt.

:active-value-set baut aus einem neuen lokalen Wert und einem alten Wert einen neuen Wert auf. Ist der alte Wert ein einfacher Wert, wird der neue lokale Wert zum neuen Wert. Ist es ein aktiver Wert, wird dieser mit modifiziertem lokalen Wert als neuer Wert übergeben. Um den modifizierten lokalen Wert zu ermitteln, wird das PUT-Behavior auf den neuen lokalen Wert und den alten Wert angewendet. Aus dem Resultat und dem lokalen Wert des alten, aktiven Werts wird mittels **:active-value-set** der modifizierte lokale Wert bestimmt. Bei genesteten aktiven Werten wird beim schreibenden Zugriff somit der **äußerste** aktive Wert **zuerst** ausgeführt.

active-value-mixin enthält Spezialisierungen der Methoden: **:get**, **:set** und **:put**, die mittels **:active-value-set** bzw. **:active-value-set** aktive Werte berücksichtigen.

7.7 Importierte Leistungen

Außer den Standard-IO-Leistungen der Benutzerschnittstelle werden keine weiteren importiert.

7.8 Exportierte Leistungen

Die Flavors **basic-frame-mixin**, **mini-frame-mixin** und **normal-frame-mixin** definieren die Leistungen der jeweiligen Varianten des Frame-Prozessors, die im Gesamtsystem verfügbar sein sollen. Der Aufbau folgt den in Kapitel 3 beschriebenen Prinzipien zur Einbindung eines Prozessors in das Gesamtsystem.

Standard-Frame-Referenzen und Standard-Behavior-Referenzen dürfen in Prämissen von Regeln bzw. Prolog-Klauseln auftreten. Da sie vom Regel- bzw. Prolog-Prozessor nicht ausgewertet werden können, werden sie dem Meta-Prozessor übergeben. Aufgrund des Typerkennungsmakros des Frame-Prozessors werden sie als Ausdrücke des Frame-formalismus erkannt und vom Frame-Prozessor ausgewertet, wobei folgende Arbeitsmodi unterstützt werden:

:recall Ermittelt die Gültigkeit einer Referenz ohne Befragen des Benutzers. Mögliche Ergebnisse sind UNBESTIMMT, NIL oder ein Wert.

:ask Ermittelt die Gültigkeit einer Referenz u.U. durch Befragen des Benutzers, falls sie noch unbestimmt ist. Mögliche Werte sind WHY, falls sich der Benutzer den Zweck der Frage nach dem Wert dieses Ausdrucks erklären lassen will, NIL oder ein Wert.

:recall-immediate Identisch mit **:recall**.

:remember Hält den Wert der Referenz fest. Mögliche Werte sind NIL, falls der Wert dem alten Wert entspricht, sonst der neue Wert.

:prolog Ermittelt die Gültigkeit einer Referenz u.U. durch Befragen des Benutzers, falls sie noch unbestimmt ist. Mögliche Werte dieser Nachricht sind NIL oder T.

Als Prämisse in Prolog-Goals können außerdem Prolog-Frame-Referenzen und Prolog-Behavio-Referenzen auftreten. Ihre Auswertung wird ebenfalls vom Frame-Prozessor geleistet. Als Arbeitsmodus ist nur **:prolog** möglich. Wieder wird die Gültigkeit der Referenz u.U. durch Befragen des Benutzers ermittelt. Mögliche Werte sind NIL, T oder eine Liste von Prolog-Klauseln.

Kapitel 8

Spezifikation des Prolog-Prozessors

8.1 Anforderungen

Logik ist seit Beginn der KI als Wissensrepräsentationsformalismus verwendet worden. Aber erst mit der Entwicklung von Prolog wurde eine Programmiersprache geschaffen, in der eine für die Praxis genügend große Teilmenge der Prädikatenlogik erster Stufe verwendet werden kann. Prolog basiert einerseits auf einer speziellen Notation dieser Teilmenge, den Horn-Klauseln, und andererseits auf dem Resolutionsverfahren von Robinson und der Unifikation.

Die in **BABYLON** integrierte Prolog-Version bedient sich beim Beweis einer Hypothese der Standard-Kontrollstrategie des Backtrackings. Diese kann in üblicher Weise durch das System-Prädikat **cut** modifiziert werden [CloMe].

Die Besonderheit von **BABYLON**-Prolog liegt darin, daß als Prämissen von Regeln Konstrukte anderer Formalismen, insbesondere Referenzen auf Objekte, vorkommen können. Ferner werden Lisp-Schnittstellen angeboten. Dies erlaubt es, den Umfang an System-Prädikaten im Vergleich zu anderen Prolog-Implementierungen gering zu halten.

Die Klauseln werden in **BABYLON**-Prolog zu Klauselmengen (axiom sets) zusammengefaßt. Klauselmengen können einer Wissensbasis fest zugeordnet oder eigenständig sein. Zu einem Beweis können mehrere Klauselmengen herangezogen werden. Sie werden als die aktuellen Klauselmengen bezeichnet.

BABYLON-Prolog verwendet anstelle der Standard-Syntax von Prolog, wie sie in [CloMe] spezifiziert ist, eine an Lisp orientierte Syntax.

8.2 Einordnung

Der Prolog-Prozessor erlaubt die Repräsentation von Wissen in Form von Horn-Klauseln und das Beweisen von Hypothesen aufgrund der Klauseln. Der Prolog-Prozessor wird in drei Varianten mit unterschiedlichem Leistungsumfang angeboten. Der **Basic-Prolog-Processor**, die einfachste Variante, erlaubt die Verwaltung von Klauselmengen und den Beweis von Hypothesen, der **Mini-Prolog-Processor** stellt zusätzlich Trace-Möglichkeiten zur Verfügung, beim **Normal-Prolog-Processor** kommen Aspekte der Erklärung und Unterstützung bei der Programmentwicklung hinzu.

8.3 Funktionsbeschreibung

8.3.1 Klauseln und Klauselmengen

Klauseln stellen logisch gesehen Implikationen dar mit einer atomaren Formel als Konklusion und mehreren (evtl. keinen) atomaren Formeln als Prämissen.

Klauseln werden extern als Lisp-Ausdrücke dargestellt:

$\langle \text{Klausel} \rangle ::= \langle \text{Fakt} \rangle \mid \langle \text{Regel} \rangle$

$\langle \text{Fakt} \rangle ::= (\langle \text{prädikativer Ausdruck} \rangle)$

$\langle \text{Regel} \rangle ::= (\langle \text{Kopf} \rangle \{ \langle - \rangle \} . \langle \text{Rumpf} \rangle)$

$\langle \text{Kopf} \rangle ::= \langle \text{prädikativer Ausdruck} \rangle$

$\langle \text{Rumpf} \rangle ::= \langle \text{Liste von Prämissen} \rangle$

Fakten können als Regeln ohne Prämissen angesehen werden.

$\langle \text{Prämisse} \rangle ::= \langle \text{prädikativer Ausdruck} \rangle \mid$
 $\langle \text{Variable} \rangle \mid$
 $\langle \text{externe Prämisse} \rangle$

$\langle \text{prädikativer Ausdruck} \rangle ::= (\langle \text{Prädikats-Term} \rangle . \langle \text{Termliste} \rangle)$

$\langle \text{Prädikats-Term} \rangle ::= \langle \text{Name eines Benutzer-Prädikats} \rangle \mid$
 $\langle \text{Name eines System-Prädikats} \rangle \mid$
 $\langle \text{Variable für ein Prädikat} \rangle$

$\langle \text{Term} \rangle ::= \langle \text{Zahl} \rangle \mid \langle \text{Name} \rangle \mid \langle \text{Variable} \rangle \mid$
 $\langle \text{String} \rangle \mid \langle \text{Struktur} \rangle$

$\langle \text{Struktur} \rangle ::= (\langle \text{Term} \rangle . \langle \text{Term} \rangle)$

$\langle \text{Name} \rangle ::= \langle \text{Zeichenkette, die nicht mit Unterstrich beginnt} \rangle$

$\langle \text{Variable} \rangle ::= \langle _ \text{Zeichenkette} \rangle \mid _$

Der Unterstrich `_` allein bezeichnet eine sog. **anonyme Variable**, deren Name nicht weiter interessiert. Jedes Vorkommen von Unterstrich in einer Klausel wird als neue Variable interpretiert.

Externe Prämissen sind Prämissen in einer dem Prolog-Prozessor unbekannten Syntax. Externe Prämissen müssen Listen sein, deren erstes Element weder ein Benutzer- noch ein System-Prädikat ist. In den Ausdrücke dürfen an beliebiger Stelle Variablen auftreten. Vor Evaluierung der Prämissen müssen diese zumeist instantiiert sein. Die Einzelheiten sind in Abschn. 8.3.3 beschrieben.

Beispiele für externe Prämissen sind die **Frame-Referenzen**. Neben den im Kapitel 6 beschriebenen **Standard-Frame-** bzw. **Standard-Behavior-Referenzen** gibt es Prolog-spezifische Varianten:

$\langle \text{Prolog-Frame-Referenz} \rangle ::= (\langle \text{Slot-Term} \rangle \langle \text{Instanz-Term} \rangle \langle \text{Term} \rangle) \mid$
 $\quad (\langle \text{Property-Term} \rangle \langle \text{Slot-Term} \rangle \langle \text{Instanz-Term} \rangle \langle \text{Term} \rangle)$

$\langle \text{Prolog-Behavior-Referenz} \rangle ::= (\langle \text{Behavior-Name} \rangle \langle \text{Instanz-Term} \rangle$
 $\quad \langle \text{Arg1} \rangle \dots \langle \text{ArgN} \rangle \langle \text{Term} \rangle)$

$\langle \text{Slot-Term} \rangle ::= \langle \text{Name eines Slots} \rangle \mid \langle \text{Variable für ein Slot} \rangle$

$\langle \text{Property-Term} \rangle ::= \langle \text{Name einer Property} \rangle \mid \langle \text{Variable für eine Property} \rangle$

$\langle \text{Instanz-Term} \rangle ::= \langle \text{Name einer Instanz} \rangle \mid \langle \text{Variable für eine Instanz} \rangle$

Die Prolog-spezifischen Formen unterscheiden sich von den Standard-Formen in der Art, wie Prolog-Variablen behandelt werden. (s. Abschn. 8.3.3).

Anfragen an Prolog werden als Hypothesen aufgefaßt, die aufgrund der aktuellen Klauseln abzuleiten sind. Als Hypothesen zulässig sind Konjunktionen atomarer Formeln. Syntaktisch entsprechen sie den Rümpfen von Regeln. Falls eine Anfrage nur aus einer atomaren Formel besteht, können die äußersten Klammern entfallen.

Klauseln werden zu **Klauselmengen** (Axiom Sets) zusammengefaßt. Die Klauseln einer Klauselmenge mit gleichem Prädikatnamen definieren ein **Benutzer-Prädikat**. Benutzer-Prädikate ohne Klauseln sind ausgeschlossen. Die Klauseln eines Prädikats gelten als linear geordnet.

Zum Beweis einer Anfrage können mehrere Klauselmengen hinzugezogen werden. Sie werden als die **aktuellen Klauselmengen** bezeichnet. Wird das gleiche Benutzer-Prädikat in mehreren aktuellen Klauselmengen definiert, wird nur die erste Definition verwendet. Die anderen Definitionen werden somit überschrieben.

System-Prädikate werden nicht durch Klauseln definiert und lassen sich **nicht** überschreiben.

Klauselmengen können Bestandteil einer Wissensbasis oder eigenständig sein. Eigenständige Klauselmengen werden durch folgendes Makro definiert:

```
(DEFAXIOM-SET <Name>
               <Klausel1>
               ...
               <Klausel N>)
```

Eine Wissensbasis kann nur eine Klauselmenge enthalten. Sie wird definiert durch:

```
(DEFRELATIONS <Klausel 1>
               ...
               <Klausel N>)
```

8.3.2 System-Prädikate (Build-In Predicates)

Die folgenden System-Prädikate sind in **BABYLON**-Prolog implementiert:

Kontrollprädikate

(CUT) Standard cut-Prädikat.

(**TRUE**) Stets wahr, keine alternative Antwort.

(**FALSE**) Stets falsch.

(**CALL** <**Variable**>)

Wenn der Prolog-Prozessor die Prämisse zu beweisen sucht, muß die Variable instantiiert sein, ihr Wert eine Prämisse, die nicht wieder eine Variable darstellt. Diese wird vom Prozessor bewiesen. Ergibt sich keine gültige Prämisse, erfolgt eine Fehlermeldung und Abbruch des Beweisprozesses. Anstelle von (CALL <Variable>) genügt es, <Variable> als Prämisse anzugeben.

(**CALLPRED** <**variabler prädikativer Ausdruck**>)

Wenn der Prolog-Prozessor die Prämisse zu beweisen sucht, muß die Variable instantiiert sein, ihr Wert ein nicht variabler Prädikatsterm. Nach Substitution wird die Prämisse bewiesen. Ergibt sich kein gültiger Prädikatsterm, erfolgt eine Fehlermeldung und Abbruch des Beweisprozesses. Das System-Prädikat CALLPRED darf weggelassen werden, d.h. der Aufruf ist gleichwertig mit <variabler prädikativer Ausdruck>.

(**AND** . <**Termliste**>)

Die Terme der Liste werden als Prämissen behandelt, die zugleich zu erfüllen sind.

(**OR** . <**Termliste**>)

Die Terme der Liste werden als Prämissen behandelt, die alternativ zu erfüllen sind. Für alternative Antworten wird nach Alternativen unter den Prämissen gesucht und zwar in der Reihenfolge, in der sie aufgeführt sind.

(**NOT** . <**Termliste**>)

Die Terme der Liste werden als Prämissen behandelt. Sind alle erfüllt, gilt der Aufruf als gescheitert, sonst als erfolgreich.

(**ONCE** . <**Termliste**>)

Wie **AND** bei erstmaligem Aufruf. Scheitert beim Backtracking.

(**REPEAT**) Stets erfolgreich, selbst beim Backtracking.

(**BAGOF** <**Variable**> <**Term**> <**Bag**>)

<Term> wird als Anfrage behandelt, für die alle Antworten bestimmt werden. Die sich jeweils ergebenden Werte für <Variable> (sie sollte in <Term> vorkommen) werden zu einer Liste zusammengefaßt, die mit <Bag> unifiziert wird. Die Anfrage liefert nur eine Lösung.

Lisp-Schnittstellen

(**LISP** . <**Lisp-Formen**>)

Die Prolog-Variablen in den Formen werden durch ihre aktuellen Werte ersetzt und der entstehende Ausdruck im Lisp-Sinne evaluiert. (Sollen Variablen nach der Substitution **nicht** evaluiert werden, muß der Benutzer dies durch **Quoten** unterbinden.) Die Prämisse gilt genau dann als bewiesen, wenn ihr Wert nicht NIL ist. Lassen sich die Formen nicht evaluieren, wird ein Fehler signalisiert und der Beweisprozeß abgebrochen. Eine alternative Antwort ist nicht möglich.

(IS <Variable> <Lisp-Form>)

Die Prolog-Variablen in der Lisp-Form werden durch ihre aktuellen Werte ersetzt und die entstehende Form im Lisp-Sinne evaluiert. Die Prämisse gilt genau dann als bewiesen, wenn sich die Variable mit dem Wert der Form unifizieren läßt. Kann die Form nicht evaluiert werden, wird ein Fehler signalisiert und der Beweisprozeß abgebrochen. Eine Alternative ist nicht möglich.

Klauselmanagement**(ASSERTA <Term>) | (ASSERTZ <Term>)**

Alle Variablen in <Term> werden durch ihre aktuellen Werte ersetzt. Ergibt sich keine Klausel, wird ein Fehler signalisiert. Ist das Prädikat der Klausel bereits definiert, wird die Klausel an erster (**ASSERTA**) bzw. letzter (**ASSERTZ**) Stelle hinzugefügt, sofern sie neu ist. Gibt es die Klausel bereits, wird nichts verändert; der Aufruf gilt auch dann als erfolgreich. Bei einem neuen Prädikat wird die Klausel zur ersten Klauselmenge hinzugefügt. Beim Backtracking wird die eingefügte Klausel nicht wieder entfernt, der Aufruf gilt als nicht erfolgreich.

(CLAUSE <Term>)

Die Variablen in <Term> werden durch ihre aktuellen Werte ersetzt. Ergibt sich keine Klausel, wird ein Fehler signalisiert. Der Aufruf gilt als erfolgreich, sobald der Term sich mit einer definierenden Klausel des Prädikats unifizieren läßt. Beim Backtracking wird eine Unifikation mit den restlichen definierenden Klauseln versucht (überschriebene Klauseln werden dabei nicht einbezogen).

(RETRACT <Term>)

Wie **CLAUSE**, nur wird zusätzlich die unifizierte Klausel aus der Klauselbasis entfernt. Backtracking ist nur solange erfolgreich bis alle definierenden Klauseln entfernt sind, d.h. Klauseln, die überschrieben wurden bleiben erhalten.

(ABOLISH <Term>)

<Term> muß bei Aufruf ein Prädikatname sein. Die zugehörigen Klauseln werden aus der Klauselbasis entfernt. (evtl. überschriebene Klauseln bleiben erhalten). Der Aufruf gilt als erfolgreich, selbst wenn das Prädikat nicht definiert wurde.

Ein-/Ausgabe

Die folgenden Prädikate setzen voraus, daß von der Benutzerschnittstelle ein Fenster als Dialog-Window ausgezeichnet wird.

(READ <Variable>)

Vom Dialog-Window wird ein Term gelesen und mit <Variable> unifiziert. Schlägt fehl beim Backtracking.

(WRITE <Term>)

<Term> wird auf das Dialog-Window geschrieben. Mißerfolg beim Backtracking.

(FORMAT <Format-String> . <Termliste>)

Die Terme der <Termliste> werden den Angaben im <Format-String> entsprechend aufs Dialog-Window geschrieben. Die Terme werden im Lisp-Sinne evaluiert. Dies muß ggf. durch Quoten vom Benutzer unterbunden werden. Mißerfolg beim Backtracking.

Sonstige Prädikate**(= <Term1> <Term2>)**

erfolgreich genau dann, wenn sich <Term1> und <Term2> unifizieren lassen.

(\= <Term1> <Term2>)

erfolgreich genau dann, wenn sich <Term1> und <Term2> nicht unifizieren lassen.

(== <Term1> <Term2>)

erfolgreich genau dann, wenn nach Substitution der Variablen <Term1> und <Term2> gleich sind.

(=\= <Term1> <Term2>)

erfolgreich genau dann, wenn nach Substitution der Variablen <Term1> und <Term2> ungleich sind.

(<Op> <Term1> <Term2>)

mit <Op> ::= =. = | =\= | < | > | >= | =<

Die Variablen in <Term1> und <Term2> werden durch ihre aktuellen Werte ersetzt, diese im Lisp-Sinne evaluiert und verglichen. Die Bedeutung der Symbole ist (in der Reihenfolge ihrer Aufschreibung): gleich, ungleich, kleiner, größer, größer-gleich, kleiner-gleich. Der Aufruf ist genau dann erfolgreich, falls sich nicht NIL ergibt.

(VAR | ATOM | INTEGER | ATOMIC <Term>)

Diese Prädikate sind erfolgreich genau dann, wenn <Term> zur Zeit des Aufrufs eine Variable (**VAR**), einen Namen (**ATOM**) oder eine ganze Zahl (**INTEGER**) verkörpert. **ATOMIC** ist wahr, wenn <Term> ein Name oder eine ganze Zahl ist. Mißerfolg beim Backtracking.

8.3.3 Externe Prämissen (Frame-Referenzen)

Prämissen, für die der Prolog-Prozessor keine Klauseln findet, werden nach Substitution der Prolog-Variablen dem Prozessor für die Wissensbasis übergeben. Als Antwort wird T, NIL oder eine Liste von Klauseln erwartet, die vom Prolog-Prozessor wie die definierenden Klauseln eines Benutzer-Prädikats benutzt werden.

Frame-Referenzen werden in dieser Weise behandelt:

Externe Prämissen werden von der Wissensbasis als **Standard-Frame-** bzw. **Standard-Behavior-Referenzen** gewertet, falls das erste Element des Ausdrucks der Name einer Instanz eines Frames ist. Bei Aufruf müssen sämtliche Variablen im Ausdruck instantiiert sein. Die Frame-Referenz wird vom Frame-Prozessor evaluiert. Kann die Referenz nicht evaluiert werden, weil ein Slot-Wert unbestimmt ist, wird der Benutzer

nach dem Wert befragt. Kann sie aus anderen Gründen nicht evaluiert werden, wird abgebrochen. Sonst wird als Antwort T oder NIL übergeben, die Prämisse dementsprechend als bewiesen oder unbeweisbar angesehen. Weitere Antworten gibt es nicht.

Ist das zweite oder dritte Element des Ausdrucks der Name einer Instanz eines Frames und das erste bzw. zweite Element ein Slotname, wird der Ausdruck als **Prolog-Frame-Referenz** behandelt und in eine **Standard-Frame-Referenz** der Form

$$(<\text{Instanz}> <\text{Slot}> \{<\text{Property}>\} = <\text{Term}>)$$

transformiert. Ist $<\text{Term}>$ ein Grundterm wird wie sonst bei **Standard-Frame-Referenzen** verfahren. Ist $<\text{Term}>$ eine Variable werden die Slot-Werte ermittelt und für jeden Wert vom System eine Klausel der Form

$$(\{<\text{Property}>\} <\text{Slot}> <\text{Instanz}> <\text{Wert}>)$$

generiert. Ist der Wert noch unbestimmt, wird der Benutzer gefragt. Die generierten Klauseln werden dem Prolog-Prozessor übergeben, der diese wie die definierenden Klauseln eines Prädikats $<\text{Slot}>$ behandelt. In diesem Fall sind im Unterschied zu Standard-Frame-Referenzen weitere Antworten möglich.

Ist das zweite Element des Ausdrucks der Name einer Instanz eines Frames und das erste Element ein Behavior-Name, wird der Ausdruck als **Prolog-Behavior-Referenz** gewertet. An die Instanz wird eine entsprechende Nachricht geschickt, das Ergebnis mit dem letzten Term des Ausdrucks verglichen, falls es sich um einen Grundterm handelt. Bei Gleichheit wird T sonst NIL zurückgegeben. Ist $<\text{Term}>$ eine Variable, wird eine Klausel der Form

$$(<\text{Behavior-Name}> <\text{Instanz}> <\text{Arg1}> \dots <\text{ArgN}> <\text{Wert}>)$$

gebildet und dem Prolog-Prozessor übergeben.

Damit Frame-Referenzen als solche erkannt werden, muß der Prozessor für die Wissensbasis eine Frame-Komponente enthalten. Andernfalls werden Frame-Referenzen wie Free-Text behandelt. Gleiches kann temporär durch Wechsel des Arbeitsmodus (s. Kapitel 3.6) erreicht werden.

8.3.4 Beweisprozeß

Der Prolog-Prozessor erlaubt jeweils nur eine Anfrage. Solange keine neue gestellt wurde, kann man auf die letzte Anfrage zurückkommen und **Antwortalternativen** anfordern.

Zum (alternativen) Beweis einer Hypothese wird die Standard-Kontrollstrategie von Prolog verwendet. Diese kann in üblicher Weise durch das System-Prädikat **cut** modifiziert werden [CloMe]. Der Unifikationsalgorithmus kennt keinen **Occur-Check**.

Die Ergebnisse eines Beweises können auf unterschiedliche Weise angezeigt werden:

- Die Anfrage wird nach Substitution aller Prolog-Variablen durch ihre Werte wiederholt.
- Alle nicht anonymen Variablen werden mit ihren Werten angezeigt.
- Alle nicht anonymen Variablen werden mit ihrem Wert angezeigt, sofern dieser keine Variable ist.

- Keine Anzeige; nur sinnvoll, falls die Anzeige vom Prolog-Programm vorgenommen wird.

Statt einer Anzeige der Ergebnisse kann eine Übergabe als Wert im Lisp-Sinn erfolgen.

8.3.5 Trace

Trace-Möglichkeiten werden erst vom **Mini-Prolog-Processor** eröffnet. Ein Trace ist global für alle Prädikate oder selektiv für einzelne Prädikate möglich, wobei zwei Modi zur Auswahl stehen:

- Im **Normal Mode** wird bei Benutzer-Prädikaten der erste bzw. jeder erneute Aufruf angezeigt sowie das jeweilige Resultat.
- Im **Full Mode** werden zusätzlich alle Klauseln angezeigt, für die eine Unifikation versucht wurde.

System-Prädikate lassen sich verfolgen, die Anzeige ist prädikat-spezifisch. Die Trace-Optionen können nach jeder Antwort auf eine Anfrage verändert werden, nicht nur beim Stellen einer neuen Anfrage.

8.4 Schnittstellenbeschreibung

Im folgenden werden die Leistungen - Methoden, Funktionen oder Makros - vorgestellt, die von einer der Varianten des Prolog-Prozessors über das entsprechende Prozessor-Mixin in einer Wissensbasis-Konfiguration zu Verfügung gestellt werden. Ausgeklammert bleiben Leistungen, die nur internen Zwecken dienen.

8.4.1 basic-prolog-mixin

Sofern bei den folgenden Methoden IO-Operationen vorkommen, wird dazu das Dialog-Window benutzt. Es ist Aufgabe des Interface-Mixins, dieses festzulegen.

Axiom Set Verwaltung

:select-load-axioms		Methode
Menügesteuerte Auswahl der aktuellen Axiom Sets.		
:add-axiom-set	<i>AxiomSet</i> &optional <i>Mode BeforeAxiomSet Check</i>	Methode
<i>AxiomSet</i> wird zu den aktuellen Axiom Sets hinzugefügt. Die Position wird durch die optionalen Parameter bestimmt.		
<i>Mode</i> darf sein: first last before		
Bei first bzw. last wird <i>AxiomSet</i> an erster bzw. letzter Stelle eingefügt, bei before vor <i>BeforeAxiomSet</i> , falls angegeben, sonst an letzter Stelle. first ist default.		
Falls <i>Check</i> nicht NIL ist, wird geprüft, ob <i>AxiomSet</i> bekannt ist.		
:remove-axiom-set	<i>AxiomSet</i>	Methode
<i>AxiomSet</i> wird aus der Liste der aktuellen Axiom Sets gestrichen.		

:reset-axiom-sets &rest *AxiomSets* **Methode**

Die *AxiomSets* werden in ihren Ausgangszustand zurückversetzt, d.h. alle per Prolog-Programm oder interaktiv durch (Re)consult bewirkten Änderungen werden zurückgenommen. Nur aktuelle Axiom Sets lassen sich zurücksetzen. Werden keine *AxiomSets* angegeben, werden alle aktuellen zurückgesetzt.

:show-axioms **Methode**

zeigt die Namen der aktuellen Axiom Sets an.

Auswertung

:display-result &optional *Dispform Redisplay* **Methode**

zeigt das Resultat des letzten Beweises (erneut) an.

Die zulässige Werte von *Dispform* sind: **form** | **vars** | **bound** | **status** | **no**

Bei **form** wird die letzte Anfrage nach Substitution aller Prolog-Variablen durch ihre Werte angezeigt, falls der Beweis erfolgreich war, ansonsten wird NO ausgegeben.

Bei **vars** werden alle nicht-anonymen Variablen mit ihren Werten angezeigt, falls der Beweis erfolgreich war, ansonsten wird NO ausgegeben.

bound entspricht vars, nur werden alle Variablen ausgelassen, deren Wert selbst wieder eine Variable ist.

Treten keine Variablen auf, wird in den beiden letzten Fällen bei Erfolg YES ausgegeben.

Bei **status** wird nur der Status des Beweises angezeigt. Mögliche Meldungen sind SUCC, FAIL, ERROR.

no entspricht status, falls der Parameter *Redisplay* nicht NIL ist, ansonsten unterbleibt eine Anzeige. Dies ist nur sinnvoll, falls die Anzeige vom Prolog-Programm selbst vorgenommen wird.

Wird kein Wert angegeben, wird die für den Prolog-Prozessor gewählte Einstellung benutzt.

:select-format **Methode**

präsentiert ein Menü zur Einstellung des Formats, in dem die Ergebnisse angezeigt werden.

:prove-display &optional *Goals Dispform* **Methode**

liefert erste bzw. nächste Antwort auf eine Anfrage.

Falls *Goals* NIL ist, wird eine Anfrage vom Benutzer angefordert und die erste Antwort ermittelt. Bei direkter Angabe einer Anfrage wird ebenfalls die erste Antwort ermittelt. Falls *Goals* * ist, wird von der letzten Anfrage ausgegangen und nach der nächsten Antwort gesucht.

Der Parameter *Dispform* steuert die Form der Anzeige. Die zulässige Werte sind: **form** | **vars** | **bound** in der üblichen Bedeutung.

:prolog-prove-loop &optional *Goals Cont Dispformat* **Methode**

liefert ein oder mehrere Antworten auf eine Anfrage.

Falls *Goals* NIL ist, wird eine Anfrage vom Benutzer angefordert, die erste Antwort ermittelt und evtl. weitere. Bei direkter Angabe einer Anfrage wird ebenfalls die erste Antwort ermittelt und evtl. weitere. Falls *Goals* * ist, wird von der letzten

Anfrage ausgegangen und nach den nächsten Antworten gesucht.

Der Parameter *Cont* steuert, wieviel Antworten gegeben werden. Ist *Cont* ONE, ALL oder eine Zahl, werden eine Antwort, alle Antworten oder soviel Antworten wie verlangt ermittelt. Ansonsten wird der Benutzer gefragt, wie weiter verfahren werden soll. Gibt er ; ein, wird die nächste Antwort ermittelt, andernfalls wird abgebrochen.

Der Parameter *Dispform* steuert die Form der Anzeige. Die zulässige Werte sind: **form** | **vars** | **bound** in der üblichen Bedeutung.

:prolog-prove &optional *Goal Disp-format* **Methode**
 liefert die erste oder nächste Antwort auf eine Anfrage.
 Falls *Goals* NIL ist, wird von der letzten Anfrage ausgegangen und nach der nächsten Antwort gesucht. Bei Angabe einer Anfrage wird die erste Antwort ermittelt.
 Für *Disp-format* sind folgende Werte zulässig: **form** | **vars** | **bound**
form liefert als Wert die letzte Anfrage nach Substitution aller Prolog-Variablen durch ihre Werte, falls der Beweis erfolgreich war, ansonsten NIL.
vars liefert als Wert eine Assoziationsliste aller nicht-anonymen Variablen mit ihren Werten, falls der Beweis erfolgreich war, ansonsten NIL.
bound liefert analog zu vars eine Liste aller gebundenen nicht-anonymen Variablen. Treten keine Variablen auf, wird in den beiden letzten Fällen bei Erfolg YES zurückgegeben.

Anzeige von Prädikaten und Axiom Sets

:list-predicate *Predicate Axiom-name* &optional *Window* **Methode**
 zeigt ein Prädikat aus einem Axiom Set auf dem angegebenen Fenster an. Wird kein Fenster genannt, wird das Dialog-Fenster verwendet.

:list-axset *Axiom-name* &optional *Window* **Methode**
 zeigt einen Axiom Set auf dem angegebenen Fenster an. Wird kein Fenster genannt, wird das Dialog-Fenster verwendet.

:list-axioms **Methode**
 zeigt ein oder alle Prädikate aus einem bekannten Axiom Set auf dem Dialog-Fenster an. Axiom Sets und Prädikate werden per Menü zur Auswahl gestellt.

8.4.2 mini-prolog-mixin

Das **mini-prolog-mixin** erweitert **basic-prolog-mixin** um Aspekte des Tracing. Die Ausgabe des Prolog-Trace erfolgt auf das an die Instanzvariable **prolog-trace-window** gebundene Fenster.

:send-prolog-trace-window *selector* &rest *args* **Methode**
 leitet Nachrichten an **prolog-trace-window** weiter.

:show-trace-status **Methode**
 zeigt den Trace-Modus an und welche Prädikate protokolliert werden.

:set-trace-mode <i>Mode</i>	Methode
setzt den Trace-Modus auf FULL bzw. NORMAL	
:trace-preds <i>Preds</i>	Methode
sorgt dafür, daß die spezifizierten Prädikate protokolliert werden. Ist <i>Preds</i> ALL, werden alle Prädikate protokolliert.	
:add-to-trace	Methode
fragt nach Prädikaten, die zusätzlich protokolliert werden sollen.	
:rem-from-trace	Methode
präsentiert ein Menü der momentan protokollierten Prädikate. Die vom Benutzer ausgewählten werden nicht länger protokolliert.	
:select-for-trace	Methode
präsentiert Menüs der aktuellen Prädikate. Die vom Benutzer ausgewählten werden protokolliert.	

8.4.3 normal-prolog-mixin

Noch nicht festgelegt.

Kapitel 9

Konstruktion des Prolog-Prozessors

9.1 Übersicht

Die Konstruktion des Prolog-Prozessors verfolgt das Ziel, eine ausgewogene Modularisierung durch Abtrennung der Aspekte Tracing, Erklärung und Entwicklungsunterstützung vom reinen Interpreter zu erreichen. Es stehen drei Varianten des Prolog-Prozessors zur Verfügung. Der **Basic-Prolog-Processor**, die einfachste Variante, besteht nur aus dem Interpreter, der **Mini-Prolog-Processor** eröffnet zusätzlich Trace-Möglichkeiten, erst der **Normal-Prolog-Processor** bietet den vollen Leistungsumfang.

Die vom **Normal-Prolog-Processor** gebotene Entwicklungsunterstützung ist z.Z. noch stark maschinenabhängig und soll im Interesse höherer Portabilität einer Revision unterworfen werden. Aus diesem Grund werden im folgenden nur die beiden ersten Varianten genau beschrieben.

9.2 Zerlegung

Da der **Mini-Prolog-Processor** eine Spezialisierung des **Basic-Prolog-Processor** darstellt und damit alle Komponenten des letzteren auch Komponenten des ersteren sind, genügt es, die Zerlegung des **Mini-Prolog-Processor** zu beschreiben (siehe Abbildung 9.1):

1. **processor-core**

Die Basiskomponente aller Prozessoren, hier des Prolog-Prozessors, die die Schnittstelle zum Meta-Prozessor realisiert und die Instanzvariablen definiert, die benötigt werden, um den Prozessor ins Gesamtsystem zu integrieren.

2. **axset-basic**

Verwaltet Klauseln und Klauselmengen.

3. **proc-sc-mixin**

Transformiert und unifiziert Klauseln im Beweisprozeß. Dabei wird ein **Structure Copying** Ansatz verfolgt, d.h. alle Klauseln werden vor der Unifikation kopiert, wobei sämtliche Prolog-Variablen durch neu generierte ersetzt werden.

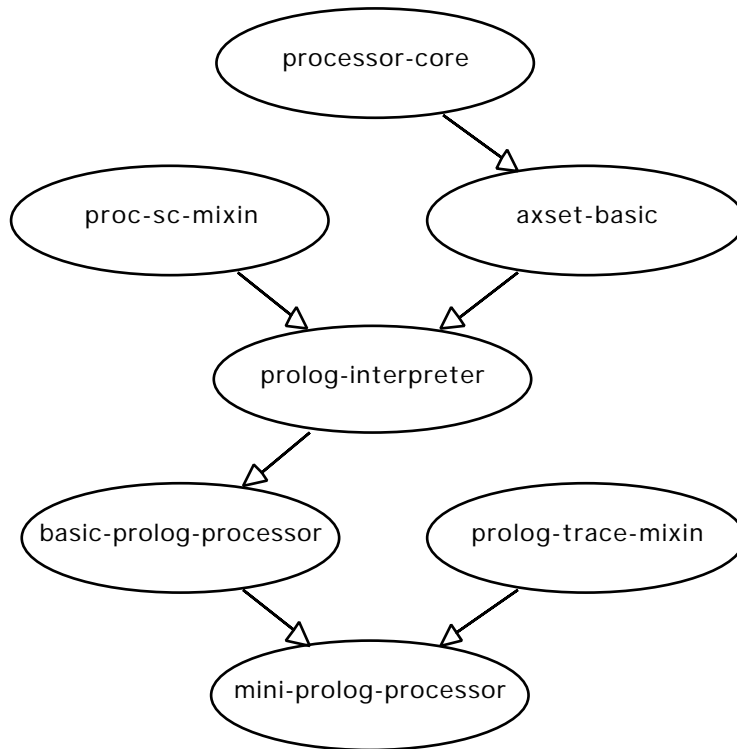


Abbildung 9.1: Architektur des Prolog-Prozessors

4. prolog-interpreter

Stößt den Beweis der vom Benutzer vorgegebenen Goals an.

5. basic-prolog-processor

Stellt noch fehlende Methoden bereit.

6. prolog-trace-mixin

Verwaltet alle Parameter, die das Tracen eines Prolog-Beweises steuern.

7. mini-prolog-processor

Kombiniert den Basic-Prolog-Processor mit der Trace-Komponente.

Der zur Beantwortung von Benutzerfragen verwendete Algorithmus lehnt sich an das **Box-Modell** von L. Byrd (siehe [CloMe, Kapitel 8.3]) an: für jedes Goal wird eine Box mit zwei Eingängen – für den ersten bzw. jeden wiederholten Beweisversuch des Goals – und zwei Ausgängen – für das Gelingen bzw. das Mißlingen des Beweises – dargestellt. Eine Box wird in **BABYLON**-Prolog durch Instanzen eines entsprechenden Flavors dargestellt. Im Zuge eines Beweises wird ein Baum aus solchen Instanzen aufgebaut. Dieser Baum verkörpert die dynamische Datenbasis des Prolog-Prozessors.

Das Flavor zur Beschreibung eines Goals hängt ab von der benutzten Prolog-Variante. Der **Basic-Prolog-Processor** benutzt das Flavor **basic-goalbox**, der **Mini-Prolog-Processor** das Flavor **mini-goalbox**. Letzteres besteht aus folgenden Komponenten (siehe Abbildung 9.2):

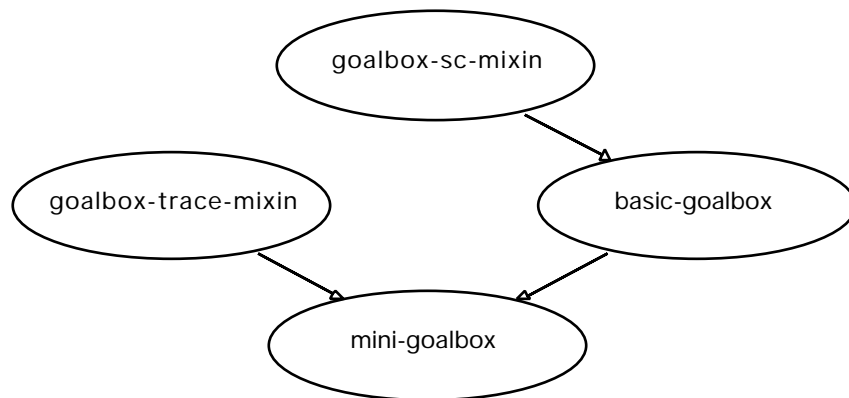


Abbildung 9.2: Goal-Repräsentation

1. **goalbox-sc-mixin**

Pendant zu **proc-sc-mixin**. Stellt für Goal-Instanzen die zur Behandlung von Klauseln benötigten Methoden bereit, soweit sie vom Structure-Copying-Ansatz abhängen.

2. **basic-goalbox**

Minimale Verkörperung eines Goals. Enthält die zum Beweis von Benutzeranfragen benötigten Methoden.

3. **goalbox-trace-mixin**

Erlaubt im Zusammenwirken mit dem **prolog-trace-mixin** das Tracen von Beweisen.

4. **mini-goalbox**

Kombiniert **basic-goalbox** mit der Trace-Komponente.

9.3 Integration ins Gesamtsystem

Die Integrationsfähigkeit des Prolog-Prozessors wird prozessorseitig durch das Flavor **processor-core** sichergestellt. Zur Verwendung innerhalb einer Wissensbasis-Konfiguration definiert das Flavor **basic-prolog-mixin** bzw. **mini-prolog-mixin** die Leistungen der Prolog-Variante, die im Gesamtsystem verfügbar gemacht werden sollen.

Das Flavor **processor-core** ist Bestandteil von **axset-basic** und realisiert das Interface – die verlangten **Instanzvariablen** – zum Meta-Prozessor. Eine Beschreibung der Rolle dieses Flavors für die Einbindung ins Gesamtsystem findet sich in Kapitel 3.

Das Flavor **basic-prolog-processor** bzw. **mini-prolog-processor** legt die Konfiguration aus dem eigentlichen Interpreter **prolog-interpreter** und der optionalen Komponente **prolog-trace-mixin** fest.

9.4 Der Basic-Prolog-Processor

Die folgenden drei Unterabschnitte beschreiben, wie die für einen Prolog-Prozessor unabhängigen Basisleistungen des Prolog-Interpreters realisiert sind.

9.4.1 Klauselverwaltung

Klauseln werden zu **Klauselmengen** (Axiom Sets) zusammengefaßt. Klauselmengen können Bestandteil einer Wissensbasis oder eigenständig sein, wobei eine Wissensbasis nur eine Klauselmenge enthalten darf. Sie werden durch eine Instanz des Flavor **axset-basic** verwaltet.

Die Klauselverwaltung generiert aus der externen Darstellung einer Klauselmenge eine interne Repräsentation. Konstruktor-Makro des Klauselteils einer Wissensbasis bzw. einer eigenständigen Klauselmenge ist

```
(DEFRELATIONS <Klausel1> ... <KlauselN>)
bzw.
(DEFAXIOM-SET <Name> <Klausel1> ... <KlauselN>)
```

Bei Klauselmengen, die zu einer Wissensbasis gehören, wird der Name der zugehörigen Wissensbasis auf der Eigenschaftsliste der Klauselmenge unter dem Indikator **kb-name** vermerkt. Die den Konstruktoren übergebenen Klauseln werden auf der Eigenschaftsliste der Klauselmenge unter dem Indikator **clauses** geführt. Dies wird zum Zurücksetzen einer Klauselmenge benutzt. Der Wert der Eigenschaft wird nur von den Konstruktoren verändert.

Die definierenden Klauseln eines Prädikats werden intern in normalisierter Form auf der Eigenschaftsliste des Prädikats geführt. Als Indikator dient der Name der jeweiligen Klauselmenge. Die in einer Klauselmenge definierten Prädikate werden als **preds**-Eigenschaft der jeweiligen Klauselmenge geführt.

Die **BABYLON** bekannten Klauselmengen werden in einer Liste geführt, die an die globale Variable ***axiom-sets*** gebunden ist. Die **aktuellen** - d.h. zum Beweis herangezogenen - Klauselmengen werden an die Instanzvariable **axioms** gebunden. Die Klauselverwaltung enthält Methoden zur Modifikation dieser Liste.

Die Funktion **get-clauses-direct** beschafft die für ein Goal relevanten Klauseln aus den aktuellen Klauselmengen. Mit den Methoden **:find-axiom-set** bzw. **:select-axiom-set** wird die aktuelle Klauselmenge ermittelt, in der Klauseln für ein Prädikat zu finden bzw. abzulegen sind.

Die Klauselverwaltung enthält ferner Methoden bzw. Funktionen, um Klauseln bzw. Klauselmengen in aufbereiteter Form drucken zu können.

9.4.2 Klausel-Transformation und Unifikation

Um beim Beweisprozeß Variablenkollisionen zu vermeiden, wird ein **Structure-Copying-Ansatz** verwendet: Jede Klausel wird vor Unifikation transformiert, wobei die Prolog-Variablen durch neu generierte, interne Variablen ersetzt werden. Alle Methoden zur Handhabung von Klauseln im Beweisprozeß beruhen auf dieser Entscheidung. Da alternative Ansätze möglich sind, wurden sie aus dem Flavor **axset-basic** zur Handhabung von Klauseln herausfaktoriert und den Mixins **proc-sc-mixin** bzw. **goalbox-sc-mixin** zugewiesen.

Prolog-Variablen werden intern durch Arrays (**Varcells** genannt) dargestellt. Sie haben eine Nummer, einen Namen (der Name der externen Variablen, die sie ersetzen), einen Wert und eine Bindungsnummer (die Position der Varcell in einem Bindungsstack). Der Bindungsstack wird geführt, um beim Backtracking Bindungen rückgängig machen

zu können. Eine Varcell wird in den Stack aufgenommen, sobald sie instantiiert wird, d.h. sobald ihr ein Wert zugewiesen wird, was im Zuge der Unifikation geschieht.

Die Funktion **trans-clause** dient dazu, in einer Klausel sämtliche Prolog-Variablen durch zugeordnete Varcells zu ersetzen. Die Funktion **subst-prolog-vars** ersetzt in einer Klausel alle Varcells durch ihre Werte. Da der Wert einer Varcell selbst ein Term mit Varcells sein kann, muß **subst-prolog-vars** rekursiv aufgerufen werden. Der Parameter *Mode* steuert, ob eine nicht instantiierte Varcell durch ihren Namen, ihre Nummer oder sich selbst substituiert wird. **rest-subst-prolog-vars** stellt eine Variante dieser Funktion dar, die Instantiierungen nur bis zu einer bestimmten Bindungsnummer berücksichtigt.

Die Methode **:get-clauses** ermittelt die für ein Goal relevanten Klauseln. Dabei wird die Funktion **get-clauses-direct** (s. Klauselverwaltung) benutzt. Liefert **get-clauses-direct** keine Klauseln, wird nach Substitution der Prolog-Variablen im Goal die Wissensbasis eingeschaltet.

Die Methode **:trans-unify** unifiziert Goal und Klausel nach deren Transformation. Das Rücksetzen des Stacks bis zu einer bestimmten Tiefe wird durch die Methode **:reset-env** besorgt. Die benötigte Tiefe wird in der Instanzvariablen **init-env-depth** von **goalbox-sc-mixin** vermerkt.

System-Prädikate können Seiteneffekte auslösen, die beim Backtracking zurückgesetzt werden. Damit auch im Falle eines **cut** das Zurücksetzen gewährleistet ist, wird ein Stack geführt, der die entsprechenden Goal-Instanzen aufnimmt. Die Methode **:cut-reset** leistet das Zurücksetzen im cut-Fall.

9.4.3 Der Beweisprozeß

Der Prolog-Prozessor kann nur jeweils einen Beweis führen, der sich nicht unterbrechen und wieder aufnehmen läßt. Wie bereits ausgeführt wird im Zuge eines Beweises ein Baum aufgebaut, dessen Knoten Goals darstellen und als Instanzen eines Flavors implementiert sind. Dies Flavor ist modular aufgebaut. **basic-goalbox** stellt die minimale Verkörperung eines Goals dar und enthält als Komponente **goalbox-sc-mixin**. **basic-goalbox** enthält eine Reihe von Instanz-Variablen, um Instanzen des Flavors miteinander zu verknüpfen: Ist ein Goal aufgrund einer Regel zu beweisen, werden entsprechend den Prämissen der Regel Instanzen erzeugt. Sie werden als Knoten gleicher Ebene untereinander durch Vorwärts- und Rückwärtsverweise verbunden und dem Goal, das sie beweisen sollen, als direkte Nachfolgeknoten zugeordnet. Die vom Benutzer eingegebenen Goals werden als Prämissen eines fiktiven Goals (**%top**) geführt.

basic-goalbox stellt Methoden zum Beweis eines Goals bereit. Für jedes System-Prädikat gibt es eine eigene Methode, für Benutzer-Prädikate eine Standardmethode. Das Makro **defprolog-method** legt die Methode auf der Eigenschaftsliste des jeweiligen Systemprädikats bzw. des Pseudoprädikats **%normal** unter dem Indikator **prolog-method** ab. Zu Laufzeit kann an die Stelle dieser Methoden eine verkapselte Version, etwa zum Tracen, treten. Die zu Laufzeit relevante Methode ist auf der Eigenschaftsliste des jeweiligen Prädikats bzw. des Pseudo-Prädikats **%normal** unter dem Indikator **curr-prolog-method** zu finden. **defprolog-method** initialisiert bei System-Prädikaten **curr-prolog-method** mit der Normalmethode und fügt außerdem das Prädikat in die Liste aller System-Prädikate ein, die an die globale Variable ***prolog-syspreds*** gebunden wird.

Die Methode **:prove-goal** ermittelt die letztlich benötigte Beweismethode und führt

sie aus. Im Standardfall ist dies **:prove-normal**. Hat **:prove-normal** ein Goal aufgrund einer Regel zu beweisen, werden die Prämissen-Instanzen durch **:generate-subgoals** erzeugt und durch **:prove-subgoals** bewiesen. Diese Methode realisiert den Backtracking-Algorithmus und benutzt dazu **:prove-goal**. Entsprechend den beiden Eingängen des Boxmodells verfügen die Beweismethoden über einen Parameter *mode* mit den Werten **try** bzw. **retry**. Das fiktive Topgoal (**%top**) wird wie ein Goal mit Systemprädikat **%top** behandelt. Die zugehörige Goalbox-Instanz wird an die Instanzvariable **root** von **prolog-interpreter** gebunden.

Das Flavor **prolog-interpreter** dient dazu, Beweise anzustoßen und die Zurückgabe von Ergebnissen zu veranlassen.

9.5 Der Mini-Prolog-Processor

Der **Mini-Prolog-Processor** fügt zu den Leistungen des **Basic-Prolog-Processors** Trace-Möglichkeiten hinzu. Die Ausgabe des Prolog-Trace erfolgt auf das an die Instanzvariable **prolog-trace-window** gebundene Fenster. Es ist Aufgabe des Interface-Mixins dieser Instanzvariablen ein geeignetes Fenster zuzuordnen. Dieses muß die Methode **:format** verstehen. Nachrichten werden nicht direkt an dies Fenster geschickt, sondern indirekt über die Methode **:send-rule-trace-window**.

Die zum Tracen benötigten Methoden werden in den Mixins **prolog-trace-mixin** und **goalbox-trace-mixin** bereitgestellt:

goalbox-trace-mixin stellt die Methoden zur Verfügung, die zum Tracen eines Beweises benötigt werden. Jeder Beweismethode eines System-Prädikats und der Standardmethode entspricht eine Trace-Variante, die eine Verkapselung der Ausgangsmethode darstellt. Sie wird aus dieser und Methoden generiert, die vor bzw. nach ihr aufgerufen werden. Der nachher auszuführenden Methode wird im Unterschied zu **:after**-Dämonen über einen zweiten Parameter das Ergebnis der Ausgangsmethode weitergereicht. Das Makro **defprolog-trace-methods** generiert die verkapselte Methode und legt diese auf der Eigenschaftsliste des jeweiligen System-Prädikats bzw. des Pseudo-Prädikats **%normal** unter dem Indikator **prolog-trace-method** ab.

prolog-trace-mixin hält fest, welche Prädikate zu tracen sind, und sorgt dafür, daß bei Änderungen die normale bzw. die Trace-Methode zur **curr-prolog-method** gemacht wird. Es wird eine Liste der momentan auf Trace eingestellten Prädikate geführt. Sie ist an die globale Variable ***prolog-preds-traced*** gebunden. Unter dem Indikator **set-by** wird vermerkt, welcher Prolog-Prozessor das Tracen veranlaßte. Vor jedem Beweis wird überprüft, ob ***prolog-preds-traced*** vom momentanen Prolog-Prozessor gesetzt wurde. Ist dies nicht der Fall, wird der für den momentanen Prozessor gültige Trace-Zustand restauriert. Dieser wird in entsprechenden Instanzvariablen (**trace-list** bzw. **trace-mode**) des Prozessors festgehalten.

Das Tracen der Klauseln im Full Mode wird durch die Methode **:before :trans-unify** bewerkstelligt. Dasselbe bewirkt die Methode **:before :clause-trans-unify** für die Systemprädikate **clause** und **retract**

9.6 Importierte Leistungen

Neben den Standard-IO-Leistungen der Benutzerschnittstelle benutzt der Prolog-Prozessor die **:eval**-Methode des Meta-Prozessors, um externe Prämissen zu evaluieren. Dabei wird der Auswertungsmodus **:prolog** benutzt und als Antwort T, NIL oder eine Liste von Prolog-Klauseln erwartet, die Klauseln von Benutzer-Prädikaten verwendet werden.

9.7 Exportierte Leistungen

Das Flavor **basic-prolog-mixin** bzw. **mini-prolog-mixin** definiert die Leistungen der Prolog-Variante - **basic-prolog-processor** bzw. **mini-prolog-processor** -, die im Gesamtsystem verfügbar sein sollen. Der Aufbau folgt den in Kapitel 3 beschriebenen Prinzipien zur Einbindung eines Prozessors in das Gesamtsystem.

Prolog-Goals dürfen im IF-Teil von Regeln auftreten. Sie werden vom Typerkennungsmacro der Flavor **basic-prolog-mixin** bzw. **mini-prolog-mixin** erkannt und von den zugehörigen Prozessoren evaluiert, falls der Arbeitsmodus **:recall** bzw. **:recall-immediate** lautet, der bei der Übergabe des Ausdrucks an den Meta-Prozessor benutzt wurde. Im Falle von **:recall** wird ein Prolog-Beweis angestoßen, bei **:recall-immediate** wird sofort NIL zurückgegeben.

Kapitel 10

Spezifikation der Benutzerschnittstelle

10.1 Anforderungen

BABYLON sollte möglichst portabel sein, andererseits sollten auf den jeweiligen Maschinen zur Verfügung stehende Fenstersysteme und Selektionstechniken (per Maus z.B.) genutzt werden können. Um diesen konträren Wünschen gerecht zu werden, wurde die Kommunikation mit dem Benutzer in einer Komponente konzentriert. Die Prozessoren für die einzelnen Formalismen kommunizieren nie direkt mit dem Benutzer, sondern wenden sich bei Wünschen nach Ein/Ausgabe an die Benutzerschnittstelle. Im Interesse der Portabilität war das dabei zu verwendende Protokoll so anzulegen, daß es sich unter alleiniger Verwendung von Funktionen, die zum Common-Lisp Standard gehören, realisieren läßt.

Über die Einhaltung des Basisprotokolls hinaus werden keine Anforderungen an die Gestaltung der Benutzerschnittstelle gestellt. Der Benutzer ist frei, seine Benutzerschnittstelle den Möglichkeiten und Gepflogenheiten der Zielmaschine entsprechend zu gestalten.

Jedes ablauffähige Expertensystem sollte mit einer eigenen, den speziellen Bedürfnissen angepaßten Benutzerschnittstelle ausgestattet werden können. Insbesondere sollte die Sprache, die bei Kommandos oder Anzeigen verwendet wird, allein von der Wissensbasis abhängen. Dies schließt die Verwendung generischer Benutzerschnittstellen nicht aus und auch nicht die gemeinsame Nutzung von Ressourcen (z.B. Fenstern) durch mehrere Expertensysteme.

10.2 Einordnung

Die Benutzerschnittstelle ist die Komponente des ablauffähigen Expertensystems, die die Kommunikation mit dem Benutzer abwickelt. Gegenstand dieses Kapitels ist das von den Standardprozessoren erwartete Protokoll. Es enthält keine Vorschriften darüber, wie die Schnittstelle und ihre Methoden zu gestalten sind. Insbesondere wird kein Fenstersystem vorausgesetzt oder spezielle Interaktionstechniken wie die Selektion per Maus.

BABYLON enthält ein Basisinterface nebst einer Erweiterung, um Operationen per Menüauswahl auslösen zu können. Diese Ausprägungen der Benutzerschnittstelle sind realisiert unter ausschließlicher Verwendung von Funktionen, die zum Common-Lisp Standard

gehören, und daher einfach zu portieren. Sie sind jedoch nur als Provisorium und Ausgangspunkt für die Entwicklung einer eigenen Benutzerschnittstelle zu werten, die auf die jeweilige Anwendung zugeschnitten ist und die Möglichkeiten der Zielmaschine nutzt. Nur als solche werden sie in diesem Kapitel herangezogen.

10.3 Funktionsbeschreibung

Im folgenden wird beschrieben, welche Anforderungen von den Standardprozessoren an die Benutzerschnittstelle gestellt werden und welche Konventionen zu befolgen sind.

Prozessorseitig wird vorausgesetzt, daß ein Stream für den Dialog mit dem Benutzer besteht, von dem gelesen und auf den geschrieben werden kann. Wie dieser Stream aussieht, bestimmt allein die Benutzerschnittstelle. Beim Basisinterface wird der an die Variable ***terminal-io*** gebundene Stream verwendet.

Im Prinzip könnte die gesamte Kommunikation mit dem Benutzer über den Dialogstream abgewickelt werden, doch ist dies bei bestimmten Aufgaben z.B. kurzen Mitteilungen an den Benutzer oder Bestätigungen durch den Benutzer vor der Ausführung von Operationen u.U. recht schwerfällig. Bei Maschinen, die über ein Fenstersystem und Auswahlmenüs verfügen, sind andere Realisierungen angemessener. Um hier differenzieren zu können, rufen die Prozessoren spezielle Methoden auf, die von der Benutzerschnittstelle bereitgestellt werden müssen. Beim Basisinterface wird auch bei den soeben angesprochenen Methoden nur wieder der Dialogstream benutzt, da dies Interface der Portabilität zuliebe keine komfortableren Interaktionstechniken voraussetzt.

Weiter erwarten die Prozessoren vom Interface, Menüs präsentieren zu können, aus denen der Benutzer Einträge auswählen kann. Benötigt werden zwei Varianten: die eine erlaubt die Auswahl eines einzelnen Eintrags, die zweite die gleichzeitige Wahl mehrerer Einträge. Das Basisinterface bietet simple Menüs an, bei denen die Auswahl durch Eingabe der Zeilennummern der Einträge erfolgt, wobei wieder der Dialogstream benutzt wird.

Die Standardprozessoren kennen neben dem Dialogstream zusätzliche Streams zum Tracen oder für Erklärungen. Von diesen Streams wird vorausgesetzt, daß es sich um Objekte handelt, die die Nachricht **:format** verstehen. Es ist Sache der Benutzerschnittstelle, diese Streams zuzuweisen. Falls ein Fenstersystem zur Verfügung steht, wird man hierzu spezielle Fenstertypen verwenden. Die Objekte werden daher auch generell als Fenster bezeichnet. Standardmäßig sind folgende Fenster bekannt: *system-trace-window*, *rule-trace-window*, *prolog-trace-window* und *explanation-window*. Die Prozessoren setzen nicht voraus, daß alle diese Fenster verschiedene Objekte sind, auch können mehrere Wissensbasen dieselben Fenster benutzen. Das Basisinterface generiert Pseudo-Fenster, d.h. Objekte, die auf den an ***terminal-io*** gebundenen Stream schreiben, der auch als Dialogstream benutzt wird.

Die primitivste Art, von einer ablauffähigen Wissensbasis Leistungen zu verlangen, und die einzig mögliche beim Basisinterface, besteht darin, an die Wissensbasis Nachrichten zu schicken. Da viele Nachrichten direkt oder indirekt die globalen Variablen ***current-knowledge-base*** und ***language*** verwenden, sollten diese zur Zeit des Aufrufs an die betroffene Wissensbasis bzw. ihre Sprache gebunden sein. Die korrekte Bindung wird durch die Methode **:select-kb** sichergestellt. **:select-kb** sorgt außerdem für die korrekte Synchronisation zwischen der Wissensbasis und ihren Prozessoren, wie sie nötig wird, wenn Teile der Wissensbasis neu evaluiert wurden. Vor anderen Nachrichten an eine

Wissenbasis sollte daher ein **:select-kb** abgesetzt werden oder der Funktionsaufruf (`call-kb <kb-name>`) erfolgen, der **:selekt-kb** aufruft. (Bestimmte Methoden wie **:start**, **:start-kb**, **:reset-kb** binden **current-knowledge-base** lokal an **self** und **language** an die zugehörige Sprache, so daß sich hier ein vorausgehendes **:select-kb** erübrigt).

Es ist Sache der Benutzerschnittstelle, ob neben dem Versenden von Nachrichten andere Möglichkeiten angeboten werden sollen, um bestimmte Operationen auszulösen z.B. durch Präsentation von Menüs. Das Miniinterface stellt gerade eine Erweiterung des Basisinterfaces um eine Komponente dar, die eine solche Auswahl gestattet.

10.4 Schnittstellenbeschreibung

Im folgenden wird das von den Prozessoren erwartete Protokoll beschrieben.

10.4.1 Kommunikation über den Dialogstream

:babylon-read *&optional special-keys* **Methode**

Diese Methode liest das nächste Zeichen vom Dialogstream, falls es zu denen gehört, die mit der Liste *special-keys* übergeben wurden, andernfalls wird die nächste Lisp-Form gelesen.

:babylon-format *fstr &rest args* **Methode**

Diese Methode schreibt formatiert auf den Dialogstream. *fstr* wird dabei in gleicher Weise als Kontrollstring benutzt, wie dies bei der Lisp-Funktion **format** der Fall ist.

:format *fstr &rest args* **Methode**

Diese Methode entspricht **:babylon-format**. Sie erlaubt es, die Wissensbasis wie ein Fenster zu behandeln, für das eine derartige Methode obligatorisch ist.

10.4.2 Spezielle Kommunikationsformen

:notify *&rest comments* **Methode**

Diese Methode ist für Mitteilungen an den Benutzer bestimmt. *comments* enthält die einzelnen Textzeilen als Zeichenketten.

:confirm *&rest comments* **Methode**

Diese Methode ist für Mitteilungen an den Benutzer bestimmt, die von ihm eine Reaktion - zumeist eine Bestätigung - verlangen. *comments* enthält den Text. Bestätigung liefert den Wert **:yes**

:type-end-to-continue *string* **Methode**

Diese Methode wartet darauf, daß der Benutzer die an ***end-key*** gebundene Taste drückt. *string* wird als Prompt für den Benutzer verwendet.

:prompt-for-input *string* **Methode**

Diese Methode fordert den Benutzer zur Eingabe einer Zeile auf, die als Zeichenkette übergeben wird. *string* dient als Prompt.

10.4.3 Auswahl aus einem Menü

:choose-from-menu *item-list* &optional *label* &rest *ignore* **Methode**

Diese Methode präsentiert ein Menü und führt die vom ausgewählten Eintrag verlangte Operation aus. Es kann nur jeweils ein Eintrag ausgewählt werden.

Jedes Element der Liste *item-list* entspricht einem Eintrag und hat im allgemeinsten Fall die Form: (<name> <operation> <argument>. <options>).

<Name> kann ein Symbol oder String sein und dient zur Anzeige.

Als <operation> sind zulässig **:value**, **:funcall**, **:eval** und **:no-select**, sie bewirken, daß <argument> als Wert übergeben, als Funktion aufgerufen oder als Lisp-Form evaluiert wird, während **:no-select** den Eintrag als nicht selektierbar kennzeichnet.

Kurzformen zur Spezifikation eines Eintrags sind:

< name>, (<name> . <value>) oder (<name> <value>)

was (<name> **:value** <name>) bzw. (<name> **:value** <value>) entspricht.

Der String *label* dient als Kopfzeile des Menüs.

:mult-choose-from-menu *mult-choose-item-list* &optional *header* **Methode**

Diese Methode präsentiert ein Menü und liefert eine Liste von Werten entsprechend den ausgewählten Einträgen. Dabei können mehrere Einträge zugleich ausgewählt werden.

Jedes Element der Liste *mult-choose-item-list* entspricht einem Eintrag und hat die Form (<value> <string> (t)).

<string> dient zur Anzeige, <value> ist der Wert des Eintrags.

Der String *header* dient als Kopfzeile des Menüs.

10.4.4 Kommunikation über spezielle Fenster

Für jeden Prozessor, der einen anderen Stream als den Dialogstream verwenden will, muß im zugehörigen Mixin ein Slot für den Stream bereitstehen, und es muß eine Methode folgender Art definiert sein:

:send-<name>-window *selector* &rest *args* **Methode**

Diese Methode schickt die Nachricht *selector* nebst *args* an das jeweilige Fenster.

Prozessorseitig wird als Selektor nur **:format** verwendet. Die Benutzerschnittstelle könnte aber noch weitere Selektoren benutzen z.B. **:expose**, **:bury**, **:clear**. Sie hat selbst dafür zu sorgen, daß nur Streamobjekte den jeweiligen Slots zugewiesen werden, die diese Methoden verstehen.

10.4.5 Schnittstelle zum Benutzer

Ob der Benutzer bei einer Wissensbasis Operationen anders als durch Versenden von Nachrichte auslösen kann, wird von den Standardprozessoren nicht präjudiziert. Das Miniinterface z.B. präsentiert Operationen in einem Menü, das sich dynamisch ändert. Es besteht aus zwei Zonen, die eine bleibt unverändert, die zweite zeigt wechselnde Untermenüs. Der Wechsel der Untermenüs wird durch Operationen im Menü selbst ausgelöst. Eine spezielle Menüoperation erlaubt es, zum vorherigen Untermenü zurückzukehren. Das Menü wird in einer Schleife präsentiert, die über eine Exit-Operation zu verlassen ist. Gestartet wird die Loop durch die Methode **:run**, die von **:select-kb** aufgerufen wird.

Kapitel 11

Konstruktion der Benutzerschnittstelle

11.1 Übersicht

Jedes ablauffähige Expertensystem enthält als Komponente seine eigene Benutzerschnittstelle. Sie ist als einzelnes Mixin oder als Gruppe von Mixins zu realisieren. Über die **:interface** Option des **def-kb-configuration** Makros wird festgelegt, welche dieser Interface-Mixins zur Konfiguration gehören sollen.

Die im Kapitel 10 vorgelegte Spezifikation der Benutzerschnittstelle legt nur das Protokoll und die Konventionen fest, die von den Standard-Prozessoren erwartet werden, läßt der Implementierung aber sonst weiten Spielraum. Das im folgenden beschriebene Minimalinterface wurde unter alleiniger Verwendung von Funktionen, die zum Common-Lisp Standard zählen, realisiert und wird hier nur als Ausgangspunkt für die Implementierung eigener Benutzerschnittstellen beschrieben.

11.2 Zerlegung

Der Minimalinterface besteht aus folgenden Teilen (siehe Abbildung 11.1):

1. **basic-menu-mixin**

Diese Komponente stellt die beiden Methoden zur Auswahl von Einträgen aus Menüs bereit.

2. **basic-interface-mixin**

Diese Komponente sorgt dafür, daß die Instanzvariablen für Fenster in den Prozessormixins gesetzt werden und stellt die in der Spezifikation genannten, nicht menüspezifischen Methoden zur Verfügung.

3. **basic-menu-item-handler**

Verwaltet Listen von Menüeinträgen.

4. **basic-menu-loop**

Diese Komponente präsentiert ein Menü von Operationen und führt die gewählten Operationen aus.

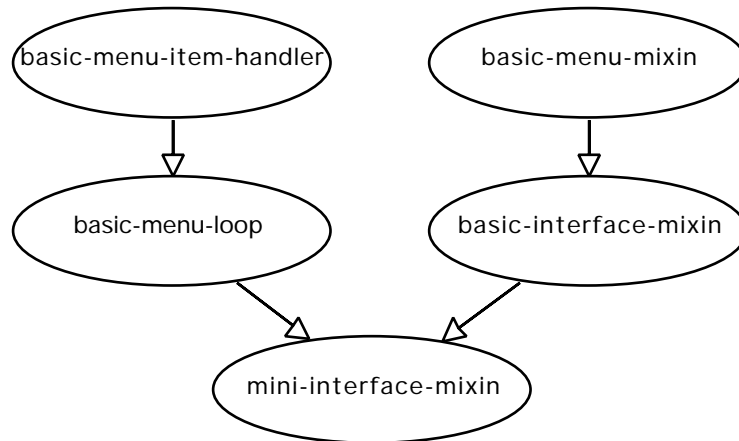


Abbildung 11.1: Architektur des Minimalinterfaces

5. mini-interface-mixin

Kombiniert **basic-interface-mixin** mit **basic-menu-loop** und beschafft die benötigten Einträge für das Operationsmenü.

11.3 Das Basisinterface

Das **basic-interface-mixin** stellt die von den Prozessoren benötigten Streams bzw. Fenster zur Verfügung und realisiert das erwünschte Protokoll. Als Dialogstream wird der an die globale Variable ***default-dialog-stream*** gebundene Stream verwendet. Defaultmäßig ist dies der an ***terminal-io*** gebundene Stream. Auf einer Lispmaschine ist diese Einstellung inadäquat. Evaluiert man nämlich eine Wissensbasis vom Editor aus, ist ***terminal-io*** an einen Background-Stream gebunden. Im Init-File für Babylon sollte man daher dafür sorgen, daß ***default-dialog-stream*** passend gesetzt, z.B. an einen Lisp Listener gebunden wird.

Die Zuweisung der Fenster erfolgt mit der Methode **:set-up-windows**. Sie generiert ein Pseudofenster, d.h. ein Objekt mit einer Instanzvariablen **Stream**. Stream wird mit dem Dialogstream besetzt. Dies Pseudofenster wird den Standardfenstern zugewiesen: **system-trace-window**, **rule-trace-window**, **prolog-trace-window** und **explanation-window**, sofern diese in der jeweiligen Konfiguration überhaupt vorkommen. **:set-up-windows** wird bei Instanziierung der Wissensbasis aufgerufen.

Sämtliche Ein/Ausgabe läuft beim Basisinterface über den Dialogstream. Bei den Methoden zur Auswahl von Einträgen aus Menüs sieht dies so aus, daß die einzelne Einträge, erweitert um die Nummer des Eintrags, über den Dialogstream ausgegeben werden, und die Auswahl durch Eingabe der Nummern erfolgt. Auf einer Lispmaschine ist dies Vorgehen völlig unangemessen, doch lassen sich beide einfach ersetzen durch Methoden, die die Funktionen **tv:menu-choose** bzw. **tv:multiple-choose** benutzen. Das Flavor **lisp-m-menu-mixin** bietet gerade derartige Realisierungen der Auswahlmethoden. Damit diese Methoden diejenigen des **basic-interface-mixin** ersetzen, muß es als zusätzliches Interface-Mixin vor **basic-interface-mixin** im **def-kb-configuration** Makro angegeben werden.

11.4 Das Minimalinterface

Das Minimalinterface präsentiert Operationen in einem Menü, das sich dynamisch ändert. Das Menü besteht aus zwei Zonen, die eine bleibt unverändert, die zweite zeigt wechselnde Einträge. Die Verwaltung der Menüeinträge wird von der Komponente **basic-menu-item-handler** bewerkstelligt.

Zur Verwaltung der Menüeinträge dient eine Assoziationsliste. Das Datum eines jeden Assoziationspaares stellt eine Liste von zusammengehörigen Menüeinträgen dar. Eine dieser Listen besteht aus dem fixen Teil des Menüs, sie ist mit dem Index **:top** verknüpft und wird im folgenden kurz als Hauptmenü bezeichnet. Die übrigen Listen umfassen in der Regel die für einen Formalismus vorgesehenen Operationen und sind mit Indizes wie **:frame**, **:rule**, **:prolog** assoziiert. Sie werden im folgenden als Untermenüs bezeichnet. Bei sehr vielen Operationen für eine Formalismus kann eine Aufsplittung in mehrere Untermenüs angebracht sein. So wurden die Prolog-Trace-Operationen in einem Untermenü **:prolog-trace** zusammengefaßt.

Angezeigt werden jeweils das Hauptmenü und maximal ein weiteres Untermenü. Die Methode **:open-menu** dient dazu, einen Wechsel im Untermenü auszulösen. Als Parameter ist der Index des gewünschten Untermenüs anzugeben. Ein Untermenü ist nur sinnvoll, falls im übergeordneten Menü (i.a. dem Hauptmenü) ein Eintrag auftritt, der das Untermenü eröffnet, d.h. **:open-menu** aufruft.

Für die Untermenüs wird ein (in der Tiefe beschränkter) Stack geführt. Die Methode **:close-menu** erlaubt es, zum letzten Zustand zurückzukehren. Das Hauptmenü enthält standardmäßig einen Eintrag (submenu-entry), der dies bewirkt.

Zum Aufbau der Menülisen stehen die folgenden Methoden zur Verfügung:

:add-operations *key operations &optional (mode :after)* **Methode**

Diese Methode fügt *operations*, i.e. eine Liste von Menüeinträgen, zu den Einträgen mit dem Index *key* hinzu und zwar am Ende oder Anfang der Liste, je nachdem ob für *mode* **:after** oder **:before** gewählt wurde.

:add-sub-operations *superkey expand-operation key operations*

&optional (mode :after)

Methode

Diese Methode baut aus *operations* ein Untermenü mit dem Index *key* auf. Zugleich wird der Menüeintrag *expand-operation* ins Menü mit dem Index *superkey* aufgenommen und zwar am Ende oder Anfang dieses Menüs, dem Wert von *mode* entsprechend. Es wird davon ausgegangen, daß *expand-operation* das fragliche Untermenü eröffnet, d.h. **:open-menu** *key* aufruft.

Zum Aufbau des Gesamtmenüs dient die Methode (**mini-interface-mixin** **:set-up-commands**). Sie baut das Hauptmenü auf und die Untermenüs für die Standard-Formalismen (Regeln, Frames, Prolog), die von der betreffenden Konfiguration unterstützt werden. Die Untermenüs richten sich dabei nach den Versionen der beteiligten Prozessoren. Das wird dadurch erreicht, daß die Prozessor-Mixin die Methoden bereitstellen, die die für sie typischen Untermenüs generieren. Sie heißen **:set-up-symb-rule-commands**, **:set-up-symb-frame-commands** bzw. **:set-up-symb-prolog-commands** und sind mithilfe von **add-sub-operations** realisiert. Die eigentlichen Menüeinträge sind in den diversen Kommando-Tabellen im Directory `cmd>` zu finden. (Dort werden auch die Methoden **:set-up-symb-rule-commands** usw. definiert.)

:set-up-commands wird bei Genenerierung einer Wissensbasis aufgerufen. Die Methode kann auch zu einem späteren Zeitpunkt aufgerufen werden. Allerdings sollten mit **:clear-menu** zuvor alle Menüeinträge gelöscht worden sein.

Die Präsentation des Menüs und die Ausführung von Operationen wird von der Komponente **basic-menu-loop** geleistet. Dabei besteht folgende Konvention: Wird in einem Menüeintrag als Operation **:value** angegeben und ist der fragliche Wert ein Schlüsselwort, so wird der Wert als Selektor für die Wissensbasis aufgefaßt und entsprechend weitergereicht. Anstelle eines Schlüsselworts kann eine Liste übergeben werden, deren erstes Element ein Schlüsselwort ist. Dieses wird wieder als Selektor interpretiert, die übrigen Elemente als Parameter. Sie werden allerdings **nicht** evaluiert. Bei den Selektoren wird mittels **:operation-handled-p** geprüft, ob der Selektor bekannt ist.

Das Menü wird in einer Schleife präsentiert. Sie wird verlassen, wenn nach Auswahl eines Eintrags als Wert das Schlüsselwort **:exit** zurückgegeben wird. Gestartet wird die Loop durch die Methode **:run**, die von **:select-kb** aufgerufen wird.

Literaturverzeichnis

- [Arb104] Müller, B.S.; Pless, E.; Richter, G., *Wissensrepräsentation durch strukturierte Objekte*. In: Bungers/Müller/Raulefs (Hrsg.) *Expertensysteme*, Bd. 1, Arbeitspapiere der GMD Nr. 104, St. Augustin, 1984, 148-197.
- [Arb124] Brewka, G.; di Primio, F.; Müller, B.S. (Hrsg.) *Materialien zu einigen Werkzeugsystemen zur Erstellung von Expertensystemen: Wissensrepräsentation*. Arbeitspapiere der GMD Nr. 124, St. Augustin, 1984.
- [BobSte83] Bobrow, D. G.; Stefik, M. *The LOOPS Manual*. XEROX PARC, Palo Alto, 1983.
- [BraLev82] Brachman, R.J.; Levesque, A.J. *Competence in Knowledge Representation*. In: Proc. of AAAI-82, 189-192.
- [Byl83] Bylander, T.; Mittal, S.; Chandrasekaran, B. *CSRL: A Language for Expert Systems for Diagnosis*. In: Proc. of IJCAI-83, 218-221.
- [Chr86] Christaller, Th. *KI-Programmiertechniken*. In: Richter, M. (Hrsg.) KIFS-86, Springer, Berlin, erscheint demnächst.
- [CloMe] Clocksin, W.F.; Mellish, C.S. *Programming in Prolog*. Springer, Berlin, 2. Aufl., 1984.
- [EngSta84] Engelmann, C.; Stanton, W.M. *An Integrated Frame/Rule Architecture*. In: Elithorn, A.; Banerji (Hrsg.) *Artificial and Human Intelligence*, North-Holland, Amsterdam, 1984, 141-146.
- [ErmLes75] Erman, L.D.; Lesser, V.R. *A multi-level organization for problem solving using many diverse, cooperating sources of knowledge*. In: Proc. of IJCAI-75, 483-490.
- [Hay80] Hayes, P.J. *The Logic of Frames*. In: Metzing, D. (Hrsg.) *Frame Conception and Text Understanding*. Berlin, de Gruyter, 1980, 46-61.
- [KEE83] *Kee User's Manual*. Menlo Park, CA, IntelliGenetics, 1983.
- [Kifs85] Christaller, Th. *LISP-Einführung*. In: Hein, H. W. (Hrsg.) KIFS-85, Springer, Berlin, erscheint demnächst.
- [Kla85] Klar, W.; Wittur, K.-H.; Henne, P. *Ein Expertensystem zur Fehlerdiagnose im automatischen Getriebe C 3 von Ford*. In: Brauer, W.; Radig, B. (Hrsg.) *Wissensbasierte Systeme*. Springer, Berlin, 1985, 105-120.

- [Kun84] Kunz, J.C.; Kehler, T.P.; Williams, M.D. *Applications Development Using a Hybrid AI Development System*. In: The AI Magazine V:3, 1984, 41-54.
- [LMM] Symbolics Inc. *Lisp Maschine Manual*. Cambridge (MA), 1984.
- [vanMel80] van Melle, W. *A domain independent system that aids in constructing consultation programs*. Rep. No. STAN-CS-80-820, Computer Science Dept., Stanford University, 1980.
- [Nov83] Novak, G.S. *Knowledge-based Programming Using Abstract Data Types*. In: Proc. AAAI, 1983, 288-291.
- [ObjHSArt] Stoyan, H.; Wedekind, H. (Hrsg.) *Objektorientierte Software- und Hardwarearchitekturen*. Berichte des German Chapter of the ACM, Band 15, Teubner Verlag, Stuttgart, 1983.
- [PriBre85] di Primio, F.; Brewka, G. *BABYLON: Kernel System of an Integrated Environment for Expert System Development and Operation*. In: Proc. of Fifth International Workshop on Expert Systems and their Applications, Avignon, 1985, 573-583.
- [Pri85] di Primio, F. *Bootstrapping in BABYLON: Aufbau und Struktur einer Wissensbasis für die Top-Down-Refine-Strategie*. GMD, St. Augustin, erscheint demnächst.
- [Pup83] Puppe, B.; Puppe, F. *Overview on MED1: a Heuristic Diagnostics System with an Efficient Control Structure*. In: Neumann, B. (Hrsg.) GWAI-83, Springer, Berlin, 1983, 11-22.
- [RefHB] Forschungsgruppe Expertensysteme *BABYLON Referenzhandbuch V 0.0/1*. Gesellschaft für Mathematik und Datenverarbeitung, Institut für Angewandte Informationstechnik, St. Augustin, 1985.
- [Rei81] Reingold, E.M.; Tilford, J.S. *Tidier Drawings of Trees* In: IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, March 1981
- [Rome87] Rome, E. *Tree Editor: Benutzer- und Programmiererhandbuch* Reihe GMD-Arbeitspapiere, Nr. 255, GMD, St. Augustin 1987
- [Sho76] Shortliffe, E.H. *Computer-based medical consultations: MYCIN*. American Elsevier, New York, 1976.
- [Ste84] Steele, G.L. *CommonLisp: The Language*. Digital Press, Burlington, 1984.
- [Sup83] Supowit, K.J.; Reingold, E.M. *The Complexity of Drawing Trees Nicely* In: Acta Informatica 18, pp. 377-392, 1983
- [Sys85] di Primio, F.; Bungers, D., Christaller, T. *BABYLON als Werkzeug zum Aufbau von Expertensystemen*. In: Brauer, W.; Radig, B. (Hrsg.) Wissensbasierte Systeme. Springer, Berlin, 1985, 70-79.
- [S1] *S.1 Reference Manual*. Framentec, Monaco, 1984.

- [Tic86] Tichy, W.F.; Ward, B. *A Knowledge-Based Graphical Editor* Submitted to ACM Transactions on Graphics, 1986
- [Tut63] Tutte, W.T. *How to Draw a Graph* In: Proc. of London Math. Soc. (3), 13, pp. 743-768, 1963
- [Vau80] Vaucher, J.G. *Pretty Printings of Trees* In: Software-Practice and Experience, Vol. 10, pp. 553-561, 1980
- [Wet79] Wetherell, C.; Shannon, A. *Tidy Drawings of Trees* In: IEEE Transactions on Software Engineering, Vol. SE-5, No. 5,