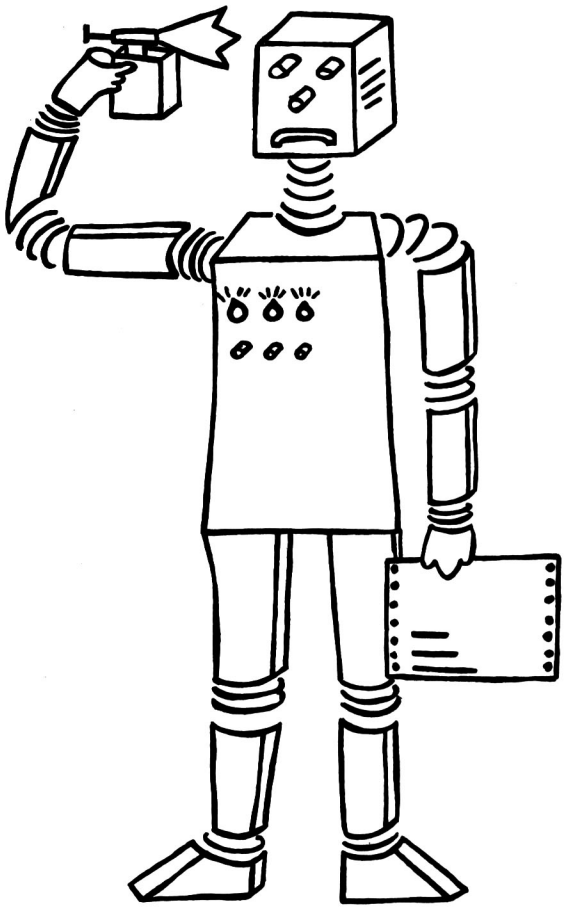
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

SEKI-Working Paper



ARI - Entwicklung von Inferenzkomponenten für wissensbasierte Systeme auf der Grundlage strukt. Produktionsregelsysteme

Winfrid Barth

July 1988 SEKI Working Paper SWP-88-05

ARI

Entwicklung von Inferenzkomponenten für wissensbasierte Systeme auf der Grundlage strukturierter Produktionsregelsysteme

Winfried Barth

Fachbereich Informatik
Universität Kaiserslautern
Mai 1988

Zusammenfassung

*Gegenstand der Arbeit ist die Konzeption und Implementierung von **ARI**, einem praktisch einsetzbaren Werkzeug zur Entwicklung von Inferenzkomponenten für wissensbasierte Systeme. Der für **ARI** gewählte Ansatz integriert einen Produktionsregelformalismus mit Repräsentationsmechanismen auf der Basis objektzentrierter, strukturierter Vererbungsnetzwerke. Zusätzlich werden Wissensstrukturen bereitgestellt, welche die Strukturierung und modulare Organisation von Regelbasen in funktionale Blöcke ("problem-solver") sowie die einheitliche Spezifikation verschiedener Kontrollmechanismen erlauben.*

*Das Werkzeugsystem **ARI** wird auf drei verschiedenen Abstraktionsebenen beschrieben. Auf der epistemologischen Ebene können für jeden "problem-solver" geeignete Inferenz- und Kontrollmechanismen spezifiziert werden. Als funktionale Einheiten betrachtet, die gemäß dieser Spezifikation die generische Klasse eines Problemtyps lösen, können "problem-solver" auf einer konzeptionellen Ebene als aufgabenspezifische Problemlöser aufgefaßt werden. Auf der Implementierungsebene werden "problem-solver", Inferenz- und Kontrollwissen effizient durch einen erweiterten RETE pattern-matching Algorithmus interpretiert.*

Abstract

*The paper describes basic concepts and the implementation of **ARI**, a practical tool for the construction of inference engines for knowledge-based systems. **ARI** provides a representation mechanism based on production rules and object-centered, structured inheritance networks. Additionally, **ARI** offers knowledge structures for organizing rulebases modularly on the basis of functional units ("problem-solvers") and for specifying various control regimes.*

***ARI** is presented at three different levels of abstraction. At the epistemological level, appropriate inference and control mechanisms can be specified for each "problem-solver". Viewed as functional units that solve a generic class of problems according to their specification, "problem-solvers" are task-specific problem-solving agents at a conceptual level. At the implementational level "problem-solvers", inferential knowledge as well as control knowledge are interpreted efficiently with an extended RETE pattern-matching algorithm.*

Inhaltsverzeichnis

Verzeichnis der Abbildungen und Algorithmen

1. Einleitung

1.1. Überblick über die Arbeit

Gegenstand der vorliegenden Arbeit ist die Konzeption und Implementierung von ***ARI*** (***A***dvanced ***R***ule ***I***nterpreter), einem Werkzeug zur Entwicklung von Inferenzkomponenten für wissensbasierte Systeme. ***ARI*** umfaßt eine *erweiterte Produktionsregelsprache*, einen *Interpretierer* (*Inferenzmaschine*) für diese Sprache, sowie eine *Methodik*, die Richtlinien zur Verwendung der Sprache bei der Entwicklung komplexer wissensbasierter Systeme bereitstellt.

Die Regelsprache basiert auf Konzepten von **OPS5** (s. /Brownston et al. 85/), integriert darüber hinaus einen Repräsentationsmechanismus auf der Grundlage strukturierter Vererbungsnetze mit speziellen Inferenztechniken (*Vererbung, taxonomische Klassifikation*, etc.), stellt mächtige Konzepte zur Strukturierung und Organisation von Regelbasen zur Verfügung und unterstützt die Spezifikation von Kontrollwissen in der Form einer *Metaebenenarchitektur*. Die Inferenzmaschine stützt sich auf einen modifizierten und erweiterten RETE-Algorithmus (s. /Forgy 82/, /Beetz 87/, /Beetz,Barth 88/), der eine sehr effiziente Interpretation der Regeln erlaubt. Methodisch präsentiert sich ***ARI*** auf drei verschiedenen Abstraktionsebenen, die unterschiedliche Sichten auf das System erlauben und verschiedene Aspekte des Systems betonen.

Das ***ARI***-System ist eine Erweiterung des von M. Beetz konzipierten Regelinterpretierers CATWEAZLE /Beetz 87/ und als Komponente in das LUIGI-Werkzeugsystem zur Entwicklung wissensbasierter Systeme im Bereich der Büroautomation integriert /Lutze 88/. ***ARI*** erweitert die *Regelkomplexe* aus CATWEAZLE um die Möglichkeit hierarchischer Strukturierung und betont für sie eine Sichtweise als *funktionale Einheiten* mit *abstrakter Beschreibung* sowie expliziter Spezifikation der *Kommunikation* zwischen ihnen. Die Integration strukturierter Objekte zur Repräsentation beschreibenden Wissens ist ebenfalls eine Erweiterung gegenüber CATWEAZLE.

Die im Rahmen der Diplomarbeit erstellte Implementierung ist der vollständige Kern einer komplexen Inferenzkomponente und geht damit über die prototypische Implementierung eines speziellen Inferenzmechanismus hinaus. Trotzdem wird keine vollständige Implementierungsbeschreibung präsentiert, sondern es werden vielmehr die Grundlagen und Methoden von ***ARI*** erläutert und ihre Realisierung als Softwaresystem aufgezeigt.

Im folgenden werden allgemeine Grundlagen wissensbasierter Systeme diskutiert und die wichtigsten Wissensrepräsentations- und Inferenzmechanismen charakterisiert. Anschließend werden Eigenschaften wissensbasierter Anwendungen im Bürobereich aufgezeigt und die daraus resultierenden Anforderungen

an Repräsentations- und Inferenzsysteme erläutert. In Kapitel 2 werden die Grundlagen des CATWEAZLE Regelinterpretierers kurz zusammengefaßt und anschließend die Methodik von *ARI* vorgestellt. Kapitel 3 präsentiert die Konzepte von *ARI* in detaillierter Form. In Kapitel 4 wird die Implementierung des *ARI*-Systems beschrieben und im abschließenden Kapitel 5 werden mögliche Weiterentwicklungen vorgeschlagen.

1.2. Wissensbasierte Systeme

Wissensbasierte Systeme* (***knowledge-based systems***) haben sich zu einer vielversprechenden Softwaretechnologie für eine Reihe komplexer, spezialisierter Anwendungsbereiche entwickelt (vgl. /Hayes-Roth 86/, /Johnson 84/). Wissensbasierte Systeme sind unter anderem Realisierung und Anwendung von Prinzipien, die im Laufe der letzten beiden Jahrzehnte auf dem Gebiet der **Wissensrepräsentation**, einem Forschungsschwerpunkt der **Künstlichen Intelligenz** (KI), entwickelt wurden. Obwohl dieses Gebiet eine Vielzahl zum Teil sehr unterschiedlicher Theorien hervorgebracht hat (vgl. /Brachman,Smith 80/, /Brachman,Levesque 85/), ist man sich weitestgehend einig, daß das Problem der Repräsentation von Wissen eine gemeinsame, zentrale Fragestellung für die verschiedenen Teilgebiete der Künstlichen Intelligenz ist (vgl. /Levesque 86/, /Delgrande,Mylopoulos-87/).

Bei der Definition des Begriffs "wissensbasiertes System" wiederum sind die Ansätze durchaus unterschiedlich. Die Charakterisierung als Softwaresystem, das menschliches Wissen anwendet, um Probleme zu lösen, die normalerweise menschliche Intelligenz erfordern (vgl. /Hayes-Roth 84/, /Shapiro 87/), reicht sicherlich nicht aus, um das Wesen dieser Systeme aufzuzeigen. Im folgenden soll eine Charakterisierung wissensbasierter Systeme vor dem Hintergrund der Wissensrepräsentationsforschung präsentiert werden, welche die Grundlage und Rahmen für die vorliegende Arbeit bildete.

Als Ausgangspunkt mag hier die **knowledge-representation hypothesis** dienen, wie sie von B. Smith formuliert wurde (s. /Smith 82/):

> *Any mechanically embodied intelligent process will be comprised of stuctural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.*

Vom Standpunkt des Logikers stellt dies genau die Forderung nach einer formalen *logischen Sprache* mit (a) einer *Modelltheorie* und (b) einer *Beweistheorie* dar. Man kann aber auch allgemein die zentrale Arbeitshypothese eines Großteils der bisherigen KI-Forschung erkennen und es ergibt sich direkt daraus die dominierende Architektur wissensbasierter Systeme. Ein solches System besteht demnach zum einen aus einer Ansammlung symbolischer Strukturen, der **Wissensbasis**, und einer Systemkomponente, die diese Strukturen syntaktisch manipuliert, oft einfach als **Inferenzmaschine** bezeichnet. Eine vorgegebene Interpretation der symbolischen Strukturen stellt dabei die Korrespondenz zum Wissen dar, das sie repräsentieren sollen.

Eine Wissensbasis stellt also im wesentlichen ein symbolisches Modell eines bestimmten, abgeschlossenen Weltausschnittes dar. Die Abgeschlossenheit dieser **Diskurswelt** ist dabei ein wichtiger Faktor was den Erfolg gegenwärtiger wissensbasierter Systeme ausmacht, denn Problemlösungsprozesse,

---

* der Begriff wird dem etwas gebräuchlicheren des ***Expertensystems*** vorgezogen, da in ihm die Schlüsselrolle von *Wissen* betont wird. Der Begriff des *Expertensystems* hingegen unterlag in den letzten Jahren einer immer stärkeren *Trivialisierung* und wird, überspitzt formuliert, immer mehr zum Synonym für *Computerprogramm* (vgl. /Steels 87/).

die allgemeines Weltwissen (***common sense knowledge***) notwendig machen, lassen sich mit derzeitigen Mitteln der Wissensrepräsentation nur unzureichend modellieren (vgl. /McCarthy,Hayes 69/, /McCarthy 87/). Zum anderen ist die Form des Modells als Menge symbolischer Strukturen im Gegensatz zu einem numerischen, mathematischen Modell von entscheidender Bedeutung für wissensbasierte Systeme. Selbst dann, wenn solche mathematischen Modelle existieren, sind sie unter Umständen unzureichend für gewisse Aufgabenbereiche, da ihre Simulationsergebnisse, im allgemeinen eine Reihe numerischer Werte von Zustandsvariablen, zusätzlicher und oft entscheidender Interpretation durch die Erfahrung und Intuition und somit wiederum Wissen eines Experten bedürfen. Dies impliziert, daß wissensbasierte Systeme an gewissen Punkten ihres diskreten, qualitativen Schlußfolgerungsprozesses durchaus auf spezialisierte mathematische Modelle in Form von Gleichungen, Tabellen, Algorithmen, etc. zurückgreifen können, aber solche formalen Modelle werden dann unter der Kontrolle eines übergeordneten symbolischen Inferenzprozesses eingesetzt (vgl. /Chandrasekaran 84/).

Ob symbolische Strukturen als Implementierungsgrundlage menschlichen Problemlösens im Sinne eines **physical symbol system** dienen können (s. /Newell 80/) oder aber dafür primitivere, "*subsymbolische*" Ebenen nötig sind (vgl. /Feldman 85/, /Smolensky 87/), ist aktueller Forschungsgegenstand und soll hier nicht weiter erörtert werden.

Das symbolische Modell der Diskurswelt eines wissensbasierten Systems ist in der Regel nur zum Teil **explizit** in der Wissensbasis des Systems enthalten. Ein Großteil des Modells wird nach Bedarf, d.h. zum Lösen eines Problems innerhalb des Anwendungsbereiches, von der Inferenzkomponente hergeleitet. Die Wissensbasis muß also neben **beschreibendem Wissen** - darunter sollen Repräsentationen für nicht näher spezifizierte Phänomene oder "*Entitäten*" der Diskurswelt sowie Aussagen und Sachverhalte über diese verstanden werden - auch Repräsentationen von **Schlußfolgerungsprinzipien** enthalten, die das Herleiten des **impliziten** Wissens aus dem tatsächlich repräsentierten expliziten Teil ermöglichen.

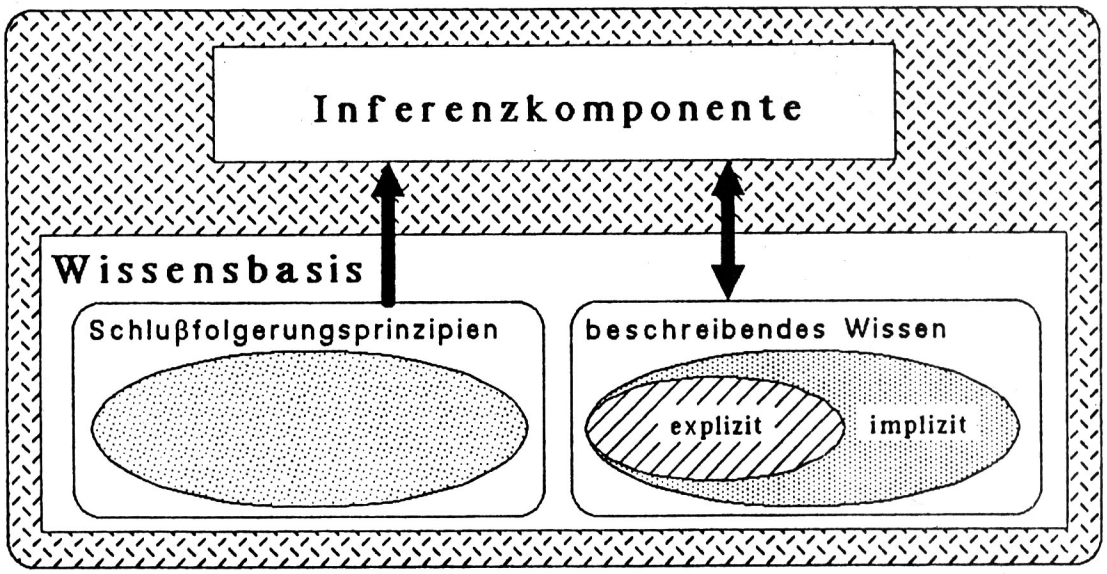


Abb. 1.1: Ein allgemeines wissensbasiertes System

Die Unterscheidung zwischen beschreibendem Wissen und Schlußfolgerungsprinzipien darf hier nicht als scharfe Trennlinie verstanden werden. Auch Schlußfolgerungsprinzipien lassen sich als Beschreibungen auffassen , die selbst Gegenstand eines Inferenzprozesses sein können (*Metaebenenkonzept*, s.u.) und Mechanismen zur Repräsentation beschreibenden Wissens können unmittelbar selbst

Inferenzmöglichkeiten besitzen (z.B. *Vererbung*, s.u.). Die Denotation eines Phänomens der Welt in Form einer Datenstruktur in der Wissensbasis wird im folgenden als **Objekt** bezeichnet, wobei der Objektbegriff hier entsprechend weit zu sehen ist, also sowohl *konkrete Dinge* (Gegenstände), als auch *abstrakte Konzepte* wie Ziele, Motive u.ä. umfaßt. Als **Sachverhalte** werden alle repräsentierten Aussagen über Relationen zwischen Objekten bezeichnet.

Die Problematik des Problemlösens durch Anwendung von Schlußfolgerungsprinzipien liegt vor allem darin begründet, daß das implizite Wissen i.a. viel zu umfangreich ist, als daß man durch systematische Konstruktion aller Ableitungen, also durch vollständige Suche, eine bestimmte Problemlösung herleiten könnte. Dieses Problem der *"kombinatorischen Explosion"* des Suchraums versuchte man in der ersten Phase der KI-Forschung durch sehr allgemeine, mit *Heuristiken* gesteuerte Suchverfahren zu lösen (vgl. /Newell,Simon 72/). Mittlerweile dominiert jedoch der wissensbasierte Ansatz, der davon ausgeht, daß eine große Menge von anwendungsspezifischem Wissen notwendig ist, um in einer Diskurswelt effektiv Probleme lösen zu können (vgl. /Feigenbaum 77/). Dieses spezifische, explizit repräsentierte Wissen wird dabei zum Steuern des Suchprozesses eingesetzt und kann im Idealfall auch sehr große Suchräume beherrschbar machen. Mit diesem Ansatz ist das **Suchproblem** jedoch keineswegs gelöst. Obwohl mehr Wissen weniger Suche auf der Ebene des Problems bedeuten kann, stellt sich für ein wissensbasiertes System zu jedem Zeitpunkt des Inferenzprozesses das Problem, welches Wissen am besten anzuwenden ist und wie dieses Wissen in der Fülle des Gesamtwissens zu finden ist. Das Suchproblem wird zum **Kontrollproblem.**

Aus dem bisher Gesagten ergeben sich die beiden folgenden Kernfragen der Wissensrepräsentationsforschung :

(1) Welche Arten von Wissen (beschreibendes Wissen, Schlußfolgerungsprinzipien) gibt es in einer Anwendungswelt und welche sind relevant für effektives Problemlösen in dieser Welt?

(2) Wie muß ein Formalismus, d.h. eine formale Sprache zur Repräsentation von Wissen aussehen, die eine adäquate Modellierung einer Diskurswelt erlaubt, und wie kann eine Inferenzkomponente diesen Formalismus zu effektivem Problemlösen nutzen?

Die vorgenommene Unterscheidung entspricht im wesentlichen den beiden Sichtweisen, die A. Newell für Wissensbasen vorgeschlagen hat (s. /Newell 82/): der **Wissensebene** (***knowledge level***), die lediglich den *Informationsgehalt* einer Wissensbasis betrachtet, also das, was sie über die Welt aussagt, die sie modelliert, und der **Symbolebene** (***symbol level***)*, die sich in der konkreten Darstellung von Wissen in Form von Datenstrukturen und Algorithmen manifestiert. Die Idee des *knowledge levels* als zentralem Ansatzpunkt für die Analyse wissensbasierter Systeme wurde in der Zwischenzeit von mehreren Forschern aufgegriffen und verfeinert (vgl. /Brachman,Levesque 86/, /Fox 86/). Als Konsequenz daraus sollten Betrachtungen von Wissensrepräsentationsmechanismen, die vorwiegend auf dem *symbol level* angesiedelt sind, stets in formalem Bezug zum *knowledge level* stehen, also zum Wissen, das sie repräsentieren können bzw. nicht können.

Formen von Wissen

Die erste der obigen Kernfragen, korrespondierend zum *knowledge level*, zeigt die Querbezüge der KI-Forschung u.a. zur Philosophie und kognitiven Psychologie auf, denn eine Erörterung über die *Natur von Wissen* kann sicherlich nicht erschöpfend innerhalb der KI abgehandelt werden. Für die vorliegende Arbeit soll auf diese Frage nur kurz eingegangen werden und auch nur mit Bezug zu Prinzipien der Wissensrepräsentation (vgl. /Delgrande, Mylopoulos 87/).

* im folgenden werden die englischen Originalbegriffe verwendet.

Unterschieden werden müssen sicherlich folgende Formen von Wissen:

- **gesichertes Wissen (*knowledge per se*)**
  ist Wissen, dessen Wahrheitsgehalt außer Zweifel steht (*"Alle Säugetiere sind sterblich."*).
- **"geglaubtes" Wissen (*belief*)**
  ist Wissen, für das gute Rechtfertigungsgründe sprechen, das sich aber in Form von Annahmen (***assumptions***) durchaus als falsch erweisen oder aus Mangel an Information nie als gesichert etabliert werden kann.
- **Hypothesen**
  sind aus Annahmen hergeleitete Schlußfolgerungen (*"Positronen bestehen aus Quarks."*).

Diese Unterscheidung betont den *Wahrheitsgehalt* von Aussagen*, ebenso können Aussagen nach ihrer *Aussagekraft* differenziert werden. **Empirisches Wissen** z.B. ("75% aller Zivilisationskrankheiten sind ernährungsbedingt!"), **ungenaues Wissen** ("Heinrich oder Friedrich ist ein Bruder von Richard."), bzw. **unscharfes (*vages*) Wissen** ("James ist ziemlich groß.") sind Erscheinungsformen von Wissen, die dazu beitragen, daß das Gesamtwissen eines Menschen oder aber eines intelligenten Computersystems immer in hohem Grade unvollständig sein wird. Für die Behandlung **unvollständigen Wissens** wurden verschiedene Theorien entwickelt (s. /Levesque 81/, /Moore 82/). Für die Behandlung empirischen Wissens seien hier die Dempster-Shafer Theorie (s. /Dempster 67/, /Shafer 76/), das Evidenzkalkül in **MYCIN** /Shortliffe,Buchanan 75/, die Anwendung des Theorems von Bayes in **PROSPECTOR** /Duda et al. 76/ und für vages Wissen die Theorie der Fuzzy-Logik /Zadeh 78/ genannt.

Auch für die Schlußfolgerungsprinzipien gibt es durchaus unterschiedliche Ansatzpunkte. Die sicherlich wichtigste Methode des Schlußfolgerns ist die logisch korrekte **Deduktion** auf der Grundlage des *Modus Ponens*. Andere Prinzipien wie die der **Abduktion** oder **Induktion** spielen jedoch beim Menschen ebenso eine signifikante Rolle, obwohl sie logisch gesehen zu nicht korrekten Schlüssen führen können. Zusätzlich können Schlußfolgerungsprinzipien ebenfalls mit Unsicherheiten und Evidenzen behaftet sein. Vor allem dann, wenn nicht nur das Ergebnis eines Schlußfolgerungsprozesses von Interesse ist, sondern auch das Vorgehen eines wissensbasierten Systems möglichst dem eines menschlichen Experten entsprechen soll - im folgenden auch als *Natürlichkeit des Schließens* bezeichnet - müssen für all diese Schlußfolgerungsprinzipien geeignete Repräsentationen und Verarbeitungs- (Interpretations-) methoden gefunden werden. Das Kontrollproblem ist auch auf dieser abstrakten Ebene sichtbar und relevant, da Wissen über die Anwendbarkeit von Schlußfolgerungsprinzipien durchaus eine wichtige Form von Wissen darstellen kann.

Wissensrepräsentationsformalismen

Die zweite Kernfrage nach einem Mechanismus zur Wissensrepräsentation, der es erlaubt, Wissen über eine Diskurswelt zu formalisieren und damit effektiver Berechnung innerhalb eines Computers zugänglich macht, ist der Schwerpunkt dieser Arbeit. In der Vergangenheit wurden eine Reihe solcher formalen Repräsentationssprachen zusammen mit entsprechenden Inferenzmechanismen entwickelt und viele davon sind bereits in kommerziell verfügbare Softwarewerkzeuge zur Erstellung wissensbasierter Systeme eingeflossen.

Die Betonung des *knowlege levels* verlangt von einer Wissensrepräsentationssprache, daß sie mit einer reichhaltigen "semantischen Theorie" versehen ist, die die Korrespondenz des Modells zur realen Welt formalisiert. Bei Datenbanksystemen, wo diese Korrespondenz offensichtlich ist - Datenbanken

* es wird hier eine Wahrheitssemantik im Sinne von Tarski zugrundegelegt /Tarski 56/.

lassen sich logisch als indizierte Menge von *Grundatomen* betrachten - liegt der Schwerpunkt auf "effizienter Berechnung", d.h. Realisierung eines einfachen Datenmodells mit sehr großen Datenmengen auf physikalisch vorgegebenen Maschinen (vgl. /Brodie,Mylopoulos 86a/). Da wissensbasierte Systeme immer mehr aus den Forschungslabors in konkrete Anwendungsumgebungen drängen, sind Gesichtspunkte effizienter Realisierung auch für sie von entscheidender Bedeutung. Die folgenden Kriterien sind deshalb zur Beurteilung eines Wissensrepräsentationsmechanismus relevant:

- **Ausdrucksstärke** (***expressive power***, ***epistemological adequacy***, vgl. /McCarthy,Hayes 69/):
  welche Formen von Wissen der Diskurswelt können repräsentiert werden und bis zu welchem Grad von Unvollständigkeit kann Wissen repräsentiert werden?
- **Natürlichkeit** und **Verständlichkeit** des Formalismus:
  ist eine direkte Korrespondenz zwischen (symbolischem) Modell und modellierter Diskurswelt erkennbar? Werden Schlußfolgerungen so gezogen, wie ein Mensch sie ziehen würde (***kognitive Adäquatheit***)?
- **Änderungsfreundlichkeit, Wartbarkeit** (***maintainability***):
  kann das Wissen in einer modularen Organisation repräsentiert werden, welche die Auswirkungen von Änderungen und Ergänzungen der Wissensbasis lokal begrenzt und überschaubar hält?
- **Entscheidbarkeit**:
  kann für jede Frage (theoretisch) ein Wahrheitswert als Antwort berechnet werden?
- **Komplexität** (***computational tractability***) von Retrieval- und Inferenzalgorithmen:
  können Antworten auf Anfragen (Problembeschreibungen) in vernünftiger Zeit berechnet werden?

Zwischen *expressive power* und *computational tractability* sieht H. Levesque einen fundamentalen Gegensatz (***fundamental tradeoff***) (s. /Levesque,Brachman 87/): je ausdrucksstärker ein Formalismus ist, desto stärker können Fragen der Berechenbarkeit und Komplexität ihn für den praktischen Einsatz disqualifizieren. H. Levesque ordnet die Vielzahl von Repräsentationsformalismen entlang dieser Skala, an deren Extrempunkten er ausdrucksstarke Logiken (z.B. Prädikatenkalkül erster Stufe) auf der einen und komplexitätsmäßig beherrschte Datenbankformalismen (ohne jegliche Inferenzmöglichkeiten) auf der anderen sieht (s. /Levesque 86/).

Diese Betrachtungen auf dem *knowledege level* müssen auf dem *symbol level* durch Fragen nach der **Organisation** und **Strukturierung** von Wissen ergänzt werden, da sie für wissensbasierte Systeme eine ähnliche Bedeutung haben, wie etwa *strukturierte Programmierung* und *abstrakte Datentypen* für Programmiersprachen. Im folgenden sollen die wichtigsten Repräsentationsformalismen kurz charakterisiert werden.

Logische Wissensrepräsentationsformalismen

Mathematische Logik, insbesondere der Prädikatenkalkül erster Stufe, war der erste Formalismus, der im Bereich der Künstlichen Intelligenz zur Wissensrepräsentation vorgeschlagen wurde (s. /McCarthy 68/). Seither wurden viele logische Formalismen entwickelt. Eine Logik besteht aus einer **formalen Sprache**, aufgebaut aus Konstanten-, Variablen-, Funktions- und Prädikatensymbolen, logischen Verknüpfungssymbolen (Junktoren) und Quantoren. Mit Hilfe des *Interpretationsbegriffes* werden *logische Konsequenz* und damit eine *Modelltheorie* entwickelt, die die **Semantik** der *Formeln* einer logischen Sprache festlegt. Auf der anderen Seite läßt sich mit dem Begriff der *Ableitbarkeit* eine *Beweistheorie* aufbauen, die rein syntaktische Manipulationen der Formeln formalisiert. Durch *Korrektheits-* und *Vollständigkeitseigenschaften* wird eine Verbindung zwischen Modelltheorie und Beweistheorie hergestellt, die für die Prädikatenlogik erster Stufe z.B. sicherstellt, daß alle aus einer

Menge logischer Formeln syntaktisch ableitbaren Formeln genau die logischen Konsequenzen dieser Formeln darstellen*. Diese formale, theoretisch sehr gut verstandene Semantik zusammen mit der Flexibilität und Einfachheit der formalen Sprache machen logische Formalismen auf den ersten Blick zu einem idealen Kandidaten für Repräsentationsmechanismen: eine Wissensbasis ist dann eine Menge logischer Formeln, die gemäß einer *intendierten Interpretation* wahr sind. Inferenzen werden syntaktisch gemäß dem Ableitungsbegriff der Beweistheorie berechnet, die Inferenzkomponente wird durch einen *Theorembeweiser* realisiert. Eine *Anfrage* an ein solches Repräsentationssystem in Form einer logischen Formel wird also durch Auffinden bzw. Konstruktion eines Beweises für die Anfrage beantwortet.

Leider gibt es schwerwiegende Probleme mit dieser Art von Repräsentationssystem. Für die Prädikatenlogik erster Stufe kann ein solcher Mechanismus wegen der Unentscheidbarkeit dieses Kalküls überhaupt nicht existieren. Ebenso problematisch ist die Tatsache, daß bereits das Entscheidungsproblem für die relativ ausdrucksschwache Aussagenlogik komplexitätstheoretisch **NP-vollständig** ist, und es damit (fast) ausgeschlossen ist, daß ein Algorithmus existiert, der dieses Problem in seiner Allgemeinheit und mit einer naiven Anwendung des Ableitbarkeitsbegriffes in akzeptabler Zeit lösen kann (***computational intractability***). Natürlich ist dies nur eine *worst-case* Betrachtung und man mag argumentieren, daß dies ein Spezialfall ist, der in der Praxis sehr selten, wenn überhaupt, auftritt. Den Gegenbeweis aus der Praxis, zumindest für die Prädikatenlogik erster Stufe, liefern aber hier R. Brachman und H. Levesque, die für das **KRYPTON**-System eingestehen (s. /Brachman,Levesque 87/, p. 36):

> *One of the things that we found was that taking on a state-of-the-art first-order logic theorem prover was somewhat of a mistake. We immediately bought into undecidability: we could ask questions of our system where the TBox would return very quickly, but the ABox might go off and never return.*

Es gibt aber noch weitere gravierende Probleme mit logischen Repräsentationsmechanismen. Klassische Logiken haben sich als nicht adäquat für die Repräsentation einer Vielzahl von Phänomenen von Wissen erwiesen. So wurden spezielle Formalismen zum Umgang mit der **Nichtmonotonie** menschlichen Schließens (s. z.B. /McDermott,Doyle 78/), mit **beliefs** (s. /Moore 77/), oder **defaults** (/Reiter 80/) entwickelt, die allerdings z.T. keine Korrektheits- und/oder Vollständigkeitseigenschaften mehr aufweisen. Die Formulierung formaler Semantiken für diese Formalismen ist also weitaus problematischer als im klassischen Fall und ist derzeitiger Forschungsgegenstand.

Ein weiteres Problem der Anwendung logischer Formalismen ist das Fehlen jeglicher **Organisations-** und **Strukturierungsmechanismen** für die Formeln in einer Wissensbasis. Eine unstrukturierte Ansammlung logischer Formeln wird im allgemeinen genausowenig handhabbar sein, wie ein Programm, das in einer Programmiersprache ohne Abstraktionsmechanismen geschrieben ist.

Ein zusätzliches Problem mit rein logischen Mechanismen ist die Schwierigkeit der Darstellung von Kontrollwissen. Die Notwendigkeit der Spezifikation von Kontrolle bei der Verwendung von Theorembeweisern ergibt sich klar aus obigen komplexitätstheoretischen Resultaten. Das Problem liegt darin, daß in einer logischen Sprache keine Unterscheidung von Objekt- und Kontrollwissen auf syntaktischer Ebene möglich ist und daß damit der Theorembeweiser nicht explizit zwischen beiden Arten von Wissen unterscheiden kann. Rein syntaktische Kontrollmechanismen, die etwa als spezielle Resolutionsstrategien fest in einem Beweiser "*verdrahtet*" sind (vgl. /Chang,Lee 73/), erscheinen für die Spezifikation verschiedener Problemlösungsstrategien wissensbasierter Systeme zu unflexibel. Die Entwicklung prozeduraler Wissensrepräsentationssprachen wie PLANNER (/Hewitt 71/) ist genau auf

* für eine umfassende Einführung in das Gebiet der mathematischen Logik siehe z.B. /Mendelson 64/.

diese Überlegungen zurückzuführen. In den letzten Jahren hat sich die Teildisziplin des **logic programming** (vgl. /Kowalski 79/, /Clark,Tarnlund 82/) und insbesondere die Programmiersprache PROLOG zu einer interessanten Variante logischer Repräsentationsmechanismen entwickelt, da für sie neben der klassischen, deklarativen Semantik auch eine prozedurale Semantik vorgeschlagen wurde (s. /vanEmden,Kowalski 76/). Die Mechanismen zur Spezifikation von Kontrolle sind in PROLOG nur recht unzureichend, auf dem Gebiet des *logic programming* wurden jedoch eine Vielzahl von Theorien zur adäquaten Formalisierung und Interpretation von Kontrollwissen entwickelt (s. z.B. /Bowen,Kowalski 82/, /Sterling 84/, /Gallaire,Lasserre 82/).

Trotz der Schwierigkeiten bei der Verwendung logischer Sprachen und Kalküle, z.B. der Prädikatenlogik erster Stufe, als adäquate Repräsentations- und kontrollierbare Inferenzmechanismen in einem wissensbasierten System ist Logik ein essentieller Bestandteil der Wissensrepräsentationsforschung. Logik mit ihren universellen Ausdrucksmitteln und fundierten semantischen Theorie stellt die ideale *Referenzsprache* dar, um die Vielzahl verschiedener Repräsentationsmechanismen exakt zu beschreiben, indem ihre Semantik durch Aufzeigen der Korrespondenz zu einer Logik formalisiert wird (vgl. /Hayes 77/, /Hayes 79/, /Kowalski 79a/). Diese fundamentale Bedeutung logischer Formalismen betonen auch /Levesque 86/ und /Israel,Brachman 81/.

Netzwerkformalismen und strukturierte Objekte

Assoziative Netzwerke* sind in einer Vielzahl von Erscheinungsformen als Wissensrepräsentationsformalismen benutzt worden. Gemeinsam ist all diesen Formalismen, daß sie Wissen in Form einer Ansammlung von Objekten (Knoten) modellieren, zwischen denen binäre Assoziationen (gerichtete Kanten mit Bezeichnung) existieren. Objekte stehen dabei wieder für konkrete Individuen oder abstrakte Konzepte, die Assoziationen repräsentieren Relationen zwischen diesen. Operationen auf solchen netzwerkartig organisierten Wissensbasen sind das Einfügen und Löschen von Knoten und Kanten. Inferenzen werden bei einer solchen Organisation sehr effizient durch diverse Graphsuchalgorithmen realisiert, was entscheidend zur Beliebtheit dieser Art von Formalismus beigetragen hat.

Betrachtet man assoziative Netze in dieser Allgemeinheit, so kann man sie durchaus als eine andere Notation oder auch eine bestimmte Implementierungsform logischer Sprachen mit speziellen Indizierungsmechanismen betrachten (vgl. /Hayes 79/, /Schubert et al. 79/).

Erweitert man aber diesen Formalismus um weitere Strukturierungsmöglichkeiten für die repräsentierten Objekte etwa im Sinne von Minskys *Frame-Theorie* /Minsky-75/ und organisiert das Netz entlang spezieller, ausgezeichneter Relationen, so erhält man Mechanismen, mit denen sich Wissen in vielen Fällen weitaus adäquater modellieren läßt, als in einer logischen Repräsentationssprache und die zudem durch spezielle Inferenzalgorithmen effizienteres Schlußfolgern erlaubt, als ein Allzweck-Theorembeweiser.

Frame-basierte Wissensrepräsentationsformalismen strukturieren Objekte in der Form von *records* (*"slots and fillers"*). Die einzelnen Slots können jedoch nicht nur Attributwerte aufnehmen, sondern sie können typisiert und mit Anzahlattributen sowie default-Werten versehen sein, aber auch mit Prozeduren besetzt sein, die in bestimmten Situationen aktiviert werden (*demons*, *procedural attachment*). Ferner können in solchen Formalismen Eigenschaften von Objekten entlang spezieller Relationen vererbt werden (*inheritance of properties*), was zum einen eine effiziente Speicherung ermöglicht, zum anderen ein Hilfsmittel zur Organisation großer Wissensbasen darstellt. Die Kritik an diesen speziellen

* dieser neutrale Begriff soll dem irreführenden Begriff ***semantische Netze*** vorgezogen werden.

Mechanismen begründet sich vor allem darin, daß für derart komplexe Sprachen mit *"prozeduralen Zutaten"* keine deklarative Semantik angegeben werden kann. Kritiker sehen in ihnen eher mächtige Programmiersprachen als Wissensrepräsentationssprachen, wobei ihre Semantik nur durch das Verhalten der zugehörigen Interpreterkomponente festgelegt ist. Eine grundlegende und zufriedenstellende Semantik von Vererbungsmechanismen ließ sich bisher noch nicht definieren (vgl. /Touretzky et al. 87/, /Sandewall 86/). Auf schwerwiegende Probleme bei der Verwendung von *defaults* und dem Überschreiben von *defaults* durch Ausnahmen (*exceptions*) hat R. Brachman hingewiesen /Brachman 85/.

Ein sehr wichtiger Beitrag jedoch, den Frame- und Netzwerkformalismen für die Wissensrepräsentationsforschung geleistet haben, ist die besondere Auszeichnung wichtiger Relationen und Strukturierungsmittel als **epistemische Primitiva** (vgl. /Brachman 78/). Darunter sind :

- **Klassen-/Instanzen Relation**
  Objekte lassen sich differenzieren in **Klassen** (generische Konzepte) und **Instanzen** (individuelle Konzepte). Klassen stehen für eine Menge von gleichartigen Individuen oder Instanzen, indem sie die einheitliche Struktur und gemeinsame Eigenschaften all dieser beschreiben (*"WND-XV 31 ist eine Instanz der Klasse PKW"*). Für eine vollständige Modellierung müssen Klassen jedoch noch weiter differenziert werden: steht die Klasse PKW etwa für das Konzept eines PKW oder für die Menge aller PKW oder etwa für einen prototypischen PKW?
- **Generalisierung-/Spezialisierung**
  Eine Klasse ***subsumiert*** eine andere Klasse, wenn sie eine allgemeinere Beschreibung als diese darstellt. So subsumiert die Klasse der KFZ in unserer Intuition die beiden Klassen PKW und LKW und für viele Anwendungswelten ist diese als **IS-A** berühmt gewordene Relation ein äußerst wichtiges und wertvolles Beschreibungsmittel. Aber auch hier führt eine naive Benutzung, die die Semantik der Subsumptions- oder **IS-A**-Relation nicht genau festlegt, zu erheblichen Schwierigkeiten, wie R. Brachman mit mindestens 10 möglichen unterschiedlichen Interpretationen der **IS-A**-Relation aufgezeigt hat (s. /Brachman 83/).
- **Aggregation / Dekomposition**
  Die **PART-OF** bzw. **HAS-PART**-Relation verbindet Objekte und ihre Teile. Für physikalische Objekte sind dies deren Komponenten (*"Ein Stuhl besteht aus Lehne, Sitzfläche, Beinen,..."*), für abstrakte Konzepte deren Konstituenten ("Eine Abteilung besteht aus Fachgruppen."). Wie bei der Generalisierungsrelation läßt sich die Modellierung mit dieser Relation bis zu einer endlichen Tiefe rekursiv fortsetzen.
- **Partitionierung**
  Eine wichtige Methode zur Strukturierung von Wissen, das in Form von Netzwerken strukturierter Objekte repräsentiert wird, ist die Zusammenfassung inhaltlich zusammengehörender Konzepte in sogenannte **Partitionen**, die wiederum in einer Hierarchie geordnet sein können (vgl. /Hendrix 79/). Durch Partitionierung einer Wissensbasis lassen sich auch gleichzeitig verschiedene **Kontexte** (***belief spaces***) verwalten, die für sich gesehen je eine abgeschlossene KB darstellen können, die aber untereinander nicht verträglich sein müssen, weil sie z.B. widersprüchliches Wissen enthalten.

Epistemische Primitiva dienen somit zum einen einer möglichst adäquaten Modellbildung, erlauben aber auch die Verwendung spezialisierter Inferenzalgorithmen (vgl. /Schubert et al. 83/), was ihre große Bedeutung für die Wissensrepräsentation ausmacht. Problematisch jedoch bleibt die Formalisierung der Semantik von frame-basierten und netzwerkartigen Mechanismen. Dies darf aber auch nicht verwundern, da sie im Gegensatz zu logischen Sprachen, die keine Interpretation für ihre Symbole mitliefern,

symbolische Strukturen mit einer vordefinierten intendierten Bedeutung enthalten. Für diese Bedeutungen gibt es jedoch sehr unterschiedliche Sichtweisen, die sich zum Teil nur schwer formal definieren lassen.

Als Ausnahme und Beispiel für einen strukturierten Netzwerkformalismus, dessen Semantik sehr präzise formalisiert werden kann, sei hier das **OMEGA**-System genannt (s. /Hewitt et al. 80/, /Attardi,Simi 86/), das auf der Basis einer Logik für Beschreibungen aufbaut.

Produktionsregelsysteme

Obwohl weder wissensbasierte Systeme im allgemeinen noch die sog. Expertensysteme im speziellen zwingend regelbasierte Systeme sein müssen, sind **Produktionsregelsysteme** der dominierende Repräsentationsmechanismus für Problemlösungswissen in heutigen wissensbasierten Systemen (vgl. /Jackson 86/).

Eine Produktionsregel formalisiert ein abgeschlossenes "*Stück*" Wissen (***piece of knowledge***, ***chunk***) als Schlußfolgerungsprinzip, indem sie eine Situation oder eine ganze Klasse von Situationen mit einer oder mehreren Aktionen assoziiert. Klassen von Situationen lassen sich dabei sehr kompakt durch Strukturen mit allquantifizierten Variablen, sogenannten Mustern (***patterns***), beschreiben. Produktionsregelsprachen sind daher in der Regel Mustersprachen, Produktionsregelsysteme nehmen die Form mustergesteuerter Inferenzsysteme (***pattern-directed inference systems***) an (vgl. /Waterman, Hayes-Roth 78/). Im Idealfall sind alle Regeln unabhängig voneinander, d.h. das in ihnen repräsentierte Wissen ist abgeschlossen und ihre Anwendbarkeit wird nur durch die in ihnen beschriebenen Situationen ("linke Seiten" der Regeln) bestimmt. Regeln sind damit als solche bereits ein Strukturierungsmittel zur modularisierten Repräsentation von Wissen (*modularized know-how systems*, vgl. /Hayes-Roth 85/).

Das für wissensbasierte Systeme bedeutendste und einflußreichste Produktionsregelsystem ist **OPS5** (s. /Forgy 81/, /Brownston et al. 85/). Die grundlegende Systemarchitektur und das Operationsprinzip von **OPS5** findet sich in den meisten Produktionsregelformalismen in zum Teil leicht abgewandelter Form wieder: die Wissensbasis läßt sich unterteilen in einen dynamischen Speicher (***working-memory***) für das beschreibende Wissen und in einen Produktionsregelspeicher (***production memory***) (s. Abb. 1.2). Die Inferenzkomponente, in diesem Fall auch Regelinterpretierer genannt, wendet die Regeln auf die Daten im working-memory an, und zwar nach dem festen Operationsprinzip des sog. ***recognize-select-act*** Zyklus. In der ***recognize***-Phase werden alle zu diesem Zeitpunkt anwendbaren Regeln bestimmt, in der ***select***-Phase eine davon ausgewählt, die zur Anwendung gelangen sollen, und diese werden schließlich in der ***act***-Phase ausgeführt.

Bei einem allgemeinen Produktionsregelsystem kann die Interpretation entweder **datengetrieben** geschehen, d.h. ausgehend von einer Anfangsbeschreibung wird durch sukzessives Anwenden von Regeln und damit ständigem Erzeugen neuer Situationsbeschreibungen versucht, eine Zielbeschreibung zu erreichen (***data-driven forward-chaining***), oder aber **zielgerichtet** ausgehend von einer Zielbeschreibung durch Anwenden von Regeln und damit Erzeugen von Unterzielen versuchen, die momentane Situation zu erreichen (***goal-directed backward chaining***). In beiden Fällen stellt die dabei aufgebaute *Inferenzkette* mit den jeweiligen beim Mustervergleich (***pattern-matching***) aufgebauten Variablenbindungen die eigentliche Problemlösung dar.

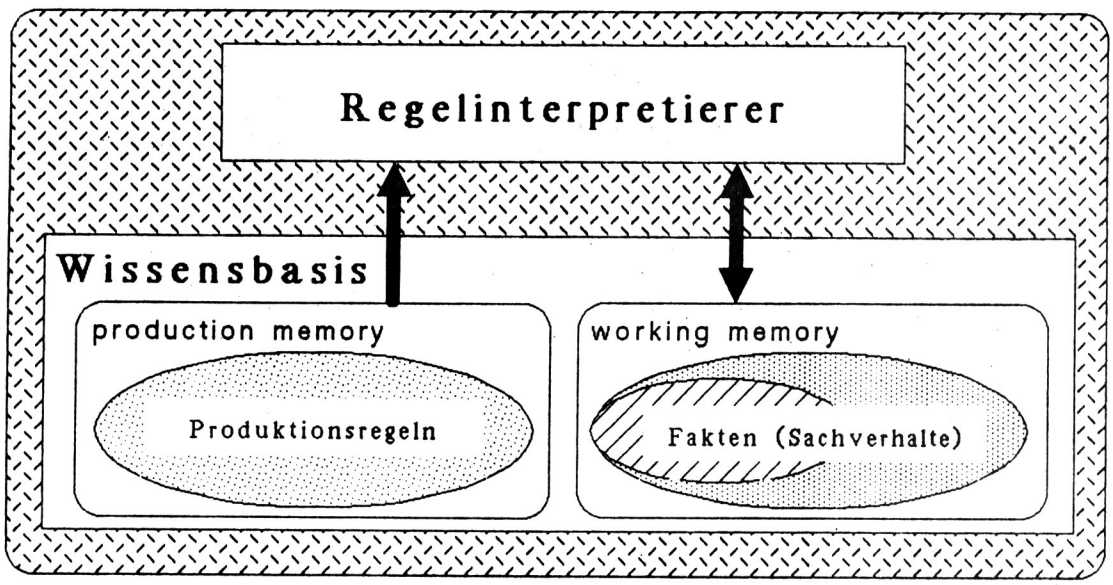


Abb. 1.2: ein einfaches Produktionsregelsystem

Unterscheiden lassen sich Produktionsregelsysteme durch die Form ihrer Elemente im working memory, diese kann von einfachen Fakten oder Sachverhalten (prädikatenlogische Grundatome) wie in **OPS5**, über Objekte in Form assoziativer Tripel bis hin zu strukturierten Objekten im Sinne frame-basierter Vererbungsnetze gehen.

Produktionsregeln wurden in vielen Anwendungen mit Evidenzkalkülen versehen, um ihnen den Charakter approximativer *"Daumenregeln"* zu geben, die plausible aber nicht zwingend korrekte Schlußfolgerungsprinzipien formalisieren. Diese Technik, die vor allem im Bereich medizinischer Diagnose wie z.B. in **MYCIN** Verwendung gefunden hat, ist aber keineswegs ein notwendiger Bestandteil von Produktionsregelsystemen.

Zwei unterschiedliche Typen von Regeln sollten unterschieden werden. Zum einen gibt es reine **Deduktionsregeln**, die lediglich neue Fakten etablieren und somit im Sinne des logischen Ableitungsbegriffes monoton neues Wissen inferieren und zum anderen **Aktionsregeln** oder **Operationen**, die auch Modifikations- und Löschoperationen auf Elementen des working-memory ausführen können, wodurch die Ableitungen nichtmonoton werden, d.h. ein abgeleitetes Faktum kann durchaus zu einem späteren Zeitpunkt invalidiert werden, weil z.B. eine der Vorbedingungen für seine Ableitung nicht mehr erfüllt ist. Mechanismen, die die durch Inferenzketten aufgebauten Abhängigkeiten explizit verwalten, werden als **Reason Maintenance Systeme** bezeichnet (s. z.B. /Reinfrank 86/). Solche Systeme erlauben die Repräsentation von Annahmen und garantieren die Konsistenz einer Wissensbasis auch nach dem Zurückziehen von Annahmen.

Die größte Problematik bei der Verwendung von regelbasierten Inferenzkomponenten birgt das bereits erörterte Kontrollproblem. Im Interpretationszyklus für Regeln muß in der ***select***-Phase aus allen anwendbaren Regeln, d.h Regeln mit erfülltem Bedingungsteil, eine Auswahl getroffen werden. Typischerweise sind in dieser **Konfliktlösung** (***conflict resolution***) einige Regeln involviert, da die behandelten Probleme im allgemeinen sehr große Suchräume bedingen. Ohne Spezifikation von Kontrollwissen sind regelbasierte Systeme genauso anfällig für eine kombinatorische Explosion des Suchraumes wie etwa ein einfacher Resolutionsbeweiser. Einfache syntaktische Konfliktlösungsmethoden wie sie z.B. in **OPS5** angewandt werden, stützen sich auf Regelcharakteristika wie die Anzahl der

Vorbedingungselemente (wie spezifisch ist eine Regel?), oder betrachten den Zeitpunkt der letzten Benutzung eines Elementes im working-memory. Zur Repräsentation von anwendungsspezifischer Kontrollinformation werden in solche Systemen oft sog. *Kontrollfakten* bzw. *Kontrollobjekte* eingeführt, auf die in anderen Regeln explizit Bezug genommen werden kann. Diese Technik ist wohl am ausgeprägtesten im Kontextmechanismus (vgl. /Brownston et al. 85/) des **OPS5**-basierten Expertensystems **XCON/R1** wiederzufinden (s. /McDermott 81/). Die Vorgehensweise hat jedoch schwerwiegende Nachteile:

- Kontrollwissen (globale Strategien, lokale Konfliktlösungstaktiken) wird nicht als solches explizit repräsentiert, sondern findet sich codiert in einer Vielzahl von Regeln. Dadurch werden die Regeln komplexer und unverständlicher, das Kontrollwissen als solches meist gar nicht mehr erkennbar.
- die Regeln nehmen durch Kontrollinformation indirekt aufeinander Bezug und verlieren ihre Abgeschlossenheit. Der modulare Charakter von Regeln geht dadurch verloren, die Verständlichkeit einer vollständigen Regelbasis wird vermindert.
- die Modifikation von Kontrollstrategien macht die Änderung sehr vieler Regeln notwendig, die typische Entwicklungsform für wissensbasierte Systeme, nämlich der inkrementelle Aufbau von Wissensbasen, wird extrem erschwert.
- die Regeln verlieren ihren deklarativen Charakter, das Formalisieren von Wissen in Form von Regeln wird immer mehr zum Programmieren mit Regeln.

Zur Spezifikation von Kontrollwissen für Produktionsregelsysteme wurden deshalb bereits weiterführende Ansätze und Techniken entwickelt. R. Davis schlägt in seinem **TEIRESIAS**-System Meta-Regeln zur Konfliktlösung vor (s. /Davis 80/), W. Clancey verwendet in **NEOMYCIN** abstrakte Tasks und Meta-Regeln über Tasks (s. /Clancey,Bock 82/, /Clancey 83/). In der kommerziellen Expertensystemshell S.1 werden prozedurale Kontrollblöcke eingesetzt (vgl. /Erman et al. 84/), während im **PRESS**-System die Technik der sog. **Metaebeneninferenz** (***meta-level-inference***) verwendet wird (s. /Bundy,Welham 81/, /Silver 86/). M. Genesereth beutzt in **META-LEVEL-ARCHITECTURE** und **MRS** eine deklarative Sprache zur Beschreibung des Verhaltens des Regelinterpretierers (s. /Genesereth,Davis 83/, /Genesereth et al. 80/). Die Konzepte zur Spezifikation von Kontrolle in ***ARI*** und dem Vorgängersystem **CATWEAZLE** werden in den Kapiteln 2 und 3 dieser Arbeit ausführlich beschrieben.

1.3. Ein Anwendungsbereich: das Büro

Der Bereich der **Büroautomation** ist zu einer wichtigen softwaretechnischen Herausforderung geworden. Man ist sich zum einen des Potentials für Automatisierung angesichts vieler Standardaufgaben, zum anderen der Notwendigkeit der Unterstützung eines Büroangestellten aufgrund immer komplexerer Vorgänge und intensiverem Informationsfluß bewußt. Mit den bisher eingesetzten Systemen jedoch können nur sehr begrenzte Aufgabenbereiche abgedeckt werden, die sich in klar strukturierte, ständig wiederkehrende Teilaufgaben gliedern und damit sehr leicht algorithmisch und prozedural formalisieren lassen. Statt einer Ansammlung fest vorgegebener Teilabläufe umfaßt Büroarbeit dem Wesen nach aber vielmehr eine Fülle koordinierter **Problemlösungsaktivitäten** in einer dynamischen und sehr stark durch Kommunikation geprägten Umgebung. Zukünftige Bürosysteme sollten deshalb vorhandene Pakete für Standardabläufe mit beratenden Teilsystemen, die einen Büroangestellten bei seinen täglichen Problemlösungsaktivitäten unterstützen, in einer hochgradig interaktiven Gesamtarchitektur integrieren (vgl. /Barber,Hewitt 82/, /Croft,Lefkowitz 84/).

Wissensbasierte Systeme sind aufgrund ihrer Fähigkeiten zur Formalisierung von Problemlösungsprozessen, ihre Flexibilität und Anpassbarkeit an eine sich ständig ändernde Umgebung, sowie durch ihre Möglichkeiten zu intelligentem Dialog und Erklärungsfähigkeit die idealen Kandidaten für solche Systemarchitekturen intelligenter Assistenten. Aufgrund der Komplexität des Anwendungsgebietes und der hohen Anforderungen ist aber auch klar, daß diese idealen Kandidaten nur mit Hilfe mächtiger Wissensrepräsentations- und Inferenztechniken sowie Methoden zur Organisation und Strukturierung zu konkreten Systemen reifen können.

Welches Wissen ist nun aber relevant und bestimmend im Bürobereich? G. Barber und C. Hewitt unterscheiden zum einen **Anwendungswissen,** d.h. Wissen über die Aufgaben und Ziele eines Büros innerhalb einer Organisation, und **Wissen über Organisationsstrukturen**, also z.B. Wissen über die Organisationseinheiten (Bereich, Abteilung, Fachgruppe, etc.), über formale Autoritätshierarchien zwischen Vorgesetzten und Untergebenen, oder auch über informelle fachliche Beziehungen zwischen Büroangestellten. Für diese verschiedenen Arten von Wissen müssen in einem wissensbasierten System geeignete Repräsentationsstrukturen bereitgestellt werden. Aufgrund seiner Vielfältigkeit läßt sich Bürowissen sicherlich nur sehr schwer und ineffizient ausschließlich in Form von heuristischen, situationsspezifischen Regeln formalisieren. Ein *"feinkörniges"* Modell der Bürowelt, das **strukturelle**, **funktionale** und **kausale** Zusammenhänge explizit modelliert, kann hier als wichtige Grundlage für effektives Problemlösen dienen. Ein solches Modell wird auch als **Tiefenmodell** bezeichnet.

Die zentrale Arbeitseinheit im Büro, die es für ein wissensbasiertes System zu unterstützen gilt, ist der **Vorgang** (vgl. /Kreifelts 84/). Vorgangsbeschreibungen sind allerdings keine formalisierten Algorithmen sondern vielmehr vorgegebene Rahmen für typische Aktivitäten, die zum einen meist unvollständig, zum anderen häufigen Änderungen oder Ausnahmesituationen ausgesetzt sind. Betrachtet man die Bearbeitung eines Vorgangs als Problemlösungsprozeß, so besteht dieser aus einer Menge von **Zielen** (***goals***), die sich über eine Zeitraum entwickeln und verändern, die **Randbedingungen** (***constraints***) der Anwendungs- und Organisationsstrukturen unterworfen sind und die es in der richtigen Reihenfolge zu richtigen Zeitpunkten zu erfüllen gilt.

Dieses problemlösungsorientierte Modell für die Unterstützung von Büroarbeit macht also das **Planen** und kontrollierte Ausführen von Plänen zur zentralen Aktivität eines intelligenten Bürosystems. Aufgrund der dynamischen Umgebung kann das Planen kooperierender Bürovorgänge nur sinnvoll als **inkrementelles Planen** mit dazu verzahnter, schrittweiser Ausführung von Teilplänen durchgeführt werden. Resultate und sonstiges Feedback der Ausführungsphase können dabei die Modifikation des Restplans bedingen oder auch eine vollständige Neuerstellung des Planes nötig machen (vgl. /Lutze 88/).

Plansynthese ist aber keineswegs die einzige Klasse von Problemlösungsprozessen für die geeignete Inferenzmethoden bereitgestellt werden müssen. Die **Klassifikation** von komplexen Situationen oder von konkreten Objekte innerhalb der Bürowelt, wie Dokumenten oder Produkten, wird eine weitere wichtige Aktivität sein, die es zu modellieren gilt. Darüberhinaus müssen **intelligente Retrievalprozesse** relevante Informationen gezielt aus der Menge des gesamten Bürowissens extrahieren können (vgl. /Eirund,Kreplin 88/). Dieses Wissen wird nämlich aufgrund seiner komplexen Strukturen, sowie vielfältigen Formen und Abhängigkeiten nicht mit einem einfachen Datenmodell wie es einer konventionellen Datenbank zugrundeliegt zu formalisieren sein. Ein weiterer Punkt ist die Notwendigkeit, bei der Ausführung geplanter Vorgänge Fehlfunktionen zu erkennen und zu beheben. Verschiedene Formen von **Diagnose** stellen hier eine adäquate Problemlösungsstrategie dar.

Das vorgestellte Szenario für wissensbasierte Systeme im Büro muß in dieser Kürze recht abstrakt bleiben, es sollen aber einige Punkte daraus festgehalten werden, die den Entwurf von ***ARI*** beinflußt haben:

(1) Wissensbasierte Systeme sind ein vielversprechender Kandidat für die Realisierung interaktiver unterstützender Bürosysteme, die Büroaktivitäten als Problemlösungsprozesse modellieren.

(2) Die Modellierung des vielfältigen Wissens innerhalb der Bürowelt (Anwendungsstrukturen, formale und informelle Organisationsstrukturen) verlangt eine **Vielfalt von Repräsentationsmechanismen,** z.B. zum Aufbau eines **Tiefenmodells.**

(3) Die Vielfalt der Klassen von Problemlösungsprozessen innerhalb der Bürowelt (*Planung, Klassifikation, Diagnose, intelligentes Retrieval*) machen **unterschiedliche Inferenzmechanismen** notwendig.

(4) Das für das System relevante Wissen ist nicht vollständig und in einer festgeschriebenen Form vorhanden, sondern kann nur **inkrementell** über einen längeren Zeitraum modelliert werden, ist also stets unvollständig, und muß jederzeit **leicht modifiziert** werden können, z.B. zur Adaption an veränderte Organisationsstrukturen, Firmenpolitiken oder Marktstrategien.

(5) Wissensbasierte Systeme im Büro müssen typischerweise sehr große Mengen von Wissen und Daten verwalten und sollten darüberhinaus bereits existierende Softwaresysteme für spezielle Teilaufgaben **integrieren** können.

(6) Ein wissensbasiertes System im Büro muß sich seinem Benutzer gegenüber **transparent** zeigen. Dies umfaßt, daß das System seine Vorgehensweise **erklären** kann, bei seinem Problemlösungsprozeß natürliche Schlußfolgerungslinien verfolgt und im Dialog die Terminologie und Sprachkonventionen des Benutzers berücksichtigt.

2. Von Catweazle zu *ARI*

2.1. Der Catweazle Regelinterpretierer

Ziel des CATWEAZLE-Systems war es, auf der Grundlage einer Produktionsregelsprache eine Inferenzmaschine bereitzustellen, mit deren Hilfe die Konstruktion komplexer wissensbasierter Systeme unterstützt wird. CATWEAZLE stellt Konzepte zur Verfügung, die Aspekte der Effizienz, der Erklärungsfähigkeit und der Wartbarkeit großer Wissensbasen innerhalb eines einheitlichen Ansatzes berücksichtigen. Dieser Ansatz erlaubt die explizite und deklarative Repräsentation verschiedener Wissensarten indem er Konstrukte zur Strukturierung bzw. Modularisierung großer Regelbasen sowie zur Spezifikation von Kontrollwissen verwendet.

Wissensbasierte Systeme auf der Grundlage der CATWEAZLE-Sprache gehören einer speziellen Klasse von Softwarearchitekturen an, den sog. **Metaebenen-Architekturen** (*meta-level architectures*), die von vielen Forschern als geeignetes Mittel zum Aufbau komplexer wissensbasierter Systeme angesehen werden (vgl. z.B. /Davis 78/, /Aiello,Levi 84/, /Clancey 85/)*. Eine einfache, zweistufige Metaebenen-Architektur besteht dabei aus einer **Basisinferenzmaschine**, der **Objektebene**, und einer **Kontrollkomponente**, der **Metaebene**.

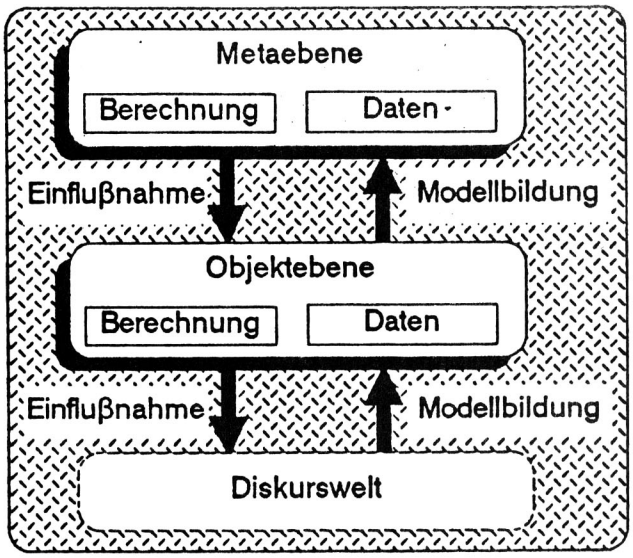


Abb. 2.1: Eine zweistufige Metaebenen-Architektur

* ein umfassender Überblick zu Metaebenen-Architekturen findet sich in /vanHarmelen 86/.

Als charakteristisch für eine Metaebenen-Architektur gilt (vgl. /Maes 87/) das Vorhandensein

- eines expliziten Modells der Objektebenenberechnungen,
- eines kausalen Zusammenhangs zwischen Aktionen auf Meta- und Objektebene,
- einer Architektur für **Introspektion** (***introspection***), die einen Wechsel zwischen Aktivitäten auf Meta- und Objektebene erlaubt.

In CATWEAZLE ist die Objektebene eine **OPS5**-artige Produktionsregelsprache. Die Kontrolle über Inferenzprozesse auf der Ebene der **Objektregeln** wird in Form von **Meta-Regeln** zur Konfliktlösung (**Kontrolltaktiken**) explizit spezifiziert. Die Regelbasis wird durch Zusammenfassen von Objektregeln, die alle für die gleiche Teilaufgabe relevant sind, und entsprechenden Kontrolltaktiken, die Konfliktlösung zwischen diesen Objektregeln formalisieren, in eine Menge **strukturierter Regelkomplexe** (***structured rulesets***) partitioniert. Während der Interpretation ist stets genau einer dieser Regelkomplexe aktiviert, und nur die darin enthaltenen Regeln und Meta-Regeln werden in der ***recognize***-Phase des Interpreters berücksichtigt und können zur Ausführung gelangen. Die Anzahl der anwendbaren Regeln wird damit auf eine kleine Menge relevanter Regeln beschränkt und der Berechnungsaufwand in der ***recognize***-Phase wird durch diese Einschränkung des Suchraumes drastisch reduziert. CATWEAZLE stellt auf einer zweiten Ebene Mechanismen für die Spezifikation von Kontrollwissen über die Aktivierung von Regelkomplexen bereit. Zu diesem Zweck werden Regelkomplexe mit einer **abstrakten Beschreibung** versehen, die neben einem *Namen* eine *Vorbedingung* und eine *Nachbedingung* enthalten. Vor- und Nachbedingung werden in derselben Mustersprache spezifiziert wie die Vorbedingungen von Objektregeln. Die Vorbedingung eines Regelkomplexes spezifiziert die Menge aller Situationen, in denen dieser vom Interpretierer *aktiviert* werden kann. Analog dazu spezifiziert die Nachbedingung eines Regelkomplexes Wissen, das in allen Situationen unmittelbar nach Abarbeitung des Regelkomplexes etabliert ist. Sie wird vom Interpretierer als *Deaktivierungsbedingung* für den Regelkomplex benutzt, d.h. sobald die Interpretation des Regelkomplexes dazu führt, das die Nachbedingung erfüllt wird und damit nichts weiter zur Problemlösung beigetragen werden kann, wird die Interpretation dieses Regelkomplexes beendet.

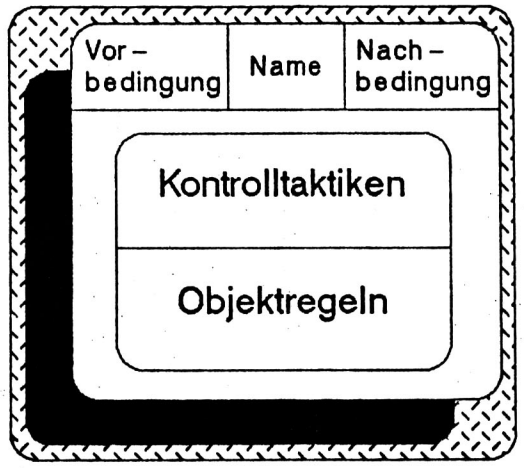


Abb. 2.2: Struktur eines Regelkomplexes

Auf der Ebene von Regelkomplexen wird Kontrollwissen entweder ebenfalls in Form von Meta-Regeln spezifiziert oder aber in Form sogenannter **Phasenfolgen**.

Meta-Regeln auf dieser Ebene werden auch als **Kontrollstrategien** bezeichnet und haben die Menge aller Regelkomplexe als zugehörige Objektebene. Die Vorbedingung einer Kontrollstrategie kann dabei mit der abstrakten Beschreibungen von Regelkomplexen auf diese Bezug nehmen.

Phasenfolgen erlauben die Spezifikation deterministischer Aktivierungsreihenfolgen für Regelkomplexe, die über ihren Namen referenziert werden können. Durch die Verwendung von Mustern innerhalb von Konstrukten für Verzweigungen und Schleifen mit Abbruchbedingungen können flexible und sehr mächtige Kontrollstrategien explizit formalisiert werden. Beispiel für die Verwendung der Kontrollmechanismen, u.a. die Modellierung der bekannten HYPOTHESIZE AND TEST Strategie mit Hilfe von Phasenfolgen, finden sich in /Beetz 87/, Abb. 2.3 gibt hier lediglich noch einmal einen allgemeinen Überblick über das dreistufige Metaebenenkonzept von CATWEAZLE.

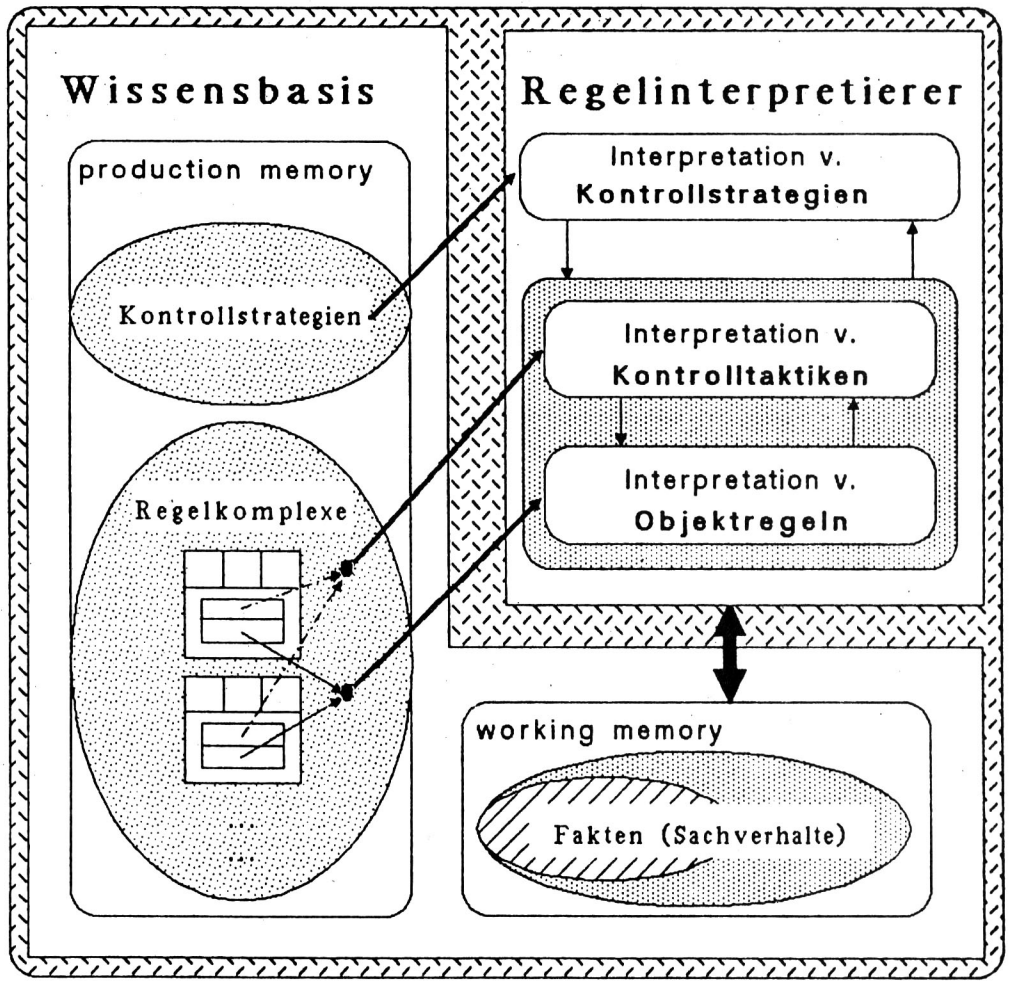


Abb. 2.3: **CATWEAZLE** *als Metaebenen-Architektur*

Auf eine detaillierte Beschreibung des Interpretationszyklus der CATWEAZLE-Inferenzmaschine wird hier verzichtet, da er in verallgemeinerter Form in *ARI* verwendet wird und in Abschnitt 3.2.4. ausführlich beschrieben wird.

Ein wesentlicher Gesichtspunkt des CATWEAZLE-Systems ist die effiziente Implementierung des Regelinterpretierers auf der Grundlage des RETE-Algorithmus. Der RETE-Algorithmus wird dabei zur Realisierung des gesamten *pattern-matching* in CATWEAZLE benutzt. Dies umfaßt

- die Berechnung der Konfliktmenge anwendbarer Objektregeln analog zu **OPS5**,

- die Berechnung der anwendbaren Kontrolltaktiken bei der Konfliktlösung zwischen mehreren anwendbaren Objektregeln,
- die Berechnung der anwendbaren Kontrollstrategien bei der Konfliktlösung zwischen mehreren anwendbaren Regelkomplexen, und
- die Abarbeitung von Phasenfolgen.

Der Algorithmus wurde so erweitert, daß die unterschiedlichen Berechnungen für *pattern-matching* mit minimalem Mehraufwand in einem Bedingungsnetzwerk abgehandelt werden. Effiziente Realisierung des *pattern-matching* für deklarative Kontrollmechanismen, mit denen wiederum eine Vielzahl verschiedener Problemlösungsstrategien spezifiziert werden können, ist ein entscheidender Faktor für die praktische Einsetzbarkeit des Konzeptes der Metaebenen-Architekturen für Produktionsregelsysteme.

2.2. Grundlagen des Inferenzsystems ARI

ARI, eine Erweiterung und Verallgemeinerung der CATWEAZLE-Konzepte, ist der Kern eines praktisch einsetzbaren Werkzeugs zur Entwicklung wissensbasierter Systeme. Die zentrale Problematik der Realisierung solcher Systeme ergibt sich vor allem aus der Vielzahl unterschiedlicher Konzepte und zu berücksichtigender Aspekte von Entwurf, Implementierung, Integration und Wartung. Diese lassen oft nur schwer ein umfassendes und kohärentes Bild eines wissensbasierten Systems aufzeigen. Im folgenden werden die allgemeinen Grundlagen der Methodik von ***ARI*** charakterisiert, die die unterschiedlichen Sichtweisen von solchen Systemen explizit identifiziert, und es wird im speziellen auf die zentrale Prämisse eingegangen, die ***ARI*** im Bezug auf Wissensrepräsentationsmechanismen zugrundeliegt.

2.2.1. Abstraktionsebenen für Inferenzsysteme

Das Hauptproblem bei der Konstruktion wissensbasierter Systeme liegt darin, daß allzu oft Konzepte des Anwendungsbereiches (z.B. die hierarchische Organisationsstruktur einer Firma) mit Konzepten der Wissensrepräsentation (z.B Vererbungsstrategien in frame-basierten Systemen) oder gar programmtechnischen Aspekten (z.B. spezielle Graphsuchverfahren) vermischt werden. Eine adäquate Modellierung von Problemlösungsprozessen eines menschlichen Experten aber kann im allgemeinen nicht auf einer Ebene gefunden werden, auf der man sich z.B. mit Prioritätsfunktionen für Agendaeinträge beschäftigen muß.

Die gleiche Problematik führte in der Forschung über **semantische Netze** in den siebziger Jahren zu sehr unterschiedlichen und untereinander unverträglichen Theorien, bis vor allem W. Woods /Woods 75/ und R. Brachman /Brachman 78/ durch explizite Unterscheidung der verschiedenen Betrachtungsweisen semantischer Netze Klarheit schafften und eine fundamentale Einsicht in dieses Teilgebiet der Wissensrepräsentationsforschung brachten, letzterer vor allem durch die Einführung verschiedener **Abstraktionsebenen**.

In der Informatik hat sich die Beschreibung komplexer, informationsverarbeitender Systeme durch eine Hierarchie von Abstraktionsstufen oder **abstrakter Maschinen** zu einem klassischen Paradigma entwickelt. Ansätze zur Modellierung wissensbasierter Systeme als abstrakte Maschinen (***knowledge system architectures***) finden sich z.B. in /Gruber,Cohen 87/.

Für den Entwurf von *ARI* unterscheiden wir in Anlehnung an die Klassifikation von R. Brachman drei Abstraktionsebenen. Sie erleichtern die Beschreibung des Systems und zeigen Leitlinien auf, wie mit Hilfe von *ARI* Architekturen wissensbasierter Systeme aufgebaut werden können.

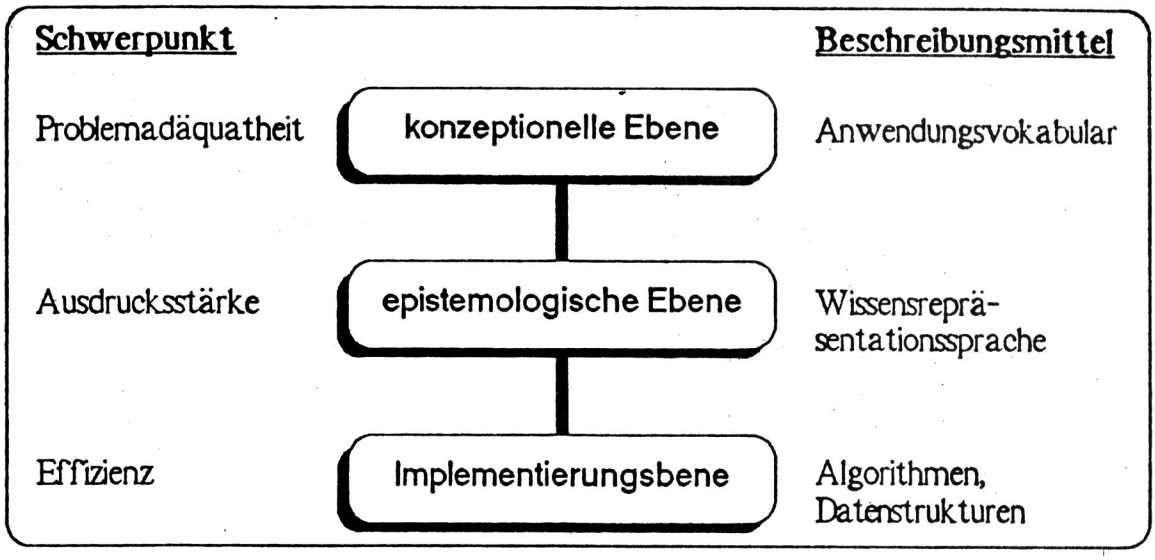


Abb. 2.4: Abstraktionsebenen für wissensbasierte Systeme

Der zentrale Ansatzpunkt ist die **epistemologische Ebene**, auf der eine hinreichend ausdrucksstarke Repräsentationssprache bereitgestellt werden muß. Diese Sprache muß es ermöglichen, ein adäquates beschreibendes Modell der Diskurswelt und die Problemlösungsmethoden innerhalb dieses Weltausschnittes symbolisch zu repräsentieren. Ein Maß für *"Adäquatheit"* muß für die jeweilige Anwendung festgelegt werden und kann alle in Abschnitt 1.2. beschriebenen Aspekte der Wissensrepräsentation berühren. Auf die zentrale Bedeutung von Mechanismen zur Organisation und Strukturierung von Wissen ist bereits mehrfach hingewiesen worden. Diese Mechanismen sollten deshalb direkt als Sprachkonstrukte auf der epistemologischen Ebene vorhanden sein, also einen integralen Bestandteil des Repräsentationsformalismus darstellen.

Die Elemente oder Primitiva einer Repräsentationssprache auf epistemologischer Ebene werden im folgenden als **Wissensstrukturen** bezeichnet. In einem adäquaten Repräsentationsmechanismus stehen also unterschiedliche Wissenstrukturen für entsprechende verschiedene Arten von Wissen zur Verfügung. Allgemeine Wissensstrukturen sind die bereits angesprochenen Objekte und Sachverhalte, aber auch für Regeln, Pläne oder Aktivitäten können in einem Formalismus eigene Wissensstrukturen vorhanden sein.

Konkrete Ausprägungen von Wissensstrukturen in einer Wissensbasis werden im folgenden als **Wissenseinheiten** bezeichnet. Wissenseinheiten sind also tatsächlich gespeicherte Daten, die das Modell oder die Repräsentation für eine instantiierte Wissensstruktur darstellen.

Neben der Wissensrepräsentationssprache muß aber auf der epistemologischen Ebene auch ein Operations- oder Interpretationsmodell für Wissenseinheiten vorgegeben werden, das ausschließlich auf den Wissensstrukturen dieser Ebene aufbaut. Dies erlaubt die Beschreibung wissensbasierter Systeme im Sinne einer abstrakten Architektur. Als Beispiel für ein solches Interpretationsmodell mag der bereits mehrfach angesprochene *recognize-select-act Zyklus* eines Produktionsregelsystems dienen.

Die **Implementierungsebene** muß Datenstrukturen und Interpretationsalgorithmen zur Verfügung stellen, die es erlauben, das repräsentierte Wissen zum Berechnen von Problemlösungen zu verwenden.

Hierbei sind Effizienzaspekte aber auch generelle theoretische Fragen der Berechenbarkeit zentrale Punkte. Erkenntnisse auf dieser Ebene werden also die Wahl der Wissensstrukturen auf epistemologischer Ebene im Sinne des von H. Levesque formulierten *tradeoffs* zwischen Ausdrucksstärke und Effizienz entscheidend beeinflussen.

Betrachtet man kommerzielle Werkzeuge zur Entwicklung wissensbasierter Systeme, so erweisen sich deren Repräsentationssprachen als komplexe Programmiersprachen, die nur von Spezialisten beherrscht und ausgenutzt werden können. Die Vorstellung, ein sogenannter **knowledge engineer** mit Grundwissen über das Anwendungsgebiet und Kenntnis der Repräsentationssprache könnte unter Verwendung solcher Werkzeuge und durch *Interviewen* eines fachlichen Experten sehr leicht ein vollständiges, *schlüsselfertiges* wissensbasiertes System entwickeln, erweist sich deshalb meist sehr schnell als Trugschluß. Dieses Szenario läßt sich nur dann realistisch aufrechterhalten, wenn die Problembeschreibungssprache direkt die Terminologie und damit die Wissensstrukturen der Anwendungswelt bereitstellt, anstatt sehr generelle, anwendungsunabhängige Konstrukte wie *Frames* und *Produktionsregeln* zur Verfügung zu stellen, für deren Verwendung keine einheitliche Methodik existiert. Ferner müßten statt allgemeiner Schlußfolgerungsmechanismen spezielle problembezogene Inferenztechniken mit spezifischen Kontrollmechanismen eingesetzt werden.

Solche spezialisierten Repräsentations- und Inferenzmechanismen können auf einer **konzeptionellen Ebene** für wissensbasierte Systeme angesiedelt werden. So sind z.B. für den Bereich diagnostischer Expertensysteme problemspezifische Repräsentationssprachen und Entwicklungswerkzeuge mit Erfolg entwickelt worden (s. z.B. /Puppe 87/, /Chandrasekaran,Mittal 83/).

Eine allgemeine Theorie, die den Ansatzpunkt zur Konstruktion wissensbasierter Systeme auf dieser Ebene ansiedelt, ist die **generic task theory** von B. Chandrasekaran (s. /Chandrasekaran 87/). Er identifiziert Klassen von Problemen, für die er spezielle Problemlösungskomponenten, die sog. *generic tasks*, vorschlägt. *Generic tasks* enthalten Wissensstrukturen sowie Inferenztechniken und Kontrollmechanismen, die an die betreffende Problemklasse angepaßt sind. Idealerweise sollten solche unterschiedlichen *generic tasks* jeweils spezialisierte Teilprobleme lösen und zur Komposition einer Gesamtlösung beitragen. Der Ansatz ist also vor allem auch im Hinblick auf komplexe aber strukturierte Probleme sehr vielversprechend.

Im Gegensatz zu Chandrasekarans Ansatz, in dem eine Vielzahl von Sprachen und Mechanismen eine einheitliche epistemologische Ebene ersetzen, können Sprachen auf konzeptioneller Ebene auch auf einer solchen einheitlichen Ebene aufsetzen. Die Semantik einer konzeptionellen Sprache ließe sich dann operational durch Transformation in entsprechende Strukturen auf der epistemologischen Ebene, also durch Benutzung der abstrakten Maschine oder Architektur dieser Ebene, definieren.

2.2.2. Die Deklarativitätsprämisse

Die Anforderungen, die wissensbasierte Systeme im Büro erfüllen müssen (vgl. Abschnitt 1.3.), sind für eine Vielzahl von Anwendungsbereichen diskutiert worden. Das Problem der **Änderungsfreundlichkeit** und **Wartbarkeit** großer Wissensbasen über einen langen Zeitraum (vgl. /Clancey 83/) und die Notwendigkeit eines gewissen Grades an **Erklärungsfähigkeit**, die die Transparenz eines Systems erhöht (vgl. /Neches,Swartout,Moore 85/), sind dabei zentrale Gesichtspunkte. Ein weiterer wichtiger Aspekt für die Konstruktion von wissensbasierten Systemen ist die Forderung nach **mächtigen Entwicklungswerkzeugen**, die der typischen inkrementellen Vorgehensweise beim Aufbau von

Wissensbasen Rechnung tragen (vgl. /Fickas 87/). Die Schlußfolgerung, daß diese Forderungen nur durch **explizite** und möglichst **deklarative** Repräsentation aller verschiedenen für eine Anwendungwelt relevanten Arten von Wissen erfüllt werden können, diente als grundlegende Richtlinie für den Entwurf des Inferenzsystems *ARI*. Die Hauptaufgabe bestand also darin, *für adäquate Wissensstrukturen zur expliziten und deklarativen Repräsentation von Wissen möglichst effiziente Interpretationsalgorithmen zu entwickeln*. Diese Richtlinie wird im folgenden als **Deklarativitätsprämisse** für *ARI* bezeichnet. Viele Systeme erfüllen diese naheliegende Forderung aus Effiziezgründen nicht, da sie explizit und deklarativ repräsentiertes Wissen nicht in effizient ablauffähige prozedurale Algorithmen übersetzen können.

2.2.3. Der *ARI*-Ansatz für Inferenzsysteme

ARI übernimmt von CATWEAZLE den Ansatz des **strukturierten Produktionsregelsystems** zur Formalisierung von Problemlösungsmethoden. Produktionsregeln mit ihrer Möglichkeit zur adäquaten Repräsentation situationsabhängigen Problemlösungswissens haben ihre Nützlichkeit in vielen Anwendungen wissensbasierter Systeme unter Beweis gestellt und der CATWEAZLE-Ansatz hat gezeigt, daß die in Abschnitt 1.2. diskutierten Probleme mit Produktionsregelsystemen durch explizite Strukturierungsmechanismen und das Konzept der Metaebenen-Architektur zur Kontrolle im Sinne der Deklarativitätsprämisse zu lösen sind. Die konkreten Erweiterungen des *ARI*-Ansatzes umfassen:

- Verarbeitung reichhaltigerer Basiswissensrepräsentationsstrukturen, insbesondere *strukturierter Objekte* in Vererbungsnetzwerken und Integration der durch diese Repräsentationsmechanismen bereitgestellten speziellen Inferenztechniken.
- Strukturierungsmechanismen, die im Gegensatz zur *flachen* Organisation der Regelkomplexe in CATWEAZLE eine *hierarchische Organisation* von Regelbasen erlauben.
- *Partitionierung* des beschreibenden Wissens durch explizite Spezifikation der beschreibenden Wissenseinheiten, die für einen Regelkomplex relevant sind und von diesem gesehen werden.
- uniforme *Integration* aller Repräsentationskonstrukte in den zugrundeliegenden RETE-Algorithmus zur effizienten Interpretation.

Im folgenden Kapitel 3 wird das *ARI*-Inferenzsystem vor dem Hintergrund des vorgestellten Modells verschiedener Abstraktionsebenen detailliert beschrieben und diskutiert.

3. Konzepte des Inferenzsystems *ARI*

3.1. Die LUIGI Umgebung

ARI ist ein Teil des prototypischen Werkzeugsystems LUIGI, das als Umgebung zur Entwicklung wissensbasierter Systeme im Büro dient (s. /Lutze 88/). LUIGI stellt eine Repräsentationssprache für die adäquate Modellierung des beschreibenden oder statischen Wissens der Bürowelt zur Verfügung. Über den Wissensstrukturen muß die Produktionsregelsprache von *ARI* Schlußfolgerungsprinzipien formulieren und der Regelinterpretierer muß konkrete Ausprägungen der Wissensstrukturen, also die Wissenseinheiten in einer LUIGI-Wissensbasis, verarbeiten können.

Im LUIGI-System werden alle Wissenseinheiten, sowohl beschreibendes Wissen als auch konkrete Strukturen der Produktionsregelsprache, in einem **Wissensrepräsentationssubsystem** (auch als **Objektsystem** bezeichnet) verwaltet. LUIGI unterscheidet nicht explizit zwischen *production-memory* und *working-memory*, sondern verwaltet alle Wissenseinheiten in einem einheitlichen Objektsystem.

Zur Spezifikation von Wissen und für Benutzerdialoge steht im LUIGI-System der interaktive graphik-basierte Wissenseditor LUKE (LUIGI Knowledge Editor) zur Verfügung. Die Gesamtstruktur des LUIGI-Systems zeigt Abb. 3.1.

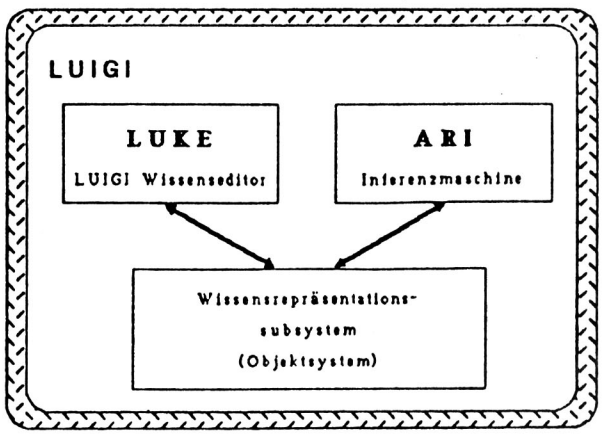


Abb. 3.1: Das Werkzeugsystem LUIGI

Im folgenden werden nur die Teile des LUIGI-Systems beschrieben, die für den Entwurf bzw. für die Arbeitsweise der *ARI*-Inferenzmaschine von Bedeutung sind, nämlich

- **Wissensstrukturen** für beschreibendes Wissen, auf die in Konstrukten der Produktionsregelsprache Bezug genommen werden kann, und
- spezielle **Inferenzalgorithmen**, die bereits im Objektsystem ausgeführt werden und die von der Inferenzmaschine benutzt und angestoßen werden können.

LUIGI stellt zur Repräsentation von beschreibendem Wissen einen Mechanismus im Sinne strukturierter Vererbungsnetze wie etwa KL-ONE (s. /Brachman,Schmolze 85/) zur Verfügung sowie ein Konstrukt zur expliziten Repräsentation von Sachverhalten zwischen Objekten. **Sachverhalte** (***affairs***) modellieren *n*-stellige Relationen zwischen Objekten bzw. Werten und entsprechen im wesentlichen den Fakten in **OPS5**.

Strukturierte Vererbungsnetze stellen sicherlich einen sehr geeigneten Repräsentationsmechanismus zum Aufbau eines Tiefenmodells für Organisations- und Anwendungswissen innerhalb eines Büros dar. Diese **terminologische Adäquatheit** (vgl. /Brachman,Levesque 82/) erleichtert den inkrementellen Aufbau großer Wissensbasen und unterstützt die Strukturierung des Wissens auf natürliche Art und Weise. Strukturierte Vererbungsnetze stellen ganz spezielle und sehr effiziente Inferenztechniken zur Verfügung, die in einem Problemlösungsprozeß zum gezielten Einschränken des Suchraumes ausgenützt werden können. Darunter fallen alle Arten von **Vererbung, Defaultmechanismen, Typisierung** von Slotwerten sowie die Ausnutzung der **Transitivität** der ausgezeichneten Strukturierungsrelationen wie SUBCLASS, PART-OF, etc. (vgl. Abschnitt 1.2.).

In LUIGI lassen sich **Objekte** zum einen in **Typen** und zu Typen gehörige **Werte**, zum anderen in **strukturierte Objektklassen** und zugehörige **Objektinstanzen** unterscheiden. Typen sind vordefinierte Wertemengen wie *integer*, *number*, *string*, *symbol* oder *list* sowie benutzerdefinierte Untermengen oder explizite Aufzählungen von Werten. Strukturierte Objekte setzen sich im allgemeinen aus mehreren Konstituenten (***slots***) zusammen, die durch einen Namen identifiziert werden. Für eine Objektklasse ***O*** ist eine Konstituente ***K*** mit folgenden Attributen versehen:

- ***K*** ist mit einer Wertemenge oder einer anderen Objektklasse **typisiert**, die die möglichen Werte für ***K*** in einer Instanz von ***O*** einschränkt;
- ein **Anzahlattribut** legt fest, wieviele Werte des mit ***K*** assoziierten Typs ***K*** in einer Instanz von ***O*** aufnehmen kann (genau einen, genau *n*, eine Zahl aus dem Intervall [*n..m*] oder die nach oben offenen Intervalle * und +);
- ein **Existenzindikator** für ***K*** legt fest, ob in einer Instanz von ***O*** ein Wert für ***K*** angegeben werden muß, oder ob er *undefiniert* bleiben kann;
- ***K*** kann mit einem optionalen **Defaultwert** versehen werden, der in einer Instanz von ***O*** als Wert für ***K*** eingetragen wird, wenn kein Wert explizit angegeben wird.

Für Konstituenten können weder Prozeduren als Werte noch "Zugriffsdämonen" spezifiziert werden.

Objektinstanzen, also die konkreten Ausprägungen der in einer Klasse beschriebenen Objekte tragen Konstituenten mit gleichen Namen wie die der zugehörigen Klasse. Diese Konstituenten sind nun aber mit **Werten** bzw. **Instanzen** anderer Klassen assoziiert, die den jeweiligen Attributen der Konstituenten der Klassendefinition genügen.

Klassen sind wie in KL-ONE in einer Taxonomie angeordnet, in der die Subklassen-/Superklassen-Relation nicht beliebig gesetzt werden kann, sondern durch eine auf der Grundlage der Klassenstruktur definierten Subsumptionsrelation festgelegt ist. Dadurch kann die Taxonomie automatisch durch eine **Klassifikatorkomponente** konsistent gehalten werden (vgl. /Schmolze,Lipkis 83/, /Bach 88/).

3.2. Epistemische Primitiva in ARI

In diesem Abschnitt werden die Sprachkonstrukte von *ARI* beschrieben, die nach dem vorgestellten Modell verschiedener Abstraktionsebenen die **epistemologische Ebene** des Inferenzsystems ausmachen. Es soll dabei vor allem eine Methodik deutlich werden, wie mit Hilfe der *ARI*-Sprache strukturierte Regelbasen zur Lösung komplexer Probleme aufgebaut werden können. Eine vollständige Beschreibung der Sprachsyntax findet sich in **Anhang A.** Die epistemischen Primitiva in *ARI* sind:

(1) Konstrukte zur **Organisation** und **Strukturierung** von Regelmengen,

(2) Konstrukte zur Repräsentation von **Kontrollwissen** auf verschiedenen Metaebenen einer allgemeinen Metaebenen-Architektur, und

(3) Konstrukte zur Repräsentation von **Inferenzwissen** auf der Objektebene eines Produktionsregelsystems.

Im folgenden werden die einzelnen Konstrukte der *ARI*-Sprache im Sinne der angesprochenen Methodik vorgestellt, zuvor sollen jedoch noch die Grundlagen der *ARI*-Mustersprache in einer abstrakten Sichtweise dargelegt werden. In Abschnitt 3.2.6. werden die Konstrukte detailliert beschrieben und an einem Beispiel erläutert.

3.2.1. Grundlagen der Mustersprache

Muster (***pattern***) sind elementare Sprachkonstrukte der *ARI*-Sprache. *Pattern* erlauben durch die Verwendung von Variablen eine kompakte und deklarative Spezifikation ganzer Klassen von Wissenseinheiten. Durch eine Menge logisch verknüpfter *pattern* für Wissenseinheiten lassen sich Klassen von **Situationen** spezifizieren, wobei eine Situation in *ARI* durch die in ihr existierenden bzw. nicht existierenden Wissenseinheiten charakterisiert wird. Situationsbeschreibungen lassen sich also als **Existenzbedingungen** für Wissenseinheiten auffassen.

> Eine **Situationsbeschreibung** besteht aus einer (implizit) **konjunktiv verknüpften** Folge von **Bedingungselementen** (***condition-elements***). Bedingungselemente sind *pattern* der *ARI*-Mustersprache. Sie können **negiert** sein, wobei das NOT-Konstrukt jedoch nicht die Bedeutung der logischen Negation trägt, sondern es wird wie z.B. in PROLOG im Sinne einer **closed-world-assumption** benutzt (vgl. /Clark 78/).

Bei der Interpretation der Sprachkonstrukte werden Situationsbeschreibungen mit einer Menge von Wissenseinheiten verglichen. Dieser Prozeß wird als ***pattern-matching*** bezeichnet. Entspricht eine Wissenseinheit der Beschreibung eines *patterns*, wird dies als **match** bezeichnet, wobei die Variablen des *patterns* an die konkreten Werte der Wissenseinheit gebunden werden. Die bei einem *match* aufgebauten Variablenbindungen werden als **Instanziierung** des *patterns* bezeichnet.

Damit läßt sich formulieren, wann eine Situationsbeschreibung bezüglich einer Situation, die durch eine Menge W_S von Wissenseinheiten gegeben ist, **erfüllt** ist.

> Ein **nichtnegiertes Bedingungselement** ist bezüglich einer Situation W_S genau dann **erfüllt**, wenn eine Wissenseinheit in W_S existiert, mit der ein **match** berechnet werden kann.
>
> Ein **negiertes Bedingungselement** ist bezüglich einer Situation W_S genau dann **erfüllt**, wenn keine Wissenseinheit in W_S existiert, mit der ein **match** berechnet werden kann.
>
> Eine **Situationsbeschreibung** ist bezüglich einer Situation W_S genau dann **erfüllt**, wenn **alle** ihre Bedingungselemente bezüglich W_S erfüllt sind und die beim *pattern-matching* der

einzelnen Bedingungselemente aufgebauten **Bindungen** alle untereinander **verträglich** sind, d.h. gleiche Variablen sind an gleiche Werte gebunden (***konsistente Bindung***).

Durch die Verwendung einheitlicher Situationsbeschreibungen werden in den verschiedenen Konstrukten der *ARI-Patternsprache* spezielle Wissensstrukturen stets mit dem gleichen *pattern* beschrieben, was sowohl die Erstellung als auch das Verständnis und die Wartbarkeit einer Regelbasis erleichtert.

In der derzeitigen Version von *ARI* sind *Objektinstanzen* und *Sachverhalte* die einzigen Wissensstrukturen, auf die in Situationsbeschreibungen Bezug genommen werden kann. Dies ist jedoch keine prinzipielle Einschränkung, da das LUIGI-System alle Wissenseinheiten einheitlich repräsentiert und *ARI* somit durch Ergänzung von epistemologischer und Implementierungsebene um Schlußfolgern über *Objektklassen*, *Pläne* oder *Aktivitäten* erweitert werden kann.

3.2.2. Konstrukte zur Organisation und Strukturierung

3.2.2.1. Problem-Solver

Der zentrale Mechanismus zur Organisation von Regelbasen in *ARI* ist die Wissensstruktur **problem-solver**. Im Gegensatz zu Systemen, die *Frames* als reine Datenstrukturen zur Strukturierung großer Regelmengen benutzen (vgl. /Fikes,Kehler 85/, /Aikins 83/), werden *problem-solver* von der Inferenzmaschine als **konzeptuelle und funktionale Einheiten interpretiert.**

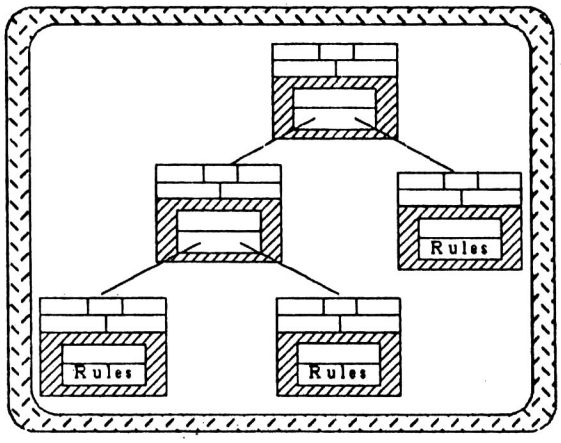


Abb. 3.2: Eine ARI-Regelbasis aus der "Vogelperspektive"

Problem-solver verallgemeinern das Konzept des strukturierten Regelkomplexes aus CATWEAZLE in mehrfacher Hinsicht. Sie erlauben die adäquate und natürliche Modellierung von komplexen Problemlösungsprozessen, die sich über mehrere Stufen hinweg hierarchisch in Teilprozesse aufteilen lassen, die jeweils Teilprobleme lösen. Zu diesem Zweck existieren zwei Arten von *problem-solvern*:

- **einfache problem-solver** (***simple problem-solvers***) interpretieren Produktionsregeln und werden zur Lösung einer Klasse von Problemen spezifiziert,
- **komplexe problem-solver** (***complex problem-solvers***) setzen rekursiv die Teillösungen einer Menge von *sub-problem-solvern* zusammen, wobei diese sowohl einfache als auch komplexe *problem-solver* sein können.

Dieses allgemeine Organisationsprinzip ergibt für *ARI*-Regelbasen eine Struktur wie in Abb. 3.2.

Beide Arten von *problem-solvern* sind komplex strukturierte Einheiten, sie sich lediglich in der Form ihres **Schlußfolgerungswissens** (***reasoning knowledge***), also des **Kontrollwissens** (***control knowledge***) und des **Inferenzwissens** (***inferential knowledge***) unterscheiden. Ein *problem-solver* setzt sich allgemein wie in der folgenden Abb. 3.3. dargestellt zusammen.

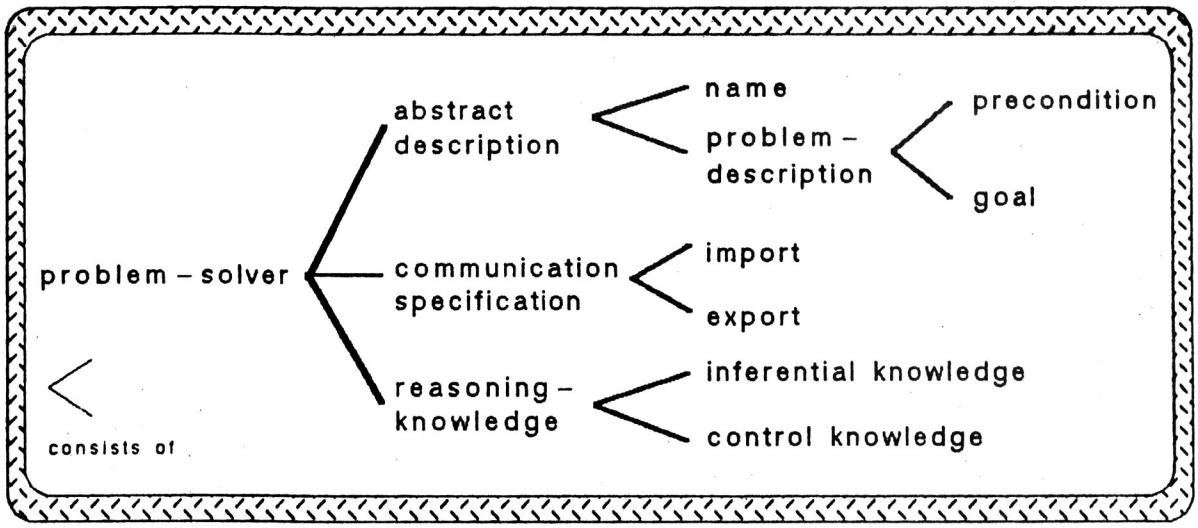


Abb. 3.3: Die Struktur allgemeiner problem-solver

Inferenzwissen in *simple problem-solvern* umfaßt eine Menge **OPS5**-artiger **Objektregeln** (***domain rules***), Kontrollwissen besteht aus einer Menge von **Kontrolltaktiken** (***control-tactics***) in Form von Meta-Regeln wie sie bereits in **CATWEAZLE** verwendet wurden. *Complex problem-solver* haben als Inferenzwissen eine Menge von *sub-problem-solvern*, das Kontrollwissen ist eine Menge von **Kontrollstrategien** (***control-strategies***) oder eine **Phasenfolge**, analog zu den Konstrukten auf oberster Ebene der **CATWEAZLE**-Architektur. Phasenfolgen sind in der derzeitigen Version des ***ARI***-Systems noch nicht realisiert und werden deshalb hier nicht weiter beschrieben. Die beiden Arten von *problem-solvern* als Spezialisierung des allgemeinen Konzeptes *problem-solver* zeigt Abb. 3.4.

3.2.2.2. Abstrakte Beschreibungen

Die **abstrakte Beschreibung** eines *problem-solvers* besteht aus seinem **Namen** und einer **Problembeschreibung** (***problem-description***), die eine Klasse von (Teil-)Problemen spezifiziert, für die der *problem-solver* als "*Problemlöser*" vorgesehen ist. Eine Problembeschreibung umfaßt

- eine **Vorbedingung** (***precondition***), die mit Konstrukten der Mustersprache alle Situationen beschreibt, in denen der *problem-solver* aktiviert werden kann, und
- eine **Zielbeschreibung** (***goal***), die spezifiziert, in welchen Situationen das von ihm bearbeitete Problem gelöst ist.

Die Aktivierung eines *problem-solvers* ist mit einer Instanziierung von dessen Vorbedingung verbunden. Der Geltungsbereich der in der Vorbedingung gebundenen Variablen umfaßt die gesamte abstrakte Beschreibung sowie die Kommunikationsspezifikation (siehe Abschnitt 3.2.2.3.). Damit bearbeitet ein aktivierter *problem-solver* genau eine **Probleminstanz**.

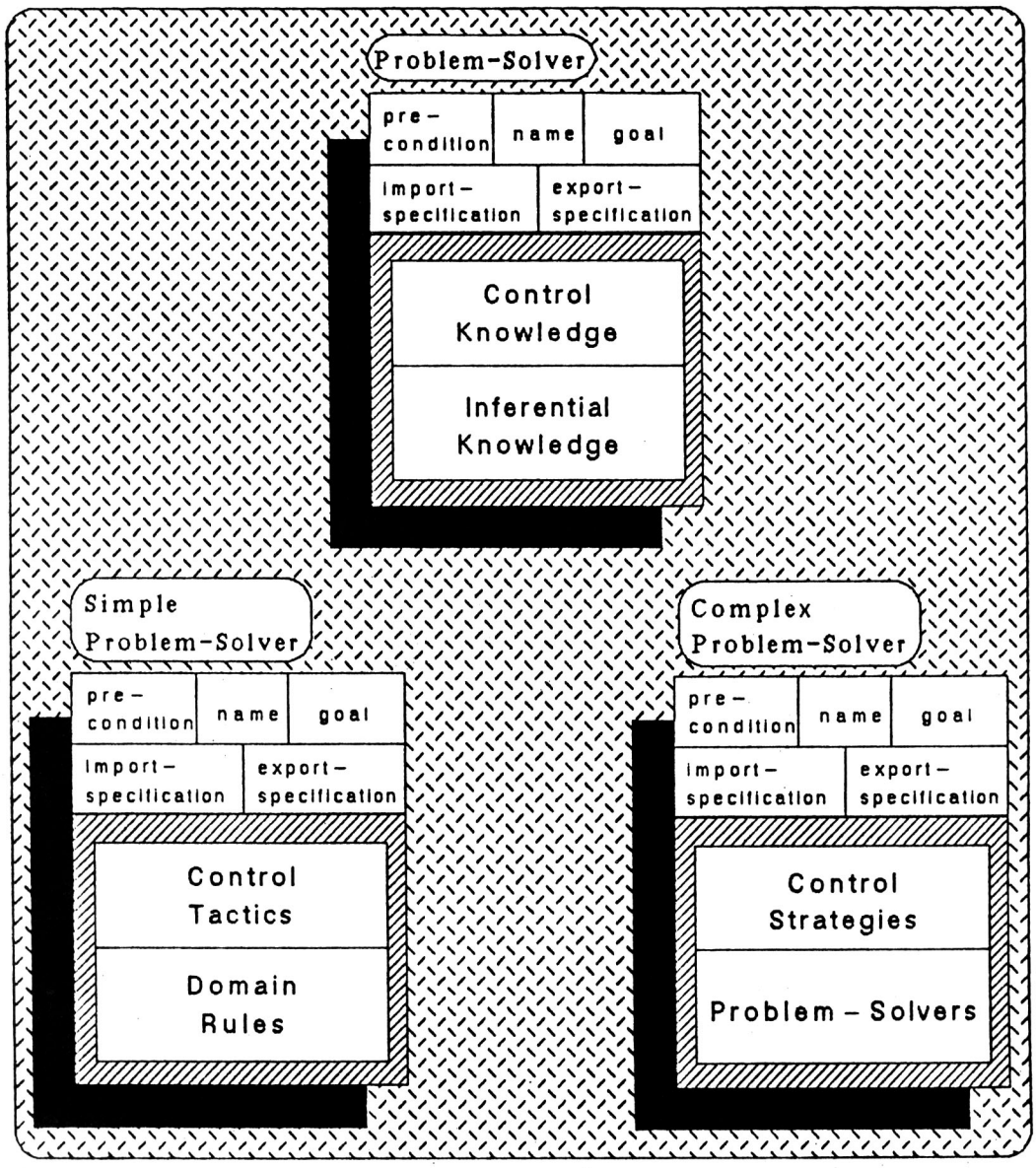


Abb. 3.4: Arten von problem-solvern

Man beachte, daß diese abstrakte Betrachtungsweise von *problem-solvern* als "*Teilproblemlöser*" erst durch die explizite Repräsentation von Vorbedingung und Zielbeschreibung im Sinne der Deklarativitätsprämisse gerechtfertigt wird. Die abstrakte Beschreibung erfüllt aber auch einen anderen Zweck. Durch sie können *problem-solver* in der Metaebenen-Architektur von ***ARI*** selbst zum Objekt bzw. Gegenstand des Schlußfolgerungsprozesses auf einer entsprechenden Metaebene werden (s. Abschnitt 3.2.3.2.).

3.2.2.3. Kommunikationsspezifikationen

Neben der Tatsache, daß das Partitionieren einer Regelbasis gemäß der Dekomposition von Problemlösungsprozessen eine natürliche Strukturierung induziert, kann dadurch auch rein technisch gesehen die Interpretation innerhalb eines Produktionsregelsystems effizienter realisiert werden, da in der ***recognize***-Phase des Interpreters nur die zu diesem Zeitpunkt relevanten Regeln betrachtet werden müssen. ***ARI*** verallgemeinert dieses Prinzip, indem es auch die beim *pattern-matching* betrachteten Daten, also die vorhandenen Wissenseinheiten, auf die jeweils relevanten beschränkt.

Zu diesem Zweck wird die **Kommunikation** zwischen *problem-solvern* in Form von **Import-** und **Exportspezifikationen** ebenfalls explizit beschrieben. Innerhalb eines *problem-solvers* wird dadurch eine **Umgebung** sichtbarer Wissenseinheiten für den *problem-solver* aufgebaut. Zu dieser Umgebung gehören während der Interpretation alle importierten Wissenseinheiten, sowie alle Wissenseinheiten, die erzeugt, modifiziert oder gelöscht werden, während der *problem-solver* aktiviert ist (siehe Abschnitt 3.2.3.).

Neben dem technischen Aspekt des *pattern-matching* unterstützt diese explizite Spezifikation den systematischen Entwurf wissensbasierter Systeme und vereinfacht deren Testen. In diesem Sinne ist die explizite Formalisierung minimaler Schnittstellen zwischen Systemkomponenten bereits zu einem klassischen Paradigma im Bereich des Software Engineering geworden.

Abstrakte Beschreibung und Kommunikationsspezifikation von *problem-solvern* machen diese zu **funktionalen Einheiten**, die als *"Black-Box"* mit formalisierter Funktionalität betrachtet werden können. Dies ermöglicht unter anderem die Entwicklung mächtiger **Erklärungskomponenten** für wissensbasierte Systeme auf der Grundlage von *ARI*, die über das übliche ***"Tracen"*** von Regelanwendungen weit hinausgehen können.

3.2.3. Regeln in ARI

Neben *problem-solvern* als Mechanismen zur Organisation und Strukturierung von Regelbasen sind verschiedene Formen von Regeln die wichtigsten epistemische Primitiva der *ARI*-Sprache:

- **Objektregeln** (***domain-rules***) sind die **Inferenzregeln**, die auf *Sachverhalten* und *Objektinstanzen* des beschreibenden Wissen der LUIGI-Wissensbasis operieren;
- **Kontrolltaktiken** (***control-tactics***) sind **Meta-Regeln** über Objektregeln und dienen zur Konfliktauflösung zwischen verschiedenen gleichzeitig anwendbaren Objektregeln;
- **Kontrollstrategien** (***control-strategies***) sind **Meta-Regeln** über *problem-solvern*, die eine Auswahl zwischen gleichzeitig anwendbaren *problem-solvern* treffen.

Alle drei Typen von Regeln in *ARI* bestehen aus einem **Bedingungsteil** (***condition-part***), der eine Situationsbeschreibung aus einer Folge von Bedingungselementen umfaßt, und einem **Aktionsteil** (***action-part***) mit einer Folge für den jeweiligen Regeltyp vordefinierter **Aktionen**. Sie unterscheiden sich lediglich in der Art der Bedingungselemente und in den Aktionen.

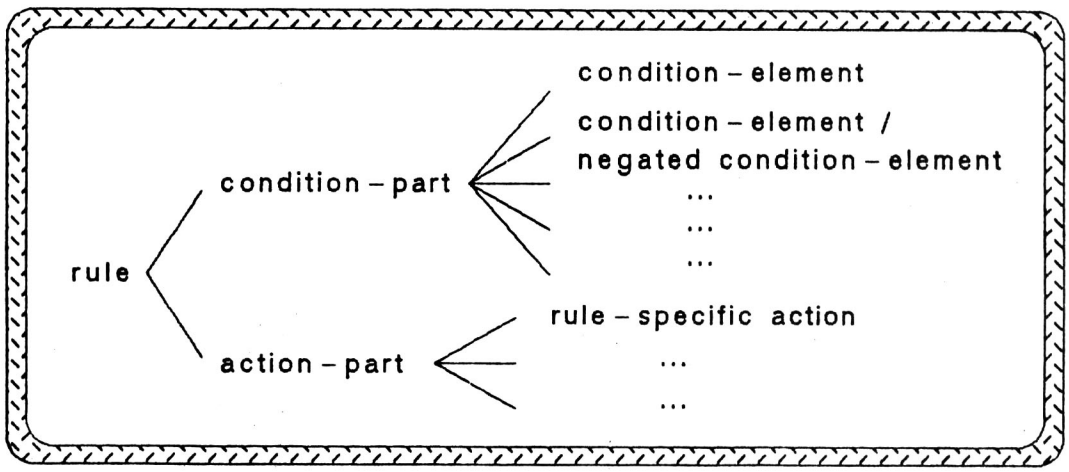


Abb. 3.5: Allgemeine Struktur von Regeln

Die **Anwendbarkeit** von Regeln während der Interpretation läßt sich auf die **Erfüllbarkeit** von Situationsbeschreibungen zurückführen:

> Eine **Regel** ist während der Interpretation genau dann **anwendbar**, wenn sie zum Schlußfolgerungswissen des zu diesem Zeitpunkt aktivierten *problem-solvers* ***P*** gehört, und die Situationsbeschreibung ihres Bedingungsteils zu diesem Zeitpunkt bezüglich der **Umgebung** sichtbarer Wissenseinheiten innerhalb von ***P*** **erfüllt** ist.

3.2.3.1.Inferenzregeln

Inferenzregeln sind die grundlegenden Repräsentationen für Schlußfolgerungsprinzipien. Sie operieren auf *Objekten* und *Sachverhalten* einer **LUIGI**-Wissensbasis. Inferenzregeln sind in ***ARI*** dem Wesen nach keine Deduktionsregeln, die allgemeingültige logische Schlüsse ziehen und nur Wissen zu einer Wissensbasis hinzufügen, sondern sie führen Operationen auf dieser Wissensbasis aus, die typischerweise auch Wissenseinheiten modifizieren und löschen.

Situationsbeschreibungen im **Bedingungsteil** von können Inferenzregeln drei verschiedene Arten von Bedingungselementen enthalten:

(1) **object-pattern** sind Muster für *Objektinstanzen*, die durch ihren *Namen*, den Namen ihrer zugehörigen *Klasse*, *Werte* von Konstituenten sowie *Prädikaten* über diesen Konstituentenwerten beschrieben werden können.

(2) **affair-pattern** sind Muster für *Sachverhalte*, die durch ihren *Sachverhaltsnamen* und durch *Werte* für die Argumente beschrieben werden können.

(3) **test-pattern**, stellen keine *pattern* im eigentlichen Sinne dar, sondern spezifizieren durch vordefinierte Prädikate *Tests* auf Bindungen für Variablen und dienen somit als *Filter für Instanziierungen*.

Die Operationen, die im **Aktionsteil** spezifiziert werden können, nehmen durch Variablen auf die im Bedingungsteil gebundenen Wissenseinheiten Bezug. In ***ARI*** gibt es außerdem die Möglichkeit, im Aktionsteil lokal neue Variablenbindungen aufzubauen und zwar durch ein ***let***- bzw. ein ***let****-Konstrukt, wie es in LISP verwendet wird. Werte für die Variablen werden dabei direkt von beliebigen LISP-Funktionen berechnet. Dieser vollständige Durchgriff auf den LISP-Interpreter ist zum einen ein sehr mächtiger Mechanismus, zum anderen birgt er natürlich die Gefahr, daß der deklarative Charakter von Inferenzregeln durch die Verwendung von Funktionen mit Seiteneffekten vollends verloren geht. Nützlich aber ist der Mechanismus sicherlich dann, wenn man sich auf eine Menge fest vordefinierter, seiteneffektfreier Funktionen beschränkt, um z.B. eine Schnittstelle zu Inferenzmechanismen innerhalb des Vererbungsnetzwerkes der **LUIGI**-Taxonomie zu schaffen. Im Rumpf eines ***let***-Konstruktes können nur **elementare Aktionen** verwendet werden, eine Schachtelung von ***let***-Konstrukten ist nicht möglich, um komplexe Programme auf den rechten Seiten von Regeln zu vermeiden. Im einzelnen gibt es in ***ARI*** folgende **elementare Aktionen** :

(1) **assert-affair** etabliert einen neuen Sachverhalt, **retract-affair** invalidiert einen Sachverhalt, der im Bedingungsteil oder in einem umgebenden ***let***-Konstrukt gebunden wurde.

(2) **create-object** und **delete-object** erzeugen bzw. löschen analog dazu Objektinstanzen. Zusätzlich erlaubt **modify-object** die Modifikation von Konstituentenwerten einer bereits bestehenden Objektinstanz.

(3) Zur Benutzerinteraktion stehen momentan die Aktionen **yes-or-no-query** zum Erfragen eines booleschen Wertes, **select-query** zur Auswahl eines Wertes aus einer vorgegebenen Wertemenge, **query-variable-value** zum Erfragen eines beliebigen Wertes und schließlich **display** zur formatierten Ausgabe zur Verfügung.

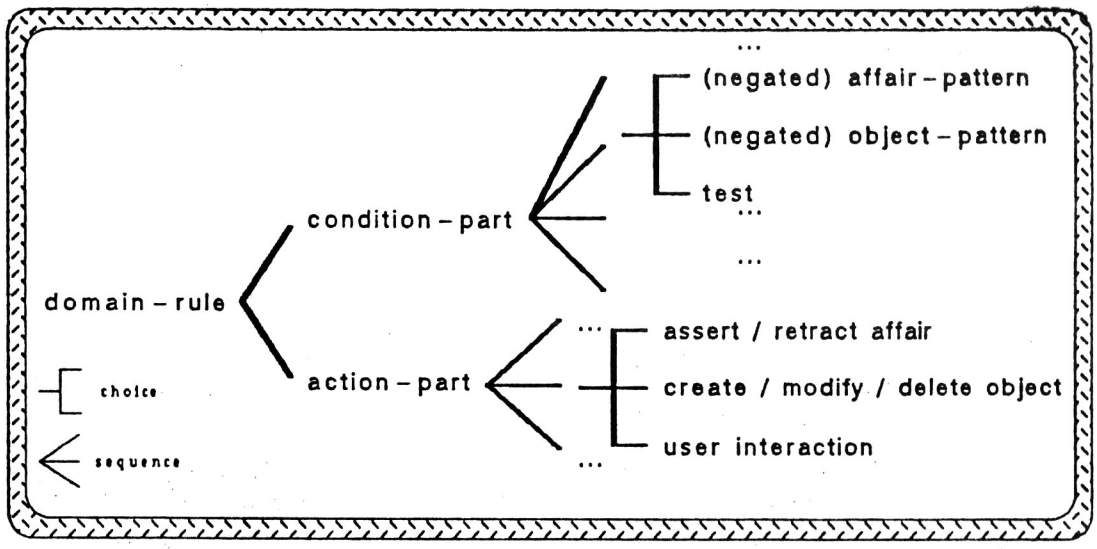


Abb. 3.6: Struktur von Objektregeln

3.2.3.2. Kontrolltaktiken

Kontrolltaktiken als Meta-Regeln treffen eine Auswahl zwischen gleichzeitig anwendbaren Objektregeln und formalisieren damit Kontrollwissen zur Konfliktauflösung. Im Bedingungsteil muß deshalb auf die Beschreibung anwendbarer Objekt- oder Inferenzregeln Bezug genommen werden. Anwendbare Objektregeln werden auch als **Regelinstanzen** bezeichnet, da in ihnen alle in den *pattern* der Regeln vorkommenden Variablen durch *pattern-matching* an einen Wert gebunden sind.

Regelinstanzen können in der Mustersprache als **Objektregelmuster** (*domain-rule patterns*) beschrieben werden. *Domain-rule-pattern* beschreiben Inferenzregeln, indem sie Bedingungselemente bzw. Aktionen der Regeln mit genau den selben Mustern beschreiben, die auch diese verwenden. Außerdem können in Kontrolltaktiken alle Bedingungselemente auftreten, die auch im Bedingungsteil von Objektregeln verwendet werden, um auch Kontrolltaktiken situationsspezifisch bezüglich des beschreibenden Wissens formulieren zu können. Situationsbeschreibungen im Bedingungsteil von Kontrolltaktiken beinhalten aber mindestens ein *domain-rule-pattern*.

Der Aktionsteil einer Kontrolltaktik besteht aus genau einer Operation, für die es nur zwei mögliche elementare Aktionen gibt:

- **suspend-domain-rule** kennzeichnet eine anwendbare Regel als "*suspendiert*" und verhindert deren Anwendung bis diese Suspendierung wieder aufgehoben ist.
- **prefer-domain-rule** kennzeichnet eine anwendbare Regel als "*vorrangig*". Vorrangige Objektregeln werden im ***apply***-Zyklus des Interpreters bevorzugt vor anderen anwendbaren Regeln angewendet, unter einer Menge als vorrangig gekennzeichneter Regeln wird eine beliebige Auswahl getroffen.

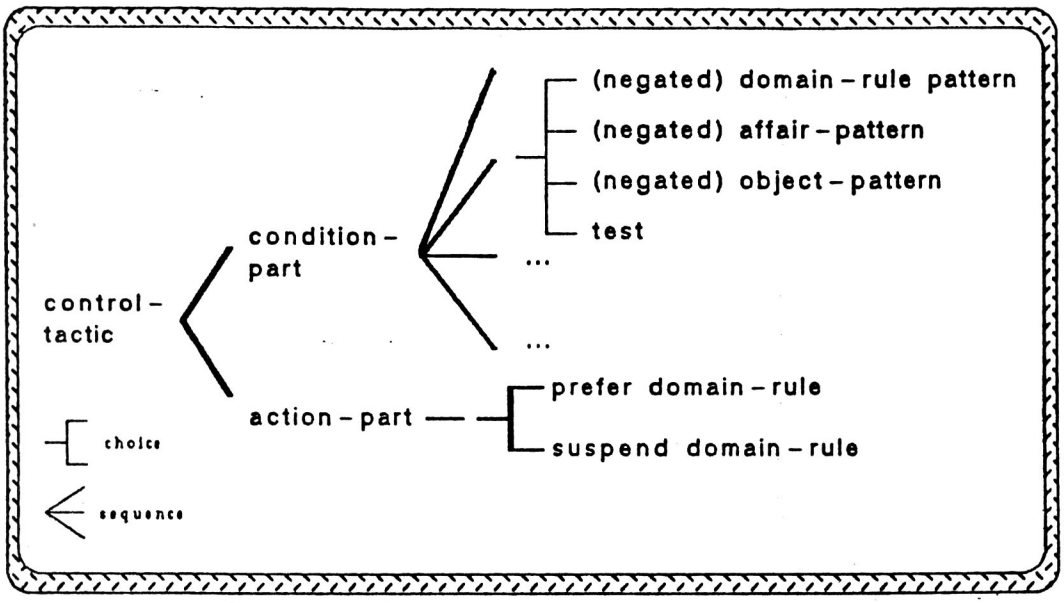


Abb. 3.7: Struktur von Kontrolltaktiken

Die Mechanismen zur Auswahl zwischen Objektregeln sind damit bewußt denkbar einfach gehalten. Ein Konzept zur Formulierung und Verwaltung von mehrstufigen Prioritäten könnte hier mehr Ausdrucksstärke bringen, es bestünde aber auch die Gefahr, daß dann Kontrollwissen nicht mehr deklarativ repräsentiert, sondern implizit in numerische Prioritätsberechnungen codiert wird. Für *ARI* sollen künftige Anwendungen die Anforderungen für ein erweitertes Konzept bestimmen.

Eine Beispielregel in abstrakter Formulierung mag die Verwendung von Kontrolltaktiken verdeutlichen:

```
KONTROLL-TAKTIK AUSGABEN_MINIMIEREN
WENN eine Regel anwendbar ist, die
       in einer Vorbedingung
         auf einen Sachverhalt Bezug nimmt, der
           eine Abteilung für ein Land zuständig erklärt
         und
         auf eine Anfrage aus diesem Land Bezug nimmt
       und
       im Aktionsteil
         einen Sachverhalt etabliert, der
           diese Abteilung als zuständig für die Anfrage erklärt
    UND
     eine Regel anwendbar ist, die
       in einer Vorbedingung
         auf einen Sachverhalt Bezug nimmt, der
           eine Abteilung für ein Sachgebiet zuständig erklärt
         und
         auf eine Anfrage zu diesem Sachgebiet Bezug nimmt

       und
       im Aktionsteil
         einen Sachverhalt etabliert, der
           diese Abteilung als zuständig für die Anfrage erklärt
DANN
 bevorzuge die erste Regel
```

Die Kontroll-Taktik AUSGABEN_MINIMIEREN soll die Auswahl von Regeln kontrollieren, die in einer Firma eine Abteilung für die Beantwortung von Kundenanfragen ermitteln. Dabei sollen z.B. aus Gründen der Kostenersparnis Regeln bevorzugt werden, die eine Abteilung vorschlagen, die für das Land zuständig ist, aus dem eine Anfrage vorliegt.

Man beachte, daß die Kontrolltaktik dabei nicht explizit beschreiben muß, ob noch andere und wenn ja, welche Bedingungen noch in die anwendbaren Objektregeln eingeflossen sind. Objektbeschreibungsmuster müssen lediglich einen Teil des Bedingungs- bzw. des Aktionsteils beschreiben und die Reihenfolge von Bedingungselementen ist dabei irrelevant. Der Vorteil der Verwendung expliziter Meta-Regeln zur Formulierung solcher Kontrollentscheidungen liegt darin, daß die Änderung einer Meta-Regel ohne Änderung der betroffenen Objektregeln eine Modifikation der Kontrolle des Problemlösungsprozesses erlaubt.

3.2.3.3. Kontrollstrategien

Kontrollstrategien dienen zur Konfliktlösung zwischen anwendbaren *problem-solvern* und werden analog zu Kontrolltaktiken interpretiert. Sie nehmen dabei durch ***problem-solver-pattern*** auf aktivierbare Instanzen von *problem-solvern* Bezug. *Problem-solver-pattern* benutzen dabei im Vergleich zu *domain-rule-pattern* die Vorbedingung von *problem-solvern* anstelle des Bedingungsteils und die Zielbeschreibung anstelle des Aktionsteils von Objektregeln. Im Aktionsteil von Kontrollstrategien sind **suspend-problem-solver** und **prefer-problem-solver** die analogen Operationen zu **suspend-domain-rule** und **prefer-domain-rule**.

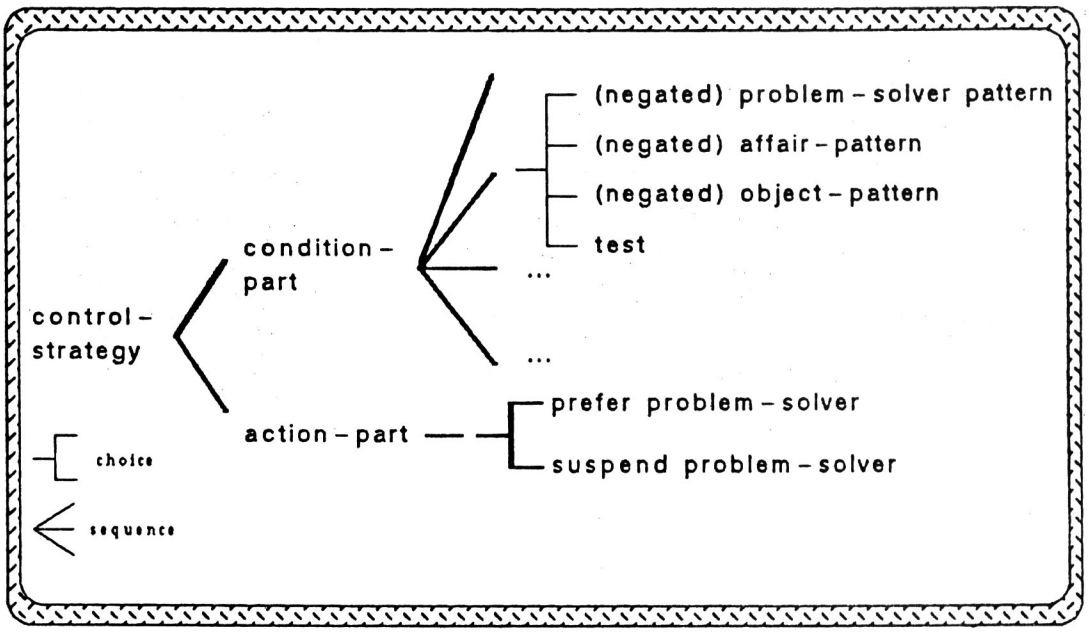


Abb. 3.8: Struktur von Kontrollstrategien

Als Beispiel soll hier wieder eine abstrakt formulierte Kontrollstrategie das Konzept von Kontrollstrategien erläutern. Die Kontrollstrategie BETONE_KUNDENBETREUUNG formalisiert eine Firmenpolitik, die vorgibt, daß eine gewisse Klasse von Anfragen zuerst beratend durch den Kundendienst der Firma erledigt wird, anstatt sofort den Vertrieb einzuschalten, um möglicherweise weitere Produkte zu verkaufen.

```
KONTROLL-STRATEGIE BETONE_KUNDENBETREUUNG
WENN ein Problem-Solver anwendbar ist, der
       zum Ziel hat
         einen zuständigen Vertriebsbeauftragten zu bestimmen
   UND
     ein Problem-Solver anwendbar ist, der
       zum Ziel hat
         einen zuständigen Kundendienstbeauftragten zu bestimmen
   UND
    eine Anfrage nach 'Hardwareanpassungen' vorliegt
DANN
 bevorzuge den zweiten problem-solver
```

Sollte das Management der betreffenden Firma einmal entscheiden, daß eine agressive Verkaufspolitik wichtiger ist, als die intensive Betreuung der Kunden, so ließe sich diese geänderte Firmenpolitik durch Modifikation der Kontrollstrategie, die eine explizite Repräsentation dieser Firmenpolitik darstellt, in der Regelbasis nachvollziehen.

Man beachte wiederum, daß die in der Kontrollstrategie beschriebenen *problem-solver* keinesweges genau die geforderte Zielbeschreibung tragen müssen, sondern daß sie durchaus speziellere Zielformulierungen haben können, wie zum Beispiel das Bestimmen eines Kundendienstbeauftragten für Hardware. *Problem-solver-pattern* genau wie *domain-rule-pattern* werden also durch alle spezielleren konkreten Ausprägungen von *problem-solvern* bzw. *domain-rules* erfüllt. Das Beispiel zeigt auch, daß *pattern* für beschreibendes Wissen, in diesem Fall eine Objektinstanz der Klasse ANFRAGE mit einem Konstituentenwert 'Hardwareanpassung', direkt im Bedingungsteil von Kontrollstrategien auftreten können.

3.2.4. Das Interpretationsmodell der Inferenzmaschine

Zur adäquaten Modellierung von Problemlösungsprozessen mit den vorgestellten Sprachkonstrukten auf epistemologischer Ebene des *ARI*-Inferenzsystems muß ein abstraktes Interpretationsmodell für diese Konstrukte vorgegeben sein. Dieses Operationsprinzip soll hier ausschließlich durch Bezugnahme auf die beschriebenen epistemischen Primitiva und ohne Berücksichtigung von Mechanismen der Implementierungsebene von *ARI* beschrieben werden.

Eine *ARI*-Regelbasis besteht auf oberster Ebene aus genau einem *complex-problem-solver*, der weitere *problem-solver* als Schlußfolgerungswissen enthält. Dieser wird im folgenden als **top-level-problem-solver** bezeichnet.

Zu jedem Zeitpunkt der Interpretation gibt es **genau einen aktivierten** *problem-solver*, der von der Inferenzmaschine interpretiert wird. Für jeden *problem-solver* existiert eine **Umgebung**, die alle für ihn **sichtbaren** Wissenseinheiten enthält, die **aktuelle Umgebung** ist die Umgebung des aktivierten *problem-solvers*.

Zu Beginn der Interpretation agiert die Inferenzmaschine selbst als aktivierter *problem-solver*. Eine **globale Umgebung** ist die aktuelle Umgebung. Sie besteht aus der gesamten LUIGI-Wissensbasis für beschreibendes Wissen, sowie einer **globalen Problemdefinition**. Die Problemdefinition wird ebenfalls mit LUIGI-Wissensstrukturen formuliert.

Der **Kontrollfluß** der Interpretation kann nun wiefolgt mit Hilfe der Begriffe des **aktivierten** *problem-solvers* sowie der **aktuellen Umgebung** formuliert werden:

- der Interpreter kann im **Ausgangszustand** den top-level-problem-solver P_{top} **aktivieren**, falls dessen Vorbedingung in der globalen Umgebung erfüllt ist;
- der Interpreter kann zu jedem späteren Zeitpunkt der Interpretation einen *problem-solver* P_i genau dann **aktivieren**, wenn dieser ein *sub-problem-solver* eines aktivierten *problem-solvers* P ist und die **Vorbedingung** von P_i in der **aktuellen Umgebung** von P erfüllt ist;
- ist zu einem Zeitpunkt der Interpretation kein *problem-solver* aktivierbar, so wird die Interpretation mit einem **Fehler** abgebrochen;
- bei der Interpretation eines aktiverten *problem-solvers* wird das Schlußfolgerungswissen durch iteriertes Durchlaufen eines ***recognize-select-act***-Zyklus abgearbeitet;
- ein aktivierter *problem-solver* P_i wird vom Interpreter **deaktiviert**, wenn die **Zielbeschreibung** von P_i vor einem Durchlauf des Interpretationszyklus in der **aktuellen Umgebung** von P_i erfüllt ist. Gleichzeitig wird der P_i übergeordnete *problem-solver* P aktiviert. Bei der Deaktivierung des top-level-problem-solvers P_{top} wird die Interpretation der Regelbasis beendet;
- läßt sich im Interpretationszyklus fur das Schlußfolgerungswissen von P_i **kein Inferenzwissen mehr anwenden**, ist aber gleichzeitig die **Zielbeschreibung** von P_i in der **aktuellen Umgebung** von P_i nicht erfüllt, so wird die Interpretation mit einem **Fehler** abgebrochen.

Diese Form von Kontrollfluß entspricht dem Interpretationsmodell für höhere prozedurale Programmiersprachen mit *problem-solvern* als Äquivalente zu Prozeduren.

Der **Datenfluß** bei der Interpretation läßt sich auf ähnliche Weise als **Kommunikation** zwischen *problem-solvern* formulieren:

- die mit der Aktivierung eines *problem-solvers* P_i verbundene **Instanziierung der Vorbedingung** von P_i bewirkt eine **Teilinstanziierung** von **Zielbeschreibung**, sowie der **Import-** und **Exportspezifikation** von P_i ;
- anschließend wird für P_i eine **aktuelle Umgebung** U_i aufgebaut, und zwar aus allen Wissenseinheiten der Umgebung U des übergeordneten *problem-solvers* P, die der teilinstanziierten Importspezifikation von P_i genügen;
- die **Interpretation des Schlußfolgerungswissens** von P_i erfolgt in der Umgebung U_i. Wissenseinheiten, die während dieser Interpretation erzeugt, gelöscht oder modifiziert werden bewirken ausschließlich eine Änderung der Umgebung U_i ;
- bei der **Deaktivierung** des *problem-solvers* P_i werden diejenigen Wissenseinheiten der Umgebung U_i von P_i, die der **Exportspezifikation** von P_i genügen, in die Umgebung U von P übernommen und U wird zur aktuellen Umgebung für den aktivierten *problem-solver* P ;

Das abstrakte Interpretationsmodell für *problem-solver* wird im folgenden ***Algorithmus 3.1*** ***INTERPRET_PROBLEM_SOLVER*** zusammengefaßt.* Dabei wird, wie auch im den folgenden Algorithmen auf Fehlerbehandlung nicht eingegangen:

```
INTERPRET_PROBLEM_SOLVER (PS, Environment, Precond-Bindings) is
  ; die beim Matchen der Vorbedingung aufgebauten Bindungen werden
  ; an goal-, import- und export-Spezifikation von PS weitergegeben:
 let goal-pattern   = INSTANTIATE (PS.goal-spec, Precond-Bindings)
     import-pattern = INSTANTIATE (PS.import-spec, Precond-Bindings)
     export-pattern = INSTANTIATE (PS.export-spec, Precond-Bindings)
    ; die in der Umgebung sichtbaren Wissenseinheiten, die der
    ; Import-Spezifikation genügen werden importiert:
    let visible-knowledge-units =
          MATCH_IMPORT (import-pattern, Environment)
      ; solange die Zielbeschreibung bezüglich der sichtbaren Wissenseinheiten
      ; nicht erfüllt ist, wird ein Interpretationszyklus ausgeführt. Dabei
      ; können neue Wissenseinheiten erzeugt, alte modifiziert oder gelöscht werden:
     while not MATCH_GOAL (goal-pattern, visible-knowledge-units) do
       let new-knowledge =
             INTERPRET_REASONING_KNOWLEDGE (PS, visible-knowledge-units)
        visible-knowledge-units :=
           UPDATE_ENVIRONMENT (new-knowledge,
                               visible-knowledge-units)
       end-let
     end-while
      ; die sichtbaren Wissenseinheiten, die der Export-Spezifikation von PS genügen,
      ; werden als Funktionswert zurückgeliefert:
     return-value MATCH_EXPORT (export-pattern,
                                visible-knowledge-units)
    end-let
 end-let
end INTERPRET_PROBLEM_SOLVER
```

***Algorithmus 3.1**: Interpretation eines problem-solvers*

Die **Interpretation des Schlußfolgerungswissens** (***INTERPRET_REASONING_KNOWLEDGE)*** verläuft wie bereits beschrieben für *simple problem-solver* in einem ***recognize-select-apply***-Zyklus, für *complex problem-solver* in einem analogen ***recognize-select-activate***-Zyklus.

* die verwendete algorithmische Beschreibungssprache benutzt ausschließlich Standardkonstrukte höherer Programmiersprachen, deren Bedeutung klar ist. Kommentare werden durch ein Semikolon (";") eingeleitet.

Ein Durchlauf durch diesen Zyklus umfaßt für einen *simple problem-solver* ***sPS*** folgende Schritte:

(1) ***recognize*** oder ***match***-Phase:
Unter den Inferenzregeln von ***sPS*** werden durch *pattern-matching* diejenigen bestimmt, die in der aktuellen Umgebung sichtbarer Wissenseinheiten **anwendbar** sind. Diese Menge der anwendbaren Objektregelinstanzen wird im allgemeinen auch als **Konfliktmenge** bezeichnet.

(2) ***select***-Phase:
Ebenfalls durch *pattern-matching* werden die **anwendbaren** Kontrolltaktiken von ***sPS*** bezüglich der Konfliktmenge anwendbarer Objektregelinstanzen und der aktuellen Umgebung bestimmt. Alle anwendbaren Kontrolltaktiken werden ausgeführt, d.h. bestimmte Objektregelinstanzen werden als "vorrangig" bzw. "suspendiert" gekennzeichnet. Gibt es nun vorrangige Objektregelinstanzen, so wird eine davon beliebig als anzuwendende Regel ausgewählt, ansonsten wird eine Objektregelinstanz ausgewählt, die nicht als suspendiert gekennzeichnet ist. Existiert keine solche Regel, so wird die gesamte Interpretation mit einem **Fehler** abgebrochen. Am (regulären) Ende dieser Phase ist also genau eine Objektregelinstanz aus der Konfliktmenge **ausgewählt** worden.

(3) ***apply-Phase:***
Die in der ***select***-Phase ausgewählte Regelinstanz wird **angewandt**, d.h. die Operationen im Aktionsteil dieser Regelinstanz werden ausgeführt und ändern sowohl das beschreibende Wissen im Objektsystem als auch die aktuelle Umgebung von ***sPS***. Das *pattern-matching* im nächsten Durchlauf wird dann auf der modifizierten aktuellen Umgebung ausgeführt.

Analog dazu läßt sich der Interpretationszyklus für das Schlußfolgerungswissen innerhalb eines *complex problem-solvers* ***cPS*** formulieren:

(1) ***recognize*** oder ***match***-Phase:
Bezüglich der aktuellen Umgebung wird durch *pattern-matching* mit den Vorbedingungen der *sub-problem-solver* von ***cPS*** eine **Konfliktmenge aktivierbarer Instanzen von *problem-solvern*** bestimmt.

(2) ***select***-Phase:
Analog zur Konfliktauflösung bei *simple-problem-solvern* wird durch Anwendung von Kontrollstrategien eine aktivierbare *problem-solver*-Instanz **ausgewählt**.

(3) ***activate-Phase:***
Die ausgewählte *problem-solver*-Instanz wird **aktiviert** und erhält die aktuelle Umgebung von ***cPS*** zum Bestimmen seines Importwissens. ***cPS*** wird selbst **deaktiviert** bis die Interpretation des aktivierten *problem-solvers* abgeschlossen ist. Diese Interpretation liefert neue bzw. modifizierte Wissenseinheiten zurück, die ***cPS*** zum Anpassen seiner aktuellen Umgebung benutzt. In dieser aktuellen Umgebung kann der nächste Durchlauf durch den Interpretationszyklus stattfinden.

Die Schritte eines Durchlaufs des Interpretationszyklus für allgemeine *problem-solver* sind im *Algorithmus 3.2. **INTERPRET_REASONING_KNOWLEDGE*** zusammengefaßt.

```
INTERPRET_REASONING_KNOWLEDGE (PS, Environment) is
 typecase PS
  of-type 'simple problem-solver' :
  ; im Falle eines simple problem-solvers werden zuerst alle anwendbaren Regeln und
  ; dann alle anwendbaren Kontrolltaktiken bestimmt. Mithilfe der Kontrolltaktiken wird die
  ; wird die Instanz einer Regel ausgewählt und anschließend interpretiert.
  ; Die Ausführung liefert neue, zu modifizierende oder zu löschende Wissenseinheiten zurück.
   let* applicable-domain-rules =
           MATCH_RULES (PS.domain-rules, Environment)
        applicable-control-tactics =
           MATCH_CONTROL_TACTICS (PS.control-tactics,
                                  applicable-domain-rules,
                                  Environment)
        selected-rule =
           SELECT_RULE (applicable-domain-rules,
                        applicable-control-tactics)
    return-value
       APPLY-DOMAIN-RULE (selected-rule, Environment)
   end-let*
  of-type 'complex problem-solver' :
  ; im Falle eines complex problem-solvers werden zuerst alle anwendbaren sub-problem-solver
  ; und dann alle anwendbaren Kontrollstrategien bestimmt. Mit Hilfe der Kontrollstrategien wird
  ; die Instanz eines problem-solvers ausgewählt und anschließend interpretiert.
  ; Die Interpretation liefert das Exportwissen des sub-problem-solvers zurück.
   let* applicable-problem-solvers =
           MATCH_PS_PRECONDS (PS.sub-problem-solvers,
                              Environment)
        applicable-control-strategies =
           MATCH_CONTROL_STRATEGIES (PS.control-strategies,
                                     applicable-problem-solvers,
                                     Environment)
        selected-ps-instance =
           SELECT_PS (applicable-problem-solvers,
                      applicable-control-strategies)
    return-value
       INTERPRET-PROBLEM-SOLVER
          (CORRESPONDING_PS (selected-ps-instance),
           PRECOND_BINDINGS (selected-ps-instance),
           Environment)
   end-let*
 end-typecase
end INTERPRET_REASONING_KNOWLEDGE
```

Algorithmus 3.2: Interpretationszyklus für Schlußfolgerungswissen

Unter Verwendung der beiden präsentierten Algorithmen läßt sich damit der Interpreter für *ARI* abstrakt als ***Algorithmus 3.3 SOLVE_PROBLEM*** formulieren, der als Eingabeparameter den top-level-problem-solver PS_{top}, die LUIGI-Wissensbasis für beschreibendes Wissen `Descriptive-Knowledge` und die globale Problemdefinition `Problem-Definition` trägt.

```
SOLVE_PROBLEM (PS_top, Descriptive-Knowledge, Problem-Definition) is
 let* environment =
        UPDATE_ENVIRONMENT (Problem-Definition,
                            CREATE_ENVIRONMENT (Descriptive-Knowledge))
      ps-instance =
        MATCH_PS_PRECOND (PS_top, environment)
      problem-solution =
        INTERPRET_PROBLEM_SOLVER (PS_top,
                                  environment,
                                  PRECOND_BINDINGS (ps-instance))
  return-value problem-solution
 end-let*
end SOLVE_PROBLEM
```

***Algorithmus 3.3**: Der ARI-Regelinterpretierer als "Problemlöser"*

3.2.5. *ARI* als Metaebenen-Architektur

ARI erweitert das Konzept einer dreistufige Metaebenen-Architektur in CATWEAZLE (vgl. Abb. 2.3.) durch die Möglichkeit, Regelbasen hierarchisch zu strukturieren. Für Objektregeln, die Inferenzen auf dem beschreibenden Wissen einer LUIGI-Wissensbasis ausführen, läßt sich auch in *ARI* in Form von Kontrolltaktiken genau eine Metaebene zur Spezifikation von Kontrollwissen definieren. Bezüglich der Ebene von *problem-solvern*, die als funktionale Einheiten eine Problemklasse bearbeiten, läßt sich für jede Stufe der hierarchischen Organisation einer *ARI*-Regelbasis eine eigene Metaebene durch Kontrollstrategien spezifizieren.

Das für *ARI* entwickelte Strukturierungsprinzip erlaubt also die Beschreibung mehrstufiger Metaebenen-Architekturen, ohne dabei Meta-Regeln über Meta-Regeln, oder beliebige weitere Stufen von Meta-Regeln zu formulieren. Die Trennung von Metaebene und Objektebene in Form von Kontrollstrategien und kontrollierten *problem-solvern* entspricht vielmehr der natürlichen Vorgehensweise menschlichen Problemlösens, bei dem ein Problem in Teilprobleme zerlegt wird, die unter Umständen mit verschiedenen Problemlösungsstrategien bearbeitet werden.

Die Einführung von Meta-Regeln über Meta-Regeln erscheint im Rahmen von *ARI* nicht sinnvoll, da *problem-solver* als abgeschlossene Einheiten für Regelmengen die Größenordnung von etwa 50-100 Regeln nicht überschreiten sollen. Eine realistische Anzahl von Kontrolltaktiken dürfte in etwa bei einem Zehntel davon liegen und dies impliziert, daß auf einer weiteren Metaebene nur noch eine bis zwei Meta-Regeln sinnvoll wären, die den Mehraufwand für die Interpretation einer weiteren Ebene nicht rechtfertigen. Einen Überblick über *ARI* als Metaebenen-Architektur gibt Abb. 3.9.

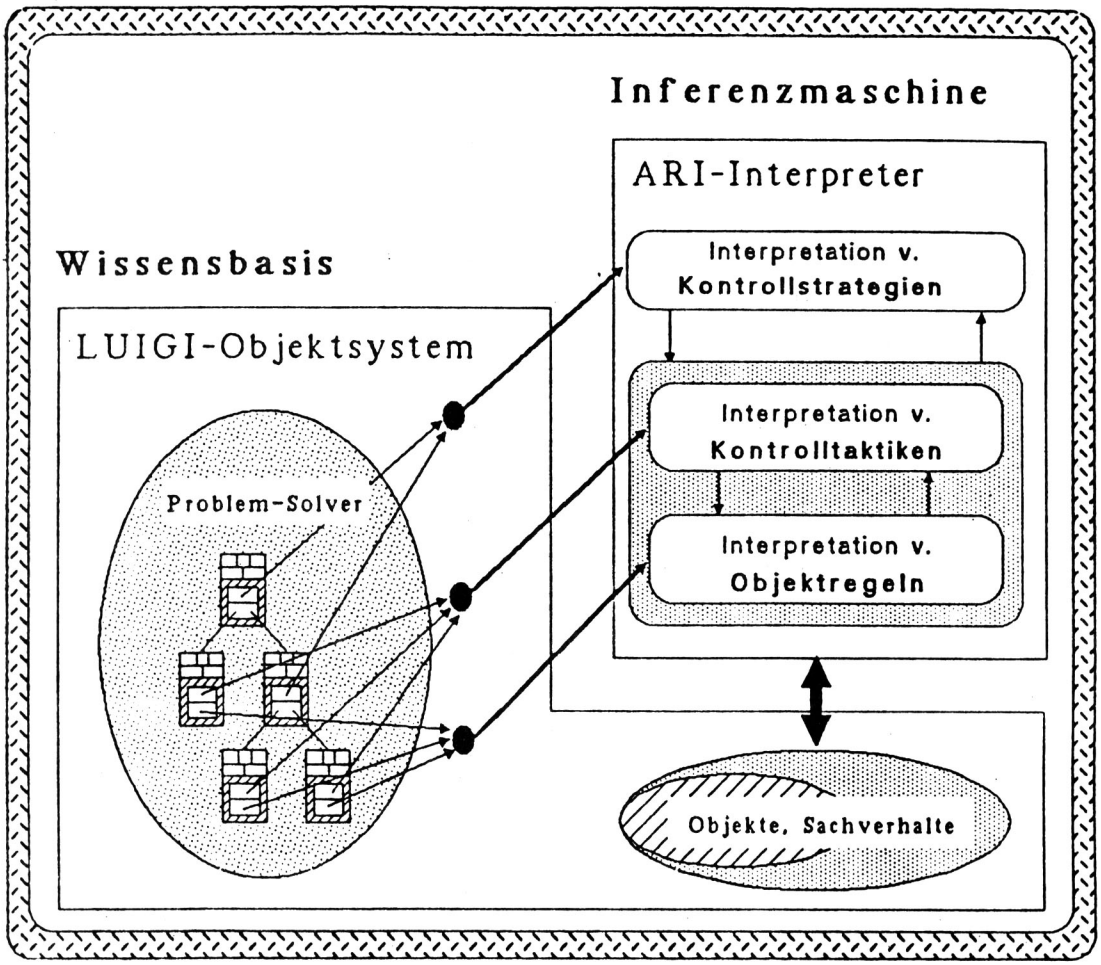


Abb. 3.9: ARI als Metaebenen-Architektur

Gemäß der Klassifikation von F. van Harmelen (s. /vanHarmelen 87/) läßt sich ***ARI*** als Metaebenen-Architektur zur Kontrolle mit Inferenzen auf verschieden Ebenen (***meta-level architecture with mixed-level inference***) charakterisieren. In solchen Systemen wechselt die Interpretation zwischen Metaebene und Objektebene im Gegensatz zu Systemen, die entweder alle Inferenzen auf der Metaebene ausführen (***meta-level inference***) oder ausschließlich auf der Objektebene operieren, indem alle Konstrukte der Metaebene in Strukturen der Objektebene übersetzt werden.

In ***ARI*** wird Kontrollwissen ausschließlich auf der Metaebene in der select-Phase interpretiert, während das System darüber ***reflektiert***, welche Regel anzuwenden oder welcher *problem-solver* zu aktivieren ist. Die Inferenzregeln auf der Objektebene ziehen Schlußfolgerungen über das beschreibende Wissen. Von diesem abstrakten Standpunkt aus gesehen führt das ***ARI***-Inferenzsystem einem ***reflect-act***-Zyklus aus.

3.2.6. Die ARI Repräsentationssprache im Detail

In diesem Abschnitt werden die wichtigsten Sprachkonstrukte in *ARI* anhand von Syntaxbeschreibungen und Beispielen detailliert erläutert. Eine vollständige Beschreibung der Syntax im Backus-Naur Formalismus findet sich in **Anhang A.** Die dort erläuterten Konventionen bezüglich der Konstrukte der Syntaxbeschreibung gelten hier analog.

Pattern für Wissensstrukturen

Sachverhalte, Objekt- und Regelinstanzen sowie Instanzen von *problem-solvern* sind momentan die einzigen Wissensstrukturen in LUIGI, die durch *pattern* der *ARI*-Sprache beschrieben und von der Inferenzmaschine verarbeitet werden können. *Pattern* für Sachverhalte und Objektinstanzen enthalten immer eine Variable (durch ein "?" gekennzeichnet), die beim *pattern-matching* an die LUIGI-interne Identität der Wissenseinheit gebunden wird. Referenzen auf die Wissenseinheit z.B. in Situationsbeschreibungen oder aber Aktionsteilen von Regeln können nur über diese Variable und nicht über den Namen der Wissenseinheit realisiert werden. Es existiert eine anonyme "dont-care" Variable ?*, die alles *matcht* und für die keine Bindung aufgebaut wird.

Sachverhalte werden in *ARI* durch ***affair-pattern*** beschrieben. Dabei muß im *pattern* der Name des Sachverhaltes und für jede Konstituente bzw. jedes Argument des Sachverhaltes muß eine Konstante bzw. eine Variable angegeben sein.

Affair-Pattern:

```
<affair-pattern>
     ::= AFFAIR  <variable>
          <affair-name> "(" (<variable> | <constant>)* ")"
```

Beispiel: das *affair-pattern*

```
AFFAIR  ?responsibility
 responsible-department-for-topic ( ?department, ?topic )
```

matcht alle zweistelligen Sachverhalte mit dem Namen `responsible-department-for-topic`. Bei einem *match* wird die Variable `?responsibility` an den Sachverhalt selbst, `?department` an das erste Argument und `?topic` an das zweite Argument des Sachverhaltes gebunden.

Objektinstanzen werden in *ARI* durch ***object-pattern*** beschrieben. Als Beschreibung der zugehörigen Klasse der Objektinstanz muß im *pattern* entweder der **Name der Klasse** oder einer Oberklasse angegeben werden. Die Menge der durch ein *pattern* beschriebenen Instanzen kann durch die **Konstituentenbeschreibungen** weiter eingeschränkt werden. Eine Konstituentenbeschreibung besteht aus Konstituentennamen, einem zweistelligen Testprädikat und einem Wert in Form einer Variablen oder einer Konstanten. Eine Konstituentenbeschreibung *matcht* die Konstituente einer Objektinstanz, wenn beide Konstituentennamen übereinstimmen und das Testprädikat wahr ist, wobei der Konstituentenwert der Objektinstanz als erstes Argument und der Wert aus der Konstituentenbeschreibung als zweites Argument verwendet wird.

Object-Pattern:

```
<object-pattern>
     ::= OBJECT  <variable>
        ( (OF-CLASS <class-name>) |
          (OF-SUBCLASS-OF <class-name>) )
        [ WITH (<constituent-name>
                <constituent-test-predicate>
                (<variable> | <constant>)+  )+ ]
```

Beispiel: das *object-pattern*

```
OBJECT  ?query
 OF-CLASS QUERY
  WITH
       topic  = "LISP"
       source > 121
```

matcht Instanzen der Klasse QUERY, die in der Konstituente `topic` den Wert `"LISP"` und in der Konstituente `x-code` eine Wert größer als 121 tragen. Bei einem **match** wird die Variable `?country` an den Wert der Konstituente `source` gebunden.

Situationsbeschreibungen

Eine **Situationsbeschreibung** besteht aus einer Folge von Bedingungselementen, die implizit konjunktiv verknüpft sind. In Situationen über beschreibendem Wissen, wie sie im Bedingungsteil von Inferenzregeln und in Vorbedingung, Zielbeschreibung, Import- und Exportspezifikation von *problem-solvern* benutzt werden, können Bedingungselemente die Form von *affair-pattern*, *object-pattern* oder *test-pattern* annehmen, wie bereits in Abschnitt 3.2.3.1. erläutert wurde. *Test-pattern* sind Aufrufe von LISP-Funktionen, die Tests auf Variablenbindungen ausführen.

Test-Pattern:

```
<test-pattern>
     ::= TEST  "(" <test-predicate-name>
                   (<variable> | <constant>)* ")"
```

Beispiel:

```
TEST (<= ?value1 ?value2)
```

Situationen des Diskursbereiches:

```
<domain-situation>
     ::= ( ([ NOT ]
             (<affair-pattern> | <object-pattern>) |
           <test-pattern> )+
```

Beispiel: die Situationsbeschreibung

```
OBJECT  ?query
 OF-CLASS QUERY
  WITH status = pending
NOT AFFAIR ?*
     (responsible-for ?department ?query)
```

beschreibt eine Situation, in der eine Instanz der Klasse QUERY mit dem Status `pending` existiert und es keinen Sachverhalt gibt, der bereits eine zuständige Abteilung für diese Anfrage etabliert hat.

Situationsbeschreibungen in Kontrolltaktiken können zusätzlich Beschreibungen von Objektregelinstanzen (*domain-rule-pattern*), in Kontrollstrategien zusätzlich Beschreibungen von *problem-solver*-Instanzen (*problem-solver-pattern*) enthalten:

Domain-Rule-Pattern:

```
<general-rule-pattern>
     ::= RULE  <variable>  WITH
         [ CONDITIONS  <domain-situation>  ]
         [ ACTIONS     <primitive-action>+ ]
```

Problem-Solver-Pattern:

```
<general-problem-solver-pattern>
     ::= PROBLEM-SOLVER  <variable>  WITH
         [ PRECONDITION  <domain-situation> ]
         [ GOAL          <domain-situation> ]
```

Regeln

Die drei verschiedenen Arten von Regeln in *ARI*, Inferenzregeln, Kontrolltaktiken und Kontrollstrategien sind in Abschnitt 3.2.3. ausführlich erläutert worden. Es sollen hier nun für jede Art ein konkretes Beispiel vorgestellt werden, als Beispiel für eine Kontrolltaktik und eine Kontrollstrategie genau die *ARI*-Formalisierungen der abstrakten Formulierungen aus 3.2.3.2. bzw. 3.2.3.3. Eine *constrained-domain-situation* ist eine Situationsbeschreibung, bei der das erste Bedingungselement nicht negiert sein darf.

Objektregeln:

```
<domain-rule>
     ::= DOMAIN-RULE  <name>
          <constrained-domain-situation>
         "-->"
          <domain-action>+
```

Beispiel:

```
DOMAIN-RULE  SUGGEST-MOST-SPECIFIC-DEPARTMENT
  OBJECT  ?query
   OF-CLASS QUERY
   WITH topic = ?topic
  AFFAIR ?*
   responsible-for-topic (?department ?topic)
 -->
  ASSERT-AFFAIR
   responsible-department-for (?department ?query)
```

Diese Regel ist anwendbar, wenn eine Anfrage `?query` zu einem speziellen Sachgebiet `?topic` vorliegt (Instanz der Klasse QUERY) und wenn gleichzeitig eine Abteilung `?department` bekannt ist, die allgemein für dieses Gebiet zuständig ist (Sachverhalt **responsible-for-topic**). Bei Anwendung der Regel wird in Form eines Sachverhaltes **responsible-department-for** die Abteilung `?department` als zuständig etabliert.

Kontrolltaktiken:

```
<control-tactic>
     ::= CONTROL-TACTIC  <name>
          <rule-conflict-situation>
          "-->"
          <control-action> <variable>
```

Beispiel: die Kontrolltaktik

```
CONTROL-TACTIC SAVE_MONEY
 RULE ?rule-1 WITH
  CONDITIONS  AFFAIR ?*
               responsible-for-country (?department ?country)
              OBJECT  ?query
               OF-CLASS Query
               WITH source = ?country
  ACTIONS  ASSERT-AFFAIR
            responsible-department-for (?department ?query)
 RULE ?rule-2 WITH
  CONDITIONS  AFFAIR ?*
               responsible-for-topic (?department ?country)
              OBJECT  ?query
               OF-CLASS Query
               WITH topic = ?topic
  ACTIONS  ASSERT-AFFAIR
            responsible-department-for (?department ?query)
-->
PREFER-DOMAIN-RULE ?rule1
```

bevorzugt Regeln, die Länderzuständigkeiten für Anfragen berücksichtigen gegenüber Regeln, die Fachzuständigkeiten berücksichtigen.

Kontrollstrategien:

```
<control-strategy>
     ::= CONTROL-STRATEGY  <name>
          <problem-solver-conflict-situation>
          "-->"
          <control-action> <variable>
```

Beispiel: Die Kontrollstrategie

```
CONTROL-STRATEGY EMPHASIZE_SUPPORT
 PROBLEM-SOLVER ?ps-1 WITH
  GOAL  AFFAIR ?*
         responsible-sales-manager (?employee)
 PROBLEM-SOLVER ?ps-2 WITH
  GOAL  AFFAIR ?*
         responsible-support-manager (?employee)
 OBJECT  ?query
  OF-CLASS Query
   WITH topic = "Hardware Services"
-->
PREFER-PROBLEM-SOLVER ?ps-2
```

bevorzugt problem-solver, die bei Anfragen Kundenbetreuung betonen vor solchen, die den Verkauf in den Vordergrund stellen.

Problem-Solver

Precondition und ***goal*** der abstrakten Beschreibung von *problem-solvern* spezifizieren ebenfalls Situationen über beschreibendem Wissen. Die *precondition* kann genau die gleiche Form annehmen, wie der Bedingungsteil von Objektregeln, für das *goal* ist eine Menge disjunktiv verknüpfter Situationsbeschreibungen erlaubt.

```
<precondition-spec>
     ::= PRECONDITION
          <constrained-domain-situation>
<goal-spec>
     ::= GOAL
          <constrained-domain-situation>
          (OR <constrained-domain-situation>)*
```

Import- und ***Exportspezifikation*** beschreiben Klassen von Wissenseinheiten, die jedoch auch jeweils durch eine Situationsbeschreibung ergänzt werden können, die als zusätzliche Bedingungen die Menge der relevanten Wissenseinheiten einschränken lassen.

```
<communication description>
     ::= COMMUNICATION
          <import-spec>
          <export-spec>

<import-spec>
     ::= IMPORT  <domain-pattern-in-situation>+

<export-spec>
     ::= EXPORT  <domain-pattern-in-situation>+

<domain-pattern-in-situation>
     ::= <domain-pattern>
        [ WITH-CONDITIONS <domain-situation> END ]
```

Ein ***domain-pattern-in-situation*** spezifiziert diejenigen Wissenseinheiten, die das `<domain-pattern>` *matchen*, wobei gleichzeitig `<domain-situation>` erfüllt sein muß. Der Geltungsbereich der verwendeten Variablen ist das gesamte Konstrukt.

Zum Abschluß wird ein *problem-solver* FIND-RESPONSIBLE-DEPARTMENT beschrieben, der für Anfragen eine zuständige Abteilung bestimmen soll. Es sind hier lediglich abstrakte Beschreibung und Kommunikationsspezifikation angegeben.

Der *problem-solver* ist gemäß ***precondition*** anwendbar, wenn eine Anfrage in Form einer Instanz der Klasse QUERY vorliegt und noch kein Sachverhalt mit einer Zuständigkeit für diese Anfrage etabliert wurde.

Ziel des *problem-solvers* ist es gemäß ***goal***, entweder eine zuständige Abteilung in Form eines Sachverhaltes **responsible-for** vorzuschlagen, oder festzustellen, daß keine zuständige Abteilung für die Anfrage existiert (Sachverhalt **there-is-no-responsible-department-for**).

```
PROBLEM-SOLVER FIND-RESPONSIBLE-DEPARTMENT
SOLVES-PROBLEM-WITH
  PRECONDITION OBJECT ?query
                OF-CLASS QUERY
                status = pending
               NOT AFFAIR ?* responsible-for (?* ?query)
  GOAL AFFAIR ?* responsible-for (?* ?query)
       OR AFFAIR ?* there-is-no-responsible-department-for (?query)
COMMUNICATION
  IMPORT OBJECT ?query
         OF-CLASS QUERY
          ...
          ...
  EXPORT OBJECT ?department
         OF-CLASS DEPARTMENT
          WITH-CONDITIONS
           AFFAIR ?* responsible-for (?department ?query)
    ...
    ...
    ...
    ...
```

Als **Importwissen** werden unter anderem alle Instanzen der Klasse Anfrage spezifiziert, als **Exportwissen** genau die Instanz der Klasse DEPARTMENT, die als Abteilung zur Beantwortung der Anfrage vorgeschlagen wird.

3.3. Grundlagen der Implementierung von ARI

Auf der Implementierungsebene für ***ARI*** müssen effiziente Interpretationsalgorithmen für die auf epistemologischer Ebene vorgegebene Repräsentationssprache bereitgestellt werden. Im Abschnitt über wissensbasierte Systeme wurde bereits darauf hingewiesen, daß zwischen Ausdrucksstärke eines Formalismus und der Komplexität von Interpretationsalgorithmen ein fundamentaler *tradeoff* besteht, d.h. je ausdrucksstärker ein Repräsentationsmechanismus ist, desto komplexer werden die Algorithmen zur Interpretation der Wissenseinheiten.

Für ***ARI***, das eine mächtige Produktionsregelsprache mit Konstrukten zur Organisation und Strukturierung von Regelbasen und Mechanismen zur expliziten und deklarativen Spezifikation von Kontrollwissen zur Verfügung stellt, ist dies sicherlich ein ganz zentraler Punkt, an dem sich entscheidet, ob der vorgegeben Rahmen geeignet ist, komplexe Architekturen wissensbasierter Systeme für den praktischen Einsatz zu entwickeln.

Worin liegt nun für regelbasierte Systeme wie ***ARI*** die Quelle der Zeitkomplexität? Mustergesteuerte Systeme hängen in ganz besonderem Maße von der Effizienz des verwendeten *pattern-matching* Algorithmus ab. Ch. Forgy hat bei solchen Systemen einen Anteil von über 90% der Gesamtzeit der Interpretation für die Operationen des *pattern-matching* gemessen (s. /Forgy 79/). Da nun in ***ARI*** alle Sprachkonstrukte auf *pattern* aufbauen, ist eine effiziente Realisierung von *pattern-matching*-Algorithmen die wichtigste Aufgabe für die Implementierungsebene des Inferenzsystems.

Für das Problem des *pattern-matchings* zwischen einer großen Menge von Mustern und einer großen Menge von Daten (***many pattern/many object pattern matching problem***), bei denen sich die Menge der Daten nur schrittweise ändert, hat Ch. Forgy den RETE-Algorithmus entwickelt. RETE wird im Produktionsregelsystem **OPS5** verwendet und hat entscheidend zum Erfolg dieses Systems beigetragen.

Da alle Sprachkonstrukte in ***ARI*** auf einer einheitlichen Mustersprache aufbauen, kann ein uniformer RETE-artiger Algorithmus die Grundlage der Implementierung des ***ARI***-Interpretierers bilden. Die RETE-Erweiterungen im entwickelten Algorithmus sind dabei im einzelnen

- Verarbeitung der Konstrukte zur Strukturierung von Regelbasen,
- Verarbeitung der Konstrukte zur Spezifikation von Kontrollwissen und
- die Integration verschiedener effizienzsteigernder Erweiterungen des ursprünglichen RETE-Algorithmus, die in der Literatur vorgeschlagen wurden.

Im folgenden werden zunächst die Konzepte und Verarbeitungsprinzipien des RETE-Algorithmus erläutert. Anschließend werden die angesprochenen Erweiterungen in der für ***ARI*** entwickelten Implementierung detailliert erläutert.

3.3.1. Der RETE-Algorithmus

Die Idee des RETE-Algorithmus läßt sich anhand dreier grundlegender Prinzipien erläutern, wobei weder eine spezielle Repräsentationsform für Daten noch für *pattern* zugrundegelegt wird.

(1) ein *pattern* kann prozedural interpretiert werden. Es faßt als *pattern-matching*-Prozedur mehrere elementare Testoperationen zusammen und bestimmt für eine Menge von Daten, ob es von diesen erfüllt wird.

(2) durch explizites Speichern von Informationen, die bei der Instanziierung von Teilmustern aufgebaut werden (*partial matches*), können Mehrfachberechnungen (***temporal redundancy***) vermieden werden.

(3) Gleichartige Teilstrukturen verschiedener *pattern* können zu einer einzigen *pattern-matching*-Prozedur zusammengefaßt werden, die von den Prozeduren für die Gesamtmuster gleichzeitig benutzt werden (***sharing of structural similarities***).

Für die folgenden Ausführungen werden als *pattern* einfache Listen mit Symbolen als Konstanten und Variablen (gekennzeichnet durch ein "?") verwendet, Daten sind dementsprechend Listen von Symbolen. Betrachtet man die folgende Regel

```
RULE SUPERIOR_CLOSURE
  (superior ?p the-employee)
  (superior the-boss ?p)
-->
ASSERT (superior the-boss the-employee)
```

so läßt sich dafür eine *pattern-matching*-Prozedur angeben, die für je zwei Datenelemente berechnet, ob sie den Bedingungsteil der Regel erfüllen und somit eine Regelinstanz für die Konfliktmenge anwendbarer Regeln liefern:

```
PATTERN-MATCHING-SUPERIOR-CLOSURE ( d1, d2 ) is
 if      LENGTH (d1) = 3
    and FIRST (d1) = superior
    and THIRD (d1) = the-employee
    and LENGTH (d2) = 3
    and FIRST (d2) = superior
    and SECOND (d2) = the-boss
    and SECOND (d1) = THIRD (d2)
 then (d1,d2) instatiantes SUPERIOR_CLOSURE
```

Diese Sichtweise hat den Vorteil, daß sie direkt ein Operationsmodell für die Berechnung des *pattern-matching* liefert. Konventionelle Prozedurkonzepte erlauben allerdings weder das Speichern von Teilresultaten zwischen verschiedenen Prozeduraufrufen, noch lassen sich Teile von verschiedenen Prozeduren auf einfache Weise von anderen Prozeduren gemeinsam benutzen.

Dies wird jedoch vom Berechnungsprinzip eines **Datenflußmodells** direkt unterstützt, bei dem eine Menge von Prozeduren als Datenflußnetzwerk repräsentiert wird, in dem **Knoten** elementare Operationen auf den Daten ausführen, die in Form sogenannter **Token** entlang der Pfade des Netzwerkes geschleust werden. Die Verarbeitung eines einzigen Datums durch eine Prozedur entspricht dabei also genau einem vollständigen Pfad von der Quelle zur Senke des Netzwerkes

An obigem Beispiel soll gezeigt werden, wie das *pattern-matching* zwischen einem *pattern* und einer Menge von Daten in diesem Modell aussieht. In einem Datenflußnetzwerk als *pattern-matching*-Prozedur sind die Operationen, die in den Knoten ausgeführt werden, Tests auf den als Token durchgeschleusten Daten. In einem ersten Modell werden alle Token, auf denen der Test fehlschlägt, weggeworfen.

Knoten haben entweder eine Eingangskante (**alpha-Knoten**) oder zwei Eingangskanten (**beta-Knoten**). Alpha-Knoten propagieren Token, auf denen der Test erfolgreich ist, an den oder die Nachfolgeknoten weiter. Beta-Knoten propagieren im Erfolgsfall ein aus beiden Eingangstoken **zusammengesetztes komplexes Token weiter.**

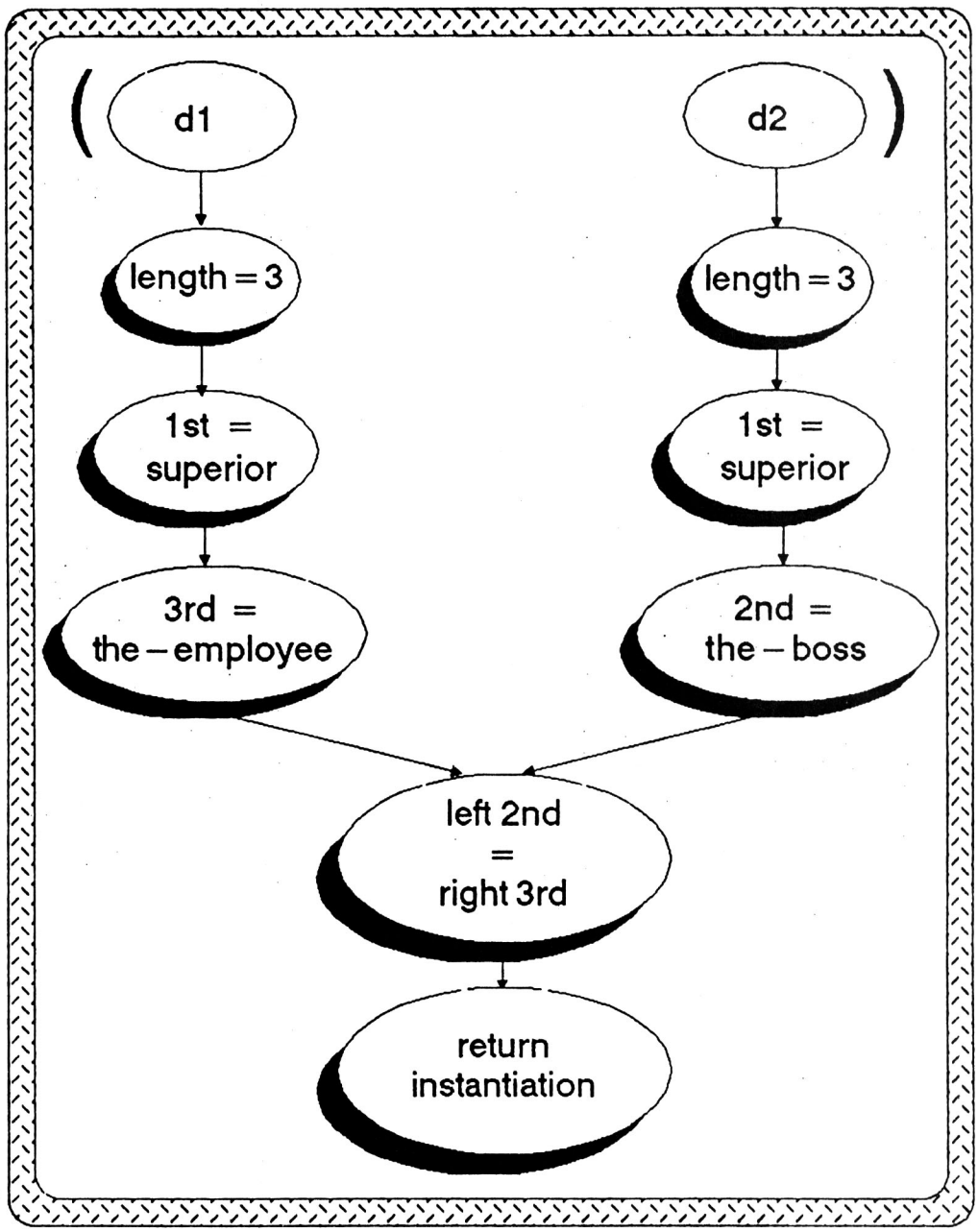


Abb. 3.10: Eine pattern-matching-Prozedur als Netzwerk

Der linke Pfad des Netzwerkes in Abb. 3.10 enthält drei elementare Tests für das erste Bedingungselement der Regel, der rechte analog dazu drei Tests für das zweite Bedingungselement. Der Knoten, bei dem die beiden Pfade zusammenlaufen führt einen Test aus, der die Konsistenz der Bindungen für die Variable ?p überprüft. Der Endknoten des Netzes empfängt nur Token, für die alle Tests erfolgreich durchgeführt wurden und kann deshalb alle ankommenden Token als Instanziierung für das *pattern* zurückliefern. Beim konkreten Vergleich des Musters mit einer Menge von Daten müssen alle Paare von Daten für d1 bzw. d2 eingesetzt werden. Tests in alpha-Knoten werden auch als **intra-element**

Tests bezeichnet, da sie sich nur auf ein Bedingungselement beziehen. Analog dazu heißen Tests in beta-Knoten **inter-element** Tests, da sie stets die Konsistenz von Variablenbindungen in verschiedenen Bedingungselementen überprüfen.

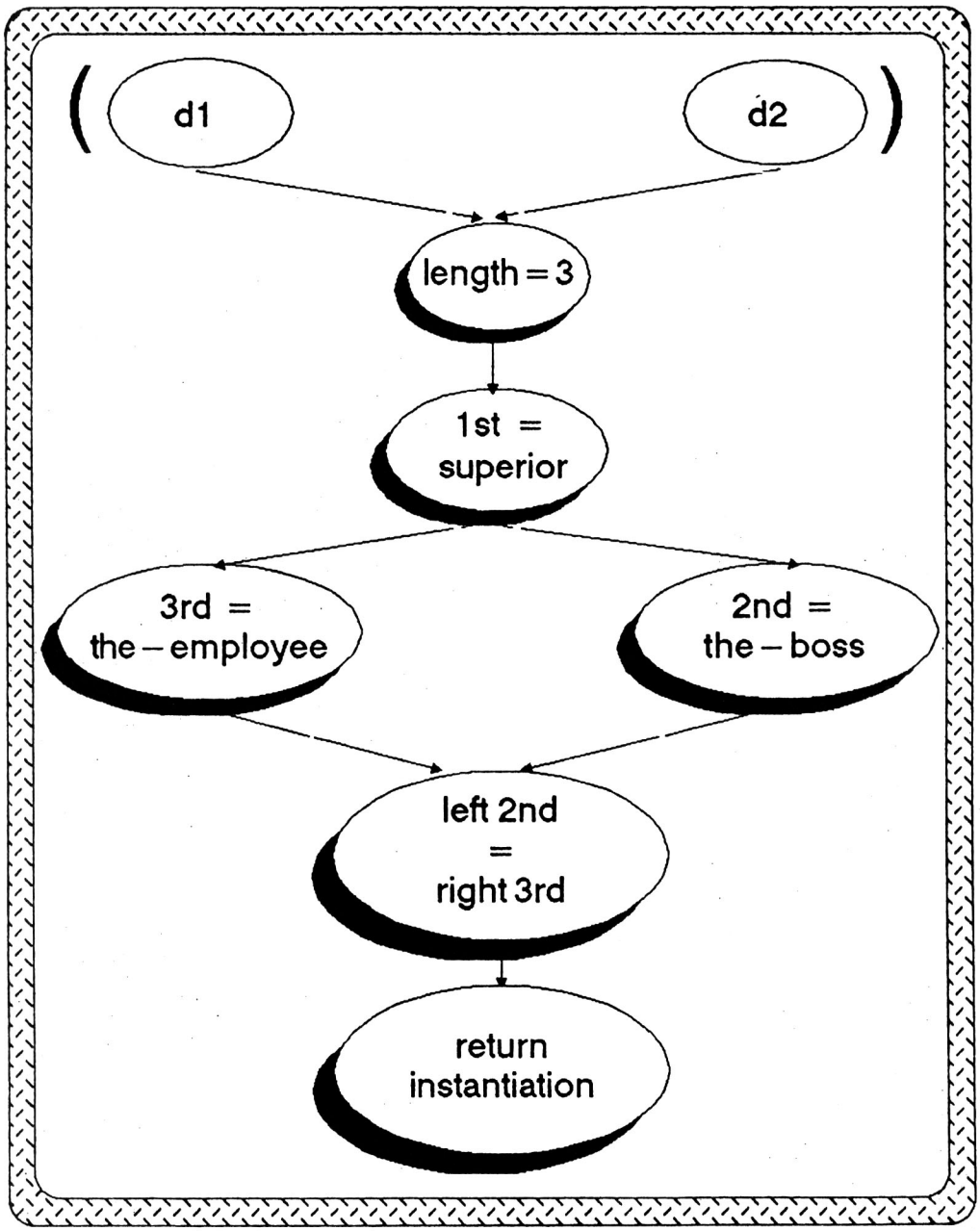


*Abb. 3.11: Ein Netzwerk mit **sharing***

Im Datenflußmodell können Knoten, die gleiche Tests ausführen und zu denen gleiche Pfade führen, zusammengefaßt werden, wodurch sich in einfacher Weise das *sharing* für gleiche Teilpattern wie in Abb. 3.11 ergibt. Die Möglichkeit des *sharing* wird natürlich durch die Reihenfolge der durchgeführten Tests beeinflußt, da nur Anfangspfade zusammengefaßt werden können.

Im bisherigen Modell können beta-Knoten nur dann erfolgreich einen Test ausführen, wenn über beide Kanten gleichzeitig Token ankommen. Versieht man die beta-Knoten mit getrennten Speichern für Token von links und von rechts, so kann die Information, die in den zum Knoten führenden Pfaden

berechnet wurde, gespeichert werden. Ankommende Token bilden nun alle möglichen Paare mit den Daten im *"gegenüberliegenden"* Speicher und auf diesen Paaren werden die Tests der Knoten ausgeführt.

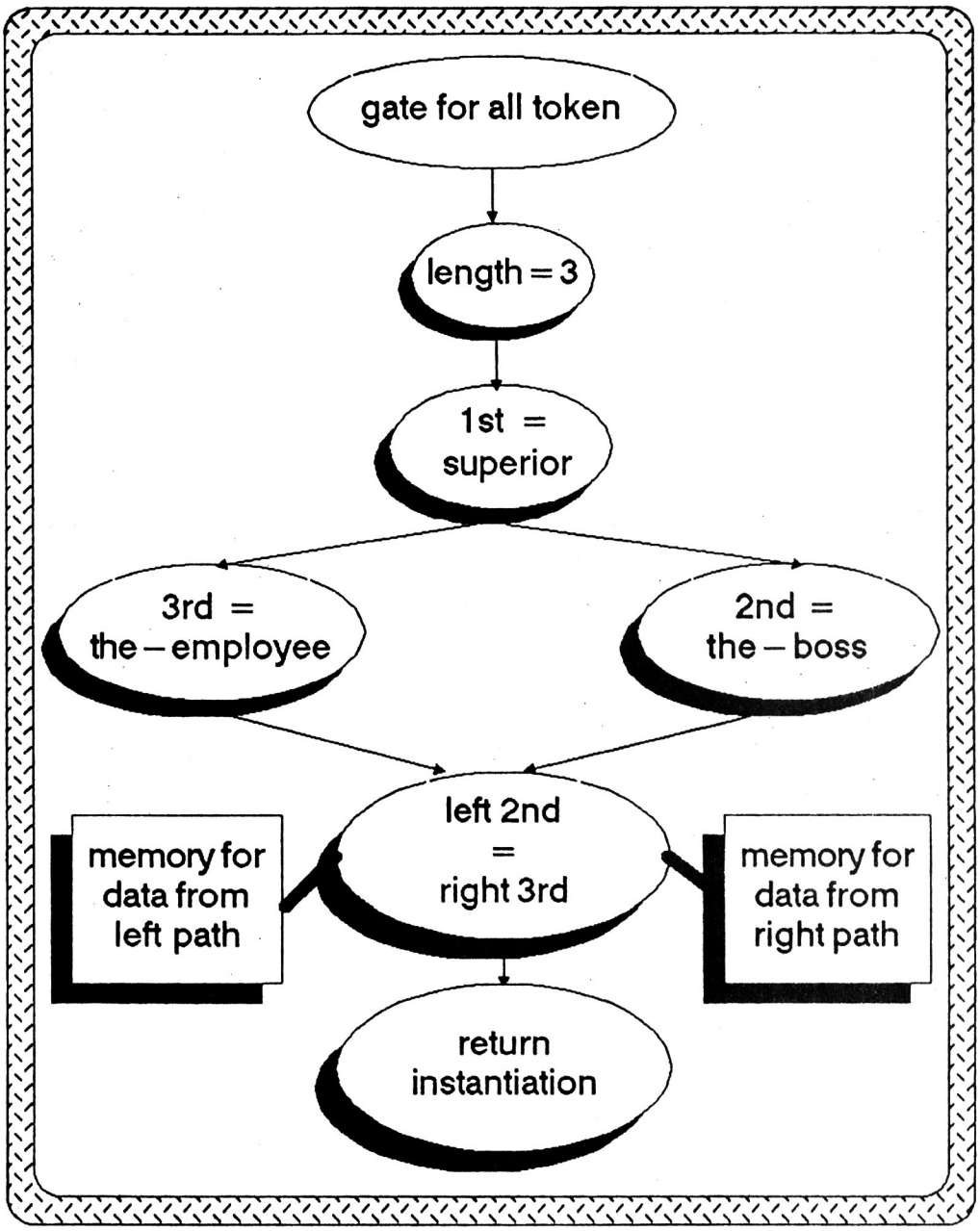


*Abb. 3.12: Ein Netzwerk mit **sharing** und Speichern*

Dieses Prinzip erlaubt das Speichern aller **partiellen matches** entlang eines Pfades und vermeidet, daß alle Paare, Tripel, usw. gebildet werden und gleichzeitig in entsprechende Testpfade eingeschleust werden müssen. Mit dieser Technik können die Daten in beliebiger Reihenfolge einzeln in alle Testpfade eingebracht werden, Synchronisation erfolgt über die beta-Knoten und deren Speicher.

In den Aktionsteilen der Regeln werden sowohl Wissenseinheiten hinzugefügt als auch gelöscht. Dadurch können Daten zu der beim *pattern-matching* betrachteten Datenmenge hinzukommen, oder aus ihr entfernt werden. Dementsprechend werden Token die durch ein RETE-Netzwerk geschleust werden, mit einem **add-** bzw. **delete**-Indikator versehen. Beim Löschen eines Datums muß es wieder genauso

durch das Netz geschleust werden, wie beim Hinzufügen, nur jetzt muß es aus allen Speichern entfernt werden, in denen es vorkommt.

Für eine Regel

```
RULE Rx
  b1
  b2
  ...
  bn
 -->
  <action-x>
```

sieht ein Bedingungsnetzwerk wie in Abb. 3.13 aus:

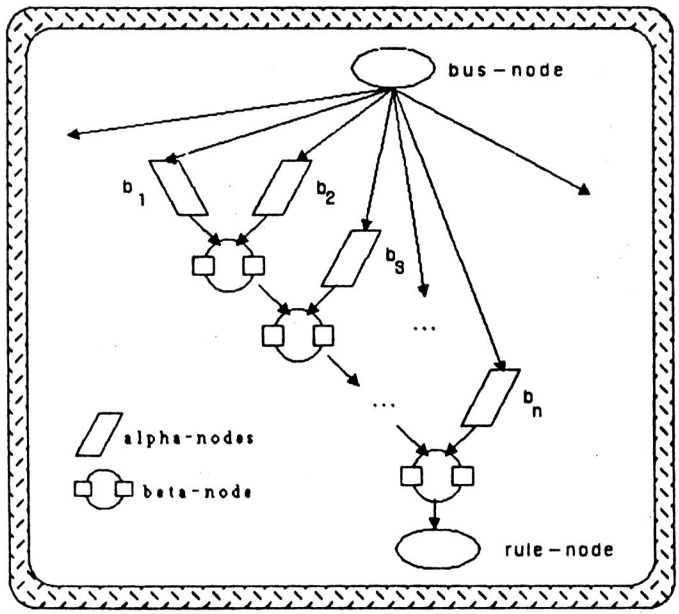


Abb. 3.13: Typisches RETE-Bedingungsnetzwerk für eine Regel

Der RETE-Algorithmus kann nun wiefolgt zur Berechnung der anwendbaren Regelinstanzen in der Konfliktmenge eines allgemeinen mustergesteuerten Produktionsregelsystems verwendet werden:

- Die Bedingungsteile aller Regeln werden in einer Vorverarbeitungsphase vor der Interpretation von einem Compiler in ein Bedingungsnetzwerk übersetzt. In diesem Netzwerk gibt es genau einen Eingangsknoten (***bus-node***), durch den alle Wissenselemente eingeschleust werden und an alle Testpfade weiterpropagiert werden. Die Endknoten im Netz repräsentieren jeweils genau eine Regel und ein ankommendes Token wird in ihnen als Regelinstanz interpretiert, die in die Konfliktmenge eingefügt wird.
- Alle Wissenseinheiten für beschreibendes Wissen (*working-memory*) werden zu Beginn der Interpretation durch das Bedingungsnetzwerk propagiert. Damit werden alle möglichen *matches* berechnet und die Konfliktmenge anwendbarer Regelinstanzen bestimmt.
- In jedem weiteren Durchlauf durch den ***recognize-select-apply*** Zyklus wird eine Regel angewendet, die Änderungen des beschreibendes Wissens im working-memory verursacht.
- Diese Änderungen werden wiederum durch das Netzwerk propagiert, wodurch neue *matches* berechnet werden, die wiederum Änderungen der Konfliktmenge nach sich ziehen.

Dies ist der RETE-Basisalgorithmus, wie er im Produktionsregelsystem **OPS5** verwendet wird. Datenelemente oder Wissenseinheiten im *working-memory* können dabei *records* von Werten elementarer Grundtypen sein, in denen der Prädikats- oder Faktenname an erster Stelle steht und die als Vektoren

repräsentiert sind. **Einfache Token** sind die netzwerkinternen Repräsentationen für einzelne Datenelemente und setzen sich aus diesen Vektoren und einem Operator '+' oder '-' zusammen, der beim Propagieren durch das Netz angibt, ob das betreffende Datenelement im *working-memory* erzeugt oder gelöscht wurde. **Komplexe Token,** die eine Folge von instanziierten Bedingungselementen repräsentieren, werden durch eine Liste einfacher Token und dem entsprechenden Operator dargestellt. Modifikationen an Datenelementen werden in **OPS5** als Kombination einer *delete*-Operation für das alte und einer *add*-Operation für das modifizierte Element realisiert.

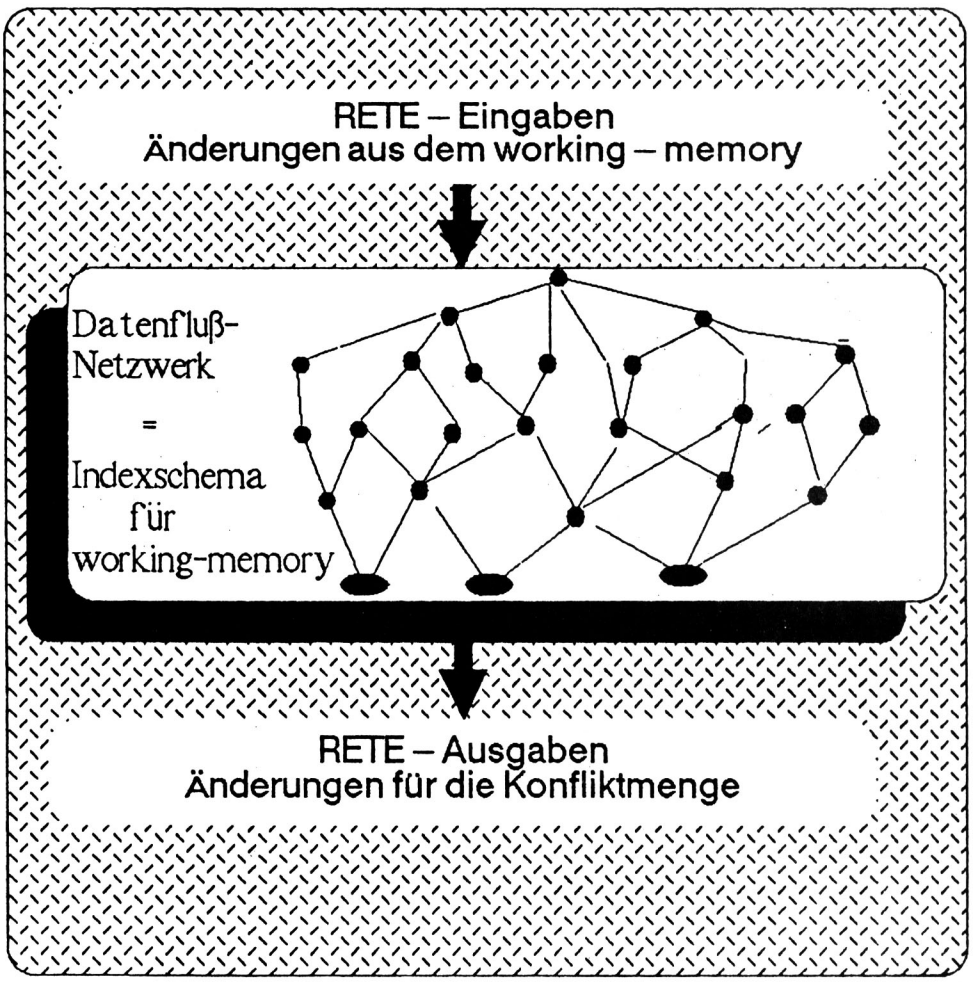


Abb. 3.14: Funktionale Sicht des RETE-Algorithmus in einem Produktionsregelsystem.

Die aufgebauten Bindungen für Variablen werden nicht explizit, etwa in einer Tabelle von Variablen, gespeichert , sondern der Wert, an den eine Variable zur Interpretationszeit gebunden ist, wird durch indizierten Zugriff auf Vektoren ermittelt. Die dazu benötigten Indizes für die als Vektoren dargestellten Token werden bei der Übersetzung von Regeln berechnet und in den Knoten abgelegt.

Eine wesentlichen Abweichung gegenüber dem präsentierten Modell ist die Zuordnung der Speicher für Token zu entsprechenden Knoten. In der konreten RETE-Realisierung werden Speicher nicht den beta-Knoten zugeordnet, sondern stellen einen eigenen Knotentyp dar und werden den beta-Knoten *"vorgeschaltet"*. Dies erlaubt zusätzliches *sharing* und vermindert die Kosten für das Speichern von Token (vgl. /Forgy 79/, pp. 63-66)

Als Repräsentationsstruktur für das Netzwerk schlägt Ch. Forgy eine Art Assemblersprache ähnlich der Maschinensprache konventioneller von Neumann Architekturen vor, in der Knoten als elementare

Maschineninstruktionen dargestellt werden. Der Interpreter wird damit zur virtuellen Maschine für diese Sprache. Die Propagation von Token wird durch Sprungbefehle und entsprechende *Label* für die Instruktionen, also die Knoten des Netzwerkes, realisiert (vgl. /Forgy 82/,pp. 26-32).

Die verschiedenen Knotentypen für den RETE-Basisalgorithmus sind in der folgenden Aufzählung zusammengefaßt*. In jedem Knoten außer den Endknoten des Netzwerkes sind die Nachfolgerknoten abgelegt.

Alpha-Knoten realisieren Tests für ein Bedingungselement und operieren somit auf je einem einfachen Token.

- ***constant-node***:
  ein *constant-node* testet, ob das ankommende Datenelement an bestimmten Positionen betimmte Werte (Konstanten) trägt. Da der Faktenname in **OPS5** immer das erste Element eines Datenvektors ist, kann auch der Namenstest für ein Faktum in einem *constant-node* realisiert werden.
- ***variable-node***:
  ein *variable-node* testet, ob mehrfache Auftreten einer Variablen innerhalb eines einzelnen Bedingungselementes konsistent gebunden werden können, d.h. ob an bestimmten Positionen innerhalb eines Tokens gleiche Werte stehen.
- ***alpha-memory-nodes***:
  ein *alpha-memory-node* schließt den Pfad einer Folge von alpha-Knoten ab und speichert alle einfachen Token, die auf diesem Pfad alle Tests passiert haben. Diese entsprechen also genau den Datenelementen, die ein einzelnes Bedingungselement *matchen*.

Beta-Knoten realisieren Tests für die konsistente Bindung von Variablen in verschiedenen Bedingungselementen, operieren also sowohl auf einfachen als auch auf komplexen Token. Beta-Knoten behandeln auch die negierten Bedingungselemente.

- ***and-node***:
  ein *and-node* überprüft die konsistente Bindung von Variablen innerhalb verschiedener Bedingungselemente einer Konjunktion. Der rechte Vorgängerknoten eines *and-node* ist immer ein *alpha-memory-node*, der einen Testpfad für ein einzelnes Bedingungselement abschließt. Der linke Vorgängerknoten eines *and-node* ist genau dann ein *alpha-memory-node*, wenn der Knoten Tests zwischen erstem und zweitem Bedingungselement einer Konjunktion ausführt, ansonsten ist es ein *beta-memory-node*. *And-nodes* propagieren komplexe Token an nachfolgende *beta-memory-nodes* weiter, wenn die Tests auf Token erfolgreich sind.
- ***not-node***:
  ein *not-node* überprüft die konsistente Bindung zwischen einem einzelnen negierten Bedingungselement einer Konjunktion und einer Teilfolge von Bedingungselementen dieser Konjunktion. Rechter Vorgängerknoten ist dabei immer ein *alpha-memory-node*, der Datenelemente speichert, die das Muster des negierten Bedingungselementes erfüllen. Der *not-node* propagiert nun das von links kommende Token genau dann an Nachfolgerknoten weiter, wenn sich im rechten *alpha-memory-node* **kein** Datenelement befindet, für den der Test erfüllt ist. Datenelemente, die das negierte Bedingungselement *matchen*, können also die Propagation von Token blockieren. Dabei speichert ein *not-node* für blockierte Token explizit die Anzahl der blockierenden Token.
- ***beta-memory-node***:
  ein *beta-memory-node* ist Nachfolger eines *and-nodes* oder eines *not-nodes* und speichert die Token, die die Tests des Vorgängerknoten passiert haben.

* die verwendeten Knoten und Knotennamen entsprechen nicht den ursprünglich von Ch. Forgy verwendeten Knoten. Die hier verwendeten Typen abstrahieren jedoch im wesentlichen nur von der konkreten Datendarstellung für Token, realisieren aber den gleichen Basisalgorithmus.

Sonstige Knoten im Netz sind:

- ***bus-node***:
  der *bus-node* ist nur einmal im Netz vorhanden und dient als Einstiegspunkt für alle Token, die durch das Netz geschleust werden sollen.
- ***rule-node:***
  ein *rule-node* schließt einen vollständigen Testpfad für den Bedingungsteil einer Regel ab. Ankommende Token können also als Instanziierungen von Regeln angesehen werden, die vom *rule-node* in die Konfliktmenge anwendbareer Regeln eingetragen wird.

3.3.2. Eine RETE-Realisierung für ARI

Es werden nun im Detail die Erweiterungen des in *ARI* verwendeten RETE-Algorithmus präsentiert, der die effiziente Interpretation aller in Abschnitt 3.2. vorgestellten Konstrukte der *ARI*-Sprache erlaubt. Zuvor sollen jedoch noch einige grundlegende Informationen zur *ARI*-Implementierung des Algorithmus gegeben werden.

In der *ARI*-Umgebung sind die Wissenseinheiten des LUIGI-Objektsystems, insbesondere Sachverhalte und Objektinstanzen, die Äquivalente zu Einträgen oder Datenelementen im *working-memory* des **OPS5**-Systems. Die internen Datenstrukturen zur Implementierung der LUIGI-Wissensstrukturen sind weitaus komplexer als die **OPS5**-internen Vektoren für Datenelemente. Zugriffe auf Konstituentenwerte von Wissenseinheiten sind aber aufgrund der Vielzahl durchzuführender Tests innerhalb des Datenflußnetzwerkes für die Effizienz des Gesamtalgorithmus ein entscheidender Faktor. Um diese Zugriffe trotzdem möglichst effizient ausführen zu können, wurde für *ARI* eine spezielle Schnittstelle zum LUIGI-Objektsystem entwickelt, die Datenzugriffe wenn möglich in indizierte Zugriffe auf Vektoren abbildet. Dies ist allerdings nicht immer möglich. Bei Ausnutzung der Flexibilität des Objektsystems durch Verwendung des ***of-subclass-of*** Konstruktes der *ARI*-Mustersprache für Objektinstanzen, von denen nur eine Oberklasse aber nicht die zugehörige Klasse bekannt ist, kann der Compiler keine Indizes für Zugriffe berechnen, da die innere Struktur der Instanz zur Übersetzungszeit nicht bekannt ist. *ARI* realisiert Zugriffe auf solche Objektinstanzen assoziativ über Konstituentennamen, als Kompromiß für die größere Ausdrucksstärke muß also ein etwas langsamerer Wertzugriff in Kauf genommen werden.

Die Repräsentation des RETE-Netzwerkes folgt in *ARI* nicht der **OPS5**-Implementierung als linearisierte Maschinensprache im Sinne einer virtuellen Maschine, sondern stellt das Netz intern durch Vektoren als Knoten und Pointer als Kanten dar. Der **OPS5**-Ansatz liefert sicherlich entscheidende Vorteile, wenn die Netzwerksprache auf möglichst niedrige Maschinenebene *"abgesenkt"* werden kann, auf der spezielle Mechanismen einer konkreten Rechnerarchitektur (*Paging*-Algorithmen, *Cache*-Speicher, usw.) ausgenutzt werden können. Für *ARI* gibt es jedoch keine konkrete Zielarchitektur und so wurde die Repräsentation in einer portierbaren höheren Sprache (COMMON-LISP) gewählt, in der die Formulierung einer virtuellen Maschine für ein symbolisch repräsentiertes Datenflußnetzwerk keinerlei Effizienzvorteile bringt.

3.3.2.1. Verarbeitung von Pattern für unterschiedliche Situationsbeschreibungen

ARI unterscheidet sich von **OPS5** vor allem darin, daß für die vielfältigen Konstrukte zur Strukturierung und Formalisierung von Kontrollwissen eine Vielzahl unterschiedlicher Situationsbeschreibungen möglich ist, während in **OPS5** nur der Bedingungsteil von Inferenzregeln mit *pattern* spezifiziert wird.

Um die Interpretation unterschiedlicher Situationsbeschreibungen effizient realisieren zu können lassen sich verschiedene neue Typen von Endknoten in den Netzwerkformalismus einfügen. Endknoten führen situationstypspezifische Operationen für ankommende Instanziierungen von *pattern* aus und verfügen über spezielle Datenstrukturen zum Zwischenspeichern relevanter Information. ***ARI*** benutzt diese Technik zur spezifischen Verarbeitung von Inferenzregeln, Kontrolltaktiken und Kontrollstrategien sowie für die Vorbedingung, Zielbeschreibung, Import- und Exportspezifikation von *problem-solvern*. Diese unterschiedlichen Sprachkonstrukte werden jeweils in spezielle Teilnetze übersetzt. Die Teilnetze enden in Knoten, die die zugehörigen Situationen identifizieren und entsprechend verarbeiten.

Am Beispiel der abstrakten Beschreibung des folgenden *problem-solvers* SATISFY-GOAL soll dies verdeutlicht werden.

```
PROBLEM-SOLVER SATISFY-GOAL
  SOLVES-PROBLEM-WITH
    PRECONDITION
      OBJECT ?goal
       OF-CLASS GOAL WITH
         goal-status = UNSATISFIED
    GOAL
        OBJECT ?goal WITH
         goal-status = SATISFIED
      OR AFFAIR ?*
         operation-failed ()
```

Die Übersetzung dieser abstrakten Beschreibung erzeugt für die Vorbedingung ein Teilnetz mit einem *precondition-node* als Endknoten, sowie für beide Teile der disjunktiven Zielbeschreibung je ein Teilnetz mit je einem *goal-node*. Während der Interpretation ist die Vorbedingung von SATISFY-GOAL auf der Ebene der RETE-Implementierung genau dann erfüllt, wenn ein Token den *precondition-node* erreicht. Dieses Token beinhaltet dann eine Instanz der Klasse GOAL mit dem Wert `UNSATISFIED` in der Konstituente `goal-status`. Die Operation bei der Interpretation des *precondition-node* besteht darin, einen Aktivierungsblock (***activation-record***) mit der Bindung für die Variable `?goal` in die Konfliktmenge aktivierbarer *problem-solver* einzutragen. Wird SATISFY-GOAL vom Interpreter aktiviert, so wird dieser Aktivierungsblock dazu benutzt, den Test des ersten *goal-nodes* zu instanziieren. Während der Interpretation von SATISFY-GOAL bewirkt das Eintreffen eines Tokens am *goal-node* die Deaktivierung gemäß des in Abschnitt 3.2.4. vorgestellten Interpretationsmodells.

3.3.2.2. Techniken zur Verarbeitung von Meta-Pattern

Zur Interpretation von Kontrolltaktiken und Kontrollstrategien muß der *pattern-matching*-Algorithmus um die Verarbeitung von *meta-pattern*, also *domain-rule-pattern* und *problem-solver-pattern*, erweitert werden. In der RETE-Realisierung von ***ARI*** wird dies durch die Konstruktion zusätzlicher Teilnetzwerke erreicht, in die Regelinstanzen bzw. instanziierte *problem-solver* eingeschleust werden. Da sowohl im Bedingungsteil von Kontrolltaktiken als auch von Kontrollstrategien *domain-pattern* vorkommen können, müssen auch Token für Sachverhalte oder Objektinstanzen in diesen Netzen verarbeitet werden können. Die Struktur der Netze ist deshalb ganz analog der für Bedingungsteile von Objektregeln. Für jedes *meta-pattern* wird jedoch statt einer Folge von *alpha-memory-nodes* genau ein spezieller Knoten eingeführt, der Tests auf eingeschleusten *pattern*-Instanzen ausführt. Für *domain-rule-pattern* heißen diese ***domain-rule-description-nodes*** und sind im Netz Nachfolger von *domain-rule-nodes*, für *problem-solver-pattern* heißen sie analog ***problem-solver-description-nodes*** und sind Nachfolger von *precondition-nodes*.

Die Endknoten der Teilnetze zur Verarbeitung von Kontrollregeln, entsprechend mit *control-tactic-node* und *control-strategy-node* bezeichnet, fügen ankommende Token mit Instanzen von Meta-Regeln in Listen anzuwendender Kontrolltaktiken bzw. Kontrollstrategien ein, die vom Interpreter verwaltet werden.

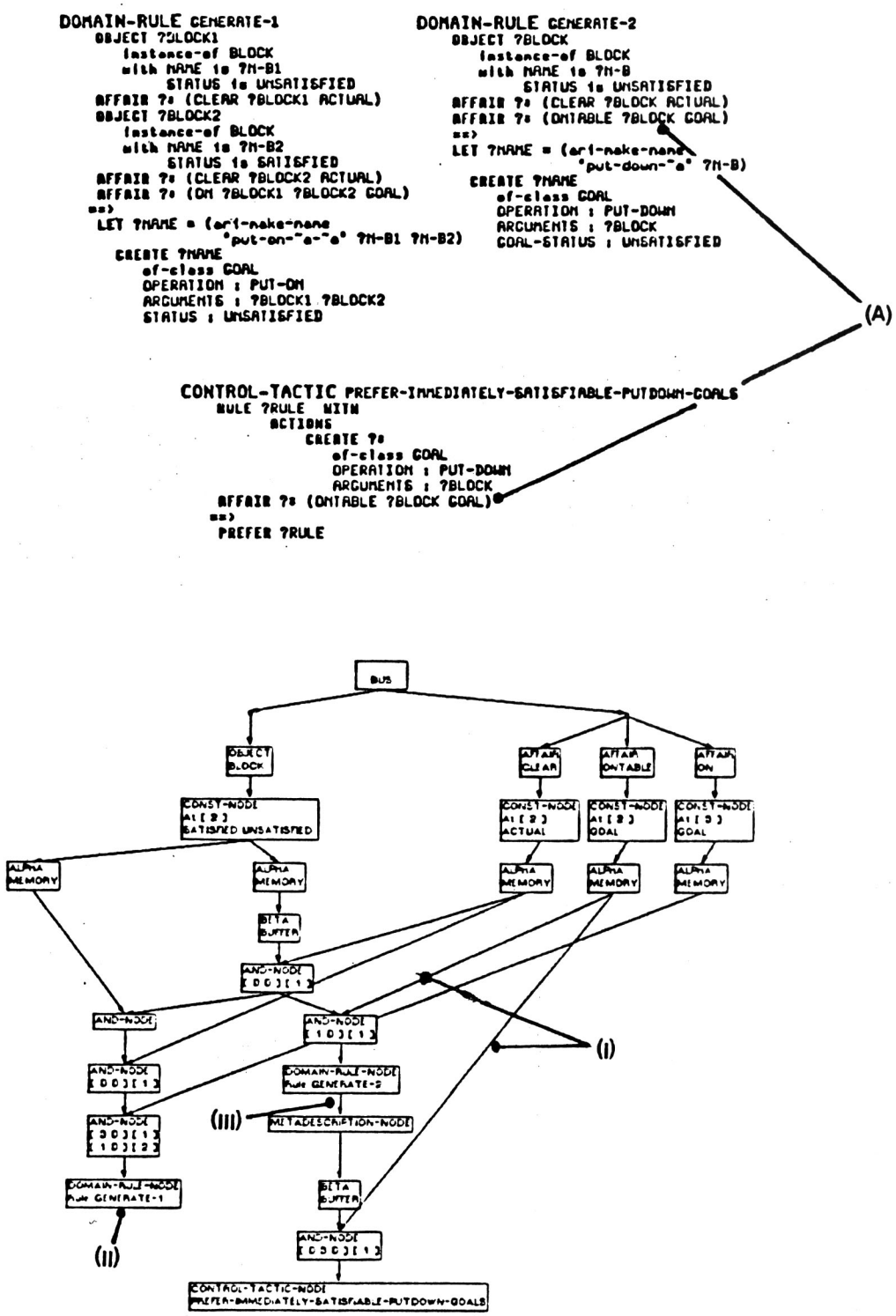


Abb. 3.15: Eine Kontrolltaktik und ihre Übersetzung

Das beschriebene Prinzip wird im Beispiel in Abb. 3.15 verdeutlicht, das aus einer Demonstrationsanwendung zum Planen in der "Klötzchenwelt" stammt. Die Kontrolltaktik beschreibt im

Bedingungsteil eine Regel, die als Aktion eine Instanz der Klasse GOAL mit der OPERATION PUT-DOWN erzeugt. Die Intention dieser Taktik ist es, Regeln zu bevorzugen, die einen Block auf den Tisch stellen, der auch in der Zielbeschreibung des Problems auf dem Tisch stehen soll.

An diesem Beispiel lassen sich auch die beiden Techniken erläutern, die in ***ARI*** eingesetzt werden, um eine möglichst effizientes *pattern-matching* für *meta-pattern* zu erreichen:

(1) Strukturgleichheit zwischen *pattern* in Objektregeln und Kontrolltaktiken wird vom ***ARI***-Compiler in *sharing* für Knoten umgesetzt, d.h. die RETE-Knoten für *domain-* und *meta-pattern* werden zusammengefaßt. Im Beispiel gilt das für den Pfad im Netz, der den Sachverhalt (ONTABLE ?BLOCK GOAL) testet (s. **(A)** und **(I)**).

(2) Zur Übersetzungszeit können *rule-pattern* in einer Kontrolltaktik mit allen Objektregeln im gleichen *problem-solver* verglichen werden. Durch diesen Vergleich kann der Compiler bereits alle Tests zwischen *meta-pattern* und Regel ausführen, die keine Bindungen von Variablen innerhalb der Objektregel betreffen. Dadurch kann die Menge der Objektregeln, die ein *rule-pattern* zur Interpretationszeit *matchen* können, bereits zur Übersetzungszeit drastisch eingeschränkt werden. Im Beispiel kann also der Compiler berechnen, daß die Objektregel GENERATE-2 zur Interpretationszeit das *rule-pattern* der Kontrolltaktik *matchen* kann, nicht aber GENERATE-1. Im Netz zeigt sich dies in der Tatsache, daß nur für den *domain-rule-node* von GENERATE-2 ein *domain-rule-description-node* als Nachfolger angelegt wurde (s. **(II)** und **(III)**).

Domain-rule-description-nodes führen im allgemeine diejenigen Tests auf Regelinstanzen aus, die im Bedingungsteil der Regel gebundene Variablen betreffen. Im Beispiel ist kein Test mehr auf der Regelinstanz von GENERATE-2 auszuführen, da die Konsistenzüberprüfung für die Variable ?BLOCK bereits im Teilnetz für domain-pattern durchgeführt wird. Dies ist also ein Beispiel dafür, wie eine Kontrolltaktik durch obige Techniken mit minimalem Mehraufwand zur Interpretationszeit verarbeitet werden kann. Für Kontrollstrategien und *problem-solver-description-nodes* gelten die gleichen Prinzipien analog.

3.3.2.3. Verarbeitung strukturierter Regelbasen

Die Strukturierung von Regelbasen, wie sie durch *problem-solver* vorgegeben ist, erlaubt es, beim *pattern-matching* sowohl die betrachtete Menge der *pattern*, als auch die betrachtete Menge der Wissenseinheiten einzuschränken. Die Menge der *pattern* ist zu jedem Zeitpunkt der Interpretation durch die zum aktivierten problem-solver gehörenden *pattern* beschränkt, die Menge der Wissenseinheiten durch die in der aktuellen Umgebung vorkommenden.

Um dieses Prinzip auf der Implementierungsebene im verwendeten RETE-Algorithmus auszunutzen, mußten entsprechende Mechanismen zur Partitionierung des RETE-Datenflußnetzwerkes entwickelt werden. In ***ARI*** wurde der folgende Ansatz gewählt:

- Die strukturellen Bestandteile von *problem-solvern* entsprechen bestimmten Teilnetzen im gesamten Datenflußnetzwerk. Diese Teilnetze müssen entsprechend dem Operationsprinzip des Interpretierers aktiviert und deaktiviert werden können.
- Die Aktivierung bzw. Deaktivierung eines Teilnetzes wird realisiert durch die Aktivierung bzw. Deaktivierung aller *alpha-memory-nodes* die auf Pfaden in dieses Teilnetz führen. *Alpha-memory-nodes* sind zu diesem Zweck mit einem Schalter versehen, der vom Interpreter entspechend gesetzt werden kann. Ein aktivierter *alpha-memory-node* arbeitet wie im Basisalgorithmus beschrieben, ein deaktivierter *alpha-memory-node* wirft ankommende Token weg. Diese Zuordnung von *alpha-memory-nodes* zu Teilnetzen und die Arbeitsweise dieser Knoten als "Schleusen" für die Teilnetze

impliziert, daß *sharing* von *alpha-memory-nodes* und aller nachfolgenden Knoten nur noch innerhalb von zusammengehörenden Teilnetzen möglich ist. *Sharing* von alpha-Knoten außer *alpha-memory-nodes* ist hingegen weiterhin für das gesamte RETE-Netzewerk möglich. Diese Einschränkung der Möglichkeiten für *sharing* von beta-Knoten ist kein entscheidender Nachteil, da der selbe Effekt in der RETE-Realisierung von **OPS5** auftritt, wenn das erste Bedingungselement von Regeln als Indikator für einen Kontext verwendet wird.

- Die Teilnetze, die der Import- und Exportspezifikation eines *problem-solvers* entsprechen, enden in einem ***import-node*** bzw. einem ***export-node***. Für die Aktivierung und Deaktivierung dieser Teilnetze nach dem soeben beschriebenen Prinzip muß jedoch beachtet werden, daß das Import-Teilnetz dann aktiviert werden muß, wenn der übergeordnete *problem-solver* aktiviert wurde, das Export-Teilnetz hingegen dann, wenn der *problem-solver* selbst aktiviert wird. Zur Realisierung der Kommunikation zwischen *problem-solvern* speichert ein *import-node* alle ankommenden Token in einer Schlange, die dem zugehörigen *problem-solver* zugeordnet ist, ein *export-node* alle ankommenden Token in einer Schlange die dem übergeordneten *problem-solver* zugeordnet ist.

3.3.2.4. Zusammenfassung der RETE-Erweiterungen in *ARI*

In diesem Abschnitt sollen die Erweiterungen des RETE-Algorithmus, insbesondere die zusätzlich eingeführten Knoten noch einmal im Überblick dargestellt werden. Zuvor werden noch kurz einige effizienzsteigernde RETE-Erweiterungen aus der Literatur angegeben, die in die *ARI*-Implementierung übernommen wurden.

(1) Datenelemente oder Fakten in **OPS5**, aber auch die Wissenselemente in **LUIGI** tragen ein festes Attribut, das ihnen immer zugeordnet ist. Für **OPS5**-Fakten ist dies der Faktenname, der an erster Stelle steht und für jedes Datenelement angegeben sein muß, für **LUIGI**-Sachverhalte ist dies der Sachverhaltsname und für Objektinstanzen der Name der zugehörigen Klasse. Diese einem Wissenselement anhaftende Information kann direkt als Indexinformation über den Einstiegspunkt eines Tokens für das Wissenselement benutzt werden und vom Compiler beim jeweiligen Symbol abgelegt werden. Dadurch werden Knoten und damit Tests zum Bestimmen dieses jeweiligen Attributs für alle Wissenseinheiten eingespart. In obigem Beispiel für *meta-pattern* sind die in Abb 3.15 gezeigten BUS-, OBJECT- und AFFAIR-Knoten also nur virtuell vorhanden. Mit einer ähnlichen Technik läßt sich durch die Verwendung eines case-Konstruktes innerhalb eines *constant-node* für alle möglichen Token der Nachfolgeknoten direkt berechnen (s. /Scales 86/, p. 33).

(2) Zur Modifikation der Konstituentenwerte von Objektinstanzen wurde in ***ARI*** die sogenannte ***update-in-place*** Technik adaptiert (s. /Schorr et al. 86/), die neben *add-* und *delete*-Operationen für Token eine *modify*-Operation vorsieht mit der das Anpassen des Netzes an Modifikationen durch das Propagieren eines einzigen Tokens anstelle einer Kombination aus *delete* und *add* durchgeführt werden kann.

(3) Eine Technik zur effizienten Ausführung des Aktionsteils einer Regel ist das Transformieren der Folge von Aktionen in eine compilierte LISP-Funktion, die die Variablenbindungen aus dem Bedingungsteil als Parameter übernimmt (s. /Allen-83/). Dadurch wird jeglicher Effizienzverlust durch symbolische Interpretation vermieden.

Die Modifikationen an bzw. die Einführung von Knotentypen in der RETE-Implementierung von ***ARI*** sind in der folgenden Tabelle aufgelistet:

- ***alpha-memory-node***:
  ein *alpha-memory-node* in ***ARI*** ist mit einem aktiv/inaktiv-Schalter versehen und verarbeitet Token nur, wenn er aktiviert ist. Somit kann eine Menge von *alpha-memory-nodes* als *Schleuse* benutzt werden, die ein gesamtes Teilnetz deaktiviert.
- ***beta-test-nodes***:
  beta-test-nodes sind in Testpfaden zwischen beta-Knoten plaziert sind und führen zusätzliche Tests auf Variablen verschiedener Bedingungselemente aus. Sie realisieren die *test-pattern* aus Situationsbeschreibungen.
- ***precondition-node***:
  ein *precondition-node* für einen *problem-solver* ist aktiv, wenn der übergeordnete *problem-solver* aktiv ist. Der Knoten konstruiert für ankommende Instanzen von Vorbedingungen einen Aktivierungsblock, der in die Konfliktmenge anwendbarer *problem-solver* eingetragen wird und der die durch Instanziierung aufgebauten Variablenbindungen enthält. Der Knoten propagiert außerdem die Instanz des *problem-solvers* an nachfolgende *ps-description-nodes*.
- ***goal-node***:
  die Tests des *goal-node* eines *problem-solvers* werden bei dessen Aktivierung durch den vom *precondition-node* aufgebauten Aktivierungsblock instanziiert und speichern beim Eintreffen eines Tokens in einer Variablen des Interpreters, daß die Zielbeschreibung des *problem-solvers* erfüllt ist. Dies wirkt dann als Deaktivierungsbedingung zwischen zwei Interpretationszyklen.
- ***import-node***:
  ein *import-node* eines *problem-solvers* ist ebenfalls zusammen mit dem übergeordnete *problem-solver* aktiv. Ankommende Token werden in einer dem eigenen *problem-solver* zugeordneten Schlange gespeichert und bei dessen Aktivierung in das Netzwerk eingeschleust.
- ***export-node***:
  ein *export-node* wird wie ein *goal-node* bei der Aktivierung des zugehörigen *problem-solvers* durch den Aktivierungsblock instanziiert und speichert während der Interpretation des *problem-solvers* alle ankommenden Token. Diese werden bei der Deaktivierung des *problem-solvers* als Exportwissen an den übergeordneten *problem-solver* übergeben.
- ***domain-rule-node***:
  ein *domain-rule-node* fügt die Regelinstanzen aus ankommenden Token in die vom Interpreter verwaltete Liste von anwendbaren Objektregeln ein. Zusätzlich werden die Regelinstanzen an nachfolgende *domain-rule-description-nodes* weiterpropagiert.
- ***control-tactic-node***:
  ein *control-tactic-node* fügt die instanziierten Kontrolltaktiken aus ankommenden Token in die vom Interpreter verwaltete Menge anzuwendender Kontrolltaktiken ein.
- ***control-strategy-node***:
  ein *control-strategy-node* fügt die instanziierten Kontrollstrategien aus ankommenden Token in die vom Interpreter verwaltete Menge anzuwendender Kontrollstrategien ein.
- ***domain-rule-description-node***:
  ein *domain-rule-description-node* führt wie ein alpha-Knoten Tests auf einem Token aus. Dies sind Tests auf Variablenbindungen der im Token enthaltenen Regelinstanz.
- ***ps-description-node***:
  ein *ps-description-node* führt Tests auf der im ankommenden Token enthaltenen Instanz eines *problem-solvers* aus.

3.4. Eine konzeptionelle Ebene für ARI

In Abschnitt 2.2.1 wurden die Möglichkeiten zur Realisierung einer konzeptionellen Ebene für wissensbasierte Systeme diskutiert. Eine solche Ebene kann die Modellierung einer Anwendungswelt problemspezifisch mit Repräsentationssprachen unterstützen, welche die spezielle Terminologie einer Diskurswelt mit elementaren Wissenstrukturen beschreiben läßt. Warum ist aber eine epistemologische Ebene wie die von ***ARI*** nicht ausreichend?

ARI stellt eine Menge von Wissensstrukturen für beschreibendes Wissen, Schlußfolgerungsprinzipien und Problemlösungsstrategien zur Verfügung. Der Anwendungsbereich und die **generische Problemklasse** (vgl. /Chandrasekaran 87/), die mit einer Anwendung gelöst werden soll, bestimmen für ein konkretes wissensbasiertes System aber in entscheidender Weise, welche dieser Wissensstrukturen den Aufbau einer Wissensbasis und die Spezifikation einer geeigneten Kontrollstrategie erleichtern (z.B. die *Establish-Refine*-Strategie für das Problem der hierarchischen Klassifikation).

Verwendet man die ganze Vielfalt der Strukturen auf epistemologischer Ebene ohne sich auf die für die vorliegende Problemklasse geeigneten zu beschränken und diese explizit herauszuarbeiten, so wird die Transparenz und Erklärungsfähigkeit der resultierenden Systeme stark darunter leiden, zeitaufwendigere Konstruktion und Wartung sind die natürliche Folge.

Kommerzielle Werkzeuge mit problemspezifischen Repräsentationssprachen begegnen diesem Problem, stellen jedoch nur eine eingeschränkte, feststehende Menge solcher anwendungsbezogenen Konstrukte zur Verfügung und sind deshalb zu inflexibel, wenn es gilt, die Menge der Wissensstrukturen zu erweitern. Dies ist aber für viele Diskurswelten notwendig, da die für sie bedeutsamen Phänomene und entsprechenden Wissensstrukturen zu Beginn einer Modellierung nicht alle bekannt sind.

Für die Bürowelt sind beispielsweise Phänomene wie *Aktivitäten*, *Vorgänge*, *Organisationseinheiten*, *Stellen* und *Firmenrichtlinien* elementare Wissensstrukturen, die bei einer Modellierung sicherlich eine Rolle spielen, aber eine Formalisierung dieser Phänomene und ihrer Rolle im Büro ist bisher nur zum Teil gelungen (vgl. /Lutze 88/).

Die Möglichkeit der Modellierung flexibler generischer Problemklassen und die Unterstützung terminologisch adäquater Wissensstrukturen erlaubt es, Anwendungsexperten direkt in den Prozeß der Konstruktion wissensbasierter Systeme einzuschließen, da sie bei der Repräsentation ihres Wissens mit gewohnten Beschreibungsmitteln arbeiten können. Die epistemischen Primitiva in ***ARI*** sind jedoch sicherlich zu komplex, als daß sie von Anwendungsexperten direkt in einer methodisch adäquaten Weise benutzt werden könnten.

Die Wissensstruktur des *problem-solvers*, die in ***ARI*** die Spezifikation funktionaler Einheiten zur Lösung von Problemklassen erlaubt, entspricht jedoch durchaus den Entwurfskriterien für problemspezifische Sprachen auf einer konzeptionellen Ebene und es läßt sich für den ***ARI***-Ansatz ein Szenario entwickeln, wie die epistemischen Primitiva, insbesondere die Wissensstruktur des *problem-solvers*, eine Implementierungsbasis für konzeptionelle Sprachen bilden können. Da für *problem-solver* spezifische beschreibende Wissensstrukturen, Typen von Inferenzregeln und geeignete Kontrollstrategien definiert werden können, ist es möglich, auf der konzeptionellen Ebene formulierte Problemlösungsprozesse in entsprechende *problem-solver* zu transformieren. Im Idealfall lassen sich also konzeptionelle Sprachen als eine "Macro-Sprache" definieren, die in Wissensstrukturen auf der epistemologischen Ebene expandiert werden kann.

4. Implementierung des Inferenzsystems *ARI*

In diesem Kapitel wird die Realisierung von ***ARI*** als Softwaresystem beschrieben. Dabei soll zuerst die Integration des ***ARI***-Systems in die Gesamtarchitektur von **LUIGI** aufgezeigt werden, bevor das Inferenzsystem selbst in seinem modularen Aufbau charakterisiert wird. In einem weiteren Abschnitt werden Teile der aktuellen Entwicklungsumgebung für ***ARI*** erläutert und in Form von Bildschirminhalten visualisiert, um einen Eindruck der konkreten Arbeitsweise des Systems zu vermitteln.

4.1. ARI innerhalb der LUIGI-Architektur

Das Werkzeugsystem **LUIGI** wurde in Abschnitt 3.1. bereits kurz als Umgebung zur Entwicklung wissensbasierter Systeme im Büro vorgestellt. **LUIGI** ist momentan als Prototyp auf SYMBOLICS LISP-Maschinen implementiert und ablauffähig. Das System ist bis auf die Benutzerschnittstelle vollständig im COMMON-LISP implementiert und wird zur Zeit als Ablaufumgebung auf SUN-Workstations portiert. Das Gesamtsystem läßt sich folgende Komponenten zerlegen (s. Abb. 4.1):

- Das **Wissensrepräsentationssubsystem** dient der Repräsentation und Verwaltung von Wissenseinheiten.
- Das **Inferenzsystem *ARI*** realisiert das Problemlösen mit Schlußfolgerungsprinzipien.
- Die einheitliche **Entwicklungsumgebung LUKE** dient als Schnittstelle zum System und verwaltet eine Menge verschiedener **LUKE-Anwendungen**.

Jeder **LUKE**-Anwendung ist ein spezielles Fenster mit fest vorgegebener Funktionalität zugeordnet. **LUKE**-Anwendungen sind momentan eine Reihe von wissensstrukturspezifischen Editoren und eine Entwicklungsumgebung für ***ARI***. Für ***ARI*** gibt es also zum einen die Sichtweise als Inferenzsystem und zum anderen als Entwicklungsumgebung innerhalb von **LUKE**.

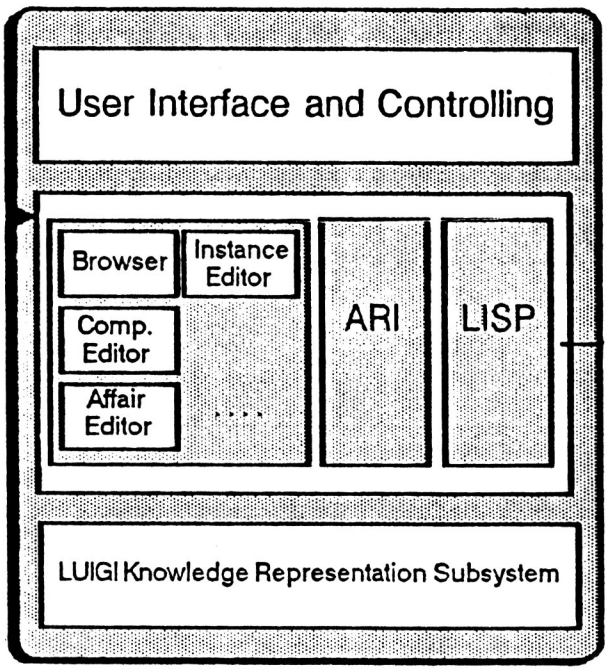


Abb. 4.1: Architektur der LUIGI-Entwicklungsumgebung

Das Inferenzsystem *ARI* benutzt die vom LUKE-Kern bereitgestellte Funktionalität zur Benutzerinteraktion. Eine Schnittstelle zum LUIGI-Objektsystem erlaubt den Zugriff auf die dort verwalteten Repräsentationen für Wissenseinheiten. Das Inferenzsystem selbst wird über die *ARI*-Entwicklungsumgebung angesprochen. Die Funktionalität des Inferenzsystems umfaßt den Aufruf von *ARI*-Compiler und *ARI*-Interpreter sowie Funktionen für Testwerkzeuge (Debugger). Daraus ergibt sich für *ARI* die in Abb. 4.2 beschriebene Stellung innerhalb der LUIGI-Gesamtarchitektur.

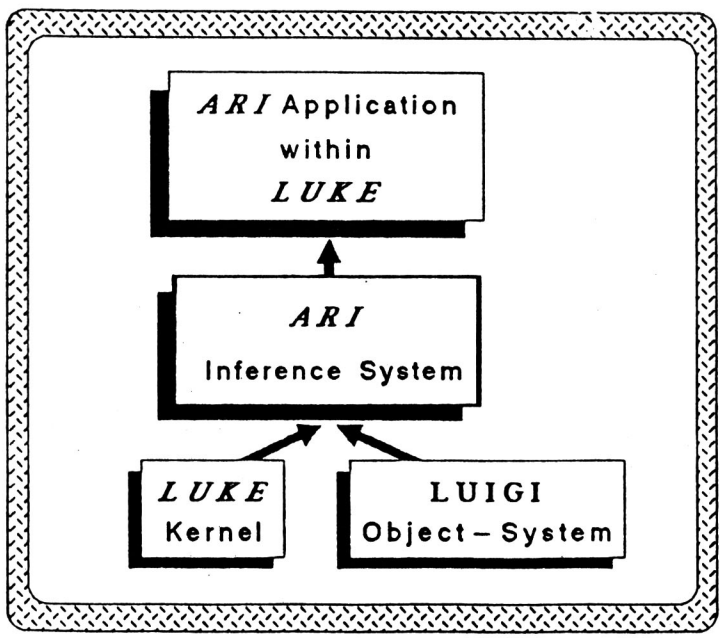


Abb. 4.2: ARI innerhalb der LUIGI-Gesamtarchitektur

4.2. Die ARI Systemstruktur

Das *ARI*-Inferenzsystem ist auf einer SYMBOLICS LISP-Maschine entwickelt worden und ist vollständig in COMMON-LISP implementiert. Die Systemstruktur des Systems ist in Abb. 4.3. als Hierarchie über sechs Ebenen von Moduln aufgezeigt und daneben noch einmal als SYMBOLICS-COMMON-LISP Systemdefinition (`defsystem`). Entlang der Pfeile bestehen Daten- und Funktionsabhängigkeiten, wobei nicht alle Abhängigkeiten explizit dargestellt sind, sondern implizit durch den transitiven Abschluß aller Pfeile gegeben sind.

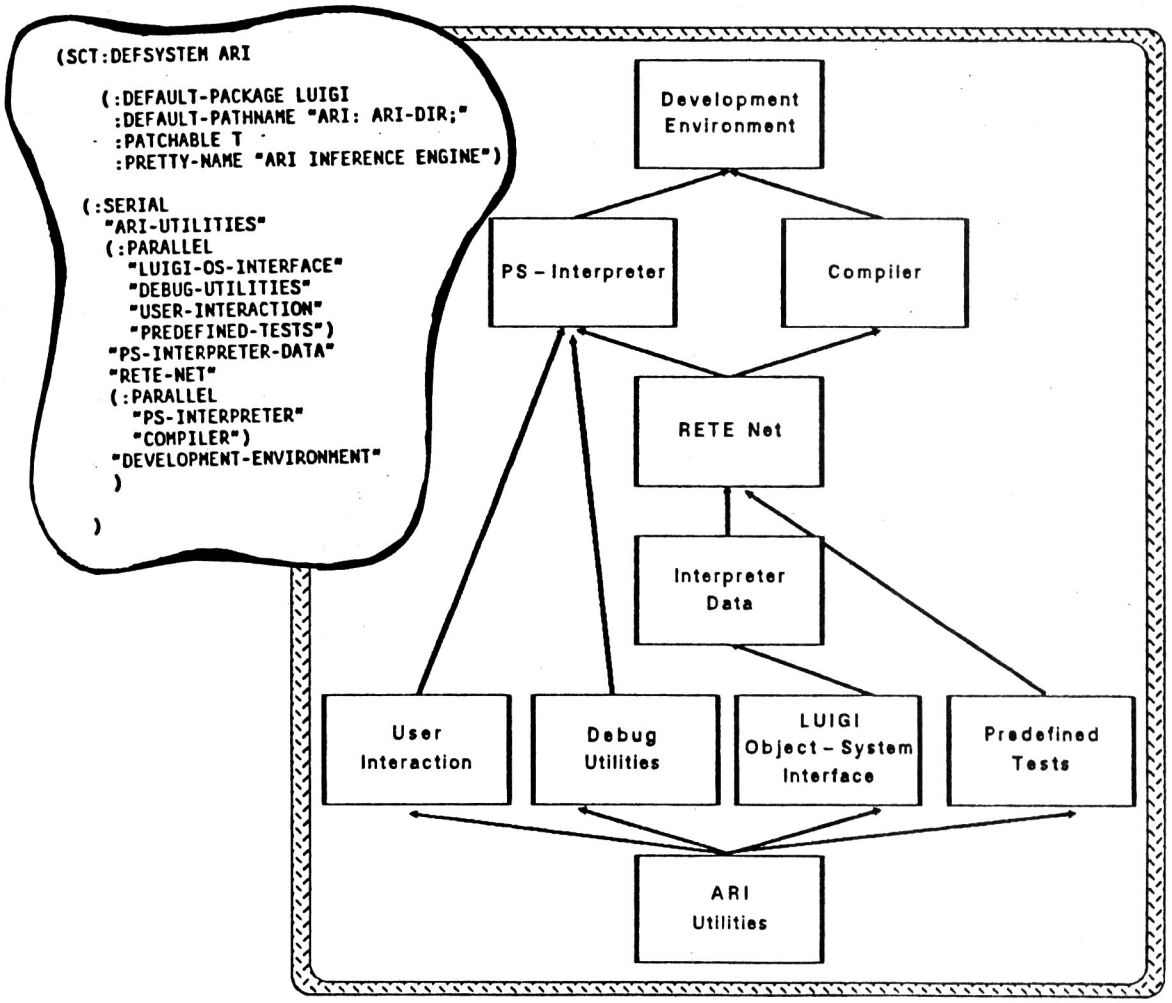


Abb. 4.3: Die Systemstruktur des ARI-Inferenzsystems

Auf der nächsten Seite werden die Funktionalität der einzelnen Moduln bzw. die in ihnen verwalteten Datenstrukturen beschrieben.

- **module** *ARI-UTILITIES*:
  ARI-UTILITIES enthält eine Reihe von Hilfsfunktionen, die im System an verschiedenen Stellen gebraucht werden. Unter anderem ist ein abstrakter Datentyp QUEUE für Schlangen implementiert.
- **module** *LUIGI OBJECT-SYSTEM INTERFACE*:
  LUIGI OBJECT-SYSTEM INTERFACE stellt für alle anderen Moduln höherer Ebene die funktionale Schnittstelle zum **LUIGI**-Objektsystem dar. Um eine möglichst klar definierte Abhängigkeit vom zugrundeliegenden Wissensrepräsentationssystem zu erhalten, werden alle Aufrufe von Funktionen des Objektsystems innerhalb des Inferenzsystems über diese Schnittstelle realisiert (eine Ausnahme bildet der Modul *PREDEFINED TESTS* auf gleicher Abhängigkeitsebene!). Ein Großteil der Funktionen dieses Moduls ist deshalb in Form von LISP-Macros realisiert, die direkt zu Funktionen des Objektsystems expandieren.
  Außerdem enthält dieser Modul Funktionen, die vom Interpreter für das RETE-Netz zum effizienten Zugriff auf die Daten von Wissenseinheiten benutzt werden.
- **module** *DEBUG UTILITIES*:
  DEBUG UTILITIES enthält eine Reihe von Funktionen und globaler Variablen, über die die *ARI*-Entwicklungsumgebung auf Zustandsinformationen des Interpreters zugreift. Da diese Daten von mehreren Moduln aus referenziert und manipuliert werden, wurden sie in diesem Modul separiert.
- **module** *PREDEFINED TESTS*:
  PREDEFINED TESTS enthält Funktionen, die als Testprädikate in *object-pattern* und *test-pattern* der *ARI*-Repräsentationssprache verwendet werden können. Ein Großteil der Funktionen dient als funktionale Schnittstelle zwischen der *ARI*-Sprache und dem Objektsystem. Beispiele für Prädikate sind INSTANCE-OF-CLASS-P und INSTANCE-OF-SUBCLASS-P.
- **module** *USER INTERACTION*:
  USER INTERACTION stellt die Schnittstelle zwischen dem *ARI*-Interpreter und der Benutzeroberfläche der **LUKE**-Umgebung dar. Es werden Funktionen für die Benutzerinteraktionen auf den rechten Seiten von Inferenzregeln bereitgestellt.
- **module** *INTERPRETER-DATA*:
  INTERPRETER-DATA enthält globale Daten des *ARI*-Interpreters. Darunter fallen:
  - die Konfliktmengen anwendbarer Objektregeln und aktivierbarer *problem-solver*,
  - die Daten der Umgebung des aktivierten *problem-solvers*,
  - Tabellen mit den Aktivierungsdaten aller *problem-solver* und
  - eine Tabelle mit den Einstiegsknoten in das RETE-Netz (vgl. Abschnitt 3.3.2.4.).
- **module** *RETE-NET*:
  RETE-NET enthält die Datenstrukturen für alle Knotentypen des RETE-Datenflußnetzwerkes, sowie Funktionen für die jeweiligen Knotenalgorithmen, welche die Testoperationen auf den Token ausführen und Token an Nachfolgeknoten weiterpropagieren.
- **module** *PS-INTERPRETER*:
  PS-INTERPRETER enthält die Implementierung des in Abschnitt 3.2.4. beschriebenen Interpretationsmodells für die *ARI*-Inferenzmaschine, realisiert also die Algorithmen 3.1 - 3.3 als LISP-Funktionen.
- **module** *COMPILER*:
  COMPILER enthält den Übersetzer, der eine Regelbasis in ein RETE-Datenflußnetzwerk transformiert.
- **module** *DEVELOPMENT ENVIRONMENT*:
  DEVELOPMENT ENVIRONMENT enthält alle Funktionen mit denen die *ARI*-Entwicklungsumgebung als Anwendung innerhalb der **LUKE**-Umgebung auf das Inferenzsystem zugreifen kann.

4.3. Die ARI Entwicklungsumgebung

Die verschiedenen Werkzeuge der Entwicklungsumgebung als LUKE-Anwendungen sind

- ein **Klassenbrowser** für das LUIGI Objektsystem,
- ein **Objektklasseneditor,**
- ein **Instanzeneditor,**
- ein **Sachverhaltseditor** und
- die **Entwicklungsumgebung** für die Inferenzmaschine *ARI*.

Zusätzlich ist noch ein Browser zur Visualisierung und Inspektion des RETE-Netzwerkes von der *ARI* Entwicklungsumgebung aus aktivierbar (siehe Abb. 4.9).

Jedem Werkzeug ist ein spezielles Fenster zugeordnet, das mehrmals instanziierbar ist. Alle Werkzeuge arbeiten auf der momentan geladenen LUIGI Wissensbasis. Der Zugriff auf die Wissensbasis und die Kommunikation zwischen den einzelnen Werkzeugen findet über die zentrale Kontrollkomponente der LUKE-Umgebung statt. Jedes Werkzeug stellt eine Menge von Operationen und Funktionen zur Verfügung, die durch Einträge in den Menüleisten der zugehörigen Fenstern angestoßen werden können. Abb. 4.4 zeigt die Fenster der Entwicklungsumgebung von *ARI*, des Klassenbrowsers, des Instanzeneditors und des Browsers für das RETE Netzwerk.

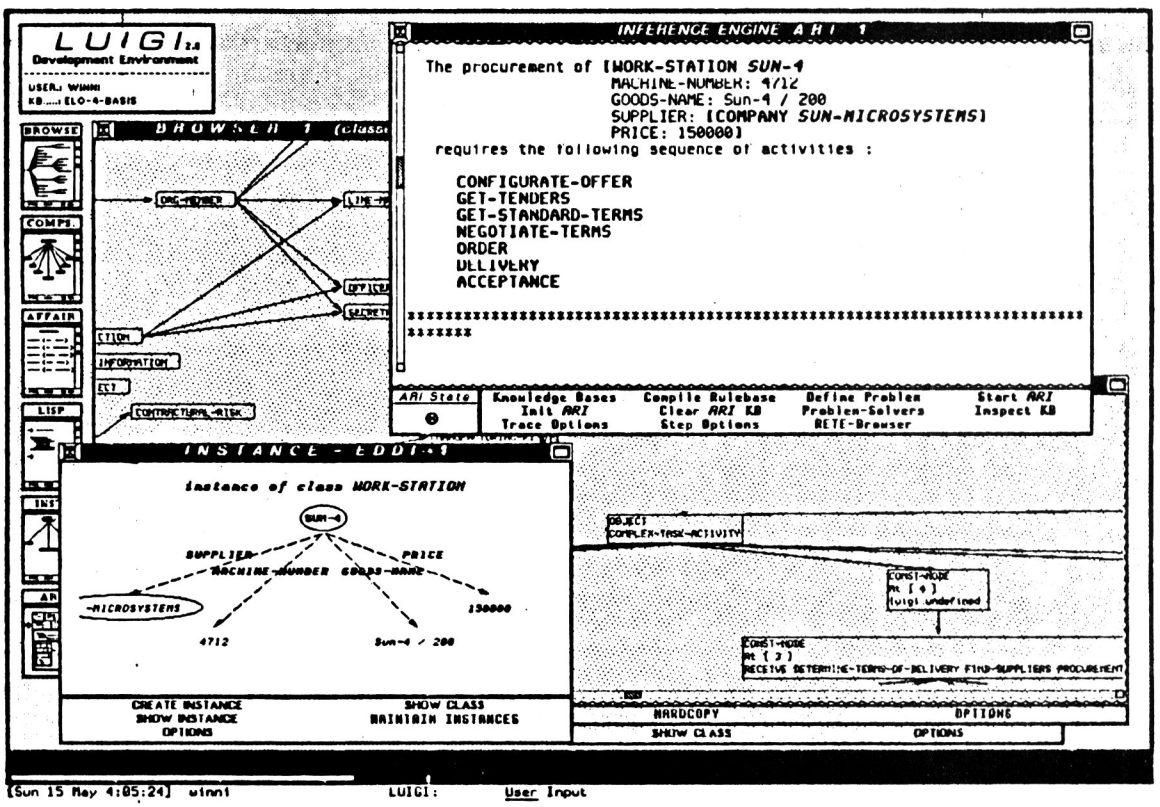


Abb. 4.4: Benutzerschnittstelle der LUIGI Entwicklungsumgebung

Die nachfolgende Abb. 4.5 zeigt die einzelnen Arbeitsschritte, die zur Interpretation einer Regelbasis auf einer beschreibenden Wissensbasis notwendig sind. Zunächst wird eine LUIGI Wissensbasis von einer Datei geladen und das *ARI* Inferenzsystem initialisiert. Der zweite Schritt ist das Laden einer Wissensbasis, die das Problemlösungswissen enthält. Über ein Menü wird anschließend ein zu übersetzender *problem-solver* ausgewählt und vom *ARI*-Compiler in ein entsprechendes RETE-Netzwerk

Netzwerk compiliert. Danach wird die Problemdefinition ebenfalls von einer Datei geladen und die darin enthaltenen Wissenseinheiten durch das RETE-Netzwerk propagiert. Der eigentliche Inferenzprozess kann dann durch den Eintrag "START ARI" im Hauptmenü des *ARI* Fensters angestoßen werden.

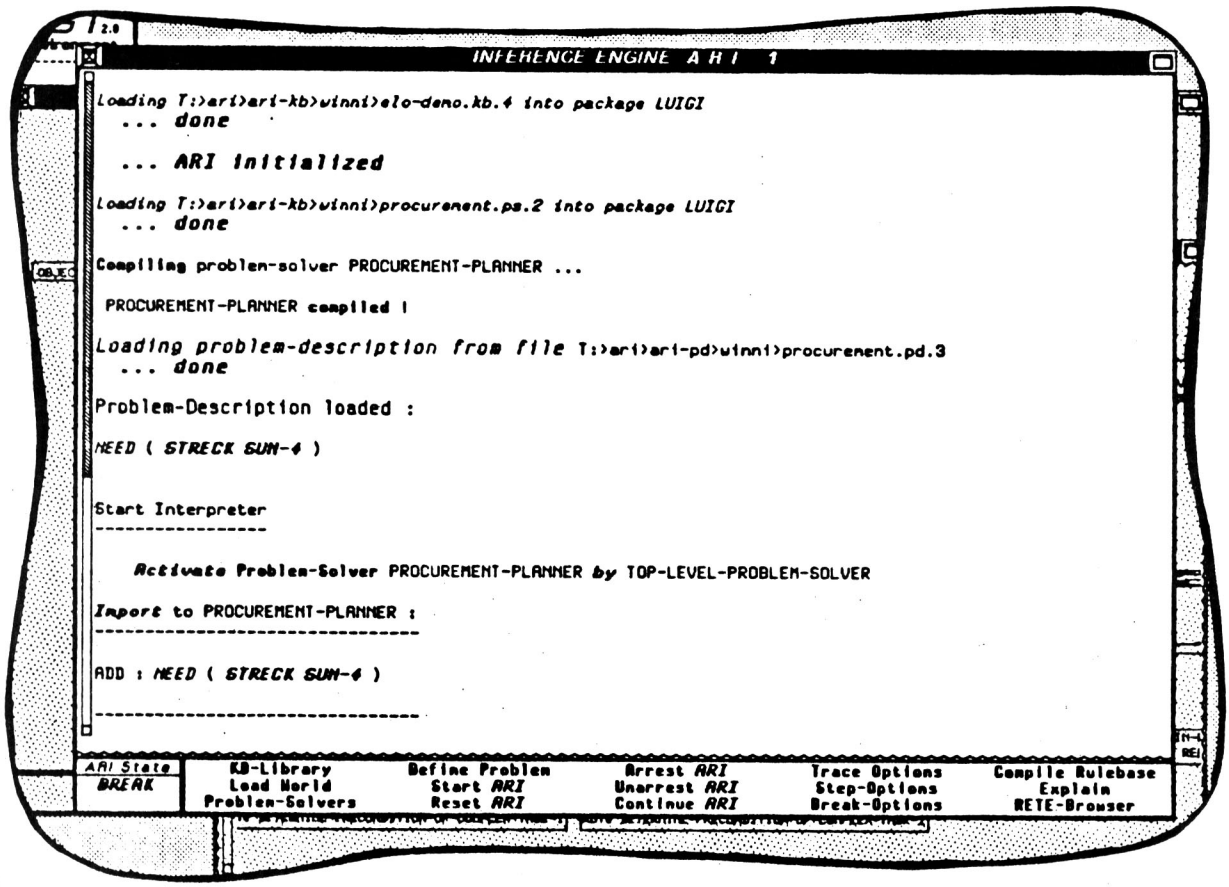


Abb. 4.5: Typische Vorgehensweise bei der Interpretation einer ARI Regelbasis

Alle Ausgaben der Problemlösungskomponente werden auf das *ARI* Fenster ausgegeben. Diese Ausgaben umfassen sowohl die des Anwendungssystems, die in den Aktionsteilen der Inferenzregeln spezifiziert werden, als auch die Ausgaben des *ARI* Inferenzsystems. Eingaben vom Benutzer werden durch "pop-up"-Menüs angefordert. Abb. 4.6 zeigt die Systemausgaben und eine aufgeblendete Auswahlfrage des Anwendungssystems.

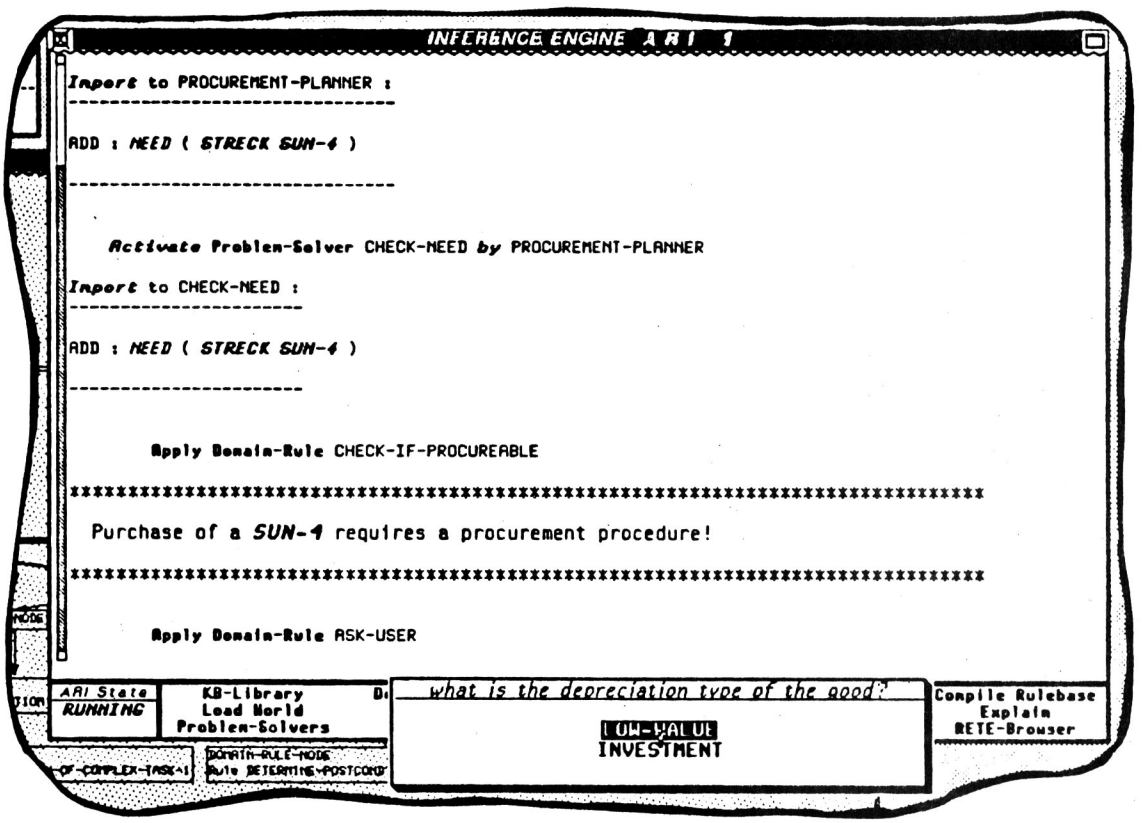


Abb 4.6: Benutzerinteraktion in der ARI Entwicklungsumgebung

Die Entwicklungsumgebung des *ARI* Inferenzsystems umfaßt derzeit einen *Trace*-Mechanismus, eine Komponente zur schrittweisen Interpretation von *ARI* Regelbasen (*Stepper*) und einen Inspektionsmechanismus für *problem-solver*. Der *Tracer* protokolliert den Problemlösungsprozess auf verschiedenen Abstraktionsebenen, die über ein Menü ausgewählt werden können. Der *Stepper* erlaubt das Anhalten des Problemlösungsprozesses und das Inspizieren des momentanen Zustandes der Wissensbasis.

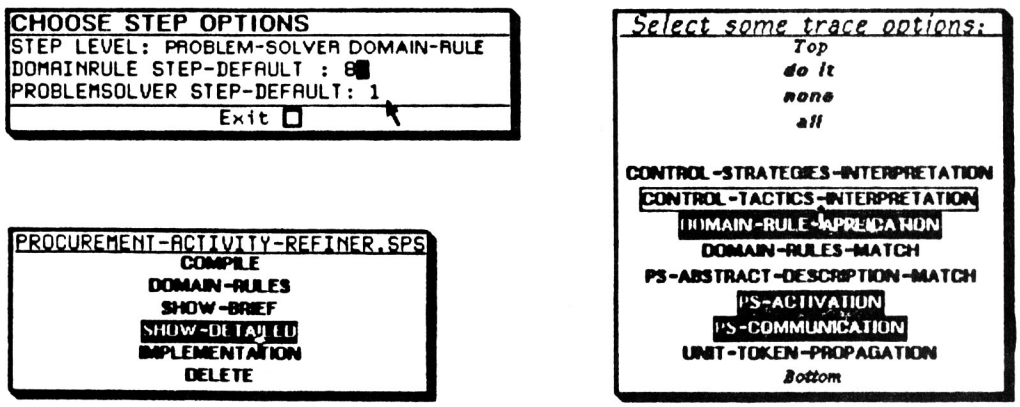


Abb. 4.7: Menüs der ARI Entwicklungsumgebung

Mit dem Inspektionsmechanismus können sowohl die externe Darstellung von *problem-solvern* als auch ihre Implementierung als Wissenseinheit im Objektsystem angezeigt werden.

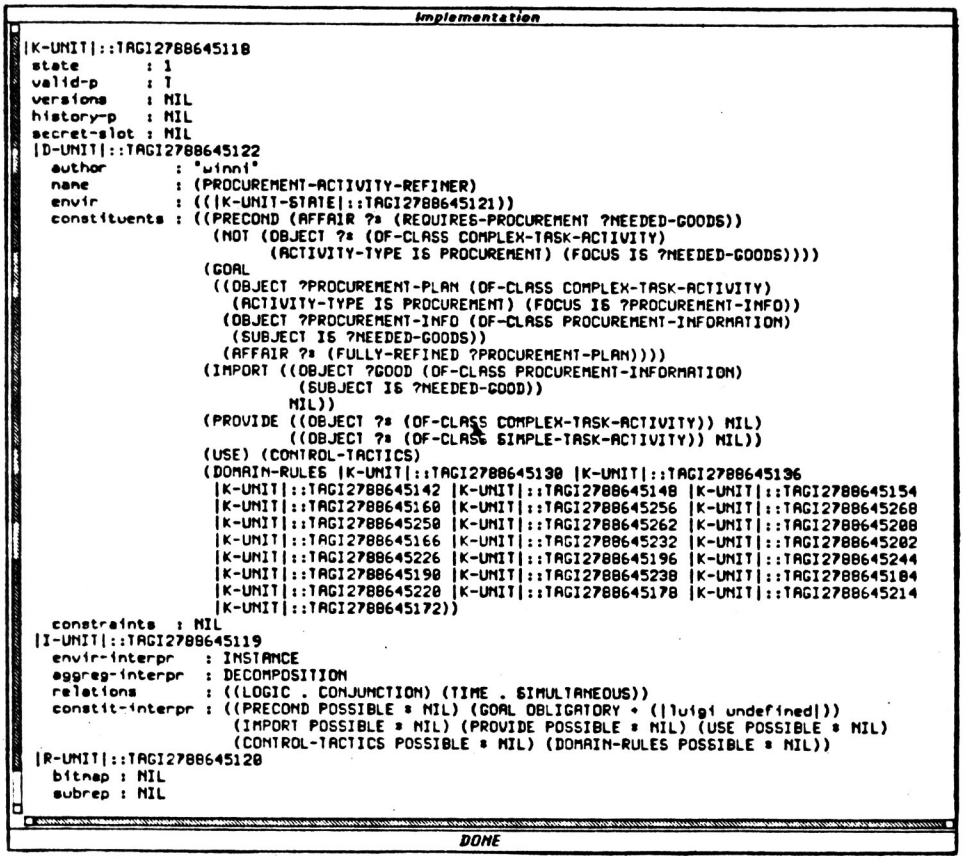


Abb. 4.8: Datenstruktur für einen problem-solver

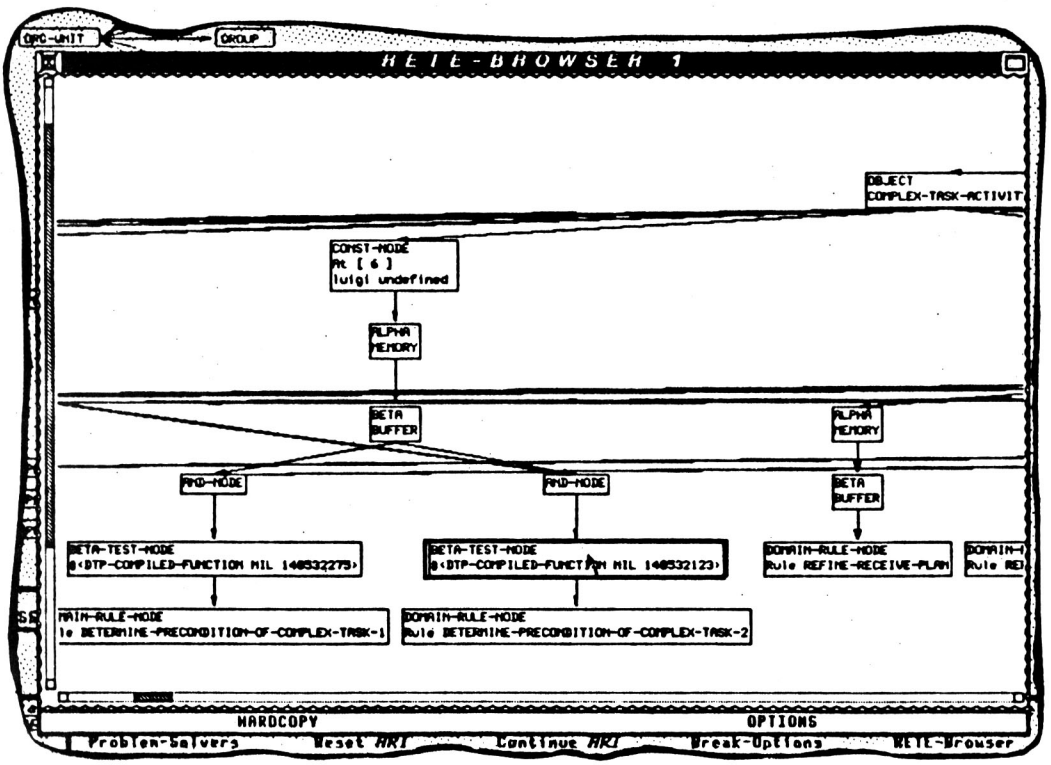


Abb. 4.9: RETE Netzwerk Browser

5. Diskussion und Ausblick

Das Inferenzsystem *ARI* ist ein praktisch einsetzbares Werkzeug zur Konstruktion von Problemlösungskomponenten für wissensbasierte Systeme. *ARI* basiert auf zwei fundamentalen Paradigmen der Wissensrepräsentation. Zum einen benutzt *ARI* Produktionsregeln als grundlegenden Mechanismus zur Formalisierung einer Vielfalt von Schlußfolgerungsprinzipien, zum anderen integriert das System einen Repräsentationsformalismus auf der Basis objektzentrierter, strukturierter Vererbungsnetzwerke.

Die entscheidenden Beiträge des *ARI*-Systems zum Gebiet der Inferenzmechanismen allgemein und der regelbasierten Systeme im speziellen sind die entwickelten epistemischen Primitiva, welche die Strukturierung und modulare Organisation von Regelbasen sowie die einheitliche Spezifikation verschiedener Kontrollmechanismen erlauben. Eine effiziente Realisierung der Interpretation dieser Wissensstrukturen wird durch einen erweiterten RETE-Algorithmus sichergestellt und garantiert die Einsetzbarkeit des Systems für komplexe Problembereiche.

Diese Erweiterungen des Produktionsregelparadigmas erlauben den Einsatz des Systems bei der Konstruktion wissensbasierter Anwendungssysteme im Bereich der Büroautomation, wie sie zur Zeit im Rahmen des WISDOM-Projektes unter Verwendung des *ARI*-Systems entwickelt werden. Anhang B dieser Arbeit charakterisiert den Prototyp eines Planungssystems zur Koordination von Beschaffungsvorgängen.

Mit der Weiterentwicklung des *ARI*-Systems ist bereits begonnen worden, die wichtigsten Erweiterungen betreffen dabei zum einen die Anreicherung der Konstrukte auf epistemologischer Ebene, zum anderen die Implementierung einer mächtigen **Entwicklungsumgebung** mit komfortablen Werkzeugen zu Spezifikation, Test und Wartung von Regelbasen.

Als erweiterte epistemische Primitiva werden in naher Zukunft **nichtmonotone** Inferenzregeln mit expliziter Abhängigkeitsverwaltung, sowie Regeln zur **Rückwärtsverkettung** in das System integriert werden. Zur effizienten Interpretation dieser Konstrukte auf der Implementierungsebene werden ein bereits entwickeltes ***Assumption-Based Truth-Maintenance System*** (**ATMS**, s. /deKleer 86/) und Teile von ***Warren's Abstract Machine*** (**WAM**, s. /Warren 83/) in den RETE-Formalismus eingebracht werden.

Die Bedeutung einer mächtigen Umgebung zur Erstellung, Test und Wartung von Systemen auf der Basis von *ARI* ist dabei als sehr wichtig einzuschätzen, wie die bisherigen Erfahrungen von Entwicklern der oben angeführten Büroanwendungen zeigen.

In einem zeitlich etwas weiteren Rahmen wird für *ARI* eine **Erklärungskomponente** entwickelt werden, die die Methoden expliziter und deklarativer Spezifikation von Strukturierung und Kontrolle ausnutzen soll.

Eine wichtige längerfristige Forschungsaufgabe ist die Identifikation verschiedener Klassen generischer Problemtypen durch Erfahrungen, die bei der Konstruktion von Anwendungen gewonnen werden. Hierzu wird das Planungssystem für Beschaffungsvorgänge unter Verwendung und Bewertung aller Wissensstrukturen auf der epistemologischen Ebene ausgebaut werden. Basierend auf Wissenstrukturen, die sich als typisch für spezielle Klassen generischer Problemtypen erweisen, sollen entsprechende Sprachen auf einer konzeptionellen Ebene für *ARI* definiert werden.

Die entwickelten Anwendungen sollen auch zeigen, inwiefern die Betonung des Produktionsregelansatzes ausreicht, um komplexe Systeme zu konstruieren. *Problem-solver* als funktionale Einheiten könnten in Zukunft ihr Inferenzwissen auch auf der Basis anderer Formalismen repräsentieren, eine Integration von konventionellen prozeduralen Programmen ist auf dieser Ebene ebenfalls vorstellbar.

In diesem Sinne möchte ich hier mit der These schließen, daß der Ansatz des *ARI*-Inferenzsystems seine Tragfähigkeit weit über den derzeitigen Rahmen hinaus zeigen kann. Der Beweis dafür muß allerdings noch erbracht werden und auf dem Weg dahin werden sicherlich noch eine Vielzahl von Problemen zu lösen sein.

6. Literaturverzeichnis

/Aiello,Levi 84/
L. Aiello, G. Levi, "The Uses of Metaknowledge in AI Systems",
Proceedings ECAI 84, Pisa, September 1984, pp. 707-717.

/Aikins 83/
J.S. Aikins, "Prototypical Knowledge for Expert Systems",
Artificial Intelligence **20**(1983), pp. 163-210.

/Allen-83/
E. Allen, "YAPS - A Production Rule System Meets Objects",
Proceedings *AAAI-83*, 1983, pp. 5-7.

/Attardi,Simi 86/
G. Attardi, M. Simi, "A description-oriented logic for building knowledge bases",
Proceedings of the IEEE, Vol. 74, No. 10, October 1986, pp. 1335-1344.

/Bach 88/
B. Bach,
Entwurf und Implementierung eines Algorithmus zur Klassifikation in semantischen Netzen,
Diplomarbeit, FB Informatik, Universität Kaiserslautern, 1988.

/Barber,Hewitt 82/
G. Barber, C. Hewitt, *Foundations for Office Semantics*,
MIT/LCS/TM-225, MIT Laboratory for Computer Science, Cambridge, Ma., 1982.

/Beetz 87/
M. Beetz, *Specifying Meta-Level Architectures for Rule-Based Systems*,
SEKI-REPORT 87-06, Universität Kaiserslautern, 1987.

/Beetz,Barth 88/
M. Beetz, W. Barth, "Towards Structured Production Systems: Efficient Implementation of Meta-Level Architectures Using an Extended RETE Algorithm",
Research Report FB-TA-88-10, TA Triumph Adler AG, Nürnberg, 1988.

/Bowen,Kowalski 82/
K.A. Bowen, R.A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming",
in /Clark,Tarnlund 82/, pp. 153-172.

/Brachman 78/
R.J. Brachman, "On the Epistemological Status of Semantic Networks",
in N.V. Findler (ed.), *Associative Networks: Representation and Use of Knowledge by Computers*,
Academic Press, New York, 1979, pp. 3-50.

/Brachman 83/
R.J. Brachman, "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks",
IEEE Computer, October 1983, pp. 30-36.

/Brachman 85/
R.J. Brachman, "I Lied about the Trees, Or, Defaults and Definitions in Knowledge Representation", AI Magazine, Fall, 1985, pp. 80-93.

/Brachman,Schmolze 85/
R.J. Brachman, J.G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science* **9**, 1985, pp. 171-216.

/Brachman,Smith 80/
R.J. Brachman, B.C. Smith (eds.), Special Issue on Knowledge Representation, *SIGART Newsletter* No. 70, Feb. 1980.

/Brachman,Levesque 82/
R.J. Brachman, H.J. Levesque, "Competence in Knowledge Representation", *Proceedings AAAI 82*, Pittsburgh, Pa., pp.189-192.

/Brachman,Levesque 85/
R.J. Brachman, H.J. Levesque (eds.), *Readings in Knowlege Representation*, Morgan Kaufman Publishers, Inc., 1985

/Brachman,Levesque 86/
R.J. Brachman, H.J. Levesque, "The Knowledge Level of KBMS", in /Brodie,Mylopoulos 86/, pp. 9 12.

/Brachman,Levesque 87/
R.J. Brachman, H.J. Levesque, "Tales from the Far Side of KRYPTON", in /Kerschberg 87/, pp. 3 43.

/Brodie,Mylopoulos 86/
M.L. Brodie, J. Mylopoulos, *On Knowledge Base Management Systems*, Springer Verlag, 1986.

/Brodie,Mylopoulos 86a/
M.L. Brodie, J. Mylopoulos, "Knowledge Bases vs. Databases", in /Brodie,Mylopoulos 86/, pp. 83-86.

/Brownston et al. 85/
L. Brownston, R. Farrell, E. Kant, N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison Wesley, 1985.

/Bundy,Welham 81/
A. Bundy, B. Welham, "Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation", *Artificial Intelligence*, Vol. **16**, No. 2, pp. 189-212.

/Chandrasekaran 84/
B. Chandrasekaran, "Expert Systems: Matching Techniques to Tasks", in *Artificial Intelligence Applications for Business*, W.Reitman (ed.), Ablex, Norwood, New Jersey, 1984, pp. 116-132.

/Chandrasekaran 87/
B. Chandrasekaran, "Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks", *Proccedings IJCAI 87*, Milan, 1987, pp. 1183-1192.

/Chandrasekaran,Mittal 83/
B. Chandrasekaran, S. Mittal, "Conceptual Representation of Medical Knowledge for Diagnosis: MDX and Related Systems", in *Advances in Computers*, Academic Press, New York, 1983, pp. 217-293.

/Chang,Lee-73/
C. Chang, R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

/Clancey 83/
W. Clancey, "The Epistemology of a rule-based expert system: A framework for explanation", *Artificial Intelligence*, Vol. **20** (1983), pp.215-251.

/Clancey 85/
W. Clancey, "Representing Control Knowledge as Abstract Tasks and Metarules", Stanford Knowledge Systems Laboratory, Working Paper No. KSL-85-16, April 1985.

/Clancey,Bock 82/
W. Clancey, C. Bock, "MRS/NEOMYCIN: Representing Metacontrol in Predicate Calculus", Stanford Heuristic Programming Project, Report No. HPP-82-31, 1982.

/Clark 78/
K.L. Clark, "Negation as Failure", in *Logic and Databases*, H. Gallaire, J. Minker (eds.), Plenum, New York, 1978, pp. 293-322

/Clark,Tarnlund 82/
K.L. Clark, S.A. Tarnlund (eds.), *Logic Programming*, Academic Press, 1982.

/Croft,Lefkowitz 84/
W.B. Croft, L.S. Lefkowitz, "Task Support in an Office System", *ACM Transactions on Office Information Systems*, Vol. 2, No. 3, July 1984, pp. 197-212.

/Davis 78/
R. Davis, "Knowledge acquisition in rule-based systems: Knowledge about representations as a basis for system construction and maintenance", in /Waterman,Hayes-Roth 78/, pp.99-134.

/Davis 80/
R. Davis, "Meta-Rules: Reasoning about Control", *Artificial Intelligence*, Vol. **15** (1980), pp.179-222.

/DeKleer 86/
J. deKleer, "An Assumption Based Truth Maintenance System", *Artificial Intelligence* **28**(1986), pp. 127 162.

/Delgrande, Mylopoulos 87/
J.P. Delgrande, J. Mylopoulos, "Knowledge Representation: Features of Knowledge", in *Fundamentals in computer understanding: speech and vision*, J.-P. Haton (ed.), Cambridge University Press, Cambridge, 1987, pp. 23-59

/Dempster 67/
A. Dempster, "Upper and Lower Probabilities Induced by a Multivalued Mapping", *Annals of Mathematical Statistics*, **38** (1967), pp. 325-339.

/Duda et al. 76/
R.O. Duda, P.E. Hart, N.J. Nilsson, "Subjective Bayesian methods for rule-based inference systems", *Proceedings of the 1976 National Computer Conference*, AFIPS Press, 1976, pp. 1075-1082.

/Eirund,Kreplin 88/
H. Eirund, K.Kreplin, "Knowledge Based Document Classification Supporting Integrated Document Handling" *Proceedings Conference on Office Information Systems* (COIS), Palo Alto, 1988.

/Feigenbaum 77/
E. Feigenbaum, "The Art of Artificial Intelligence: Themes and case studies of knowledge engineering", *Proccedings IJCAI 77*, Cambridge, Ma., 1977, pp.1014-1029.

/Feldman 85/
J.A. Feldman (ed.), Special Issue on Connectionist Models and their Applications, *Cognitive Science* 9(1), 1985.

/Fickas-87
S. Fickas, "Supporting the Programmer of a Rule-Based Language", *Expert Systems*, Vol. 4, No. 2, 1987, pp. 74-87.

/Fikes,Kehler-85/
R. Fikes, T. Kehler, "The Role of Frame-based Representation in Reasoning", *Communications of ACM*, Vol. 28, No. 9, 1985, pp. 921-932.

/Forgy-79/
C. Forgy, *On the Efficient Implementation of Production Systems*, Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, 1979.

/Forgy 81/
C.L. Forgy, *OPS5 User Manual*, CMU-CS-81-135, Computer Science Dept., Carnegie Mellon University, Pittsburgh, Pa., 1981.

/Forgy 82/
C.L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Problem", *Artificial Intelligence* 19 (1982), pp. 17-37.

/Fox 86/
M.S. Fox, "Beyond the Knowledge Level", in /Kerschberg 87/, pp. 455-464.

/Gallaire,Lasserre 82/
M. Gallaire, C. Lasserre, "Meta-Level Control for Logic Programs", in /Clark,Tarnlund 82/, pp.173-188.

/Genesereth,Davis 83/
M. Genesereth, D. Smith, "An Overview of Meta-Level Architecture", *Proceedings AAAI 83*, 1983, pp. 119-124.

/Genesereth et al. 80/
M. Genesereth, R. Greiner, D. Smith, *MRS Manual*, Stanford Heuristic Programming Project, Memo HPP-80-24, 1980.

/Gruber,Cohen 87/
T. Gruber, P. Cohen, "Knowledge Engineering Tools at the Architecture Level", *Proceedings IJCAI 87*, Milan, Italy, 1987, pp. 100-103.

/Hayes 77/
P.J. Hayes, "In Defense of Logic", *Proceedings IJCAI 77*, Cambridge, Ma., 1977, pp. 559-565.

/Hayes 79/
P.J. Hayes, "The Logic of Frames", in D. Metzing (ed.), *Frame Conceptions and Text Understanding*, Walter de Gruyter & Co., Berlin, 1979, pp. 46-61.

/Hayes-Roth 84/
F. Hayes-Roth, "The Knowledge-Based Expert System: A Tutorial", IEEE *Computer*, Vol. 17, No. 9, 1984, pp. 11-28.

/Hayes-Roth 85/
F. Hayes-Roth, "Rule-Based Systems", Communications of the ACM, Vol. 28, No. 9, September 1985, pp.921-932.

/Hayes-Roth 86/
F. Hayes-Roth, "Knowledge-Based Expert Systems - The State of the Art in the US",
The Knowledge Engineering Review, 1986.

/Hendrix 79/
G.G. Hendrix, "Encoding Knowledge in Partitioned Networks",
in /Findler 79/, pp. 51-92.

/Hewitt 71/
C. Hewitt, "PLANNER: A Language for Proving Theorems in Robots",
Proceedings IJCAI 71, London, 1971, pp. 167-181.

/Hewitt et al. 80/
C. Hewitt, G. Attardi, M. Simi, "Knowledge embedding in the description system OMEGA",
Proccedings AAAI 80, Stanford, Ca.,1980.

/Israel,Brachman 81/
D.J. Israel, R.J. Brachman, "Distinctions and Confusions: A Catalogue Raisonne",
Proceedings IJCAI 81, Vancouver, B.C., Canada, August 1981, pp. 452-459.

/Jackson 86/
P. Jackson, *Introduction to Expert Systems*,
Addison Wesley Publishing Company, 1986.

/Johnson 84/
T. Johnson, *The Commercial Application of Expert System Technology*, Ovum, London, 1984.

/Kerschberg 87/
L. Kerschberg (ed.), *Proceedings from the First International Conference on Expert Database Systems*,
Benjamin/Cummings Publishing Company, 1987.

/Kowalski 79/
R. Kowalski, *Logic for Problem Solving*,
North Holland, Amsterdam, 1979.

/Kowalski 79a/
R. Kowalski, "Algorithm = Logic + Control",
Communications of the ACM, Vol. 22, No. 7, 1979, pp. 105-118.

/Kreifelts 84/
Th. Kreifelts, "DOMINO: Ein System zur Abwicklung arbeitsteiliger Vorgänge im Büro",
Angewandte Informatik 4(1984), pp. 137-146.

/Levesque 81/
H.J. Levesque, *A formal treatment of incomplete knowledge bases*,
PhD thesis, Dep. Comput. Sci., Univ. Toronto, Ontario, 1981.

/Levesque 86/
H.J. Levesque, "Knowledge Representation and Reasoning",
in *Annual Review in Computer Science*, 1986, pp. 255-287.

/Levesque,Brachman 87/
H.J. Levesque, R. Brachman, "Expressiveness and tractability in knowledge representation and reasoning",
Computational Intelligence, Vol. 3, No. 2, 1987, pp. 78-93.

/Lutze 88/
R. Lutze, "Customizing Cooperative Office Procedures by Planning",
Proceedings Conference on Office Information Systems (COIS), Palo Alto, 1988.

/Maes 87/
P. Maes, *Computational Reflection*,
AI-Laboratory technical report 87_2, Vrije Universiteit Brussel, 1987.

/McCarthy 68/
J. McCarthy, "Programs with Common Sense",
in M. Minsky (ed.), *Semantic Information Processing*, MIT Press, Cambridge, MA., 1968, pp. 403-418.

/McCarthy 87/
J. McCarthy, "Generality in Artificial Intelligence",
Communications of the ACM, Vol 30, No. 12, Dec. 1987, pp. 1030-1035.

/McCarthy,Hayes 69/
J. McCarthy, P.J. Hayes, "Some philosophical problems from the standpoint of Artificial Intelligence",
in *Machine Intelligence* 4, D. Michie (ed), Elsevier North Holland, New York, 1969, pp. 463-502.

/McDermott 81/
J. McDermott, "R1: The Formative Years",
AI Magazine, No. 2 (1981), pp. 21-29.

/McDermott,Doyle 78/
D. McDermott, J. Doyle, "Non-Monotonic Logic I",
Memo MIT/AIM-486, MIT Laboratory for Artificial Intelligence, 1978.

/Mendelson 64/
E. Mendelson, *Introduction to Mathematical Logic*,
Van Nostrand, Princeton, N.J., 1964.

/Minsky-75/
M. Minsky, "A Framework for Representing Knowledge",
in P.H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, pp. 211-277.

/Moore 77/
R. Moore, "Reasoning about Knowledge and Action",
Proceedings IJCAI 77, Cambridge, MA., 1977.

/Moore 82/
R. Moore, "The role of logic in knowledge representation and commonsense reasoning",
Proceedings AAAI 82, Pittsburgh, Pa., 1982, pp. 428-433.

/Neches,Swartout,Moore-85/
R. Neches, W. Swartout, J. Moore, "Explainable (and Maintainable) XPS",
Proceedings IJCAI 85, Los Angeles, 1985, pp. 382-389.

/Newell 80/
A. Newell, "Physical Symbol Systems",
Cognitive Science, Vol. 4, No. 2, 1980, pp. 135-183.

/Newell 82/
A. Newell, "The Knowledge Level",
Artificial Intelligence, Vol. 18 (1982), pp. 87-127.

/Newell,Simon 72/
A. Newell, H.A. Simon, *Human Problem Solving*,
Prentice Hall, Englewood Cliffs, N.J., 1972.

/Puppe 87/
F. Puppe, *Diagnostisches Problemlösen mit Expertensystemen*,
Informatik Fachberichte, Springer Verlag, 1987.

/Reinfrank 86/
M. Reinfrank, "Reason Maintenance Systems",
in H. Stoyan (ed.), *Proc. Workshop on Truth Maintenance Systems*, Berlin, Okt. 1986, Springer Verlag.

/Reiter 80/
R. Reiter, "A Logic for Default Reasoning",
Artificial Intelligence, Vol. 13, 1980, pp. 81-132.

/Sandewall 86/
E. Sandewall, "Non-monotonic inference rules for multiple inheritance with exceptions", Proceedings of the IEEE, vol. 74, 1986, pp. 1345-1353.

/Scales 86/
D.J. Scales, *Efficient Matching Algorithms for the SOAR/OPS5 Production Systems*, Knowledge Systems Lab STAN CS 86 1124, Stanford University.

/Schmolze,Lipkis 83/
J. Schmolze, T. Lipkis, "Classification in the KL ONE Knowledge Representation System", *Proceedings IJCAI 83*, Karlsruhe, 1983, pp. 330 332.

/Schor et al. 86/
M. Schor, T. Daly, H. Lee, B. Tibbits, "Advances in RETE Pattern Matching", Proceedings *AAAI-86*, 1986, pp. 226-231.

/Schubert et al. 79/
L. Schubert, R. Goebel, N. Cercone, "The Structure and Organization of a Semantic Net for Comprehension and Inference", in N.V. Findler (ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, New York, 1979, pp. 121-175.

/Schubert et al. 83/
L.K. Schubert, M.A. Papalaskaris, J. Taugher, "Determining Type, Part, Color, and Time Relationships", *IEEE Computer*, October 1983, pp. 53-60.

/Shafer 76/
G. Shafer, *A Mathematical Theory of Evidence*, Princeton, New Jersey, Princeton University Press, 1976.

/Shapiro 87/
S.C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, Wiley-Interscience Publishers, 1987.

/Shortliffe,Buchanan 75/
E.H. Shortliffe, B.G. Buchanan, "A model of inexact reasoning in medicine", *Mathematical Biosciences* **23** (1975), pp. 351-379

/Silver 86/
B. Silver, *Meta-level Inference*, Studies in Computer Science and Artificial Intelligence, North Holland, 1986.

/Smith 82/
B.C. Smith, "Reflection and Semantics in a procedural language", Techn. Report MIT/LCS/TR-272, Mass. Inst. Technol., Cambridge, 1982.

/Smolensky 87/
P. Smolensky, "Connectionist AI, Symbolic AI, and the Brain", *Artificial Intelligence Review*, Vol. 1, pp. 95-109, 1987.

/Steels 87/
L. Steels, "The deepening of Expert Systems" *AI Communications*, Vol. 0, No. 1, 1987, pp.9 16.

/Sterling 84/
L. Sterling, Implementing Problem Solving Strategies Using the Meta-Level, D.A.I. Research Paper No. 209, Dept. Artificial Intelligence, University of Edinburgh, 1984.

/Tarski 56/
A. Tarski, *Logic, Semantics, Metamathematics*, Oxford University Press, 1956.

/Touretzky et al. 87/
D.S. Touretzky, J.F. Horty, R.H. Thomason,
"A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems",
Proceedings IJCAI 87, Milan, 1987, pp. 476-482.

/vanEmden,Kowalski 76/
M.H. van Emden, R.A. Kowalski, "The Semantics of Predicate Logic as Programming Language",
Journal of the ACM, Vol. 23, No. 4, 1976, pp. 733-742.

/vanHarmelen 86/
F. van Harmelen, *Improving the Efficiency of Meta-Level Reasoning*, DAI Discussion Paper No. 40;
Dept. of Artificial Intelligence, University of Edinburgh, December 1986.

/Warren 83/
D. Warren, *An Abstract PROLOG Instruction Set*,
Technical Note 309, SRI International, 1983.

/Waterman,Hayes-Roth 78/
D.A. Waterman, F. Hayes-Roth, *Pattern-Directed Inference Systems*,
Academic Press, New York, 1978.

/Woods 75/
W. Woods, "What's in a link: Foundations for Semantic Networks",
in *Representation and Understanding: Studies in Cognitive Science*, D.G. Bobrow and A.M. Collins (eds.),
Academic Press, New York, pp. 35-82.

/Zadeh 78/
L.A. Zadeh, "Fuzzy Sets as a basis for a theory of possibility",
Fuzzy Sets Sys. **1** (1978), pp. 3-28.

Anhang A: Syntax der ARI Repräsentationssprache

Zur Beschreibung der Syntax der Repräsentationssprache des *ARI*-Inferenzsystems wird eine Backus-Naur Notation verwendet. Die benutzten Konstrukte werden wiefolgt unterschieden:

(1)	***SCHLÜSSELWORTE***	: fett und kursiv gedruckt.
(2)	**"Terminalsymbole"**	: fett gedruckt in Anführungszeichen.
(3)	<Nichtterminalsymbole>	: in spitzen Klammern.
(4)	[...]	: optionales Konstrukt.
(5)	(...)	: Gruppierung
(6)	|	: Alternativen (infix)
(7)	+	: ein- oder mehrmaliges Auftreten (postfix).
(8)	*	: kein, ein- oder mehrmaliges Auftreten (postfix).
(9)	^	: Konkatenation (infix, z.B. **"?"^"abc"** --> **"?abc"**).

Kommentare werden mit einem Semikolon (";") eingeleitet

I. Die ARI Mustersprache (pattern language)

```
<domain-pattern>
     ::= <affair-pattern> | <object-pattern>


<affair-pattern>
     ::= AFFAIR  <variable>
          <affair-name> "(" (<variable> | <constant>)* ")"


<object-pattern>
     ::= OBJECT  <variable>
        <class-spec>
        [ WITH <constituent>+ ]

<class-spec>
     ::= (OF-CLASS <class-name>) |
         (OF-SUBCLASS-OF <class-name>)

<constituent>
     ::= <constituent-name>
         <constituent-test-predicate>
         (<variable> | <constant>)

<constituent-test-predicate>
     ::= "=" | <two-place-predicate-name>


<test-pattern>
     ::= TEST  "(" <test-predicate-name>
                   (<variable> | <constant>)* ")"
```

<rule-pattern>
::= <named-rule-pattern> | <general-rule-pattern>

<named-rule-pattern>
::= ***RULE*** <name>

<general-rule-pattern>
::= ***RULE*** <variable> ***WITH***
[***CONDITIONS*** <domain-situation>]
[***ACTIONS*** <primitive-action>+]

<problem-solver-pattern> ::=
::= <named-problem-solver-pattern> |
<general-problem-solver-pattern>

<named-problem-solver-pattern>
::= ***PROBLEM-SOLVER*** <name>

<general-problem-solver-pattern>
::= ***PROBLEM-SOLVER*** <variable> ***WITH***
[***PRECONDITION*** <domain-situation>]
[***GOAL*** <domain-situation>]

<rule-conflict-pattern>
::= <domain-pattern> | <rule-pattern>

<problem-solver-conflict-pattern>
::= <domain-pattern> | <problem-solver-pattern>

II. Situationsbeschreibungen:

```
<domain-situation>
     ::= ( ([ NOT ] <domain-pattern>) |
           <test-pattern> )+


<constrained-domain-situation>
     ::=  <domain-pattern>
        [ <domain-situation> ]


<disjunctive-constrained-domain-situations>
     ::= <constrained-domain-situation>
         (OR <constrained-domain-situation>)*


<rule-conflict-situation>
     ::= <rule-conflict-pattern>
         ( ([ NOT ] <rule-conflict-pattern>) |
           <test-pattern> )+


<problem-solver-conflict-situation>
     ::= <problem-solver-conflict-pattern>
         ( ([ NOT ] <problem-solver-conflict-pattern>) |
           <test-pattern> )+


<domain-pattern-in-situation>
     ::= <domain-pattern>
        [ WITH-CONDITIONS <domain-situation> END ]
```

III. Elementare Sprachkonstrukte:

<variable> ::= "?"^<name>
; ?x, ?y ?object-1 sind z.B. zulässige Variablen.

<constant> ::= <LISP-symbol> | <LISP-number> |
<LISP-string> | <list>
; Konstanten sind Werte der angegebenen LISP-Typen.

<list> ::= <LISP-list-of-<constant>>

<string> ::= <LISP-string>

<name> ::= <LISP-symbol>
<affair-name> ::= <LISP-symbol>
<class-name> ::= <LISP-symbol>
<object-instance-name> ::= <LISP-symbol>
<slot-name> ::= <LISP-symbol>
<two-place-predicate-name> ::= <LISP-symbol>
<test-predicate-name> ::= <LISP-symbol>
<function-name> ::= <LISP-symbol>

IV. Problem-Solver:

```
<problem-solver>
     ::= <simple-problem-solver> |
         <complex-problem-solver>


<simple-problem-solver>
     ::= PROBLEM-SOLVER  <name>
          <problem-description>
          <communication-description>
          <control-tactics>
          <domain-rules>
         END-PROBLEM-SOLVER


<complex-problem-solver> ::=
     PROBLEM-SOLVER  <name>
      <problem-description>
      <communication-description>
      <control-strategies>
      <problem-solvers>
     END-PROBLEM-SOLVER


<problem-description>
     ::= SOLVES-PROBLEMS-WITH
          <precondition-spec>
          <goal-spec>

<precondition-spec>
     ::= PRECONDITION
          <constrained-domain-situation>

<goal-spec>
     ::= GOAL
          <disjunctive-constrained-domain-situations>
```

```
<communication description>
     ::= COMMUNICATION
          <import-spec>
          <export-spec>

<import-spec>
     ::= IMPORT  <domain-pattern-in-situation>+

<export-spec>
     ::= EXPORT  <domain-pattern-in-situation>+
```

V. Konstrukte zur Spezifikation von Kontrollwissen:

```
<control-strategies>
     ::= <control-strategy>*

<control-strategy>
     ::= CONTROL-STRATEGY  <name>
          <problem-solver-conflict-situation>
          "-->"
          <control-action> <variable>


<control-tactics>
     ::= <control-tactic>*

<control-tactic>
     ::= CONTROL-TACTIC  <name>
          <rule-conflict-situation>
          "-->"
          <control-action> <variable>


<control-action>
     ::= SUSPEND | PREFER
```

Phasenfolgen (derzeit in ARI nicht realisiert)

```
<phase-sequence-spec> ::= PHASE-SEQUENCE  <name>
                           <phase-sequence-element>+
                          END-PHASE-SEQUENCE

<phase-sequence-element> ::= <name>
                             <IF-sequence-element> |
                             <LOOP-sequence-element>

<IF-sequence-element> ::= IF  <pattern>  THEN
                           <name>
                          [ ELSE  <name> ]
                          END-IF

<LOOP-sequence-element> ::=  LOOP
                               <name>*
                               EXIT WHEN  <pattern>
                               <name>*
                             END-LOOP
```

VI. Objektregeln:

```
<domain-rules>
     ::= <domain-rule>+

<domain-rule>
     ::= DOMAIN-RULE  <name>
          <constrained-domain-situation>
         "-->"
          <domain-action>+


<domain-action>
     ::= <primitive action> |
         <compound-action>  |
         <user-interaction>


<compound-action>
     ::= <let-action> | <let*-action>

<let-action>
     ::= LET (<variable> "=" <function-call>)+
          <primitive action>*
         END-LET

<let*-action>
     ::= LET* (<variable> "=" <function-call>)+
          <primitive action>*
         END-LET*

<function-call>
     ::= "(" <function-name> (<variable> | <constant>)* ")"
```

```
<primitive action>
     ::= ASSERT  <affair-name>
                 "(" (<variable> | <constant>)* ")"
         RETRACT  <variable> |
         CREATE   <object-instance-name>
          OF-CLASS  <class-name>
            <object-instance-slot-value>* |
         DELETE   <variable> |
         MODIFY   <variable>
            <object-instance-slot-value>+

<object-instance-slot-value>
     ::= <slot-name> ":" (<variable> | <constant>)


<interactive-action>
     ::= <yes-or-no-query>          |
         <select-query>             |
         <query-for-variable-value> |
         <display>


<yes-or-no-query>
     ::= QUERY <question>
          IF-TRUE  <primitive-action>+
          IF-FALSE <primitive-action>+


<select-query>
     ::= QUERY  <question>
          SELECT  <variable>  FROM  <list>
           DO  <primitive-action>+  END-DO
```

<query-for-variable-value>
    ::= *QUERY-VALUE* <variable> <question>
        DO <primitive-action>+ *END-DO*

<question>
    ::= <string>

<display>
    ::= *DISPLAY* <list>

Anhang B: ARI-OFFICEPLAN - Ein Anwendungsbeispiel

ARI-OFFICEPLAN ist ein Anwendungsbeispiel des *ARI* Inferenzsystems in einem System zur Unterstützung kooperativer Bürovorgänge. Problemgeenstand sind Bürovorgängen zur Beschaffung von Gütern. Die Aufgabe des Systems ist das Planen bzw. Anpassen dieser Vorgänge an individuelle Gegebenheiten. ARI-OFFICEPLAN ist mit der LUIGI Repräsentationssprache für beschreibendes Wissen und der *ARI* Sprache für Problemlösungswissen implementiert.

Die Arbeitsweise des Systems stellt sich von einem abstrakten Standpunkt folgendermaßen dar:

- Das System startet mit einer Problemdefinition, die feststellt, daß ein bestimmtes Gut von einem Sachbearbeiter benötigt wird.
- ARI-OFFICEPLAN bestimmt dann, ob zur Beschaffung des Gutes ein Beschaffungsvorgang initiiert werden muß.
- Ist dies der Fall, so konstruiert das System einen geeigneten Plan für diesen Beschaffungsvorgang, der den Firmenrichtlinien entspricht.
- Das System bestimmt außerdem notwendige kooperative Aktivitäten wie Genehmigungen, Konsultationen, usw. und fügt sie in den Plan ein. Die Personen, die diese kooperativen Aktivitäten ausführen sollen, werden aus einer beschreibenden Wissensbasis über die Organisationsstruktur und die Funktionen von Organisationseinheiten bestimmt.

ARI-OFFICEPLAN ist als Planungssystem organisiert, das aus vier Problemlösungskomponenten mit unterschiedlichen Problemlösungsaktivitäten besteht. Die erste Komponente klassifiziert das angeforderte Produkt und bestimmt auf der Basis dieser Klassifikation, ob zum Erhalt des Produkts ein Beschaffungsvorgang erforderlich ist. Die zweite Komponente ist das Planexpansionsmodul, das die Synthese des Beschaffungsvorgangs durchführt. Dieses konstruierte Planskelett wird durch die dritte Komponente analysiert, um herauszufinden, welche Teilaktivitäten im Planskelett Risiken nach sich ziehen (z.B. finanzielle oder terminbezogene Risiken). Firmenrichtlinien werden von der vierten Komponente auf die Risiken und sie verursachenden Teilaktivitäten angewendet, um Genehmigungen und Konsultationen in das Planskelett einzufügen, die die Risiken bezüglich der Firmenrichtlinien abdecken.

Die Realisierung des Systems in ARI ist ein hierarchisch strukturiertes Produktionsregelsystem, das über einer taxonomisch organisierten Wissensbasis für beschreibendes Wissen inferiert. Die Wissensbasis enthält sowohl strukturierte Objekte als auch Sachverhalte, also Relationen über diesen Objekten.

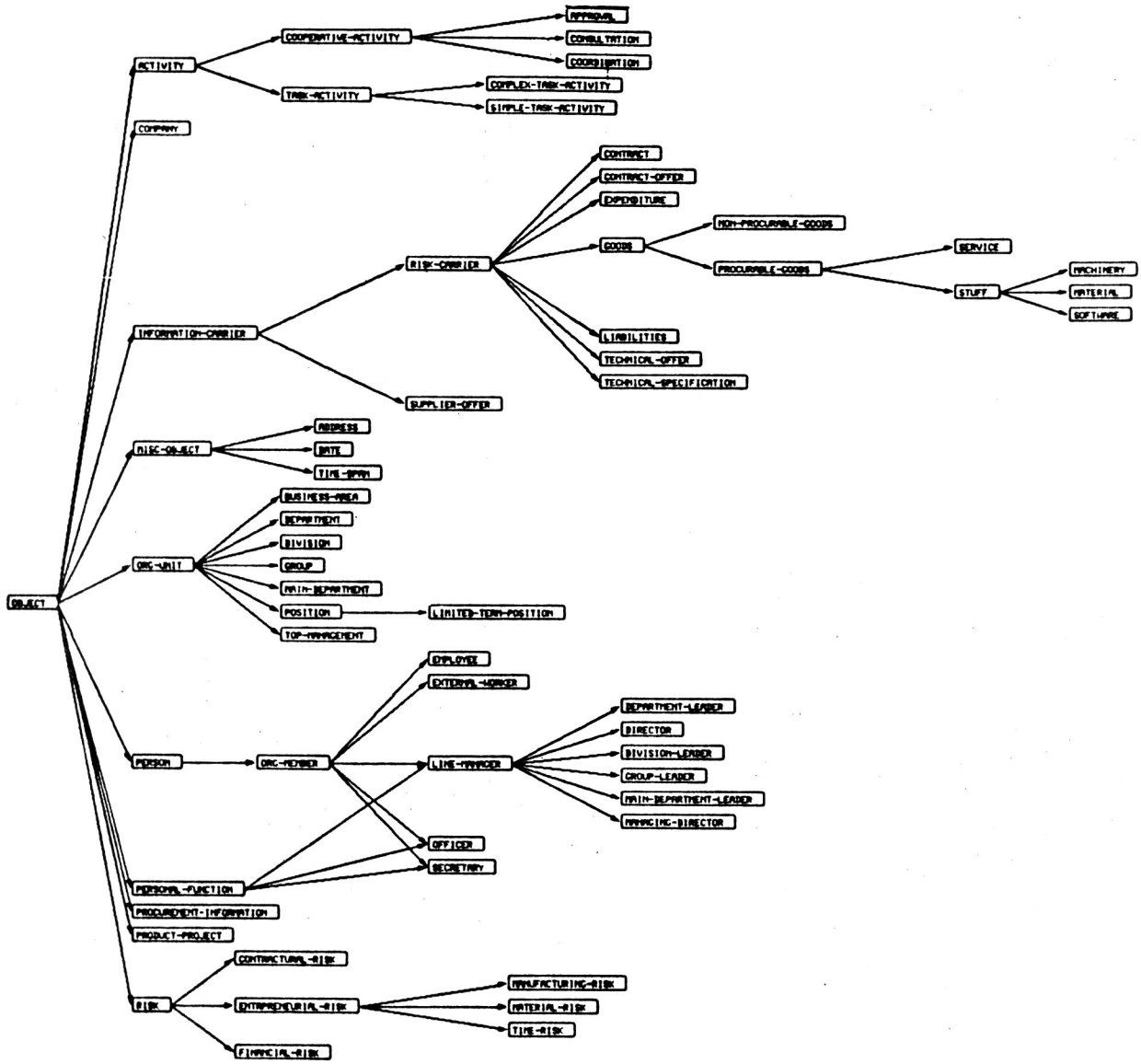


Abb. B.1: Die Objekttaxonomie für ARI-OFFICEPLAN

Die Systemarchitektur realisiert das Prinzip verteilter wissensbasierter Planungssysteme (distributed knowledge-based planners) und ist in ***ARI*** explizit durch *problem-solver* modelliert, die als funktionale Einheiten Teilprobleme lösen (siehe Abb. B.2). Die *problem-solver* werden in verschiedenen Phasen aktiviert, die vom System sequentiell durchlaufen werden.

Für jeden *problem-solver* ist das Teilproblem, das er löst, explizit als abstrakte Beschreibung repräsentiert und die Wissenseinheiten, die er importiert bzw. exportiert sind als Import- bzw. Exportwissen spezifiziert. Die Abb. B.3 zeigt die Spezifikation des *problem-solvers* PROCUREMENT-ACTIVITY-REFINER. Dieser *problem-solver* löst die folgende Klasse von Problemen:

> *Falls ein Gut benötigt wird, das einen Beschaffungsvorgang erforderlich macht, der noch nicht bekannt ist (Vorbedingung), dann konstruiere einen vollständig verfeinerten Beschaffungsvorgang (Zielbeschreibung).*

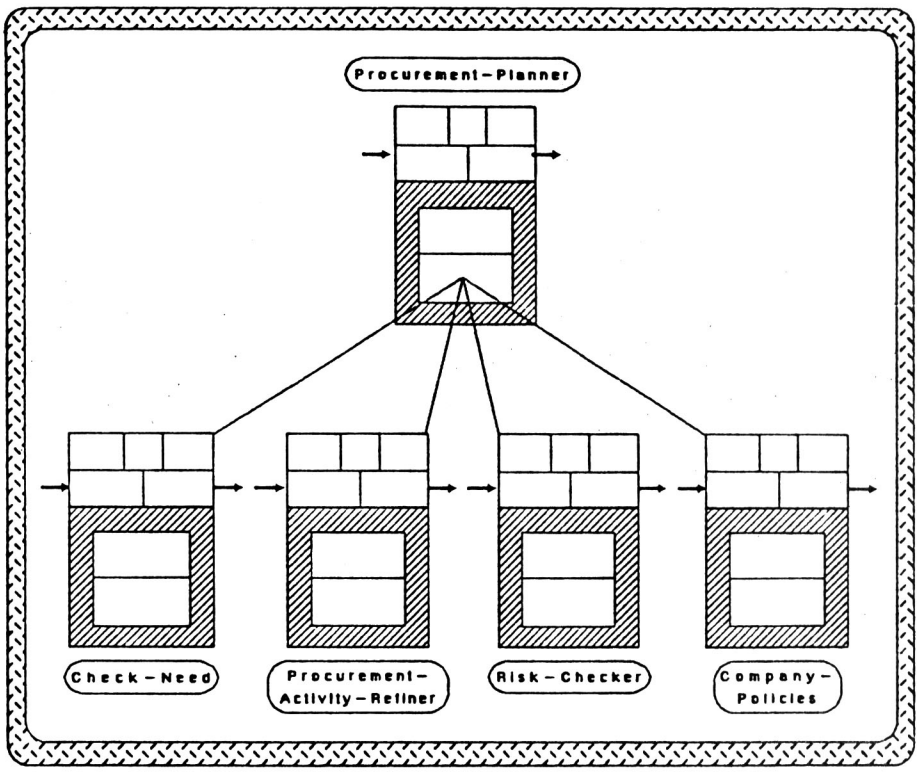


Abb. B.2: Struktur der Regelbasis für ARI-OFFICEPLAN

Die Vorbedingung dient gleichzeitig als Aktivierungs-, die Zielbeschreibung als Deaktivierungsbedingung. PROCUREMENT-ACTIVITY-REFINER importiert die Informationen über das zu beschaffende Gut und exportiert bei seiner Deaktivierung den vollständig verfeinerten Beschaffungsvorgang.

PROBLEM-SOLVER *PROCUREMENT-ACTIVITY-REFINER*

```
SOLVES-PROBLEMS-WITH
    PRECONDITION
        AFFAIR ?* (REQUIRES-PROCUREMENT ?NEEDED-GOODS)
        NOT OBJECT ?*
                of-class COMPLEX-TASK-ACTIVITY
                with ACTIVITY-TYPE is PROCUREMENT
                     FOCUS is ?NEEDED-GOODS
    GOALS
        OBJECT ?PROCUREMENT-PLAN
           of-class COMPLEX-TASK-ACTIVITY
           with ACTIVITY-TYPE is PROCUREMENT
                FOCUS is ?PROCUREMENT-INFO
        OBJECT ?PROCUREMENT-INFO
           of-class PROCUREMENT-INFORMATION
           with SUBJECT is ?NEEDED-GOODS
        AFFAIR ?* (FULLY-REFINED ?PROCUREMENT-PLAN)
COMMUNICATION
    IMPORT
        OBJECT ?GOOD
           of-class PROCUREMENT-INFORMATION
           with SUBJECT is ?NEEDED-GOOD
    EXPORT
        OBJECT ?*
           of-class COMPLEX-TASK-ACTIVITY
        OBJECT ?*
           of-class SIMPLE-TASK-ACTIVITY
```

Abb. B.3: Abstrakte Beschreibung

Die *problem-solver* lösen verschiedene generische Problemklassen:

- CHECK-NEED löst das Problem der **Klassifikation** von Produkten,
- ACTIVITY-REFINER führt **Plansynthese** (Synthese von Bürovorgängen) durch,
- RISK-CHECKER **diagnostiziert Risiken, und**
- COMPANY-POLICIES führt **intelligent Retrieval aus.**

Das Lösen von verschiedenen generischen Problemklassen spiegelt sich in ARI-OFFICEPLAN im Vorhandensein typischer Inferenzregeln wieder. Ein typischer Planexpansionsoperator repräsentiert als Inferenzregel ist REFINE-FIND-SUPPLIERS. Die Regel *matcht*, falls eine komplexe Aktivität noch nicht verfeinert ist, erzeugt die Teilaktivitäten und trägt sie in die komplexe Aktivität ein.

```
DOMAIN-RULE REFINE-FIND-SUPPLIERS
   OBJECT ?FIND-SUPPLIERS-PLAN
      of-class COMPLEX-TASK-ACTIVITY
      with ACTIVITY-TYPE is FIND-SUPPLIERS
           AGENT is ?AGENT
           SUBACTIVITIES is *UNDEFINED*
           FOCUS is ?PROCUREMENT-INFO
   ==>
   LET ?SUBACTIVITIES =
            (LIST (CREATE CONFIGURATE-OFFER
                          SIMPLE-TASK-ACTIVITY
                          (ACTIVITY-TYPE CONFIGURATE-OFFER)
                          (FOCUS ?PROCUREMENT-INFO)
                          (AGENT ?AGENT))
                  (CREATE GET-TENDERS
                          SIMPLE-TASK-ACTIVITY
                          (ACTIVITY-TYPE GET-TENDERS)
                          (FOCUS ?PROCUREMENT-INFO)
                          (AGENT ?AGENT)))

     MODIFY-OBJECT ?FIND-SUPPLIERS-PLAN
        SUBACTIVITIES : ?SUBACTIVITIES
   END-LET
```

Abb. B.4: Eine Regel zum Expandieren von Plänen

```
DOMAIN-RULE FIND-CONTRACTURAL-RISK-1
   OBJECT ?ACTIVITY
      of-class SIMPLE-TASK-ACTIVITY
      with POSTCONDITION is ?POSTCONDITION
           FOCUS is ?SUBJECT
   TEST (AND (NOT (EQ ?POSTCONDITION *UNDEFINED*))
             (INSTANCE-OF-SUBCLASS-P ?POSTCONDITION 'CONTRACT))
   ==>
   LET* ?NAME =
             (ARI-MAKE-NAME "contractural-risk-wrt-~a" (ARI-CONSTIT-NAMED ?SUBJECT 'SUBJECT))
        ?RISK =
             (CREATE ?NAME
                     CONTRACTURAL-RISK
                     (SUBJECT ?SUBJECT)
                     (CAUSED-BY ?ACTIVITY)
                     (AMOUNT 2))

     ASSERT-AFFAIR (CAUSES ?ACTIVITY ?RISK)
     DISPLAY (Activity ~'pc~s~ɔ~% causes a ~'bccontractural-risk~ɔ~% ~
                                          because it produces an ~'bcorder~ɔ !
              (ARI-FIRST-UNIT-NAME ?ACTIVITY))
   END-LET*
```

Abb. B.5: Eine Inferenzregel zum Erkennen von Risiken

FIND-CONTRACTURAL-RISK-1 ist eine typische Inferenzregel zum Erkennen von Risiken. Bestimmte Aktivitäten, die Risiken bedingen, werden erkannt, ein strukturiertes Objekt, welches das

Risiko beschreibt, wird erzeugt und durch den Sachverhalt mit Namen causes mit der Teilaktivität verknüpft, die das Risiko bedingt hat.

Die Inferenzregel POLICY-TO-COMPENSATE-CONTRACTURAL-RISK fügt eine Koordination in den Plan für den Beschaffungsvorgang ein, falls eine Teilaktivität ein Vertragsrisiko bedingt.

DOMAIN-RULE *POLICY-TO-COMPENSATE-CONTRACTURAL-RISK*

```
OBJECT ?CONTRACTURAL-RISK
   of-class CONTRACTURAL-RISK
   with SUBJECT is ?SUBJECT
        CAUSED-BY is ?RISK-CARRIER
        COMPENSATED-BY is *UNDEFINED*
AFFAIR ?* (CAUSES ?ACTIVITY ?CONTRACTURAL-RISK)
OBJECT ?ACTIVITY
   of-class SIMPLE-TASK-ACTIVITY
==>
LET* ?COORD-NAME =
          (ARI-MAKE-NAME "coordination-of-procurement-of-~a"
                         (ARI-CONSTIT-NAMED ?SUBJECT 'SUBJECT))
     ?COORDINATION =
          (CREATE ?COORD-NAME
                  COORDINATION
                  (ACTIVITY-TYPE COORDINATION)
                  (PRECONDITION ?RISK-CARRIER)
                  (GOAL ?SUBJECT)
                  (MOTIVE ?CONTRACTURAL-RISK)
                  (SUBJECT ?SUBJECT))

  ASSERT-AFFAIR (COMPENSATES ?COORDINATION ?CONTRACTURAL-RISK)
  MODIFY-OBJECT ?CONTRACTURAL-RISK
     COMPENSATED-BY : ?COORDINATION
  ASSERT-AFFAIR (BEFORE ?COORDINATION ?ACTIVITY)
END-LET*
```

Abb. B.6: Eine Kompensationsregel für Vertragsrisiken

FIND-RESPONSIBLE-PERSON ist eine Regel, die intelligentes Retrieval ausführt. Sie ist anwendbar, falls die Agenten einer kooperativen Aktivität noch nicht bestimmt sind. Agenten für kooperative Aktivitäten werden mit Hilfe der Funktion **ari-find-reponsible-for-task** aus der Wissensbasis für beschreibendes Wissen bestimmt.

DOMAIN-RULE *FIND-RESPONSIBLE-PERSON*

```
OBJECT ?COORDINATION
   of-class COORDINATION
   with AGENT is *UNDEFINED*
AFFAIR ?* (COMPENSATES ?COORDINATION ?CONTRACTURAL-RISK)
OBJECT ?CONTRACTURAL-RISK
   of-class CONTRACTURAL-RISK
   with SUBJECT is ?PROCUREMENT-INFO
OBJECT ?PROCUREMENT-INFO
   of-class PROCUREMENT-INFORMATION
   with SUBJECT INSTANCE-OF-SUBCLASS-P WORK-STATION
==>
LET* ?AGENT =
          (ARI-FIND-RESPONSIBLE-FOR-TASK COMPUTER LEGAL-ADVISORY)

  MODIFY-OBJECT ?COORDINATION
     AGENT : ?AGENT
END-LET*
```

Abb. B.7: Auffinden einer verantwortlichen Person durch Retrieval