UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

REPRESENTAÇÃO DE CONHECIMENTO: PROGRAMAÇÃO EM LÓGICA E O MODELO DAS HIPERREDES

40

por

Luiz Antonio Moro Palazzo

Dissertação submetida como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof. José Mauro Volkmer de Castilho
Orientador

Prof. Antônio Carlos da Rocha Costa
Co-orientador

SABi

UFRGS

05225470

Porto Alegre, julho de 1991

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Palazzo, Luiz Antonio Moro

Representação de Conhecimento: Programação em Lógica e o Modelo das Hiperredes / Luiz Antonio Moro Palazzo.- Porto Alegre: Curso de Pós-Graduação em Ciência da Computação da UFRGS, 1991.

291p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1991. Orientador: Castilho, José Mauro Volkmer de. Co-orientador: Rocha Costa, Antonio Carlos da.

Dissertação: Inteligência Artificial: Representação de Conhecimento: Programação em Lógica: Problema do Controle: Bases de Conhecimento: Modelagem Semântica: Hiperredes.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

5935

PALAZZO, LUIZ ANTONIO MORO

REPRESENTACAO DE CONHECIMENTO

681.3.011(043)
P155R

INF
1993/60365-0
1993/08/02

MOD. 2.3.2

AGRADECIMENTOS

A concepção e posterior desenvolvimento do presente trabalho deve-se em grande parte ao aconselhamento, estímulo e orientação dos professores José Mauro Volkmer de Castilho e Antônio Carlos da Rocha Costa, respectivamente orientador e co-orientador da presente dissertação, aos quais sou especialmente grato. Registro também, neste particular, meu agradecimento à professora Rosa Maria Viccari pela leitura e sugestões apresentadas, que foram de grande valia. Entre os diversos colegas do curso de mestrado, agradeço especialmente a Adagenor Lobato Ribeiro, Graçaliz Dimuro e Paulo Barella, cuja amizade, compreensão e incentivo foram muito importantes ao longo dos últimos três anos. Sou também agradecido aos colegas do CPD/UFPel, na pessoa de Bayard Silva Centeno, seu diretor e Adenauer Corrêa Yamin pelo apoio e estímulo que me ofereceram. Também aos colegas do DCET V da UCPel, na pessoa de Aroldo Roberto dos Santos Peduzzi e Francisco de Paula Marques Rodrigues, registro sinceros agradecimentos. Finalmente, pelo carinho, compreensão e encorajamento, bem como pelo tempo roubado ao convívio familiar, reservo a mais profunda gratidão à minha esposa e filhas, que pacientemente esperaram a conclusão deste trabalho.

Para

Mara,
Daniela e
Renata.

SUMÁRIO

LISTA DE ABREVIATURAS 9
LISTA DE SINAIS 10
LISTA DE FIGURAS 11
RESUMO 13
ABSTRACT 15

1 INTRODUÇÃO 17

2 CONHECIMENTO E REPRESENTAÇÃO 21

2.1 Conhecimento 21
2.2 Representação de Conhecimento 25
2.3 O Problema da Representação de Conhecimento 30
2.3.1 Considerações Epistemológicas 30
2.3.2 Expressividade 34
2.3.3 Potencial Heurístico 37
2.3.4 Conveniência Notacional 41
2.4 Esquemas de Representação de Conhecimento 42
2.4.1 Sistemas de Produções 42
2.4.2 Redes Semânticas 47
2.4.3 Sistemas de Frames 49
2.4.4 Linguagens Lógicas 53
2.4.5 Krypton 58

3 PROGRAMAÇÃO EM LÓGICA 63

3.1 Programação em Lógica de Primeira Ordem 63
3.1.1 Programas em Lógica 66
3.2 Semântica Modelo-Teorética 75
3.2.1 Modelos de Programas em Lógica 75
3.2.2 Substituições-Resposta 84
3.3 Semântica Prova-Teorética 90

4 EXTENSÕES DA LÓGICA DE PRIMEIRA ORDEM 94

4.1 Lógica Modal 95
4.2 Lógica em Nível Meta 99
4.3 Reflexão Computacional e Metaconhecimento 105
4.4 metaProlog 111
4.4.1 A Natureza do metaProlog 112
4.4.2 Sintaxe do metaProlog 116
4.4.3 Teorias e Nomes 118
4.5 Programação em Lógica Contextual 123
4.5.1 Teoria Básica 123
4.5.2 Derivação Top-Down 125
4.5.3 Derivação Bottom-Up 127
4.5.4 Semântica Declarativa 128
4.5.5 Semântica de Ponto Fixo 132

5 PROGRAMAÇÃO EM LÓGICA E BASES DE DADOS 134

5.1 Bases de Conhecimento e Bases de Dados 134
5.2 A Lógica como ER 137
5.3 Lógica e Bases de Dados 139
5.4 A Linguagem Prolog e Sistemas de Bases de Dados 142

6 REPRESENTAÇÃO DE CONHECIMENTO EM HIPERREDES 154

6.1 O Modelo das Hiperredes 154
6.2 A Linguagem HYPER 159
6.3 Operações Básicas 163
6.4 Comandos 169
6.5 Padrões de Ligação em Estratégias de Controle 173

7 REPRESENTAÇÃO DE HIPERREDES EM LÓGICA 178

7.1 Representando Hiperredes em Prolog 178
7.2 Raciocínio Sobre Hiperredes 187
7.2.1 Seleção de Objetivos 189
7.2.2 Geração de Aspectos 191

7.2.3 Contextualização de Aspectos 194
7.2.4 Derivando Objetivos em Sistemas de Contextos 199
7.3 Operações Sobre Hiperredes 202
7.3.1 Instanciação 202
7.3.2 Composição 203
7.3.3 Decomposição 204
7.3.4 Abstração 204
7.3.5 Modificação 205
7.3.6 Deleção 205
7.3.7 Renomeação 206
7.4 Visões em Hiperredes 207
7.4.1 Preliminares 207
7.4.2 Representação de Visões em Hiperredes 209

8 SISTEMAS GERENCIADORES DE BASES DE CONHECIMENTO 212

8.1 Arquitetura 212
8.2 O Nível da Aplicação 216
8.3 O Nível da Engenharia de Conhecimento 218
8.3.1 Aspectos Descritivos 219
8.3.2 Aspectos Operacionais 221
8.3.3 Aspectos Organizacionais 224
8.3.3.1 Classificação 225
8.3.3.2 Generalização 226
8.3.3.3 Herança 227
8.3.3.4 Associação de Elementos 228
8.3.3.5 Associação de Conjuntos 229
8.3.3.6 Agregação 231
8.3.3.7 Integração de Abstrações 232
8.4 O Nível de Implementação 237
8.4.1 Características de Acesso 237
8.4.2 Tipos de Acessos 238
8.4.3 Frequências de Acesso 238
8.4.4 Processamento de Conhecimento 239
8.4.5 Manutenção Temporária do Conhecimento Dinâmico 240
8.4.6 Ambientes Multiusuários 240
8.4.7 Estruturas de Conhecimento 241

9 RHESUS: UM SISTEMA EXPERIMENTAL 242

9.1 A Arquitetura do Sistema Rhesus 243
9.2 A Base de Conhecimento 245
9.3 O Sistema de Processamento de Conhecimento 251
9.3.1 O Sistema Resolutivo 252
9.3.1.1 Raciocínio Estrutural 253
9.3.1.2 Raciocínio Dedutivo 254
9.3.1.3 Raciocínio Circunstancial 254
9.3.2 O Sistema Cognitivo 255
9.3.2.1 Raciocínio por Classificação/Generalização 256
9.3.2.2 Raciocínio por Associação 258
9.3.2.3 Raciocínio por Agregação 261
9.4 O Sistema de Comunicação 263
9.5 O Sistema Explanativo 266
9.6 Diretrizes de Implementação 267
9.6.1 A Modelagem do Conhecimento 268
9.6.2 O Sistema de Processamento de Conhecimento 270
9.6.2.1 O Sistema Cognitivo 271
9.6.2.2 O Sistema Resolutivo 273
9.6.3 O Sistema de Comunicação 275
9.6.3.1 O Ambiente 275
9.6.3.2 O Processador de Consultas 277
9.6.3.3 O Sistema de Apresentação 278
9.6.4 O Sistema Explanativo 279

10 CONCLUSÕES 280

BIBLIOGRAFIA 285

LISTA DE ABREVIATURAS

Asp	Aspecto
BC	Base de Conhecimento
BD	Base de Dados
EC	Engenharia de Conhecimento
ER	Esquema de Representação de Conhecimento
Ext	Estrutura Externa
fbf	Fórmula Bem-Formada
IA	Inteligência Artificial
Int	Estrutura Interna
ipc	Introdução de Peça de Conhecimento
ju	Justificativa
LC	Linguagem de Consulta
LO	Linguagem Objeto
lr	Linha de Raciocínio
MI	Máquina de Inferência
ML	Metalinguagem
pbd	Problema Bem-Definido
PC, pc	Peça de Conhecimento
PCon	Processador de Consultas
PR	Definição da Relação de Provabilidade
Prop	Conjunto de Propriedades
RC	Representação de Conhecimento
rc	Requisição de Conhecimento
SApr	Sistema de Apresentação
SC	Sistema Baseado em Conhecimento
SCog	Sistema Cognitivo
SCom	Sistema de Comunicação
SE	Sistema Especialista
SExp	Sistema Explanativo
SF	Sistema de Frames
SGBC	Sistema Gerenciador de Bases de Conhecimento
SGBD	Sistema Gerenciador de Bases de Dados
SP	Sistema de Processamento de Conhecimento
SRes	Sistema Resolutivo

LISTA DE SINAIS

$\square$	Cláusula Vazia
$\wedge$	Conjunção
$\models$	Consequência Lógica
$\subseteq$	Contenção
$\vdash\!\urcorner$	Derivação *bottom-up*
$\setminus$	Diferença
$\vee$	Disjunção
$\equiv$	Equivalência
$\gg$	Extensão
F	Falso
Δ	Fórmula Nula
$\rightarrow$	Implicação
$\vdash$	Inferência, Derivação *top-down*
$\cap$, $\bigcap$	Intersecção
$\top$	Maior Elemento
$\bot$	Menor Elemento
$\Box$	Necessidade
$\neg$	Negação
$\in$	Pertinência
$\Diamond$	Possibilidade
$\forall$	Quantificador Universal
$\exists$	Quantificador Existencial
$\lceil$	Restrição
$/$	Substituição
$\cup$, $\bigcup$	União
V	Verdadeiro

LISTA DE FIGURAS

Figura 2.1 O Ciclo Cognitivo 23
Figura 2.2 Funcionamento de um ER Baseado em Regras........ 46
Figura 2.3 Exemplo de uma Rede Semântica 47
Figura 2.4 Um Sistema de Frames Simplificado 51

Figura 3.1 Supercontextos das Cláusulas de Horn 74

Figura 4.1 Axiomatização Explícita do metaProlog 113
Figura 4.2 Axiomatização Implícita do metaProlog 114
Figura 4.3 Relações **add_to** e **drop_from** 114
Figura 4.4 Uma Contradição da Semântica Declarativa 117
Figura 4.5 O Predicado **dbm** 119
Figura 4.6 Restrições de Inserção 120
Figura 4.7 Modificando Restrições 121
Figura 4.8 Associando Fatores de Confiança 122

Figura 5.1 Distinção entre Conhecimento e Dados 135
Figura 5.2 Uma Analogia entre BDs e Conceitos Prolog 143

Figura 6.1 Refinamento de um Hipernodo 154
Figura 6.2 Refinamento de uma Hiperrelação 158
Figura 6.3 Padrões de Ligação Livres 174
Figura 6.4 Padrões de Ligação Conjuntivos 175
Figura 6.5 Padrões de Ligação Condicionais 176

Figura 7.1 Especificando BCs como Hiperredes 179
Figura 7.2 Especificação Sintática de um Identificador 180
Figura 7.3 Especificação Sintática de um Sort 180
Figura 7.4 Especificação de Estruturas Internas 181
Figura 7.5 Especificação de Estruturas Externas 183
Figura 7.6 Especificando Listas de Propriedades 184
Figura 7.7 Estrutura de uma BC Geográfica 185
Figura 7.8 Derivação Top-Down em Hiperredes 188
Figura 7.9 Extraindo Aspectos de uma BC 192

Figura 7.10 Contextualização de Aspectos 196
Figura 7.11 Um Sistema de Contextos 197
Figura 7.12 Representando a Rede Taxonômica 198
Figura 7.13 Demonstrando um Conjunto de Objetivos 199
Figura 7.14 Enfoques Semânticos do Modelo das Hiperredes ... 201
Figura 7.15 A Operação de Instanciação 202
Figura 7.16 Composição Inclusiva e Composição Exclusiva 203
Figura 7.17 Decomposição de PCs 204
Figura 7.18 Abstração Conjuntiva e Abstração Disjuntiva 205
Figura 7.19 Modificação de uma PC 205
Figura 7.20 Deletando uma PC de uma BC 206
Figura 7.21 Renomeando PCs 206
Figura 7.22 Visões em Hiperredes 208
Figura 7.23 Definindo Visões Próprias em Hiperredes 210
Figura 7.24 Visões como Hiperredes 210

Figura 8.1 Requisitos da Arquitetura de SGBCs 214
Figura 8.2 Arquitetura Geral de um SGBC 215
Figura 8.3 Esquema Semântico dos Conceitos de Abstração ... 233
Figura 8.4 Organização do Conhecimento 240

Figura 9.1 Arquitetura Proposta para o Sistema Rhesus 244
Figura 9.2 Entidades em BCs 246
Figura 9.3 O Sistema de Processamento de Conhecimento 251
Figura 9.4 Estrutura Funcional do Sistema de Comunicação .. 264
Figura 9.5 O SCog Enfatizado no Contexto do SP 272
Figura 9.6 O SRes Enfatizado no Contexto do SP 273

RESUMO

Apesar de sua inerente indecidibilidade e do problema da negação, extensões da lógica de primeira ordem tem se mostrado capazes de superar a questão da monotonicidade, vindo a constituir esquemas de representação de conhecimento de expressividade virtualmente universal. Resta entretanto solucionar ou pelo menos amenizar as consequências do problema do controle, que limitam o seu emprego a aplicações de pequeno a médio porte. Investigações nesse sentido **[BOW 85] [MON 88]** *indicam que a chave para superar a explosão inferencial passa obrigatoriamente pela estruturação do conhecimento, de modo a permitir o exercício de algum controle sobre as possíveis derivações dele decorrentes. O modelo das hiperredes* **[GEO 85]** *parece atingir tal objetivo, dado o seu elevado potencial de estruturação e o instrumental que oferece para o tratamento de construções descritivas, operacionais e organizacionais. Além disso, a simplicidade e uniformidade sintática de suas entidades primitivas possibilita uma interpretação semântica bastante clara do modelo original, por exemplo, baseada em grafos. O presente trabalho representa uma tentativa de associar a programação em lógica ao formalismo das hiperredes, visando obter um novo modelo capaz de preservar as expressividade da primeira, beneficiando-se simultaneamente do potencial heurístico e estrutural do segundo.*

Inicialmente procura-se obter uma noção clara da natureza do conhecimento e de seus mecanismos com o objetivo de caracterizar o problema da representação de conhecimento. Diferentes esquemas correntemente empregados para esse fim (sistemas de produções, redes semânticas, sistemas de frames, programação em lógica e a linguagem Krypton) são estudados e caracterizados do ponto de vista de sua expressividade, potencial heurístico e conveniência notacional. A programação em lógica é objeto de um estudo em maior profundidade, sob os enfoques modelo-teorético e prova-teorético. Sistemas de programação em

lógica - particularmente a linguagem Prolog e extensões em nível meta - são investigados como esquemas de representação de conhecimento, considerando seus aspectos sintáticos e semânticos e a sua relação com Sistemas Gerenciadores de Bases de Dados.

O modelo das hiperredes é apresentado introduzindo-se, entre outros, os conceitos de hipernodo, hiperrelação e protótipo, assim como as propriedades particulares de tais entidades. A linguagem Hyper, para o tratamento de hiperredes, é formalmente especificada. Emprega-se a linguagem Prolog como formalismo para a representação de Bases de Conhecimento estruturadas segundo o modelo das hiperredes. Sob tal abordagem uma Base de Conhecimento é vista como um conjunto (possivelmente vazio) de objetos estruturados ou peças de conhecimento, que por sua vez são classificados como hipernodos, hiperrelações ou protótipos. Um mecanismo top-down para a produção de inferências em hiperredes é proposto, introduzindo-se os conceitos de aspecto e visão sobre hiperredes, os quais são tomados como objetos de primeira classe, no sentido de poderem ser valores atribuídos a variáveis. Estuda-se os requisitos que um Sistema Gerenciador de Bases de Conhecimento deve apresentar, do ponto de vista da aplicação, da engenharia de conhecimento e da implementação, para suportar efetivamente os conceitos e abstrações (classificação, generalização, associação e agregação) associadas ao modelo proposto. Com base nas conclusões assim obtidas, um Sistema Gerenciador de Bases de Conhecimento (denominado Rhesus em alusão à sua finalidade experimental) é proposto e especificado, objetivando confirmar a viabilidade técnica do desenvolvimento de aplicações baseadas em lógica e hiperredes.

PALAVRAS-CHAVE: Inteligência Artificial, Representação de Conhecimento, Programação em Lógica, Problema do Controle, Bases de Conhecimento, Modelagem Semântica, Hiperredes.

TITLE: "KNOWLEDGE REPRESENTATION: LOGIC PROGRAMMING AND THE HYPERNETS MODEL."

ABSTRACT

In spite of its inherent undecidability and the negation problem, extensions of first-order logic have been shown to be able to overcome the question of the monotonicity, establishing knowledge representation schemata with virtually universal expressiviness. However, one still has to solve, or at least to reduce the consequences of the control problem, which constrains the use of logic-based systems to either small or medium-sized applications. Investigations in this direction **[BOW 85] [MON 88]** *indicate that the key to overcome the inferential explosion resides in* the *proper knowledge structure representation, in order to have some control over possible derivations. The Hypernets Model* **[GEO 85]** *seems to reach such goal, considering its high structural power and the features that it offers to deal with descriptive, operational and organizational knowledge. Besides, the simplicity and syntactical uniformity of its primitive notions allows a very clear definition for its semantics, based, for instance, on graphs. This work is an attempt to associate logic programming with the hypernets formalism, in order to get a new model, preserving the expressiveness of the former and the heuristic and structural power of the latter.*

First we try to get a clear notion of the nature of knowledge and its main aspects, intending to characterize the knowledge representation problem. Some knowledge representation schemata (production systems, semantic networks, frame systems, logic programming and the Krypton language) are studied and characterized from the point of view of their expressiveness, heuristic power and notational convenience. Logic programming is the subject of a deeper study, under the model-theoretic and proof-theoretic approaches. Logic programming systems - in

particular the Prolog language and metalevel extensions - are investigated as knowledge representation schemata, considering its syntactic and semantic aspects and its relations with Data Base Management Systems.

The hypernets model is presented, introducing the concepts of hypernode, hyperrelation and prototype, as well as the particular properties of those entities. The Hyper language, for the handling of hipernets, is formally specified. Prolog is used as a formalism for the representation of Knowledge Bases which are structured as hypernets. Under this approach a Knowledge Base is seen as a (possibly empty) set of structured objects, which are classified as hypernodes, hyperrelations or prototypes. A mechanism for top-down reasoning on hypernets is proposed, introducing the concepts of aspect and vision, which are taken as first-class objects in the sense that they could be assigned as values to variables. We study the requirements for the construction of a Knowledge Base Management System from the point of view of the user's needs, knowledge engineering support and implementation issues, actually supporting the concepts and abstractions (classification, generalization, association and aggregation) associated with the proposed model. Based on the conclusions of this study, a Knowledge Base Management System (called Rhesus, refering to its experimental objectives) is proposed, intending to confirm the technical viability of the development of applications based on logic and hypernets.

KEYWORDS: Artificial Intelligence, Knowledge Representation, Logic Programming, Control Problem, Knowledge Bases, Semantic Modeling, Hypernets.

1 INTRODUÇÃO

Os últimos anos vem testemunhando uma crescente demanda por sistemas computacionais ditos *inteligentes* ou *especialistas* ou ainda **sistemas baseados em conhecimento** (SCs). Tais sistemas se diferenciam dos sistemas convencionais de processamento de dados por apresentarem, entre outras características, elevado grau de independência entre o conhecimento e os mecanismos que o transformam e aplicam. Assim, a principal diferença entre os SCs e os sistemas convencionais não reside na forma sob a qual o conhecimento é aplicado, mas sim na forma em que este é *organizado*, isto é, na sua particular arquitetura. Enquanto que nos sistemas convencionais o conhecimento se encontra implícito no código de seus programas, nos SCs este constitui uma entidade separada, perfeitamente identificável, denominada **base de conhecimento** (BC). A BC é manipulada por um componente de controle, usualmente uma máquina de inferência de propósito geral, que é capaz de tomar decisões com base no conhecimento armazenado, responder perguntas sobre esse conhecimento e determinar as conseqüências que ele implica. Isso significa que o conhecimento incorporado a um SC é independente dos programas que o manipulam, de modo que é mais facilmente acessado, modificado ou ampliado por seus usuários.

Essa particular organização subentende a existência de um formalismo capaz de *representar* adequadamente o conhecimento, tornando-o manipulável por computadores. Diversas metodologias tem sido propostas para esse fim, caracterizando uma área de concentração de pesquisas no campo da Inteligência Artificial (IA) denominada **representação de conhecimento** (RC). A RC tem por objetivo o estudo dos **esquemas de representação de conhecimento** (ERs), que podem ser associados a conjuntos de convenções formais capazes de capturar e permitir a manipulação em computadores de certos aspectos de um determinado universo de discurso.

É fundamental, na especificação de um ER, garantir que o mesmo represente precisamente o universo de discurso (ou domínio de aplicação) que pretende descrever. Isto significa que o ER deve ser capaz de capturar corretamente o significado de cada uma das entidades que compõem o domínio e formular com precisão a semântica de todas as relações que podem ocorrer entre tais entidades. À essa característica de um ER denomina-se *expressividade*. Segundo [JAC 86], além da expressividade, um ER se caracteriza pelo seu *potencial heurístico* e sua *conveniência notacional*, isto é, pela eficiência que oferece à utilização do conhecimento que representa e pela facilidade de compreensão direta e inequívoca das construções que emprega para esse fim. Tais características muitas vezes encontram-se em conflito, tornando necessário um balanceamento entre elas.

Com relação à expressividade, verifica-se que certos ERs apresentam-se mais adequados que outros, dependendo do aspecto do conhecimento que se pretende enfatizar. As redes semânticas, por exemplo, destacam as características declarativas do conhecimento e são, por esta razão, muito apropriadas para descrição da parte passiva do domínio de aplicação. Já outros ERs, como os sistemas de produções, se concentram na representação dos aspectos procedimentais, sendo portanto voltados à descrição da parte ativa ou comportamental do universo de discurso. Finalmente, uma terceira categoria de ERs, como os sistemas de frames, se direciona à descrição dos aspectos estruturais do conhecimento, sendo empregada principalmente para propósitos de organização de BCs. A expectativa de se vir a obter um ER de expressividade geral esbarra na complexidade do mundo real, o qual, segundo [HOF 79], não pode ser representado em um único formalismo.

O emprego da lógica como ER, estabelecido por Kowalski [KOW 74] [KOW 78], vem resistindo ao longo dos anos ao surgimento de uma multiplicidade de novos formalismos propostos com o objetivo de solucionar algumas questões cruciais a ela inerentes. Dentre tais questões destaca-se o *problema do controle*, que pode

ser definido como a dificuldade em controlar o crescimento potencial do número de inferências possíveis a partir de um determinado objetivo. Verifica-se que mesmo subconjuntos muito mais manejáveis da lógica de primeira ordem, como é o caso das cláusulas de Horn, possuem essa limitação, que restringe o seu emprego a aplicações de pequeno a médio porte. Por outro lado, os formalismos baseados em lógica apresentam diversas vantagens bem conhecidas sobre as demais técnicas de RC **[REI 84]**, tais como uma semântica bem definida, notação simplificada, economia conceitual, uniformidade operacional e de representação, etc.

O problema do controle tem sido abordado em diversos estudos, entre os quais **[BOW 85]** e **[MON 88]**, que propõem a estruturação do espaço de prova e o controle das inferências mediante metaconhecimento incorporado ao próprio formalismo. Por outro lado Georgescu, em **[GEO 85]**, propõe um modelo para a representação de conhecimento baseado em estruturas recursivas denominadas *hiperredes*, dotadas de amplo potencial de estruturação. O trabalho aqui apresentado pode ser caracterizado como uma tentativa da associar a programação em lógica ao formalismo das hiperredes, visando obter um novo modelo capaz de preservar a expressividade da primeira ao mesmo tempo em que se beneficia do potencial heurístico e da capacidade de estruturação do segundo.

O ponto focal da investigação aqui desenvolvida é demonstrar que a associação entre a programação em lógica e o modelo das hiperredes, além de possuir indiscutíveis vantagens sobre cada um dos dois formalismos tomados isoladamente, apresenta características únicas, normalmente não encontradas na maioria dos ERs em utilização atualmente, como por exemplo a facilidade em se definir dinamicamente a forma de raciocínio a utilizar e o tratamento uniforme que permite dispensar às diferentes categorias de conhecimento. Com a finalidade de confirmar a viabilidade técnica do desenvolvimento de aplicações baseadas no modelo apresentado, é também proposta uma arquitetura para um Sistema Gerenciador de Bases de Conhecimento (denominado *Rhesus* em alusão à sua finalidade experimental).

A presente dissertação está organizada em 10 capítulos. No capítulo 2 são examinados os conceitos fundamentais da RC, caracterizando-se alguns dos ERs mais utilizados, de modo a melhor situar a proposta apresentada. No capítulo 3 conceitualiza-se a sintaxe usual da programação em lógica e as semânticas modelo-teorética e prova teorética. Algumas propostas de extensão à programação em lógica de primeira ordem são consideradas no capítulo 4, em particular o sistema metaProlog, proposto em [BOW 85] e o sistema de programação em lógica contextual de Monteiro e Porto [MON 88]. No capítulo 5 é discutido o emprego da programação em lógica na descrição de bases de dados, considerando-se as divergências semânticas envolvidas. O modelo das hiperredes é apresentado no capítulo 6, enfatizando-se suas propriedades de estruturação, expressividade e potencial heurístico. O emprego da programação em lógica a nível meta para a descrição de hiperredes é investigado no capítulo 7, onde se apresenta um mecanismo para a derivação top-down e se introduz os conceitos de *visão* e *aspecto* sobre hiperredes. No capítulo 8 examina-se os requisitos que devem ser considerados no projeto de SGBCs do ponto de vista da aplicação, da engenharia do conhecimento e da implementação. No capítulo 9 propõe-se uma arquitetura para o desenvolvimento de um SGBC experimental, baseado no modelo das hiperredes representado em lógica. Finalmente, no capítulo 10 são apresentadas as conclusões obtidas sobre o estudo realizado.

UFRGS
INSTITUTO DE INFORMATIC
BIBLIOTECA

2. CONHECIMENTO E REPRESENTAÇÃO

Uma idéia clara do que significa *conhecimento* e como este pode ser *representado* é de fundamental importância para os propósitos do presente trabalho. Neste capítulo examinamos tais conceitos caracterizando o *problema da representação de conhecimento* e introduzindo alguns dos modelos de representação mais utilizados atualmente de modo a melhor situar nossa proposta.

2.1 CONHECIMENTO

O problema de definir precisamente o que significa *conhecimento* parece não ter sido ainda completamente solucionado em nenhum dos ramos da Ciência que o tomam como objeto de estudo. Na tentativa de estabelecer uma noção apropriada à abrangência pretendida para presente trabalho iremos inicialmente apresentar o pensamento de alguns autores a esse respeito. Em **[FIS 87]**, por exemplo, *conhecimento* é definido como sendo

> "... *a informação armazenada ou os modelos empregados por uma pessoa ou máquina para interpretar, predizer e apropriadamente responder aos estímulos do mundo exterior.*"

Os autores de tal definição destacam a importância de se estabelecer uma distinção clara entre o conteúdo e a forma de um corpo de conhecimento, isto é, entre o conhecimento propriamente dito e a sua representação. Essa mesma idéia foi defendida por Allen Newell, no seu artigo antológico "The Knowledge Level" **[NEW 82]**, onde sustenta que a concepção de conhecimento é logicamente anterior à de representação e, até que uma noção clara do primeiro exista, a última permanecerá confusa.

Na visão de Newell:

> *"Conhecimento é tudo o que pode ser atribuído a um agente tal que o seu comportamento possa ser computado de acordo com o Princípio da Racionalidade."*

Fica evidenciado, a partir das duas definições apresentadas, que a noção de conhecimento está intimamente relacionada com uma série de outros conceitos dos quais não pode ser dissociada. O Princípio da Racionalidade, por exemplo, é formulado por Newell como sendo a lei de comportamento que governa um agente e permite a previsão de seu comportamento, correspondendo à idéia de que o conhecimento que ele possui será empregado a serviço de seus objetivos, isto é:

> *"Se um agente tem o conhecimento de que uma de suas ações atenderá a um de seus objetivos, então o agente selecionará esta ação."*

Uma definição mais próxima dos objetivos da presente dissertação e que adotaremos como hipótese de trabalho é formulada em [**MAT** 89] e corresponde ao conceito usualmente adotado na literatura corrente associada à IA, segundo o qual conhecimento é:

> *"... a soma das percepções de um indivíduo acerca de um dado universo de discurso em um determinado tempo."*

Em outras palavras, considera-se conhecimento como sendo alguma coisa que um determinado indivíduo (o agente cognitivo) "sabe" a respeito de um universo específico em um dado instante. Essa formulação nos parece mais adequada porque explicita os três conceitos que contribuem decisivamente para caracterizar um corpo de conhecimento independentemente da representação adotada: O *agente cognitivo*, que corresponde à entidade detentora do conhecimento, o *universo de discurso*,

(também denominado *domínio da aplicação* ou simplesmente *domínio*), que pode ser qualquer subconjunto, tanto do universo real como de algum universo não-real, futuro, imaginário ou baseado em crenças individuais, e o *instante* ou *tempo* em que o conhecimento ocorre, que vem caracterizar a sua condição dinâmica.

A importância do conhecimento para nós, humanos, reside no fato de que quase tudo o que fazemos encontra-se de alguma forma baseado nele, isto é, *aplicamos* o conhecimento que possuimos para atingir nossos objetivos. A habilidade de aplicar conscientemente o conhecimento é certamente uma das diferenças óbvias existentes entre os seres humanos e as demais criaturas, entretanto, constata-se que somente somos capazes de aplicar o conhecimento que possuímos, ou seja, não temos capacidade de executar tarefas com as quais não estamos familiarizados. Essa observação sugere que a *aplicação*, apesar de ser a mais importante, não é a única das atividades humanas envolvendo o conhecimento. Na verdade, a aplicação é apenas uma das quatro atividades que constituem o que se denomina *ciclo cognitivo*, mostrado na **Figura 2.1**, abaixo, adaptada de [**MAT 89**]:

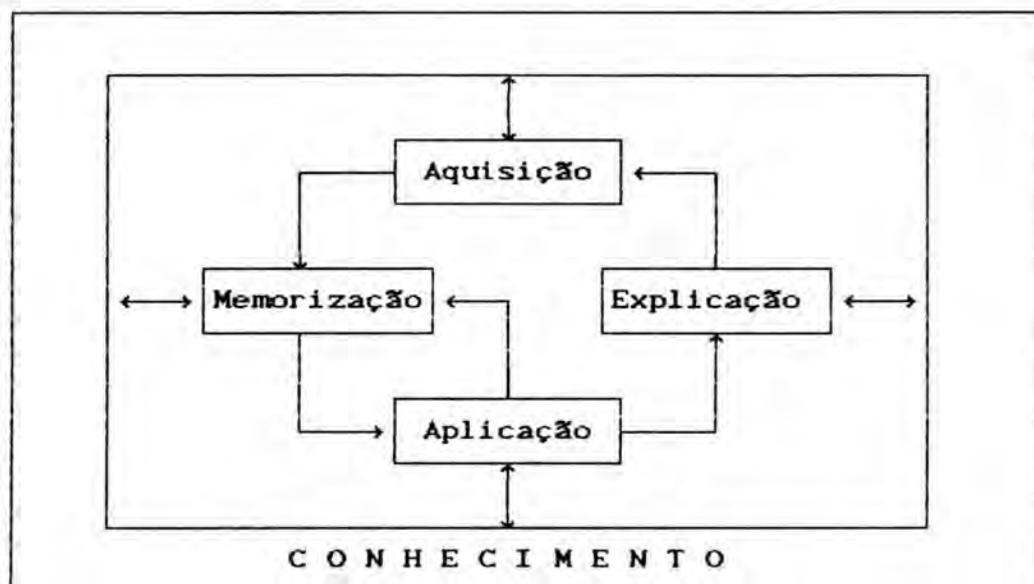


Figura 2.1

O Ciclo Cognitivo

O ciclo cognitivo representa então a sequência de atividades percorrida pelo conhecimento quando este é aplicado no processo de solução de problemas, de forma que, para entender tal processo, é também necessário analisar as demais fases do ciclo. Assim, a primeira fase de importância nesse contexto é a *aquisição de conhecimento*, isto é, a metodologia empregada para o enriquecimento da "base de conhecimento" do agente cognitivo. Isso envolve a coleta de conhecimento, a partir de diversas fontes, e a sua assimilação, correspondendo, em um certo sentido, às atividades de *percepção* e *atenção* da psicologia cognitiva. Após a assimilação, o conhecimento precisa ser de alguma forma representado e armazenado, de modo a permitir futuras referências. Esse processo chama-se *memorização* e o seu estudo corresponde ao que a psicologia cognitiva denomina *memória*.

Uma vez memorizado, o conhecimento encontra-se em condições de ser aplicado na fase seguinte, que corresponde ao processo de solução de problemas. Solucionamos problemas basicamente por meio do *pensamento*. Isso significa a transformação ativa do conhecimento memorizado de forma a produzir novos conhecimentos que por sua vez são aplicados para atingir o objetivo desejado. Alcançado o objetivo, o conhecimento inferido durante o processo é normalmente assimilado automaticamente. Tal assimilação, entretanto, não é suficiente para manter a informação obtida na aplicação do conhecimento. Para isso é necessário *transmiti-lo* a outros agentes cognitivos. A transmissão de um novo conhecimento geralmente envolve a *explicação* de como foi obtida a sua aquisição. A *explicação* é, portanto, a quarta e última fase do ciclo cognitivo. O processo de explicar alguma coisa envolve novamente a aquisição de conhecimento, completando assim o ciclo.

É importante notar que os processos de aquisição e memorização são fundamentalmente *endógenos*, uma vez que, em diferentes níveis, canalizam o conhecimento do mundo exterior para a base de conhecimento do agente cognitivo. Por outro lado, o processo de explicação é fortemente *exógeno*, isto é, transmite o conhecimento explícita ou implicitamente representado na base

de conhecimento para o mundo exterior. Já a aplicação do conhecimento possui dupla caracterização: transforma o conhecimento armazenado, produzindo (internamente) novos conhecimentos que, por sua vez, são aplicados ao mundo exterior. Destacamos, entretanto, que o único processo "*puro*" é a memorização, visto que os demais possuem em maior ou menor grau alguma característica exógena.

2.2 REPRESENTAÇÃO DE CONHECIMENTO

A idéia de que um certo agente cognitivo *possui* um determinado conhecimento conduz obrigatoriamente à intuição de que o mesmo deve ser *memorizado* (ver a definição de ciclo cognitivo apresentada na seção anterior), isto é, *representado* e *armazenado* de forma a estar disponível no momento em que for necessário. Por outro lado, para evitar que uma determinada porção deste conhecimento, comum a várias situações, se repita de maneira redundante nas diferentes peças de conhecimento de que participa, é preciso que a mesma seja representada de modo a permitir a combinação e integração com outras porções, formando a peça de conhecimento necessária no momento de sua aplicação.

Assim como a noção de conhecimento, a idéia de *representação* não possui ainda uma formulação suficientemente clara. Para Newell [**NEW 82**], por exemplo, ela é definida como sendo um sistema simbólico que codifica um corpo de conhecimento, isto é, uma *representação* é simplesmente outro termo para referir uma estrutura que denota. Por definição, **X** representa **Y** se **X** denota aspectos de **Y**, isto é, se existem processos simbólicos que tomam **X** como entrada e comportam-se como se tivessem acesso a alguns aspectos de **Y**. Ainda segundo Newell, uma representação é algumas vezes formulada em termos de um mapeamento dos aspectos de **Y** aos aspectos de **X**. Outras vezes, em termos de uma estrutura de dados com operações próprias associadas. Essa visão enfatiza

o acoplamento da estrutura estática (chamada simplesmente *a representação*) e o processamento que define o que pode ser armazenado ou recuperado sobre a estrutura e que transformações podem ser efetuadas sobre o que está representado.

Em [FIS 87] encontramos a ponderação de que nenhuma representação ou modelo único pode capturar todos os aspectos de um objeto real, de forma que uma entidade inteligente precisa empregar um largo espectro de representações para poder lidar com o mundo real. A visão desses autores é, em muitos aspectos, semelhante à formulada por Newell, uma vez que oferecem a seguinte definição para representação:

> *"Uma representação de uma situação (ou objeto, ou problema) é a tradução dessa situação para um sistema constituído por um vocabulário que nomeia objetos e relações, operações que podem ser executadas sobre tais objetos e fatos e restrições sobre eles."*

O propósito de uma representação parece ser a simplificação do problema de responder a uma classe restrita de questões sobre uma determinada situação, de modo que a seleção da representação a ser adotada deve ser dirigida por objetivos. Pelo menos duas representações distintas são requeridas para se dotar uma máquina com a capacidade de solucionar problemas do mundo real: a primeira deve oferecer o aparato simbólico adequado para a resposta de questões sobre a situação em análise, enquanto que a segunda traduz as técnicas de solução formuladas pela primeira para as operações e estruturas de armazenamento inerentes à própria máquina. Evidentemente esses dois níveis devem ser considerados em uma implementação efetiva de um sistema de representação mas, no presente capítulo, estaremos nos ocupando principalmente com o primeiro.

Podemos então pensar em **representação de conhecimento**, ou abreviadamente **RC**, como sendo o campo da IA que estuda as maneiras de fornecer aos SCs o conhecimento que eles necessitam

para serem capazes de solucionar problemas acerca de um determinado domínio. Assim, podemos conceituar RC como sendo

> *"... um conjunto de convenções sintáticas, semânticas e pragmáticas que torna possível a descrição de entidades do universo de discurso."*

Tais convenções constituem os denominados **esquemas de representação (ERs)**, dos quais se espera, segundo [JAC 86], *expressividade*, *potencial heurístico* e *conveniência notacional*, ou, em outras palavras, que permitam representar *qualquer* conhecimento sobre o universo de discurso, ofereçam uma forma eficiente de utilização do conhecimento assim representado e que sejam fáceis de ler e de escrever, permitindo compreensão de maneira direta e inequívoca. Tais requisitos frequentemente encontram-se em conflito, tornando necessário um balanceamento entre eles. Por exemplo, um ER de grande expressividade como o cálculo completo de predicados de primeira ordem, apresenta sérias dificuldades computacionais que se manifestam no difícil controle de suas inferências. Por outro lado, ERs que apresentam pouca expressividade, como é o caso das redes semânticas, podem possuir um grande potencial heurístico.

O papel desempenhado pelos ERs na construção de SCs encontra-se, como já vimos, intimamente relacionado com os objetivos de tais sistemas, que podem ser resumidos nas seguintes categorias, segundo [FIS 87]:

- **Redução de Problemas:** Conversão de um problema em outro que possui uma solução conhecida;

- **Interpretação:** Informação sensorial pode ser interpretada através do uso de representações internas de objetos reais. Por exemplo, informação visual pode ser interpretada através da comparação dos dados percebidos visualmente com as descrições de objetos armazenadas;

- **Função de Organização:** Uma representação pode permitir a organização da informação de forma que as similaridades e diferenças entre objetos e eventos sejam mais prontamente identificadas, por exemplo através da plotagem de grafos;

- **Função de Questionamento:** Os modelos internos conduzem à formulação questões sobre determinados eventos. Por que determinado evento ocorre quando o modelo conduz a predizer algo diferente? Somos assim guiados a revisar em nossos modelos a geração de alternativas que conduzem a respostas inconsistentes;

- **Função Preditiva:** Um modelo interno permite predizer os eventos que irão resultar de determinadas ações. Por exemplo, um modelo matemático de um foguete permitirá predizer o seu movimento;

- **Função Dedutiva:** Certos ERs podem ser utilizados para tornar novos conhecimento explícitos permitindo a formulação de deduções sobre o conhecimento original.

A grosso modo, os diferentes esquemas atualmente empregados para a representação de conhecimento podem ser agrupados em três modelos fundamentais: (1) baseados em regras (sistemas de produções), (2) baseados em objetos estruturados (redes semânticas, frames, roteiros, etc), e (3) baseados em lógica. Cada ER, não importando o modelo em que se situa, é caracterizado pela soma das construções que oferece para representar o modelo do domínio de aplicação e as operações que irão atuar sobre ele. Fundamentalmente, as operações sobre BCs não apresentam grandes diferenças entre os ERs, incluindo o armazenamento e a recuperação de seu conteúdo e, de alguma forma, derivando novos conhecimentos com base no já existente. Assim, as construções oferecidas pelos ERs para a modelagem do universo

do discurso é o que irá efetivamente refletir a sua expressividade. Tais construções estão ligadas a alguma noção de sintaxe para a descrição do mundo, possuindo uma semântica associada. O potencial semântico é exatamente o que diferencia esquemas pobres dos apropriadamente expressivos.

A semântica dos ERs é também particularmente importante porque define a forma de modelar o domínio de aplicação. O relacionamento entre os objetos da representação e as entidades do mundo é parte da semântica de um ER. O significado de uma declaração em uma BC é usualmente representado por *verdadeiro* (V) ou *falso* (F). A esse respeito, a interpretação de um conjunto de declarações em uma BC consiste na especificação de um conjunto não-vazio de entidades e um mapeamento dos objetos nas declarações para as entidades especificadas. Se todas as declarações na BC são verdadeiras em uma particular interpretação, essa interpretação é dita ser um *modelo* para a BC [GAL 84]. A partir das interpretações de uma BC podemos responder a várias questões a seu respeito:

- Uma BC é dita *inconsistente* se não possui modelo. Isso ocorre quando informação contraditória pode ser derivada com base no conteúdo da BC.

- Com respeito a um modelo da BC, pode ser provado que um esquema inferencial deriva somente sentenças verdadeiras. Tal fato denomina-se *validade inferencial* ou *coerência*.

- Uma BC é dita ser *completa* quando toda declaração verdadeira que pode ser exprimida a partir das construções do ER utilizado pode ser dela derivada.

As relações mais importantes entre a semântica e a sintaxe dos ERs são a **coerência** e a **completeza** [GAL 84]. Assim, uma determinada BC (uma *instância* de um ER) é dita *coerente* se e

somente se para todo conjunto de sentenças $S \in BC$, se S permite derivar uma sentença s, então s é consequência lógica de S, ou seja:

$$\text{coerente}(BC) \equiv S \vdash s \rightarrow S \models s$$

Por outro lado, a BC é dita *completa* se e somente se e somente se para todo conjunto de sentenças $S \in BC$, se uma sentença s é consequência lógica de S, então S permite derivar s, isto é:

$$\textbf{completa}(BC) \equiv S \models s \rightarrow S \vdash s$$

2.3 O PROBLEMA DA REPRESENTAÇÃO DO CONHECIMENTO

O propósito desta seção é analisar os principais problemas enfrentados pela pesquisa no campo da representação de conhecimento com o objetivo de estabelecer uma base adequada para a formulação de nossa proposta. A partir dessa premissa estudaremos o *problema da representação do conhecimento* inicialmente sob o aspecto epistemológico em sua relação com o *problema da IA* e após sob o enfoque das características estabelecidas em [JAC 86] para os ERs (expressividade, potencial heurístico e conveniência notacional).

2.3.1 Considerações Epistemológicas

O **problema da IA**, ou seja, "*como obter máquinas verdadeiramente capazes de pensar*", vem sendo alvo, ao longo dos anos, de incessante debate entre os seguidores de duas correntes filosóficas antagônicas. A primeira delas, que pode ser

denominada "*a linha do desempenho*" e que tem em John McCarthy um de seus maiores expoentes [MCC 69] [MCC 77] [MCC 80], afirma que o problema da IA pode ser solucionado por meio de programas projetados para raciocinar de acordo com linguagens extraídas da lógica matemática, não importando absolutamente se esta é ou não a forma que os seres humanos utilizam para pensar. A outra corrente, que denominaremos "*a linha da emulação*", é representada pelo pensamento de Marvin Minski [MIN 75] [MIN 82] e proclama que uma abordagem adequada para tentar resolver o problema da IA é levar os computadores a imitar o processo de raciocínio executado na mente humana, o qual, quase certamente, não ocorre através da lógica matemática.

Ambas as partes do debate concordam entretanto que um dos objetivos centrais de suas pesquisas é dotar os computadores com boa parte do conhecimento humano acerca do mundo e das entidades nele existentes. Esse corpo de conhecimento genérico e de limites indefinidos é conhecido por "*senso comum*". O problema reside então em como reproduzir tal conhecimento em uma máquina, isto é, como projetar um robô com capacidade de raciocínio suficientemente poderosa para utilizar o conhecimento de senso comum, extraindo dele o suficiente para inteligentemente explorar e adaptar-se ao seu ambiente. Podemos assumir que o conhecimento de senso comum corresponde a noções como as de que "*os objetos caem, a menos que algo os sustente*", "*os objetos físicos não desaparecem repentinamente*" e que "*podemos nos molhar se nos expusermos à chuva*".

David J. Israel em [ISR 83] propõe que o problema da IA pode ser reduzido ao "**problema da representação do conhecimento**", isto é, se for possível obter um modelo capaz de efetivamente representar o conhecimento de senso comum então o problema da IA estará substancialmente resolvido. Em [FIS 87] encontramos a ponderação de que uma BC capaz de acomodar conhecimento de senso comum deveria incluir as descrições gerais das propriedades espaciais, o comportamento de materiais e líquidos e possuir um entendimento "ingênuo" de tópicos tais como física, botânica, zoologia, ecologia, etc. Um tipo especial de raciocínio parece

estar envolvido com o emprego do conhecimento de senso comum. Apesar do mundo real se apresentar contínuo aos nossos sentidos, não necessitamos, na verdade, possuir uma representação contínua de suas entidades, como ocorre em modelos matemáticos e físicos. Parece que as pessoas lidam com o mundo real tratando-o de maneira qualitativa, empregando apenas uns poucos valores para qualquer uma das variáveis envolvidas, por exemplo: *muito grande*, *grande*, *mediano*, *pequeno* e *muito pequeno*. Quantizações similares podem ser empregadas para proximidade, força, peso, etc. Um raciocínio desse tipo parece ser adequado para a solução dos problemas do dia a dia, por ser capaz de nos transmitir uma idéia sobre como as coisas funcionam e como utilizar alguma coisa de maneira diferente daquela originalmente concebida, por exemplo: usar um tronco de árvore caído como banco. Outro problema associado com o conhecimento de senso comum é o **"problema da relevância"**, isto é, dada uma situação do mundo real, como pode um certo raciocínio determinar que outros objetos irão interagir significativamente com as entidades de interesse ou quais aspectos de quais objetos devem ser efetivamente considerados.

A modelagem do conhecimento de senso comum é abordada por diversos autores. McCarthy, por exemplo, em **[MCC 77]** propos uma regra de conjectura não-monotônica que denominou **circunscrição** e que pode ser formulada intuitivamente da seguinte maneira:

> *"Se conhecemos alguns objetos de uma mesma classe e temos algum modo de gerar mais, podemos concluir diretamente que isso nos dará todos os objetos desta classe."*

Segundo McCarthy, o raciocínio do senso comum está ordinariamente pronto para "*saltar para a conclusão*" que uma determinada ferramenta pode ser usada para o propósito pretendido a menos que alguma coisa impeça o seu uso. Considerada puramente extensional, tal declaração não carrega informação alguma. Parece que meramente assegura que uma ferramenta pode ser usada para o propósito pretendido, a menos que não possa.

Heuristicamente, entretanto, a declaração não é apenas uma disjunção tautológica. Ela sugere a formação de um plano para o uso da ferramenta.

Existe o consenso de que a maior parte dos elementos presentes em um sistema "*inteligente*" não pode funcionar sem *conhecimento* do domínio de aplicação. Em um sistema computadorizado, tal conhecimento deve ser representado através de alguma notação formal que, alem de permitir operações de armazenamento e recuperação, possibilite a realização de inferências sobre as informações nela representadas. Uma questão filosófica fundamental considera a extensão em que as complexidades do mundo real podem ser reduzidas a um conjunto manejável de relações simbólicas suscetíveis à análise lógica. Certos pesquisadores afirmam que há conceitos demasiado sutis para serem capturados por algum esquema formal de representação **[PEN 83]** como, por exemplo, um rosto, o paladar, o som de um instrumento musical e os padrões táteis e olfativos.

A hipótese de que podemos capturar o conhecimento, as ações e as experiências de uma determinada pessoa em um programa através de alguma representação formal tem sido contestada também pelos *fenomenologistas* (estudiosos dos fundamentos da experiência e da ação), que adotam a premissa de que nossas hipóteses e crenças fundamentais não podem ser tornadas explícitas por meio de qualquer formalismo. Os fenomenologistas rejeitam tanto a visão objetiva do mundo ("*o mundo físico objetivo é a realidade primária*") quanto a visão subjetiva ("*nossos sentimentos e sensações constituem a realidade primária*"), propondo, ao invés disso, que é impossível para qualquer uma delas existir na ausência da outra. Assim, a noção de interpretação fica sujeita à coexistência de um intérprete (o que *interpreta*) e de um objeto (o que *está sendo interpretado*): existência é interpretação e interpretação é existência. Se os fenomenologistas estiverem corretos, não conseguiremos capturar as sutilezas das interpretações requeridas para "*manejar o universo*" se não conseguirmos representar formalmente a natureza interativa de tais interpretações.

Um outro problema, mais concreto, pode ser formulado da seguinte maneira: Vamos supor que um determinado agente inteligente possua uma ampla gama de representações disponíveis. Como pode ele decidir qual representação ou modelo é aplicável a uma determinada situação ? Parece que as pessoas selecionam representações apropriadas para manejar situações do mundo real sem qualquer dificuldade. O problema de escolher a representação adequada para utilizar em uma determinada situação de tempo e espaço surge em diversos contextos, por exemplo, na determinação de quais os aspectos de uma dada configuração são relevantes para a solução do problema considerado.

Uma última questão deve ainda ser considerada: Se um modelo adequado não estiver disponível, como podemos obter sistematicamente uma representação eficiente para lidar com a situação enfrentada ? Pode-se observar que as pessoas geralmente conseguem construir novas representações quando as existentes são inadequadas, entretanto, o modo através do qual isso é obtido ainda permanece obscuro.

Nas seções seguintes analisaremos o problema da RC sob um enfoque mais prático, ou seja, em função das características ideais que um ER deve apresentar, segundo [JAC 86], do ponto de vista da sua *expressividade* (o que o ER permite representar), *potencial heurístico* (qual o grau de *eficiência* do ER na solução dos problemas que lhe são propostos) e *conveniência notacional* (até que ponto a notação empregada pelo ER facilita a sua interpretação).

2.3.2 Expressividade

Espera-se que um ER permita descrever adequadamente todo o universo que se propõe a representar, isto é, que seja capaz de capturar corretamente o significado de cada uma das entidades que compõem o domínio e formular precisamente a semântica de todas as relações que podem ocorrer entre tais

entidades. A essa característica denominamos **expressividade**. O grau de expressividade de um ER depende diretamente do domínio que se tenta descrever, isto é, um mesmo ER pode se revelar altamente expressivo em determinados contextos e pobre em outros. A expectativa de se vir a obter um ER de expressividade geral, ou seja, capaz de efetivamente descrever a totalidade das situações esbarra na complexidade do mundo real, o qual, como sugerem o Teorema da Incompleteza de Gödel, o Teorema da Indecidibilidade de Church, o Teorema da Parada de Turing e o Teorema da Verdade de Tarski, não pode ser representado em um único formalismo [**HOF 79**].

Uma questão filosófica que se apresenta diretamente relacionada com a noção de expressividade é "o que significa *saber* alguma coisa ?". O ponto de vista científico embasado no conceito de operacionalismo afirma que *saber* ou *entender* alguma coisa é ser capaz de predizer o seu comportamento. Usualmente expressamos nosso conhecimento através de modelos mecânicos ou simbólicos, relacionando o comportamento do modelo com a situação que desejamos entender. Assim, algumas entidades muito complexas (por exemplo, o universo) não poderiam ser completamente representadas em nenhum sistema menos complexo do que elas próprias. Por outro lado, a forma específica da representação adotada pode influir decisivamente na sua *efetividade* ou capacidade de facilitar a produção de inferências apropriadas, isto é, pequenas alterações na representação de um determinado problema podem se revelar decisivas para conduzi-lo da insolubilidade ao trivial [FIS 87].

Isso significa que na medida em que os ERs vão ficando mais ricos em termos de expressividade, mais e mais vão se tornando dependentes da escolha apropriada das construções empregadas na representação dos objetos do domínio. Neste ponto podemos, por exemplo, traçar um paralelo com as diferentes formas de se declarar um mesmo objeto empregando a lógica de predicados de primeira ordem, cujo elevado grau de expressividade permite descrever a mesma entidade de várias maneiras diferentes, todas logicamente equivalentes, porem empregando abordagens cujo grau

de efetividade (a capacidade intrínseca de produzir as inferências apropriadas) pode variar consideravelmente.

Uma vez que a informação que o mundo real nos transmite se manifesta fundamentalmente através das percepções que dele conseguimos extrair, podemos caracterizar como intimamente ligada ao conceito de expressividade (aqui entendida como capacidade de representação), a habilidade que os ERs devem possuir de elaborar suas próprias construções e de dialogar com os subsistemas perceptivos externos (ver a definição de ciclo cognitivo na seção 2.1). Isso significa que os ERs devem possuir uma ou mais *linguagens de representação* (LRs), atuando entre a base de conhecimento subjacente e os "órgãos" periféricos, capazes de traduzir as informações recebidas do mundo exterior para o formato de armazenagem e/ou aplicação e destes para algum outro que permita a transmissão do conhecimento produzido internamente ao ambiente do sistema. Um ER apropriadamente expressivo deve ainda permitir a organização do conhecimento que armazena de modo a facilitar o acesso à sua estrutura e evitar a propagação de redundâncias.

Para os propósitos do presente trabalho, podemos então considerar que a expressividade de um dado ER é determinada por um conjunto de características ou capacidades particulares que esse ER possui para bem representar um certo domínio. Denominaremos tais capacidades de *fatores de expressividade*, que podem ser considerados como sendo:

- A capacidade de representar adequadamente todos os objetos do domínio,
- A capacidade de representar adequadamente as relações existentes entre esses objetos,
- A capacidade de facilitar a produção das inferências apropriadas (efetividade),
- A capacidade de comunicação, e
- A capacidade de auto-organização.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

É importante reforçar que tais objetivos podem, em determinadas ocasiões, se encontrar em conflito. Por exemplo, ERs de elevada capacidade de representar objetos e relações, como é o caso da lógica de predicados de primeira ordem, podem apresentar dificuldades para o controle das inferências que permitem produzir. Por outro lado, ERs de alta efetividade, como as redes semânticas, muitas vezes possuem capacidade de representação limitada.

2.3.3 Potencial Heurístico

O **potencial heurístico** de um ER está vinculado ao grau de eficiência que ele apresenta na solução da classe de problemas que se propõe a resolver, isto é, à sua capacidade de obter respostas corretas de maneira econômica e no menor tempo possível. Essa característica está relacionada com os esquemas de inferências que ele permite aplicar e é fortemente influenciado pela efetividade da representação adotada.

Num sentido mais geral, podemos dizer que o potencial heurístico de um ER corresponde à sua capacidade de *raciocínio*, ou seja, à sua capacidade de solucionar problemas. Entretanto, simplesmente porque um certo dispositivo consegue solucionar determinados problemas, isso não quer dizer que o mesmo tenha capacidade de raciocínio. Por exemplo, uma calculadora pode "solucionar" diversos problemas matemáticos, mas isso não significa que a mesma seja dotada de raciocínio. Alguns critérios são, por conseguinte, necessários para distinguir entre raciocínio e mero comportamento "mecânico". Em primeiro lugar, espera-se que um ER capaz de raciocinar seja capaz de expressar e solucionar diversas classes de problemas, incluindo formulações que não correspondam aos padrões antecipados por seus projetistas (a calculadora, por exemplo, falha nesse teste). Assim o primeiro requisito para garantir a presença de raciocínio em um determinado ER implica em que o mesmo deve ser baseado em um conjunto "efetivo" de representações (ver seção 2.3.2).

Em segundo lugar, o ER deve ser capaz de tornar explícitas as informações que possui armazenadas implicitamente, isto é, deve ser capaz de obter sistematicamente todas as representações equivalentes à informação que possui. (Isso deve ser entendido em um sentido conceitual, uma vez que na prática a quantidade de tempo necessário para tal pode ser demasiadamente grande.) Por exemplo, a partir da informação (1) todos os tigres são perigosos e (2) Rajá é um tigre, o ER deve ser capaz de obter a declaração explícita (3) Rajá é perigoso.

Note que a conclusão obtida (3) está implícita em (1) e (2). Assim, necessitamos de um conjunto de operações ou transformações capazes de produzir outras informações "válidas" quando aplicadas às explicitamente representadas. O ER deve ser capaz de traduzir uma situação expressa através de alguma representação externa para a sua propria representação (interna), assim como ser capaz de transformar "sintaticamente" a informação representada internamente.

Associado à capacidade de raciocínio e de explicitar o conhecimento que representa de maneira implícita, um ER com apropriado potencial heurístico deve preencher o requisito de apresentar um certo grau de eficiência computacional, isto é, executar as operações e transformações anteriormente descritas de maneira econômica e num espaço de tempo que se configure viável.

Há diversas formas de raciocínio passíveis de utilização por um determinado ER na solução dos problemas que lhe são apresentados, as quais agruparemos em três categorias principais: raciocínio *dedutivo*, raciocínio *indutivo* e raciocínio *por analogia*. Devemos registrar, entretanto, que algumas técnicas podem combinar mais de uma delas.

No **raciocínio dedutivo** o ER tenta encontrar uma "cadeia dedutiva" ou assertivas válidas conduzindo das declarações que possui, assumidas verdadeiras, a alguma assertiva dada cuja validade se deseja estabelecer. A eficácia da abordagem dedutiva surge da capacidade que suas regras de inferência possuem de

obter novas declarações verdadeiras, a partir das existentes, que se caracterizam por sua necessidade lógica: a conclusão deve ser consequência lógica das premissas, isto é, deve estar nelas implícita.

A dedução só tem sentido no contexto de um sistema formal onde os símbolos possam ser combinados e transformados obedecendo determinadas regras de inferência dedutiva. A essência de um sistema dedutivo é a manutenção da validade da BC subjacente, isto é, de sua *consistência*: uma declaracão e a sua contradição não podem ser ambas derivadas de uma mesma BC, o que garante a validade dos resultados obtidos. O problema dos sistemas dedutivos é a sua inadequabilidade para expressar determinadas categorias de conhecimento. Por exemplo, não possuem um modo prático para lidar com probabilidades ou com declarações quantitativas que necessitem computação numérica. Sistemas matemáticos não dispõem de recursos práticos para lidar diretamente com declarações conflitantes ou qualitativas, como por exemplo: "José se parece um pouco com Carlos". Os sistemas probabilísticos, na medida em que podem ser considerados dedutivos, não possuem maneiras práticas para expressar relações e nenhum recurso efetivo para lidar com declarações que sejam estritamente verdadeiras ou falsas.

No **raciocínio indutivo** o objetivo é encontrar alguma generalização ou abstração que descreva ou caracterize um certo conjunto de dados. A maior diferença entre dedução e indução é que nesta última temos um conjunto de restrições para satisfazer, ao invés de uma determinada declaração explícita (dada), cuja validade desejamos estabelecer. Além disso os problemas indutivos não necessitam ser formulados com a precisão exigida nos problemas dedutivos.

É típico dos problemas indutivos a existência de mais de uma resposta aceitável para um mesmo problema. Assim, estes frequentemente requerem extrapolação e geralmente não possuem uma forma definitiva de verificar a correção das respostas obtidas. As premissas *suportam* as declarações derivadas mas estas podem

não ser conclusões lógicas do conhecimento armazenado. As regras de inferência indutiva não possuem meios para derivar novas declarações a partir das já existentes [FIS 87].

Uma outra importante distinção entre o raciocínio indutivo e o dedutivo é a quantidade de "evidência" que pode ser envolvida na derivação de uma nova assertiva ou na verificação de alguma hipótese. Devido à necessidade de assegurar a consistência da BC, os sistemas dedutivos geralmente utilizam longas cadeias de raciocínio constituídas de passos muito pequenos, em cada um dos quais somente um subconjunto muito reduzido do total de fatos conhecidos pelo sistema é explicitamente empregado. A estratégia dos sistemas dedutivos é construída sobre transformações sintáticas "locais", uma vez que eles não podem empregar perspectivas "globais" na solução de problemas. Por outro lado, devido à possibilidade de utilizar informações inconsistentes, os sistemas indutivos operam sobre pequenas cadeias de raciocínio constituídas de grandes passos. Sua estratégia consiste na tentativa de empregar explicitamente todas as informações disponíveis a cada passo, uma vez que dependem de "consenso" para assegurar a correção de suas conclusões.

Finalmente, no **raciocínio por analogia**, estabelecemos uma correspondência entre os elementos e as operações de dois sistemas distintos. Tipicamente, um dos sistemas é suficientemente compreendido, enquanto que o outro é aquele que estamos tentando descrever. As descrições são obtidas através de analogias estabelecidas entre este e o sistema já dominado. O maior problema do raciocínio por analogia é encontrar a correspondência entre os dois sistemas, isto é, determinar que tipo de relacionamento é relevante para estabelecer alguma analogia entre eles. O raciocínio de senso-comum combina técnicas analógicas e indutivas na solução dos problemas rotineiros do dia-a-dia, relacionados com o comportamento dos objetos físicos que compõem o mundo real. O raciocínio por analogiadesempenha também um papel muito importante nos sistemas de aprendizado.

2.3.4 Conveniência Notacional

O último requisito de um ER é a sua **conveniência notacional**, isto é, a facilidade que apresenta para a leitura e escrita, permitindo a compreensão de maneira direta e inequívoca. Em [GEO 85] encontramos a ponderação de que a conveniência notacional, sendo destinada a facilitar a interação humana no decorrer de todas as etapas de implementação dos SCs, pode ocasionar uma simplificação exagerada nas técnicas de representação de conhecimento, cujas consequências mais imediatas podem ser encontradas na falta de flexibilidade verificada nas fases de operação e manutenção de tais sistemas. Em outras palavras, a priorização da conveniência notacional poderia conduzir a restrições na expressividade, o que acabaria por se manifestar na utilizacão dos SCs que adotassem tais sistemas. Segundo [GEO 85], com o desenvolvimento dos sistemas de aquisição de conhecimento a motivação para a adoção de métodos simplificados para a representação de conhecimento torna-se obsoleta. A linguagem de representação deixa de ser a linguagem da engenharia de conhecimento, passando a constituir uma linguagem intermediária entre o sistema de aquisição e o de memorização:

> *"Há um novo desvio no paradigma, de métodos de representação de conhecimento muito simplificados para sistemas estruturados, adequados à real complexidade dos domínios de aplicação."*

Para os propósitos do presente trabalho iremos considerar como conveniência notacional apenas a capacidade de um ER de representar conhecimento de forma inequívoca, isto é, evitando qualquer ambiguidade. Para assegurar tal requisito, a notação adotada por um determinado ER para a representação das entidades do seu universo deve ser tal que não permita qualquer forma de confusão entre duas construções semelhantes, ao contrário, permitindo a identificação e o tratamento de tais semelhanças, sempre que isso for de alguma valia para o processo de raciocínio que estiver sendo empregado. Realmente, no

contexto que pretendemos explorar, qualquer esforço no sentido de tornar a linguagem de representação adequada à interação humana parece ser supérfluo, quando não contraproducente, na medida em que isso possa originar perdas no grau de expressividade do sistema ou no potencial heurístico que se deseja obter.

2.4 ESQUEMAS DE REPRESENTAÇÃO DE CONHECIMENTO

Na presente seção apresentaremos as principais características dos paradigmas de representação de conhecimento mais utilizados atualmente. Os modelos analisados são os baseados em regras (*sistemas de produção*), os baseados em objetos estruturados (*redes semânticas* e *frames*) e as *linguagens lógicas*. Também discutiremos um modelo intermediário - a linguagem *Kripton* - que combina o uso de frames e lógica. Não temos a pretensão de esgotar o assunto, uma vez que a área de pesquisa da representação de conhecimento não se reduz apenas aos enfoques aqui analisados. Assim como podemos encontrar modelos intermediários, há também alguns que não se encaixam de nenhum modo nos formalismos que estudaremos, como é o caso da representação analógica, das representações implantadas diretamente em hardware e das redes neurais.

2.4.1 Sistemas de Produções

Nesta abordagem o conhecimento é expresso por meio de regras do tipo: "**SE** condição **ENTÃO** ação". Uma regra compõe-se de *premissas* (a condição), que usualmente correspondem à descrição de algum estado, o qual, uma vez satisfeito, determina a execução de determinadas ações que por sua vez modificam o estado corrente. Dito de outra forma, uma regra pode ser vista como constituída por *condições* e *conclusões* que podem ser extraídas se

as condições forem satisfeitas. O formato básico de uma regra em um sistema de produções é o seguinte:

SE p1 e p2 e ... e pn ENTÃO q1 e q2 e ... e qm

onde tanto as condições como as conclusões podem ser pares atributo-valor (por exemplo: idade = 30, onde *idade* é o atributo e 30 é o valor), ou triplas objeto-atributo-valor (por exemplo: idade de joão = 30, onde *joão* é o objeto).

A representação baseada em regras possui ampla expressividade, sendo bastante adequada para representar associações empíricas. Possui sintaxe e semântica muito simplificadas, sendo bastante fácil de entender e aplicar. Apresenta características modulares e permite rápida prototipagem e programação experimental. Por outro lado, tem também importantes limitações. A necessidade de considerar as consequências da estratégia de resolução de conflitos utilizada dificulta a construção, compreensão e depuração do sistema. Também não facilita distinções semânticas como as existentes entre propriedades essenciais e propriedades acidentais. O conhecimento resulta expresso como um conjunto desordenado e não-estruturado de regras, não sendo possível a representação das estruturas inerentes ao domínio da aplicação (taxonomias, relações de causa e efeito, parte-todo e entre objetos e classes de objetos).

Os principais componentes de uma arquitetura baseada em regras são três: uma **memória de trabalho**, um conjunto de **regras** e uma **máquina de inferências**. A memória de trabalho é uma base de dados *global*, de símbolos representando fatos e assertivas acerca do domínio do problema. Os dados são instâncias de objetos que podem representar objetos físicos, fatos relacionados ao domínio de aplicação ou objetos conceituais relacionados com a estratégia de solução do problema. Pode-se dizer que, em um dado momento, o conteúdo da memória de trabalho está indicando o *estado de solução* do problema. Em geral os objetos da memória de trabalho são da forma (*objeto atributo valor*), como por exemplo (José

idade 28), e são utilizados para conduzir a execução das regras. A memória de trabalho contem elementos que podem ser desde simples sequências de caracteres, até objetos de estrutura completa. A forma de representar estes objetos pode variar de linguagem para linguagem, podendo ser na forma de listas, tuplas, registros, etc.

As regras constituem o *programa* ou *memória de regras*. Cada uma delas tem uma parte "se..." ou condição, que descreve a configuração de dados necessária para que a regra seja executada, e uma parte "então..." ou ação, que indica as modificações a serem realizadas na memória de trabalho. As regras não se referenciam entre si, e possuem nomes apenas para efeito de documentação. A comunicação entre elas se dá através dos dados. Por exemplo, se quisermos simular uma execução sequencial, cada regra deve colocar na memória de trabalho algum dado que somente será requisitado pela regra seguinte, de maneira que esta seja a única em condições de ser executada.

O lado esquerdo da regra (parte da condição), também denominado *antecedente*, é habitualmente uma combinação booleana de predicados. Os operadores variam, ainda que geralmente sejam apenas o **and** e o **not** e os predicados restringem os valores possíveis de um ou vários atributos de algum objeto que possa estar na memória. O lado direito da regra (parte da ação), também denominado *consequente*, é geralmente uma lista de modificações a serem feitas sobre os objetos na memória de trabalho, juntamente com eventuais chamadas a rotinas externas de entrada e saída ou de algum outro tipo.

O objetivo da máquina de inferências é a execução das regras. Ela determina quais as regras são relevantes para uma dada configuração da memória de trabalho e, dentre elas, escolhe uma para ser aplicada. Essa seleção faz parte da estratégia de controle e se denomina "*resolução de conflito*". A máquina de inferências posui um ciclo que pode ser dividido em três etapas: verificar regras, selecionar regras (resolução de conflito) e executar regras. A primeira etapa consiste em encontrar todas as

regras que são satisfeitas pelo conteúdo da memória de trabalho, de acordo com comparações efetuadas por algoritmos particulares da máquina de inferências. Todas elas são passíveis de ser executadas e formam, juntamente com os elementos da memória de trabalho que as satisfazem, o que se chama "*conjunto de conflito*".

A segunda etapa do ciclo da máquina de inferências consiste em selecionar dentre as regras contidas no conjunto de conflito, aquela que deve ser efetivamente aplicada, ou seja: *resolver o conflito*. O processo de resolução de conflito utiliza alguma estratégia de seleção para determinar um *elemento* do conjunto de conflito (composto por uma regra, juntamente com os dados da memória de trabalho que a satisfazem). A estratégia de resolução de conflitos constitui um elemento importante na definição do sistema. Duas características nela desejáveis são: (1) a *sensibilidade* às modificações realizadas sobre ns dados da memória de trabalho e (2) alguma *estabilidade*, entendida como um certo grau de continuidade em relação à linha de raciocínio. As estratégias existentes são muitas, há entretanto três critérios gerais (aplicados, por exemplo pela linguagem OPS5, que é a mais popular dentre as baseados em regras):

(1) Não permitir que uma regra se aplique mais do que uma única vez sobre os mesmos dados. Em outras palavras, um par (regra; dados que a satisfazem), após executado, não deve ser reconsiderado para execução nos ciclos seguintes. O objetivo é evitar laços infinitos.

(2) Priorizar os dados mais recentes. Uma vez que os elementos da memória de trabalho estão referenciados conforme sua ordem de inserção, este critério dá prioridade aos pares em que a regra se aplica a dados mais recentes, e

(3) Priorizar a *especificidade* da regra. Os elementos do conjunto de conflito que participam das regras mais específicas, isto é, os que possuem o maior número de condições e são, por conseguinte, mais difíceis de satisfazer, são preferíveis aos mais genéricos. Dessa maneira pretende-se que a seleção seja mais sensível.

Na etapa de execução, a regra selecionada é então acionada sobre o correspondente elemento da memória de trabalho. Esse ciclo é também denominado *"reconhecimento e ação"*. Quando é reiniciado, a memória de trabalho foi modificada pela regra executada, e assim um novo conjunto de conflito será gerado. Na **Figura 2.2** apresentamos de forma esquemática as idéias comentadas acima.

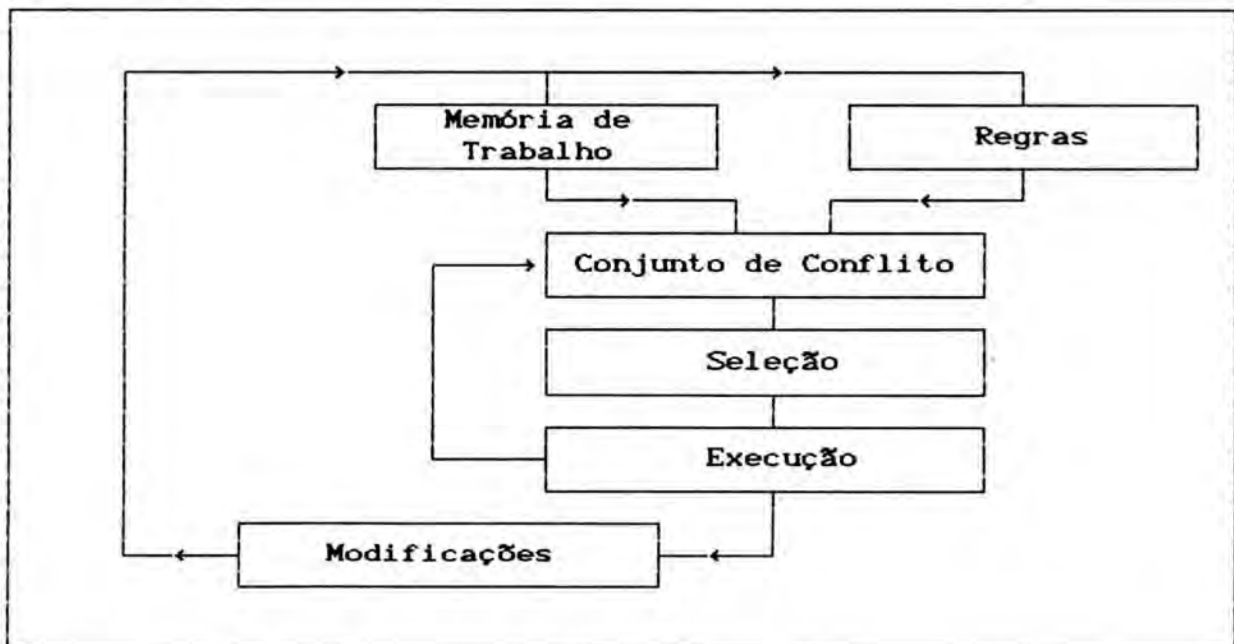


Figura 2.2
Funcionamento de um ER baseado em regras

2.4.2 Redes Semânticas

Nas redes semânticas, um dos dois modelos que estudaremos dentre os baseados em objetos estruturados, o conhecimento se expressa por meio de um grafo rotulado e direcionado, cujos nodos representam objetos, conceitos ou situações e os arcos definem os relacionamentos existentes entre eles. Um exemplo, adaptado de [MAT 89] é mostrado na **Figura 2.3**. O grafo ali mostrado pretende representar *"o carro branco, zero km, estacionado na Av. Osvaldo Aranha, que pertence a uma pessoa de 28 anos chamada Maria"*.

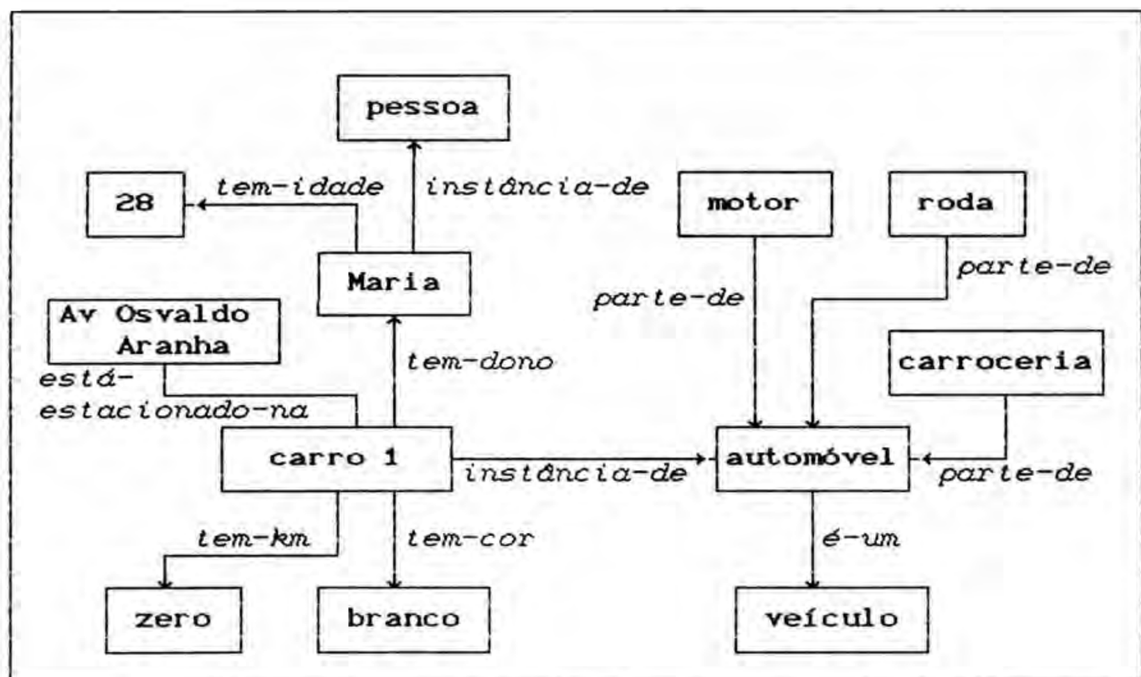


Figura 2.3
Exemplo de uma Rede Semântica

Observando a estrutura, a primeira vantagem importante das representações em rede deve tornar-se clara: Toda a informação acerca de um nodo está integrada ao seu redor e é diretamente acessável a partir dele. Basicamente os nodos podem ser de dois tipos: *"individuais"* e *"genéricos"*. Os primeiros

representam descrições ou afirmações a respeito de uma instância individual de um objeto, enquanto que os segundos, a respeito de uma classe ou categoria de objetos. A contribuição mais importante dos esquemas de redes semânticas é dada pelos eixos organizacionais, que oferecem um grande poder de estruturação à BC subjacente. A função de alguns desses eixos é apresentada a seguir:

- **Classificação (membro-de, instância-de):** Eixos desse tipo relacionam um objeto (por exemplo: Maria e carro 1) com o seu tipo de objeto genérico (por exemplo: pessoa e automóvel). A inclusão de tais eixos em um ER força uma distinção entre objetos e conceitos ou tipos de objetos.

- **Generalização (é-um):** Os eixos *é-um* relacionam um tipo de objeto (por exemplo: automóvel) com outros mais genéricos (por exemplo: veículo). O uso desse tipo de eixo ocorre principalmente para minimizar os requisitos de armazenamento, permitindo que as propriedades de tipos genéricos sejam herdadas por tipos mais especializados.

- **Agregação (parte-de):** Esse tipo de eixo é usado para relacionar um objeto aos seus componentes. Por exemplo: as partes de um automóvel são motor, carroceria e rodas.

O maior problema dos ERs baseados em redes tem sido, até o presente, a falta de uma semântica formal e uma terminologia padronizada. Esse problema existe mesmo nas mais importantes construções a partir deste esquema (ver, por exemplo, os muitos significados existentes para o relacionamento É-UM [**BRA 83**]). Isso se deve, pelo menos em parte, ao fato de que as redes semânticas tem sido usadas para a representação de conhecimento a partir de fontes muito diferentes, por exemplo: na representação de fórmulas lógicas, na organização do conhecimento, para expressar o significado de sentenças em linguagem natural e na

representação de expressões linguísticas. Alem disso, nem todos os ERs baseados em redes associam os diferentes significados dos eixos organizacionais a diferentes arcos no grafo da representação. Isso certamente conduz a mais problemas com a semântica de tais arcos. Outra desvantagem nesta abordagem devida à falta de uma semântica formal é a dificuldade em se verificar a correção do processo de inferência.

As redes semânticas parecem ser mais populares em outros campos (por exemplo, no processamento da linguagem natural) do que em representação de conhecimento. Apesar disso, existem sistemas especialistas que empregam formalismos baseados em rede, entre os quais alguns de grande porte, como o INTERNIST e o PROSPECTOR [WAT 86].

2.4.3 Sistemas de Frames

Frames são uma forma de agrupar informações em termos de registros de *slots* e *fillers* [MIN 75]. Cada registro pode ser visualisado como um nodo complexo em uma rede com um slot especial preenchido com o nome do objeto que o nodo representa e outros slots preenchidos com os valores de diversos atributos comuns associados com tal objeto. O uso de frames é considerado uma forma adequada para a representação de até mesmo aspectos mais sutis do mundo real, como expectativas e suposições.

Minski descreve os frames como sendo "*estruturas de dados para a representação de situações estereotípicas*" às quais se ligam diversos tipos de informação, incluindo a forma com que o frame deve ser utilizado. A intuição implicita na idéia dos frames é que a codificação conceitual realizada pelo cérebro humano está menos relacionada com definições exatas e exaustivas das propriedades que uma determinada entidade deve possuir para

ser considerada como pertencente a uma certa categoria e mais relacionada com "*propriedades marcantes*", associadas a objetos que são, de alguma forma, típicos de sua classe. Tais objetos denominam-se *objetos prototípicos* ou *protótipos*. Assim, um pássaro prototípico, como o pardal, pode voar, e desta maneira pensamos em "*voar*" como sendo uma propriedade típica dos pássaros, mesmo existindo alguns, como o avestruz, que não o fazem. Em outras palavras, um pardal é um melhor representante da categoria dos pássaros do que o avestruz, porque possui uma propriedade que é comum à grande maioria dos membros da classe dos pássaros.

Sistemas de frames tentam raciocinar acerca de classes de objetos utilizando representações prototípicas de conhecimento que valem para a maioria dos casos, mas que devem ser de alguma forma modificados para capturar a complexidade do mundo real, onde as exceções são abundantes e frequentemente há limites difusos entre as classes. A idéia fundamental é que as propriedades, nos níveis mais altos dos sistemas de frames, são fixas, uma vez que representam informações que são típicamente verdadeiras acerca dos objetos ou situações de interesse. Os níveis mais baixos possuem slots que devem ser preenchidos com dados reais, possuindo ainda outras especificações características da instância que representam. Consideremos o exemplo de frame mostrado na **Figura 2.4:** O frame representando **PESSOA** pode ser considerado como um tipo de declaração, na medida em que diz quais os tipos de atributos que uma pessoa tem, tais como **IDADE, ALTURA** e **PESO**, e então aplica certas restrições sobre os valores que estes atributos podem tomar. Por exemplo, a idade de uma pessoa em anos deve ser um número entre 0 e 120. Esperamos que seu peso seja até de 250 kg e a altura máxima da classe, de 240 cm. Note-se que os atributos são por sua vez representados por frames e podem possuir propriedades específicas.

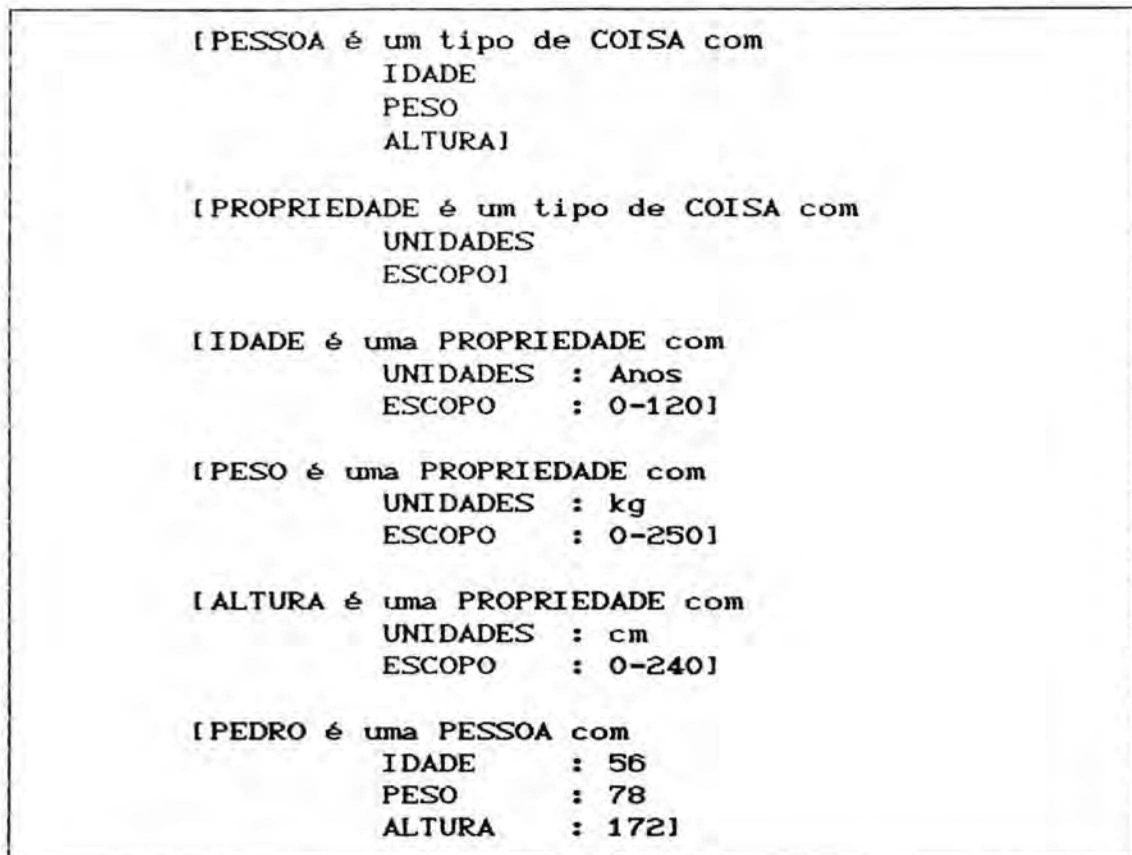


Figura 2.4

Um sistema de frames simplificado

A relação "*um tipo de*" (a kind of), algumas vezes abreviada para *utd* (ako), relaciona subclasses com suas superclasses e é análoga à relação "*subconjunto de*" da teoria dos conjuntos. A relação "*é um*" (is a) relaciona objetos às suas classes, sendo portanto similar à relação "*membro de*" da mesma teoria. Membros de uma determinada classe são normalmente denominados "*instâncias*", enquanto que as subclasses são chamadas "*tipos*". O modelo dos frames se propõe a formular uma teoria global e coerente para a representação do raciocínio de senso comum. Não é possível, com o uso de frames, a representação de qualquer conceito que não possa ser representado por meio da lógica de primeira ordem [NIL 80], mas a integração de toda a informação sobre uma entidade do mundo real em um frame, com o suporte de alguns aspectos operacionais, torna esse ER um dos mais poderosos.

Mesmo assim, parte dos problemas verificados nas redes semânticas pode ser repassado ao estudo dos frames, os quais são uma tentativa de resolver as insuficiências daquelas, adicionando estruturas de dados, incorporando uma semântica melhor definida dos objetos como indivíduos típicos de sua classe e aprimorando os aspectos de implementação para melhor explorar as estruturas inerentes aos dados. Subsiste a indefinição acerca do significado do objeto estruturado em sua relação com o domínio que pretende representar. O embasamento epistemológico dos frames continua portanto um tema bastante polêmico.

Segundo [JAC 86], os frames conduzem a abordagens relacionadas com a adequação de modelos, procurando a melhor unificação entre os dados e alguma hipótese. Assim como os sistemas de produções, os sistemas de frames não utilizam habitualmente técnicas de backtracking, entretanto os registros podem conter informação suficiente para que o interpretador seja capaz de tomar decisões relativamente precisas sobre o que deve ser feito em seguida. Alguns sistemas permitem ainda um tipo de refinamento iterativo, isto é, hipóteses alternativas são processadas em sucessivos estágios, usando diferentes estratégias até que uma delas possa vir a ser selecionada.

Os frames oferecem mecanismos para o tratamento de exceções e defaults, o que, paradoxalmente, os tornam alvo de críticas, porque o cancelamento e a alteração de propriedades herdadas torna difícil definir precisamente algum objeto ou conceito em função de outros. Um outro ponto que tem sido criticado nos sistemas de frames é a sua aparente separação da lógica e a sua flexibilidade em termos de contextualização e controle localizado, que pode tornar o seu comportamento difícil de prever e de entender.

2.4.4 Linguagens Lógicas

Definir lógica não é uma tarefa fácil, mesmo se considerarmos apenas os sistemas formais desenvolvidos com base na lógica matemática. Tais sistemas foram originalmente criados e estudados com o objetivo de caracterizar precisamente uma linguagem simbólica na qual todas as proposições matemáticas pudessem ser expressas. Mais precisamente, o que se desejava era uma linguagem na qual fosse possível representar um conjunto de verdades básicas, ou axiomas, a partir do qual todo o resto pudesse ser gerado pela aplicação de um conjunto finito de regras de prova precisamente definidas e que garantissem a preservação da coerência original. Segundo [ISR 83], as linguagens da lógica matemática não foram concebidas para uso geral. Seus projetistas não declaram que elas são simbolismos universais para aplicações irrestritas, isto é, que qualquer coisa imaginável possa ser adequadamente representada por seu intermédio, entretanto, o fato de que estes formalismos não foram projetados com o objetivo de expressar conhecimento de senso comum (isto é, de solucionar o problema da IA), não significa que eles não possam ser utilizados para tal.

A idéia de que a lógica poderia ser utilizada como uma linguagem de programação foi colocada em uso prático por volta de 1972, com o surgimento da linguagem Prolog. A lógica tradicionalmente oferece uma sólida base conceitual para a representação de conhecimento, uma vez que pode trabalhar formalmente com a noção de consequência lógica. A introdução do Prolog tornou possível não apenas a representação do conhecimento em termos de lógica, como também a obtenção automática de inferências apropriadas sobre o conhecimento representado.

Podemos expressar conhecimento em Prolog por meio de *fatos* e *regras*. As unidades básicas para a construçãm de fatos e regras são os *predicados*, isto é, expressões que declaram coisas simples acerca das entidades do universo de discurso. Por

exemplo, a peça de informação "*João gosta de Maria*" é representada por:

gosta(joão,maria).

Classes de indivíduos que seriam expressas na linguagem natural por meio de expressões como "*todos*", "*alguém*", "*ninguém*", etc, devem fazer uso de variáveis. Por exemplo, a declaração "*Toda mãe gosta de seus filhos*" pode ser representada por:

gosta(mãe(X),X).

Os predicados são representados por um nome de predicado (por exemplo "*gosta*") seguido por uma lista de argumentos. Cada argumento pode ser (1) o nome de um indivíduo, também denominado uma *constante* (por exemplo: "*joão*", "*maria*"), (2) uma variável (por exemplo "X"), ou (3) uma expressão funcional (por exemplo "*mãe(X)*"). Expressões funcionais podem ser pensadas como nomes para aqueles indivíduos que estão em relação de dependência funcional com seus argumentos. Representamos um fato em Prolog, simplesmente escrevendo um predicado seguido por um ponto. Prolog interpreta um fato como uma assertiva do que o predicado vale no mundo que estamos tentando descrever.

As regras, por outro lado, possuem a seguinte forma geral:

P1 se (P2 e P3 e ... e Pn).

onde os Pi são predicados e os parênteses são geralmente omitidos. Por exemplo, a regra geral que estabelece que "*Maria gosta de todos que gostam dela*" é dada por:

gosta(maria,Y) se gosta(Y,maria).

Prolog interpreta uma regra como a declaração de que se todas as condições P2, P3, ... ,Pn são satisfeitas, então a

conclusão P1 também o é. Se a regra contem alguma variável, então ela é aplicada a todos os possíveis valores dessa variável. Assim, em um mundo no qual os únicos indivíduos são João e Maria, a regra acima é uma generalização de:

gosta(maria,maria) se gosta(maria,maria).

e

gosta(maria,joão) se gosta(joão,maria).

Tais regras são denominadas *instâncias* da regra mais geral. (Observe que a primeira delas, se aplicada na tentativa de provar que Maria gosta de si própria, pode conduzir a um *loop* infinito).

Os indivíduos que constituem o mundo são aqueles aos quais nos referimos em nossos fatos e regras, seja diretamente através de seus identificadores (por exemplo: "*joão*"), ou indiretamente através de expressões funcionais (por exemplo, *mãe(X)* pode denotar *mãe(joão)*, *mãe(maria)*, *mãe(mãe(joão))*, etc., em um mundo em que as únicas constantes são João e Maria). Os fatos que contem variáveis são também tidos como que aplicados a todas as suas instâncias.

O conhecimento relacionado a um certo mundo ou domínio pode ser representado em Prolog de diversas maneiras diferentes. Por exemplo, a peça de conhecimento "*Toda mãe gosta de seus filhos*" pode também ser representada por:

gosta(X,Y) se mãe(X,Y)

(isto é, se X é a mãe de Y, então X gosta de Y). Aqui, entretanto, estamos usando um predicado binário ao invés de uma expressão funcional para representar a relação "*mãe*" entre dois indivíduos. Qualquer que seja a representação escolhida, esta deve resultar consistente ao longo da completa descrição e possibilidades de consulta ao conhecimento representado. Todo indivíduo representado deve possuir um nome único.

Podemos representar mais informação por meio de predicados do que é possível através de funções. Por exemplo, se "*tio(X,Y)*" significa que X é tio de Y, podemos facilmente descrever um mundo onde uma pessoa tem dois tios:

tio(josé,maria).

e

tio(júlio,maria).

Se, entretanto, a única maneira de nos referirmos ao tio de alguém for por meio de uma expressão funcional da forma *tio(X)*, então somente podemos nomear um tio para cada indivíduo, uma vez que *tio(maria)* denota um único indivíduo do universo.

Bases de conhecimento constituem uma área de aplicação particularmente interessante para a programação em lógica, se considerarmos as suas seguintes características:

- Fatos e regras podem coexistir na descrição de qualquer relacionamento,
- Definições recursivas são permitidas,
- Múltiplas respostas para a mesma consulta são possíveis (não-determinismo),
- Não há distinção entre argumentos de entrada e saída em um mesmo predicado, e
- As inferências ocorrem automaticamente.

Tais características possuem implicações importantes para aplicações em bases de conhecimento: Como fatos e regras podem ser utilizados conjuntamente, nenhum componente dedutivo adicional precisa ser utilizado, ao contrário do que acontece com bases de dados convencionais. Além disso, como regras recursivas e não-determinismo são permitidos, o usuário pode obter descrições muito claras, concisas e não-redundantes da informação

que deseja representar. Como não há distinção entre argumentos de entrada e de saída, qualquer argumento ou combinação de argumentos pode ser utilizada na recuperação de informações, enquanto que bases de dados convencionais necessitam nomear e controlar diferentes linhas de recuperação para tornar isso possível.

Finalmente, o fato de que as respostas são extraídas automaticamente da base de conhecimento, por meio da utilização de um mecanismo de inferência transparente, ao usuário resulta em elevado grau de independência de dados. O usuário pode não apenas representar suas informações em um nível muito elevado, como também não necessita descrever as operações usadas na sua recuperação. Estas operações estão implícitas no mecanismo de inferência do Prolog, que é capaz de atribuir um significado operacional às regras e fatos puramente descritivos utilizados.

Um problema operacional na utilização da lógica como formalismo para a representação de conhecimento é a dificuldade em controlar o crescimento potencial do número de inferências, o que tem limitado sua utilização, principalmente em aplicações de grande porte. Mesmo restrições muito mais manejáveis da lógica de primeira ordem, como é o caso das cláusulas de Horn, que constituem o paradigma fundamental do Prolog, possuem essa limitação, aliada à incapacidade de expressar certos aspectos do raciocínio de senso comum, como o da não-monotonicidade, além da indecidibilidade inerente à lógica de primeira ordem. Segundo [CAR 88]:

> *"... está aberto o caminho para o projeto de aplicações que, seja por requerem somente um subconjunto manejável da lógica de primeira ordem, seja por permitirem o controle das inferências mediante metaconhecimento incorporado ao próprio formalismo, consigam aproveitar as potencialidades da lógica, buscando evitar sua complexidade computacional."*

No primeiro caso, se o domínio de aplicação não exigir, podem ser eliminados os símbolos funcionais não-constantes. Se, além disso, eliminarmos as variáveis, teremos o cálculo proposicional (lógica de ordem zero), que possui a virtude de ser decidível. A segunda variante consiste em orientar as inferências de acordo com o conhecimento contido em contextos particulares. Uma possibilidade interessante é a compilação de várias teorias, cada uma com o seu aparato formal dedutivo, em um único processo de demonstração. A Teoria da Resolução de Stickel **[STI 85]**, propõe que um provador de teoremas por resolução permita a chamada de outros provadores, utilizando outras técnicas, que realizam parte do trabalho, devolvendo os resultados obtidos ao provador principal.

Bowen **[BOW 85]** e Monteiro e Porto **[MON 88]**, estabelecem a necessidade de se dispor de múltiplas *visões* de uma mesma situação, estendendo o conceito formulado por Date **[DAT 83]** para bases de dados convencionais, com a proposta que tais visões deveriam ser passíveis de *estruturação* e *composição*. Resumidamente, tal proposta se viabilizaria através de extensões a nível meta do paradigma da programação em lógica, onde diferentes visões (*teorias* para Bowen ou *contextos* para Monteiro e Porto), passam a ser tratadas como *objetos de primeira classe*, no sentido de poderem ser valores atribuídos à variáveis. Conjuntos finitos de fórmulas (as visões) são representadas por termos e cada item sintático (de variáveis a conjuntos de fórmulas) é nomeado por uma constante da linguagem.

2.4.5 Krypton

Apesar de grande parte da pesquisa corrente em representação de conhecimento encontrar-se repleta de discordâncias, algumas tendências comuns parecem estar surgindo. Em particular, grande parte do esforço tem sido focalizado no desenvolvimento de linguagens baseadas em frames. O grande apelo ao uso de taxonomias de frames parece residir em sua capacidade

em corresponder de modo muito próximo à nossa intuição de como o mundo se estrutura (como é ilustrado em taxonomias sociais, por exemplo). Elas também oferecem formas atraentes de processamento (herança, *defaults*, etc.) e encontram aplicações em outras áreas da ciência da computação, tais como o gerenciamento de bases de dados e programação orientada a objetos.

Enquanto que as idéias básicas sobre sistemas de frames são muito objetivas, no seu projeto e utilização podem surgir complicações. Essas dificuldades surgem tipicamente porque (1) as estruturas são interpretadas de modos diferentes em diferentes tempos (sendo a principal ambiguidade a existente entre as interpretações factuais e definicionais), e (2) a semântica da linguagem de representação é especificada somente em termos das estruturas de dados usadas para implementá-las (tipicamente, redes de herança).

O sistema **Krypton [BRA 83a]** distingue claramente entre informações definicionais e factuais. Em particular, Krypton possui duas linguagens de representação, uma para a formação de termos descritivos e outra para construir declarações sobre o mundo com o uso de tais termos. Além disso, Krypton oferece uma visão *funcional* de uma base de conhecimento, caracterizada em termos do que pode ser perguntado ou dito e não em termos das estruturas particulares que utiliza para a representação de conhecimento.

Um tema explorado em várias linguagens de representação é a taxonomia de descrições estruturadas. Em alguma dessas linguagens, poderíamos ter a seguinte descrição de uma família:

família

É-UM(A) estrutura-social
pai: homem (exatamente 1)
mãe: mulher (exatamente 1)
criança: pessoa

Mesmo não interpretado, esse tipo de estrutura de dados é indubitavelmente útil. Para a representação de conhecimento, entretanto, devemos impor uma significação aos objetos representados, isto é, eles tem que significar *alguma coisa*. Assim, podemos fazer um número imenso de interpretações das ligações e nodos em uma taxonomia de frames, e sistemas de frames típicos deixam muito do trabalho semântico com o usuário. Ainda, a despeito da multiplicidade de possíveis interpretações, duas abordagens ao significados dos frames parecem se destacar. Na primeira, frames são *assertivas* ou declarações a respeito de como são as coisas no mundo. Sob esta interpretação, que foi estudada em alguma profundidade em [HAY 79], a presença do frame "*família*" em um sistema de frames seria entendida como a declaração de que toda família é uma estrutura social com um pai, uma mãe e um certo número de filhos.

Infelizmente, o ponto de vista assercional se apresenta como sendo bastante restritivo, principalmente por duas razões. A primeira delas é que a *instanciação* (preenchimento dos *slots* de um frame), que é a forma básica de declaração no sistema de frames torna expressões de conhecimento incompleto difíceis, quando não impossíveis. A segunda razão é que descrições verdadeiramente compostas não podem ser expressas. Por exemplo, ao invés de sermos capazes de formar uma expressão do tipo "*Uma família sem filhos*", podemos apenas criar o frame família-sem-filhos e declarar que famílias deste tipo não tem filhos, como se isso fosse uma propriedade incidental do tipo: marido e mulher trabalham fora.

A frustração com as limitações apresentadas pela visão assercional dos frames poderia nos conduzir a adotar a outra interpretação predominante: frames como *descrições*, sem importância assercional direta. Linguagens de representação como o KL-ONE e outras, adotam a visão de que os frames e as ligações entre eles constituem a estrutura das descrições. Algum outro mecanismo é necessário para usar as descrições para a declaração de fatos. Sob essa interpretação, o símbolo "*família*" do nosso exemplo seria tomado como uma abreviação para uma descrição do

tipo "*uma estrutura social que se caracteriza por possuir, entre outras coisas, um pai que é um homem, uma mãe que é uma mulher, e um certo número de filhos, todos pessoas.*". Os que preferem essa interpretação para os frames afirmam que ela produz uma linguagem mais limpa, que não tem os problemas apresentados pelos sistemas de frames estritamente assercionais.

O principal objetivo dos criadores de Krypton foi evitar os tipos de problemas gerados pelas abordagens estruturadas mais convencionais:

> "*Focalizamos uma especificação funcional da base de conhecimento, respondendo a qualquer questão do tipo: 'Que estruturas deve o sistema oferecer ao usuário ?' com: 'O quê, exatamente, o usuário espera que o sistema faça ?' Em outras palavras, decidimos o que desejávamos que o sistema fizesse para interagir com uma base de conhecimento sem assumir nada com respeito à sua estrutura interna. Tornando tais operações disponíveis aos usuários da linguagem Krypton, sentimos que poderíamos controlar precisamente a forma com a qual o sistema seria utilizado se tivéssemos a liberdade de implementar as operações de qualquer maneira que fosse conveniente.*" **[BRA 83b]**.

Assumir uma representação funcional, entretanto, adiantaria pouco ou nada se as operações utilizadas fossem precisamente as mesmas convencionalmente adotadas. Em tal caso, o sistema sucumbiria às mesmas confusões existentes entre as visões estruturais e assercionais mencionadas anteriormente. Para evitar esse problema, as operações foram separadas em dois tipos diferentes, que conduzem aos dois principais componentes do sistema Krypton: um *terminológico*, ou **T-Box** e um *asserciona1* ou **A-Box**. O T-Box permite estabelecer taxonomias de termos estruturados e responder questões acerca do relacionamento analítico entre tais termos. O A-Box permite construir teorias descritivas de domínios de interesse e responder questões acerca desses domínios. A separação entre os dois componentes é feita

naturalmente a partir dos dois tipos de expressões usadas para representar conhecimento - *termos (nominais)* e *sentenças*. O T-Box lida com o equivalente formal de frases nominais tais como "*uma pessoa com pelo menos três filhos*" e entende que essa expressão subordina-se a "*uma pessoa com pelo menos um filho*" e é disjunta de "*uma pessoa com no máximo um filho*". O A-Box, por outro lado, opera com o equivalente formal de *sentenças*, tais como "*Toda pessoa com pelo menos três filhos possui um carro*" e entende as implicações (no sentido lógico) de assertivas desse tipo. Além disso, como o usuário não tem meios de especificar após o fato quais sentenças são consequência lógica de outras no A-Box, ele também não tem meios de especificar após o fato onde um termo se encaixa em uma taxonomia no T-Box. Os relacionamentos de subordinação e disjunção estão baseados unicamente na estrutura dos termos no T-Box, e não em qualquer dos fatos (dependentes de domínio), mantidos pelo A-Box.

3 PROGRAMAÇÃO EM LÓGICA

Dentre os diversos formalismos empregados na representação de conhecimento, os esquemas baseados em lógica se encontram entre os mais promissores, não só devido ao elevado grau de expressividade que apresentam como também em função de sua capacidade intrínseca de produção automática de inferências sobre o conhecimento representado. No presente capítulo abordamos a utilização de sistemas de programação em lógica - particularmente a linguagem Prolog - como esquemas de representação de conhecimento, considerando seus aspectos sintáticos e semânticos.

3.1 PROGRAMAÇÃO EM LÓGICA DE PRIMEIRA ORDEM

A programação em lógica originou-se em grande parte na pesquisa sobre prova automática de teoremas, particularmente no desenvolvimento do princípio da resolução, estabelecido por Robinson em **[ROB 65]**. Um dos primeiros trabalhos relacionando o princípio da resolução com a programação de computadores deve-se a Green **[GRE 69]**, que mostrou que o mecanismo de extração de respostas poderia ser usado para sintetizar programas convencionais por meio da aplicação do princípio da resolução a suas especificações expressas na forma clausal. Os sintetizadores projetados para esse propósito podem ser considerados os precursores dos interpretadores lógicos atuais.

O termo "**programação em lógica**" é devido a Kowalski **[KOW 74]** e designa o uso da lógica como linguagem de programação de computadores. O mérito de Kowalski foi o de ter identificado, em um particular procedimento de prova, um procedimento

computacional permitindo uma interpretação "procedimental" da lógica, estabelecendo assim as condições que nos permitem entende-la como uma linguagem de programação de uso geral. Esse foi um avanço essencial, necessário para adaptar os conceitos relacionados com a prova de teoremas às técnicas computacionais já dominadas pelos programadores.

Avanços nas técnicas de implementação também foram de grande importância para o emprego da lógica como linguagem de programacão. O primeiro interpretador experimental foi implementado por Colmerauer, Roussel e outros na Universidade de Aix-Marseille em 1972 com o nome de **Prolog** (um acrônimo para "**Pro**grammation en **Log**ique"). Desde então tem surgido um grande número de implementações Prolog, cobrindo as mais diversas filosofias, máquinas hospedeiras e ambientes de aplicação.

Uma das principais idéias da programação em lógica, devida à Kowalski **[KOW 79]**, é que um algoritmo é constituído por dois elementos disjuntos: a **lógica** e o **controle**. A lógica corresponde à definição do "**que**" deve ser solucionado, enquanto que o controle estabelece "**como**" a solução pode ser obtida. O ideal da programação em lógica é que o programador deva somente especificar o componente lógico de um algoritmo, deixando o controle para ser exercido apenas pelo sistema de programação em lógica subjacente, o que ainda não é totalmente possível nos ambientes atualmente disponíveis.

Para alcançar essa premissa idealizada para os sistemas de programação em lógica, dois problemas principais precisam ser solucionados. O primeiro deles é o **problema do controle**. Por enquanto é tarefa dos programadores estabelecer em seus programas um grande número de informações destinadas ao controle das inferências a ser produzidas, parte através da ordenação das cláusulas e dos objetivos nelas contidos, parte por intermédio de

mecanismos de controle extra-lógicos, como o *cut*. Tais controles tem se mostrado insatisfatórios por diversas razões. A primeira grande tarefa a ser enfrentada no sentido de superar esse problema consiste em oferecer aos programadores mecanismos de controle mais satisfatórios para uso em seus programas. A segunda tarefa seria transferir a responsabilidade sobre o controle dos programadores para o próprio sistema.

O segundo problema é o **problema da negação**. Os atuais sistemas de programação em lógica, baseados no subconjunto da lógica constituído pelas cláusulas de Horn, não implementam a negação, mas sim uma versão imperfeita e por vezes problemática desta, a *negação por falha finita*. A pesquisa nesta área procura um melhor entendimento da negação como regra de falha, de modo a obter algum formalismo mais próximo da negação lógica capaz de ser empregado em substituição.

Os termos *programação em Lógica* e *programação Prolog* tendem a ser usados indiscriminadamente, entretanto, a estratégia adotada pelo Prolog para a produção de inferências é apenas uma das diversas existentes para a execução de programas em lógica. Por exemplo, uma estratégia completamente diferente é o procedimento de prova por "conexão em grafos", proposta em **[KOW 75]**, que opera por meio de um esquema especial de ativação de ligações aplicado às conexões existentes em um grafo, que são vistas como chamadas para a execução de procedimentos. A primeira tentativa de implementação de tal método deve-se provavelmente a Tärnlund **[TAR 75]**.

Além disso, é importante registrar que os sistemas que implementam a linguagem Prolog costumam apresentar significativas diferenças entre si, no que se refere a mecanismos adicionais que oferecem para o enriquecimento de seus recursos. A maioria dos interpretadores permite o emprego de diversas alternativas para a modificacão da estratégia básica de controle, o que às vezes

conduz a computacões que não podem ser completamente justificadas em termos de inferências lógicas. Tais interpretadores são considerados como potencialmente "impuros". Um exemplo é o Prolog original desenvolvido em Marselha. Por outro lado, quando o interpretador sempre se comporta estritamente de acordo com regras inferência lógica (mesmo que não empregue o princípio da resolução), ele é considerado "puro". O IC-Prolog desenvolvido no Imperial College em Londres por Clark e McCabe é considerado um Prolog "puro", apesar de possuir diversos mecanismos sofisticados que suplementam a estratégia original.

3.1.1 Programas em Lógica

Um programa em lógica é constituído por sentenças que expressam o conhecimento relevante para o problema que se pretende solucionar. A formulação de tal conhecimento emprega dois conceitos básicos: a existência de objetos discretos, que denominaremos *indivíduos*, e a existência de *relações* entre eles. Os indivíduos, considerados no contexto de um problema particular, constituem o *domínio* do problema. Por exemplo, se o problema é solucionar uma equação algébrica, então o domínio deve incluir pelo menos os números reais.

Para que possam ser representados por meio de um sistema simbólico tal como a lógica, tanto os indivíduos quanto as relações devem receber *nomes*. A atribuição de nomes é, entretanto, apenas uma tarefa preliminar na criação de modelos simbólicos para a representação de conhecimento. A tarefa principal é a construção de *sentenças* expressando as diversas propriedades lógicas das relações nomeadas. O raciocínio sobre algum problema baseado no domínio representado é obtido através da manipulação de tais sentenças por meio de inferência lógica. Em um ambiente típico de programação em lógica, o programador

estabelece sentenças lógicas que, reunidas, formam um programa. O computador então executa as inferências necessárias para a solução dos problemas propostos.

A lógica de primeira ordem possui dois aspectos: sintático e semântico. O aspecto sintático diz respeito às fórmulas bem-formadas (fbf's) admitidas pela gramática de uma linguagem formal. O aspecto semântico está relacionado com o significado atribuído aos símbolos presentes nas fbf's. Apresentaremos a seguir os principais conceitos necessários para a definição de linguagens lógicas de primeira ordem.

- **Definição: Teoria de Primeira Ordem**

  Uma *teoria de primeira ordem* consiste em uma linguagem de primeira ordem definida sobre um alfabeto de primeira ordem, um conjunto de axiomas e um conjunto de regras de inferência. A linguagem de primeira ordem consiste nas fbf's da teoria. Os axiomas e regras de inferência são utilizados para a derivação dos teoremas da teoria. ■

- **Definição: Alfabeto de Primeira Ordem**

  Um *alfabeto de primeira ordem* é constituído por sete classes de símbolos:

  (i) Variáveis Individuais,
  (ii) Constantes Individuais,
  (iii) Constantes Funcionais,
  (iv) Constantes Predicativas,
  (v) Conetivos,
  (vi) Quantificadores, e
  (vii) Símbolos de Pontuação. ■

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

As classes (v) a (vii) são as mesmas para todos os alfabetos, sendo denominadas *símbolos lógicos*. As classes (i) a (iv) podem variar de alfabeto para alfabeto e são denominadas *símbolos não-lógicos*. Para qualquer alfabeto de primeira ordem, somente as classes (ii) e (iii) podem ser vazias. Adotaremos aqui as seguintes convenções para a notação dos símbolos do alfabeto: As variáveis individuais serão denotadas por cadeias de símbolos iniciando com letras maiúsculas (A,B,...,Z). Constantes individuais, funcionais e predicativas serão denotadas por por cadeias de símbolos iniciando com letras minúsculas (a,b,...,z). Os conetivos são $\neg$, $\wedge$, $\vee$, $\rightarrow$ e $\equiv$. Os quantificadores são $\forall$ e $\exists$. Os símbolos de pontuação são "(", ")" e ",". Adotaremos a seguinte hierarquia para a precedência entre conetivos e quantificadores. Em ordem decrescente:

$$\neg, \forall, \exists$$
$$\vee$$
$$\wedge$$
$$\rightarrow, \equiv$$

- **Definição: Termo**

Um *termo* é definido indutivamente da seguinte maneira:

(i) Uma variável individual é um termo.

(ii) Uma constante individual é um termo.

(iii) Se f é uma função n-ária e $t_1,t_2,\ldots,t_n$ são termos, então $f(t_1,t_2,\ldots,t_n)$ é um termo. ■

- **Definição: Fórmula Bem-Formada (fbf)**

Uma *fórmula bem-formada (fbf)* é definida indutivamente da seguinte maneira:

(i) Se p é uma constante predicativa e $t_1, t_2, \ldots, t_n$ são termos, então $p(t_1, t_2, \ldots t_n)$ é uma fórmula bem-formada (denominada *fórmula atômica* ou, mais simplesmente, *átomo*).

(ii) Se f e g são fórmulas bem formadas, então $(\neg f)$, $(f \wedge g)$, $(f \vee g)$, $(f \rightarrow g)$ e $(f \equiv g)$ são fórmulas bem-formadas.

(iii) Se f é uma fórmula bem-formada e X é uma variável, então $(\forall X\ f)$ e $(\exists X\ f)$ são fórmulas bem-formadas. ■

Muitas vezes pode ser conveniente escrever a fórmula $(f \rightarrow g)$ de modo reverso, isto é, $(g \leftarrow f)$. De agora em diante, por abuso da linguagem, empregaremos indistintamente a palavra "*fórmula*" para nos referirmos a "*fórmulas bem-formadas*".

- **Definição: Linguagem de Primeira Ordem**

Uma *linguagem de primeira ordem* sobre um *alfabeto de primeira ordem* é o conjunto de todas as fórmulas construídas a partir dos símbolos desse alfabeto. ■

A semântica informal dos conetivos e quantificadores é a seguinte: $\neg$ representa a negação, $\wedge$ a conjunção (e), $\vee$ a disjunção (ou), $\rightarrow$ a implicação e $\equiv$ a equivalência. $\exists$ é o quantificador existencial, tal que "$\exists X$" significa "existe um X", enquanto que $\forall$ é o quantificador universal e "$\forall X$" significa "para todo X" ou "qualquer que seja X". Assim, a semântica informal de $\forall X(p(X,g(X)) \leftarrow q(X) \wedge \neg r(X))$ é "para todo X, se $q(X)$ é verdadeiro e $r(X)$ é falso, então $p(X,g(X))$ é verdadeiro".

- **Definição: Escopo de um Quantificador. Ocorrência Ligada de uma Variável em uma Fórmula.**

O *escopo* de $\forall X$ em $\forall Xf$ e de $\exists X$ em $\exists Xf$, é f. Uma *ocorrência ligada* de uma variável em uma fórmula é uma ocorrência que imediatamente segue o quantificador e qualquer ocorrência dessa mesma variável no escopo desse quantificador. Qualquer outra ocorrência de variável é dita ser *livre*. ■

- **Definição: Fórmula Fechada**

Uma fórmula é dita ser *fechada* quando não contém nenhuma ocorrência de variáveis livres. ■

- **Definição: Fecho Universal e Fecho Existencial**

Se f é uma fórmula, então $\forall(f)$ denota o *fecho universal* de f, que é a fórmula fechada obtida pela imposição de um quantificador universal a todas as variáveis que ocorrem livremente em f. Da mesma forma, $\exists(f)$ denota o *fecho existencial* de f, obtido pela imposição de um quantificador existencial a todas as variáveis que ocorrem livremente em f. ■

- **Definição: Literal**

Um *literal* é um átomo ou a negação de um átomo. Um *literal positivo* é um átomo, enquanto que um *literal negativo* é a negação de um átomo. ■

- **Definição: Cláusula**

Uma *cláusula* é uma fórmula do tipo:

$$\forall X_1 \ldots \forall X_s \, (l_1 \vee \ldots \vee l_m)$$

onde cada l_i é um literal e $X_1, \ldots, X_s$ são todas as variáveis que ocorrem em $l_1, \ldots, l_m$. ■

Por exemplo, são cláusulas:

$$\forall X \forall Y \forall Z \, (\, p(X,Z) \vee \neg q(X,Y) \vee \neg r(Y,Z) \,)$$

$$\forall X \forall Y \, (\, \neg p(X,Y) \vee r(f(X,Y),a) \,)$$

Uma vez que as cláusulas são tão comuns na programação em lógica, é conveniente adotarmos uma notação clausal particular. Assim, a cláusula

$$\forall X_1 \ldots \forall X_s \, (a_1 \vee \ldots \vee a_k \vee \neg b_1 \vee \ldots \vee \neg b_n),$$

onde $a_1, \ldots, a_k, b_1, \ldots, b_n$ são átomos e $X_1, \ldots, X_s$ são todas as variáveis que ocorrem nestes átomos, será representada por:

$$a_1, \ldots, a_k \leftarrow b_1, \ldots, b_n$$

Assim, na notação clausal, todas as variáveis são assumidas universalmente quantificadas, as vírgulas no antecedente, $b_1, \ldots, b_n$, denotam conjunção e as vírgulas no consequente, $a_1, \ldots, a_k$, denotam disjunção. Tais convenções se justificam uma vez que a fórmula

$$\forall X_1, \ldots \forall X_s \, (a_1 \vee \ldots \vee a_k \vee \neg b_1 \vee \ldots \neg b_n)$$

é equivalente a

$$\forall X_1, \ldots, \forall X_s \, (a_1 \vee \ldots \vee a_k \leftarrow b_1 \wedge \ldots \wedge b_n)$$

- **Definição: Cláusula de Programa**

Uma *cláusula de programa* é uma cláusula do tipo:

$$a \leftarrow b_1, \ldots, b_n$$

que contém exatamente um literal positivo, a. O literal positivo a é denominado a *cabeça* da cláusula, enquanto que a conjunção de literais $b_1, \ldots, b_n$ é o *corpo* da mesma. ■

- **Definição: Cláusula Unitária**

Uma *cláusula unitária* é uma cláusula do tipo

$$a \leftarrow$$

isto é, uma cláusula de programa com o corpo vazio. ■

A semântica informal de $a \leftarrow b_1, \ldots, b_n$ é: "para todas as possíveis atribuições de cada uma das variáveis presentes na cláusula, se $b_1, \ldots, b_n$ são todos verdadeiros, então a é verdadeiro". Assim, se $n > 0$, uma cláusula de programa é *condicional*. Por outro lado, uma cláusula unitária $a \leftarrow$ é incondicional. Sua semântica informal é "para todas as possíveis atribuições de cada uma das variáveis presentes em a, a é verdadeiro".

- **Definição: Programa em Lógica**

Um *programa em lógica* é um conjunto finito de cláusulas de programa. ■

• **Definição: Definição de um Predicado**

Em um programa em lógica, o conjunto de todas as cláusulas de programa que possuem o mesmo predicado **p** na cabeça é denominado a *definição* de **p**. ■

• **Definição: Cláusula Objetivo**

Uma *cláusula objetivo* é uma cláusula da forma

$$\leftarrow b_1, \ldots, b_n$$

isto é, uma cláusula que possui o consequente vazio. Cada b_i $(i = 1, \ldots, n)$ é um *sub-objetivo* da cláusula. ■

Se $Y_1, \ldots Y_r$ são todas as variáveis na cláusula objetivo

$$\leftarrow b_1, \ldots, b_n$$

então essa notação clausal é uma abreviatura para

$$\forall Y_1, \ldots, Y_r \; (\neg b_1 \vee \ldots \vee \neg b_n)$$

ou, de modo equivalente, para

$$\neg \exists Y_1, \ldots, \exists Y_n \; (b_1 \wedge \ldots \wedge b_n)$$

• **Definição: Cláusula Vazia**

A *cláusula vazia*, denotada por $\square$, é a cláusula que possui tanto o antecedente quanto o consequente vazios. Tal cláusula deve ser interpretada como uma *contradição*. ■

- **Definição: Cláusula de Horn**

  Uma *cláusula de Horn* é uma cláusula de programa ou uma cláusula objetivo. ■

As cláusulas de Horn são assim denominadas em homenagem ao lógico Alfred Horn que lhes estudou as propriedades. Uma de suas mais importantes características é que qualquer problema solúvel capaz de ser representado por meio delas pode ser representado de forma tal que apenas uma das cláusulas seja uma cláusula objetivo, enquanto que todas as restantes serão cláusulas de programa [KOW 79].

Para um grande número de aplicações da lógica, é suficiente empregar o contexto restrito das cláusulas de Horn. Na **Figura 3.1**, abaixo, posicionamos as cláusulas de Horn em sua relação com a lógica matemática, o cálculo de predicados de primeira ordem e a forma clausal.

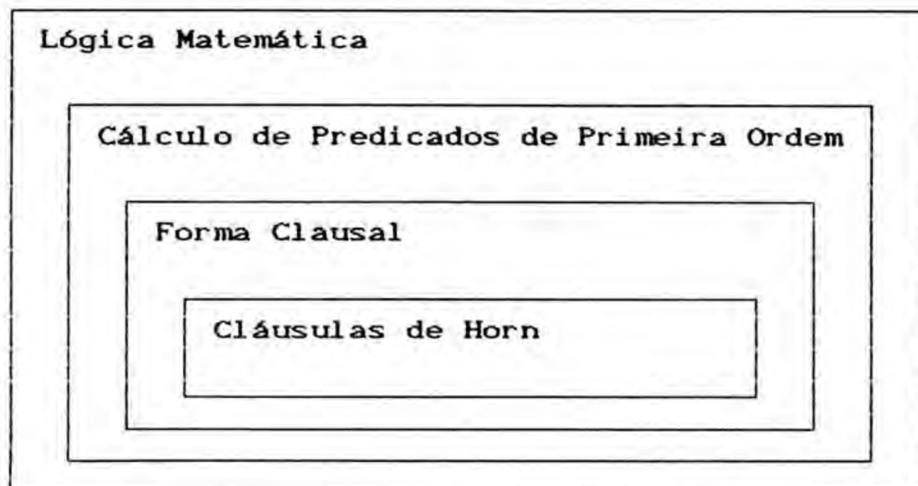


Figura 3.1

Supercontextos das Cláusulas de Horn

3.1 SEMÂNTICA MODELO-TEORÉTICA

Tratamos aqui da semântica declarativa dos programas em lógica por meio da semântica modelo-teorética de fórmulas da lógica de primeira ordem. Empregaremos, no presente trabalho, principalmente a formulação de modelos, em detrimento da formulação prova-teorética, uma vez que essa parece ser a alternativa mais comumente adotada para a definição de consultas, visões e restrições de integridade sobre BDs.

3.2.1 Modelos de Programas em Lógica

Para que sejamos capazes de discutir sobre a verdade ou falsidade representadas através de fórmulas da lógica de primeira ordem, é necessário atribuir inicialmente algum significado a cada um dos símbolos nelas presentes. Os diversos conetivos e quantificadores possuem um significado fixo, entretanto, o significado atribuído às constantes individuais, constantes funcionais e constantes predicativas pode variar. Uma *interpretação* consiste simplesmente em algum universo de discurso sobre o qual as variáveis podem assumir valores, na atribuição de um elemento desse universo a cada constante individual, na atribuição de um mapeamento sobre o domínio a cada constante funcional e de uma relação sobre o domínio a cada constante predicativa. Cada interpretação especifica assim um significado para cada símbolo na fórmula. Estamos particularmente interessados em interpretações para as quais as fórmulas expressam uma declaração verdadeira. Tais interpretações são denominadas *modelos* para as fórmulas. Normalmente haverá alguma interpretação especial, denominada *interpretação pretendida*, que irá especificar o significado principal dos símbolos. Naturalmente, a *interpretação pretendida* sempre será um modelo.

A partir de agora empregaremos os termos *constante*, *função* e *predicado* para designar respectivamente constantes individuais, constantes funcionais e constantes predicativas.

A lógica de primeira ordem nos oferece métodos para a dedução dos teoremas presentes em alguma teoria. Estes podem ser caracterizados como sendo as fórmulas que são consequência lógica dos axiomas da teoria, isto é, que são verdadeiras em todas as interpretações que são modelos para cada um dos axiomas da teoria. Em particular, cada teorema deve ser verdadeiro na interpretação pretendida da teoria. Os sistemas de programação em lógica que irão nos interessar são os que adotam o princípio da resolução como única regra de inferência. Vamos supor que desejamos provar que a fórmula

$$\exists Y_1, \ldots \exists Y_r \ (b_1 \wedge \ldots \wedge b_n)$$

é uma consequência lógica de um programa P. Com esse objetivo empregamos o princípio da resolução por meio de um sistema de refutação, isto é, a negação da fórmula a ser provada é adicionada aos axiomas e uma contradição deve ser derivada. Se negarmos a fórmula que desejamos provar obteremos a cláusula objetivo:

$$\leftarrow b_1, \ldots, b_n$$

A partir dessa cláusula objetivo e operando de forma top-down sobre os axiomas de P, o sistema deriva sucessivas cláusulas objetivo. Se, em um determinado momento for derivada a cláusula vazia, então uma contradição foi obtida (a cláusula vazia é contraditória) e esse resultado nos assegura que $\exists Y_1 \ldots \exists Y_n$ $(b_1, \ldots, b_n)$ é uma consequência lógica de P. De agora em diante usaremos simplesmente *objetivo* para designar cláusulas objetivo.

Do ponto de vista da prova de teoremas, o único interesse é demonstrar a existência da relação de consequência lógica. Por outro lado, do ponto de vista da programação em lógica estamos muito mais interessados nas *ligações* que foram realizadas sobre as variáveis $Y_1,...Y_r$, uma vez que estas nos fornecem a *saída* da execução do programa. Segundo [LLO 84], a visão ideal de um sistema de programação em lógica é a de uma caixa preta para a computação de ligações e o nosso único interesse reside no seu comportamento de entrada e saída, isto é, as operações executadas internamente pelo sistema deveriam ser transparentes para o programador. Infelizmente tal situação não ocorre, em maior ou menor grau, nos sistemas Prolog atualmente disponíveis, de forma que muitos programas Prolog somente podem ser entendidos a partir de sua interpretação operacional, devido ao emprego de *cuts* e outros mecanismos extra-lógicos.

- **Definição: Interpretação**

Uma *interpretação* de uma linguagem L de primeira ordem é constituída por:

(i) Um conjunto não-vazio D, denominado o *domínio* da interpretação.

(ii) Para cada constante em L, a atribuição de um elemento de D.

(iii) Para cada função n-ária em L, a atribuição de um mapeamento de D^n em D.

(iv) Para cada predicado n-ário em L, a atribuição de um mapeamento de D^n em {verdadeiro,falso}, isto é, de uma relação sobre D^n. ■

- **Definição: Atribuição de Variáveis**

Seja I uma interpretação de uma linguagem L de primeira ordem. Uma *atribuição de variáveis* (com respeito a I) é uma atribuição de um elemento do domínio de I a cada uma das variáveis em L. ■

- **Definição: Atribuição de Termos**

Seja I uma interpretação de uma linguagem L de primeira ordem, com domínio **D**, e seja **A** uma atribuição de variáveis. Uma *atribuição de termos* (com respeito a I e A) para os termos em L é definida da seguinte maneira:

(i) A cada variável em L é dada uma atribuição de acordo com **A**.

(ii) A cada constante em L é dada uma atribuição de acordo com **I**.

(iii) Se $t_1',\ldots t_n'$ são as atribuições dos termos $t_1,\ldots t_n$ e f' é a atribuição de f, então $f'(t_1',\ldots,t_n') \in$ **D** é a atribuição de termos de $f(t_1,\ldots,t_n)$. ■

- **Definição: Valor-Verdade de uma Fórmula**

Seja **I** uma interpretação de domínio **D** de uma linguagem **L** de primeira ordem e seja **A** uma atribuição de variáveis. Então a uma fórmula em **L** pode ser atribuído um *valor-verdade* (verdadeiro ou falso, que denotaremos por V e F respectivamente) com respeito a **I** e a **A**, da seguinte maneira:

(i) Se a fórmula é um átomo, $p(t_1,\ldots t_n)$, então o valor-verdade é obtido pelo cálculo do valor-verdade de $p'(t_1',\ldots,t_n')$, onde p' é o mapeamento atribuído a **p** por **I** e $t_1',\ldots,t_n'$ é a atribuição de termos para $t_1,\ldots,t_n$, com respeito a **I** e a **A**.

(ii) Se a fórmula tem a forma $\neg f$, $f \wedge g$, $f \vee g$, $f \rightarrow g$ ou $f \equiv g$, então o valor-verdade da fórmula é dado pela tabela verdade:

f	g	¬f	f∧g	f∨g	f→g	f≡g
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

(iii) Se a fórmula tem a forma ∃X f, então o valor verdade da fórmula é V se existe d∈D tal que f tem valor-verdade V com respeito a I e A(X/d), onde A(X/d) é A, exceto que a X é atribuído d. Caso contrário o seu valor-verdade é F.

(iv) Se a fórmula tem a forma ∀X f, então o valor verdade da fórmula é V se, para todo d∈D, temos que f tem valor-verdade V com respeito a I e a A(X/d). Caso contrário o seu valor-verdade é F. ■

- **Definição: Modelo de uma Fórmula**

Seja I uma interpretação de uma linguagem L de primeira ordem e seja f uma fórmula fechada de L. Então I é um *modelo* para f se o valor-verdade de f com respeito a I é V. ■

- **Definição: Modelo de um Conjunto de Fórmulas Fechadas**

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem e seja I uma interpretação de L. Dizemos que I é um *modelo* para S se I é um *modelo* para cada uma das fórmulas em S. ■

- **Definição: Conjunto de Fórmulas Satisfatível**

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *satisfatível* se L possui uma interpretação que é um modelo para S. ■

- **Definição: Conjunto de Fórmulas Válido**

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *válido* se toda interpretação de L é um modelo para S. ■

- **Definição: Conjunto de Fórmulas Insatisfatível**

Seja S um conjunto de fórmulas fechadas de uma linguagem L de primeira ordem. Dizemos que S é *insatisfatível* se não tem modelos em L. Note que $\{f, \neg f\}$ é insatisfatível, assim como a cláusula vazia, denotada por $\square$. ■

- **Definição: Consequência Lógica de um Conjunto de Fórmulas Fechadas**

Seja S um conjunto de fórmulas fechadas e seja f uma fórmula fechada de uma linguagem L de primeira ordem. Dizemos que f é *consequência lógica* de S, isto é, $S \models f$ se, para toda interpretação I de L, se I é um modelo para S então I é também um modelo para f. Note que se $S = \{f_1, \ldots, f_n\}$ é um conjunto finito de fórmulas fechadas, então f é consequência lógica de S se e somente se $f_1 \wedge \ldots \wedge f_n \rightarrow f$ é válida. ■

- **Proposição 3.1**

Seja S um conjunto de fórmulas fechadas e f uma fórmula fechada de uma linguagem L de primeira ordem. Então f é consequência lógica de S se e somente se S ∪ {¬f} é insatisfatível.

Prova:

Vamos supor que f seja consequência lógica de S. Se S ∪ {¬f} é satisfatível, então existe uma interpretação I da linguagem L tal que I é modelo de S ∪ {¬f}. Por outro lado, se f é consequência lógica de S, então I é também modelo de f, ou seja de {f,¬f}, o que não é possível. Logo S ∪ {¬f} é insatisfatível.

Inversamente, vamos supor que S ∪ {¬f} seja insatisfatível e seja I uma interpretação da linguagem L. Suponhamos que I seja um modelo para S. Uma vez que S ∪ {¬f} é insatisfatível, I não pode ser modelo para ¬f. Assim I é um modelo para f e portanto f é consequência lógica de S. ■

Aplicando as definições acima a programas em lógica, constatamos que quando fornecemos um objetivo G ao sistema com o programa P carregado, estamos pedindo ao sistema para provar que P ∪ {G} é insatisfatível. Se G é o objetivo $\leftarrow b_1,\ldots,b_n$, com as variáveis $Y_1,\ldots,Y_r$, então a proposição 3.1 estabelece que provar que P ∪ {G} é insatisfatível equivale a provar que $\exists Y_1,\ldots,\exists Y_r\ (b_1 \wedge \ldots \wedge b_n)$ é consequência lógica de P. Assim o problema básico é a determinação da insatisfatibilidade de P ∪ {G},onde P é um programa e G um objetivo. De acordo com a definição de insatisfabilidade, isso implica em mostrar que *nenhuma* interpretação de P ∪ {G} é um modelo.

- **Definição: Termo Básico. Átomo Básico.**

Um *termo básico* é um termo que não contém variáveis. Da mesma forma um *átomo básico* é um átomo que não contém variáveis. ■

- **Definição: Universo de Herbrand**

Seja L uma linguagem de primeira ordem. O *universo de Herbrand* $\mathbf{U_L}$ para L é o conjunto de todos os termos básicos que podem ser obtidos a partir das constantes e funções presentes em L. No caso em que L não possui constantes, introduziremos uma constante (por exemplo, "a") para a formação de termos básicos. ■

- **Definição: Base de Herbrand**

Seja L uma linguagem de primeira ordem. A *base de Herbrand* $\mathbf{B_L}$ para L é o conjunto de todos os átomos básicos que podem ser formados usando os predicados de L com termos básicos do universo de Herbrand como argumentos. ■

- Definição: **Interpretação de Herbrand**

Seja L uma linguagem de primeira ordem. Uma interpretação sobre L é uma *interpretação de Herbrand* se as seguintes condições forem satisfeitas:

(i) O domínio da interpretação é o universo de Herbrand, $\mathbf{U_L}$;

(ii) As constantes em L são atribuídas "a si próprias em $\mathbf{U_L}$;

(iii) Se f é uma função n-ária em L, então a f é atribuído o mapeamento de $(U_L)^n$ em U_L definido por $(t_1,...,t_n) \rightarrow f(t_1,...,t_n)$. ■

Nenhuma restrição é feita sobre a atribuição de predicados em L, de forma que diferentes interpretações de Herbrand surgem quando se emprega diferentes atribuições sobre eles. Uma vez que, para as interpretações de Herbrand, as atribuições de constantes e funções é fixa, é possível identificar uma interpretação de Herbrand com um subconjunto da base de Herbrand. Para toda interpretação de Herbrand, o correspondente subconjunto da base de Herbrand é o conjunto de todos os átomos básicos que são verdadeiros com respeito à essa interpretação. Inversamente, dado um subconjunto arbitrário da base de Herbrand, há uma interpretação de Herbrand que a ele corresponde.

- **Definição: Modelo de Herbrand**

Seja L uma linguagem de primeira ordem e S um conjunto de fórmulas fechadas de L. Um *modelo de Herbrand* para S é uma interpretação de Herbrand para L que é um modelo para S. ■

- **Proposição 3.2**

Seja S um conjunto de cláusulas e suponha que S tem um modelo. Então S tem um modelo de Herbrand.

Prova:

Seja I uma interpretação de S. Uma interpretação de Herbrand de S, I', é definida por:

$$I' = \{p(t_1,...,t_n) \in \mathbf{Bs} \mid p(t_1,...,t_n) \text{ é V c.r.a. } I\}$$

Segue diretamente que se I é um modelo para S, então I' também é um modelo para S. ■

Proposição 3.3

Seja S um conjunto de cláusulas. Então S é insatisfatível se e somente se S não possui modelo de Herbrand.

Prova:

Se S é satisfatível, então a **proposição 3.2** demonstra que S tem um modelo de Herbrand. ■

3.2.2 Substituições Resposta

Conforme foi anteriormente estabelecido, o propósito principal de um sistema de programação em lógica é a computação de ligações. Na presente seção introduzimos o conceito de *substituição resposta correta*, que permite um entendimento declarativo da saída desejada de um programa e um objetivo.

Definição: Substituição

Uma *substituição* θ é um conjunto finito da forma $\{v_1/t_1, \ldots, v_n/t_n\}$, onde cada v_i é uma variável e cada t_i é um termo distinto de v_i. Além disso, as variáveis $v_1, \ldots, v_n$ devem ser distintas. Cada elemento v_i/t_i é denominado uma *ligação* para v_i. Se os t_i são todos termos básicos, então θ é denominada uma *substituição básica*. Se os t_i são todos variáveis, θ é denominada uma *substituição-variável pura*. ■

- **Definição: Expressão**

Uma *expressão* é um termo, um literal ou uma conjunção ou disjunção de literais. Uma *expressão simples* é um termo ou um átomo. ■

- **Definição: Instância de uma Expressão**

Seja $\theta = \{v_1/t_1,\dots,v_n/t_n\}$ uma substituição e E uma expressão. Então $E\theta$, a *instância de* E *pela substituição* θ, é a expressão obtida a partir de E através da substituição simultânea de todas as ocorrências da variável v_i em E pelo termo t_i $(i=1,\dots,n)$. Se $E\theta$ é básica, então $E\theta$ é chamada uma *instância básica* de E. Se $S = \{E_1,\dots,E_n\}$ é um conjunto finito de expressões e θ é uma substituição, então $S\theta$ denota o conjunto $\{E_1\theta,\dots,E_n\theta\}$. ■

- **Definição: Composição de Substituições**

Sejam $\theta = \{u_1/s_1,\dots,u_m/s_m\}$ e $\sigma = \{v_1/t_1,\dots,v_n/t_n\}$ duas substituições. Emtão a *composição* $\theta\sigma$ é a substituição obtida do conjunto

$$\{u_1/s_1\sigma,\dots,u_m/s_m\sigma,v_1/t_1,\dots,v_n/t_n\}$$

retirando-se dele todas as ligações $u_i/s_i\sigma$ para as quais $u_i=s_i\sigma$ e todas as ligações v_j/t_j para as quais $v_j \in \{u_1,\dots,u_m\}$. ■

- **Definição: Substituição Identidade**

Substituição identidade é a substituição dada pelo conjunto vazio. Denotaremos a substituição identidade por ε. Note que $E\varepsilon = E$ para todas as expressões E. ■

- **Proposição 3.4**

Sejam θ, σ e γ substituições. Então:

(i) $\theta\varepsilon = \varepsilon\theta = \theta$

(ii) $\forall E\ (E\theta)\sigma = E(\theta\sigma)$

(iii) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$

Prova:

(i) Segue diretamente da definição de ε.

(ii) É suficiente provar o resultado quando E é uma variável, digamos **X**. Seja $\theta = \{u_1/s_1,\ldots,u_m/s_m\}$ e $\sigma = \{v_1/t_1,\ldots,v_n/t_n\}$. Se $X \notin \{u_1,\ldots,u_m\}\cup\{v_1,\ldots,v_n\}$, então $(X\theta)\sigma = X = X(\theta\sigma)$. Se $X \in \{u_1,\ldots,u_m\}$, digamos $X = u_i$, então $(X\theta)\sigma = s_i\sigma = X(\theta\sigma)$. Se $X \in \{v_1,\ldots,v_n\}\setminus\{u_1,\ldots,u_m\}$, digamos $X = v_j$, então $(X\theta)\sigma = t_j = X(\theta\sigma)$.

(iii) É suficiente mostrar que se **X** é uma variável, então $X((\theta\sigma)\gamma) = X(\theta(\sigma\gamma))$. De fato, $X((\theta\sigma)\gamma)=(X(\theta\sigma))\gamma=((X\theta)\sigma)\gamma=(X\theta)(\sigma\gamma)=X(\theta(\sigma\gamma))$, em função de (ii). ■

- **Definição: Variantes**

Sejam E e F expressões. Dizemos que E e F são *variantes* se existem as substituições θ e σ tais que $E = F\theta$ e $F = E\sigma$. Diz-se também que E é variante de F ou que F é variante de E. ■

- **Definição: Renomeação**

Seja E uma expressão e **V** o conjunto das variáveis que ocorrem em E. Uma *renomeação* para E é uma substituição variável pura $\{X_1/Y_1,\ldots,X_n/Y_n\}$ tal que $\{X_1,\ldots,X_n\} \subseteq V$, os Y_i são distintos e $(V\setminus\{X_1,\ldots,X_n\})\cap\{Y_1,\ldots,Y_n\}=\emptyset$. ■

- **Proposição 3.5**

Sejam E e F expressões variantes. Então existem as substituições θ e σ tais que $E = F\theta$ e $F = E\sigma$, onde θ é uma renomeação para F e σ é uma renomeação para E.

Prova:

Uma vez que E e F são variantes, então existem as substituições θ_1 e σ_1 tais que $E = f\theta_1$ e $F = E\sigma_1$. Seja V o conjunto das variáveis que ocorrem em E e seja σ a substituição obtida de σ_1 através da remoção de todas as ligações da forma X/t, onde $X \notin V$. Claramente, então, $F = E\sigma$. Além disso, $E = F\theta_1 = E\sigma\theta_1$, de onde segue que σ deve ser uma renomeação para E. ■

Estaremos interessados principalmente nas substituições que *unificam* um conjunto de expressões, isto é, que tornam as expressões contidas em um conjunto sintaticamente idênticas. O conceito de unificação remonta aos estudos de Herbrand em 1930, tendo sido empregado por Robinson [ROB 65] na regra da resolução. Restringiremos nossa atenção a conjuntos finitos (não vazios) de expressões simples (termos ou átomos).

- **Definição: Unificador**

Seja S um conjunto finito de expressões simples. Uma substituição θ é dita ser um *unificador* para S, se $S\theta$ é única. Um unificador θ é dito ser um *unificador mais geral (umg)* para S se, para todo unificador σ de S há uma substituição γ tal que $\sigma = \theta\gamma$. ■

Segue da definição de umg que se θ e σ são ambos umg's de $\{E_1, ... E_n\}$, então $E_i\theta$ é variante de $E_i\gamma$. A proposição 3.5 nos garante então que $E_i\theta$ pode ser obtida de $E_i\gamma$ por simples renomeação de variáveis.

- **Definição: Conjunto de Desacordo**

Seja S um conjunto finito de expressões simples. O *conjunto de desacordo* de S é definido da seguinte maneira: Localizamos a posição do símbolo mais à esquerda que não é o mesmo para todas as expressões de S e extraímos de cada uma delas a sub-expressão que inicia com tal símbolo. O conjunto de todas as sub-expressões assim retiradas é o conjunto de desacordo de S. ■

- **Algoritmo da Unificação**

(i) Faça $K = 0$ e $\sigma_0 = \varepsilon$

(ii) Se $S\sigma_K$ é único, então pare: σ_K é um umg de S. Senão, encontre o conjunto de desacordo D_K de $S\sigma_K$.

(iii) Se existem **V** e **t** em D_K tais que **V** é uma variável que não ocorre em **t**, então faça $\sigma_{K+1} = \sigma_K\{V/t\}$, incremente o valor de K e volte ao passo (ii). Senão pare: S não é unificável. ■

Na forma apresentada acima, o algoritmo de unificação é não-determinístico, uma vez que podem ser consideradas diversas escolhas para **V** no passo (iii), entretanto a aplicação de quaisquer dois umg's produzidos pelo algoritmo irá conduzir a expressões que diferem entre si somente pelo nome das variáveis envolvidas. Deve ficar claro também que o algoritmo sempre termina, uma vez que S contém um conjunto finito de variáveis e cada aplicação do passo (iii) elimina uma delas. Ainda devemos considerar que no passo (iii) uma verificação é feita para garantir que **V** não ocorre em **t**. Tal verificação é denominada *verificação de ocorrência* (occur check).

- Teorema 3.1 (Teorema da Unificação)

  (a) S é um conjunto unificável de expressões simples, se e somente se o Algoritmo da Unificação termina retornando um umg para S.

  (b) S não é um conjunto unificável de expressões simples se e somente se o Algoritmo da Unificação termina retornando a resposta "**não**".

  Prova:

  Por razões de espaço deixamos de apresentar aqui a prova do Teorema da Unificação. Esta pode ser encontrada em diversas obras sobre programação em lógica como, por exemplo, **[CAS 87]** ou **[LLO 84]**. ■

- **Definição: Substituições Resposta**

  Seja **P** um programa e **G** um objetivo. Uma *substituição resposta* para **P** ∪ {**G**} é uma substituição para as variáveis de G. ■

Entende-se que tal substituição não precisa necessariamente conter uma ligação para cada uma das variáveis em G. Em particular, se G não contém variáveis, a única substituição possível é a substituição identidade.

- **Definição: Substituição Resposta Correta**

  Seja P um programa, G um objetivo $\leftarrow A_1,\ldots,A_k$ e θ uma substituição resposta para P ∪ {G}. Dizemos que θ é uma *substituição resposta correta* para **P** ∪ {G} se $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ é consequência lógica de **P**. ■

A partir da proposição 3.1, podemos afirmar que θ é uma substituição resposta correta se e somente se $P \cup \{\neg\forall((A_1 \wedge .. A_k)\theta)\}$ for insatisfatível. Esta definição de substituição resposta correta captura o significado intuitivo de "resposta correta". Da mesma forma que fornece substituições respostas, um sistema de programação em lógica pode tambem retornar com a resposta "**não**". Dizemos que a resposta "**não**" é correta se $P \cup \{G\}$ for satisfatível.

3.3 SEMÂNTICA PROVA-TEORÉTICA

A lógica clássica de primeira ordem é definida pela especificação de um esquema de axiomas e regras de inferência (para as definições básicas veja a seção 3.1):

- **Axiomas:** Para todas as fbfs **f**, **g** e **h** de uma certa linguagem L da lógica de predicados de primeira ordem:

  (1) $A \rightarrow (B \rightarrow A)$

  (2) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

  (3) $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$

  (4) $\forall X\ A(X) \rightarrow A(t)$, onde **t** é um termo livre de **X** em $A(X)$, isto é, nenhuma ocorrência livre de **X** em **A** surge no escopo de qualquer quantificador $(\forall X')$ onde **X'** é uma variável em **t**.

  (5) $(\forall X)(A \rightarrow B) \rightarrow (A \rightarrow \forall X\ B)$ onde **A** não contém nenhuma ocorrência livre de **X**. ■

- Regras de Inferência:

$$\frac{A, \quad A \rightarrow B}{B} \qquad \text{[MP - } \textit{modus ponens}\text{]}$$

$$\frac{A}{\forall X(A)} \qquad \text{[GEN - } \textit{generalização}\text{]}$$

■

- **Definição: Prova**

Uma *prova* é qualquer sequência da forma $A_1, \ldots, A_n$ onde cada A_i ou é uma instância de um esquema de axiomas ou deriva dos membros anteriores da sequência por meio da aplicação de **MP** ou **GEN**. ■

- **Definição: Teorema**

Um *teorema* é qualquer fbf que resulta de uma prova, isto é, o último membro de uma sequência de prova. ■

- **Definição: Frame de Primeira Ordem**

Um *frame de primeira ordem* **M** para uma linguagem **L** da lógica de primeira ordem consiste em um domínio não-vazio **D**, juntamente com uma função que atribui a cada símbolo funcional n-ário **f** uma função **f'** de $D^n \rightarrow D$ e a cada constante relacional C um elemento C' de 2^{D^n}. ■

Para estabelecer a semântica da linguagem L com respeito a esse frame, utilizaremos uma função de atribuição **g** que atribui a cada variável individual um elemento de **D**. A notação:

$$M \models_g A$$

indica que a função de atribuição g *satisfaz* a fbf A no frame M.

(1) $M \models_g C(t_o,...,t_{n-1}) \equiv (V(t_o,g),...,V(t_{n-1},g)) \in C'$

onde $V(t,g) = g(t)$ se t é uma variável individual e em $f'(V(t'_o,g),...,V(t'_{m-1},g))$ os t_i são da forma $f(t'_o,...,t'_{m-1})$.

(2) $M \models_g \neg A \equiv M \not\models_g A$

(3) $M \models_g A \wedge B \equiv M \models_g A$ e $M \models_g B$

(4) $M \models_g \forall X\ A \equiv M \models_{g(d \backslash X)} A$

onde $g(d \backslash X)$ é uma função de atribuição idêntica a g, exceto para a variável X, à qual é atribuído o valor d.

As condições de verdade para os demais conetivos podem ser estabelecidas a partir das seguintes equivalências:

$$A \vee B \equiv \neg(\neg A \wedge \neg B)$$

$$A \rightarrow B \equiv \neg A \vee B$$

$$A \equiv B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$$

$$\exists X\ A \equiv \neg \forall X\ \neg A$$

■

• **Definição: Fórmula Universalmente Válida**

Uma fbf A é dita ser *universalmente válida* se e somente se para todo frame M e para toda função de atribuição g, $M \models A$. ■

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Essas duas noções (isto é, ser um teorema e ser universalmente válida) são conectadas por meio do teorema da completeza:

- **Teorema 3.2: Completeza do Cálculo de Predicados**

  Uma fbf do cálculo de predicados de primeira ordem é um teorema se e somente se é universalmente válida. ■

4 EXTENSÕES DA LÓGICA DE PRIMEIRA ORDEM

A expressão "*lógica não-convencional*" é um termo genérico empregado na designação de qualquer outro sistema lógico diferente da lógica proposicional ou cálculo de predicados. Segundo **[TUR 84]** tais lógicas podem, a grosso modo, ser divididas em dois grupos: as que rivalizam com a lógica clássica e as que simplesmente a estendem. No primeiro grupo encontram-se as lógicas de múltiplos valores, a lógica difusa e a lógica intuicionista, enquanto que o segundo compreende a lógica modal, a lógica temporal e as lógicas de ordem superior. Evidentemente essa divisão não é precisa, porém pode ser adotada para fins de referência. Os sistemas que rivalizam com a lógica não diferem da lógica de predicados de primeira ordem em termos da linguagem empregada, entretanto certos teoremas desta última resultam falsos se submetidos a tais sistemas como, por exemplo, a lei do terceiro excluído. As lógicas que estendem a lógica de primeira ordem sancionam todos os seus teoremas, suplementando-os de dois modos distintos. Em primeiro lugar, as linguagens empregadas pelas extensões enriquecem o potencial de expressividade da lógica clássica. Depois, os teoremas dos sistemas lógicos estendidos suplementam os teoremas da lógica clássica em decorrência da utilização desse vocabulário mais rico. Por exemplo, a lógica modal supõe o emprego de dois novos operadores: $\Box$ (é necessário que) e $\Diamond$ (é possível que). Sob este novo regime a sentença $\mathbf{A} \rightarrow \Diamond\mathbf{A}$ é tomada como axiomática. A adição de axiomas desse tipo e das apropriadas regras de inferência envolvendo tais operadores facilita a derivação de teoremas que não podem sequer ser expressos na linguagem do cálculo de predicados de primeira ordem. No presente capítulo iremos estudar algumas extensões à logica de predicados de primeira ordem, com a finalidade de verificar sua possível utilização em sistemas de programação em lógica visando incrementar a expressividade potencial de tais sistemas.

4.1 LÓGICA MODAL

A *lógica modal* opera sobre argumentos que envolvem os conceitos de **necessidade e possibilidade**. Uma verdade *necessária* é aquela que não poderia ser de outra maneira. Uma verdade *contingente*, por outro lado, poderia. A distinção é frequentemente estabelecida através da noção de *mundos possíveis*: Uma verdade necessária seria verdadeira em *todos* os mundos possíveis, enquanto que uma verdade contingente é verdadeira no mundo atual mas não em todos os mundos possíveis.

Essa diferença é de natureza metafísica **[TUR 84]** e não deve ser confundida com a distinção entre verdades *a priori* e *a posteriori*. Uma verdade *a priori* é uma verdade que pode ser reconhecida como tal independentemente da experiência, enquanto que uma verdade *a posteriori* não pode. Tais noções conduzem, evidentemente, a considerações epistemológicas além dos objetivos do presente trabalho, de modo que iremos nos concentrar apenas nos aspectos formais da lógica modal, com vistas a possíveis aplicações.

A linguagem da lógica modal de primeira ordem, L_M é obtida a partir da linguagem do cálculo de predicados, L, pela adição de dois novos operadores: $\Box$, lido como "*é necessário que*" e $\Diamond$, lido como "*é possível que*". Mais precisamente, L_M é obtida de L pela adição da seguinte condição na definição de fbf:

"*Se* A *é uma fbf de* L_M, *então* $\Box$A *e* $\Diamond$A *também o são.*"

Ao contrário dos conetivos clássicos, $\neg$, $\wedge$, $\vee$, $\rightarrow$ e $\equiv$, os operadores $\Box$ e $\Diamond$ não admitem uma interpretação funcional de valores-verdade, ao invés disso sua interpretação está relacionada com a noção de *mundo possível*. A grosso modo podemos

dizer que $\Diamond$**A** é verdadeira se **A** é verdadeira em *alguns* dos mundos possíveis, enquanto que $\Box$**A** é verdadeira se **A** é verdadeira em *todos* os mundos possíveis.

- **Definição: Frame Modal**

Um *frame modal* **M** é uma estrutura <**W**, **D**, **R**, **F**> onde:

(i) **W** é um conjunto não vazio (de *mundos possíveis*),

(ii) **D** é um domínio não-vazio (de *indivíduos*),

(iii) **R** é uma relação binária de *acessibilidade* sobre **W**, e

(iv) **F** é uma função que atribui a cada par constituído por um símbolo funcional de aridade **n** ($n \geq 0$) e por um elemento **w** de **W**, uma função de $D^n \rightarrow D$ e a cada par constituído por um símbolo relacional de aridade **n** ($n > 0$) e por um elemento **w** de **W**, um elemento de 2^{D^n}. ∎

A função **F** deve ser pensada como atribuindo a cada símbolo funcional **f** e mundo **w** a extensão da função nomeada por **f** em **w**, o mesmo ocorrendo para os símbolos relacionais. Generalizando a noção de frame, concluímos que o domínio de indivíduos poderia variar em mundos diferentes. A relação de acessibilidade pretende capturar a idéia que, do ponto de vista de um certo mundo **w**, outros mundos podem ser considerados possíveis além dos considerados como tal do ponto de vista de algum outro mundo diferente de **w**. A noção do que é possível e do que não é se apresenta de forma relativa: o que é possível ou não depende *de como as coisas são*.

A interpretação da linguagem L_M para o frame modal difere da interpretação do cálculo de predicados tendo em vista o papel crucial desempenhado pelo domínio **W**. A interpretação é, entretanto, tomada de maneira semelhante em relação à função de atribuição g que atribui elementos de **D** às variáveis individuais. Por questões de conveniência escreveremos **wRw'** para indicar que <w,w'> satisfaz a relação **R**. Empregaremos a notação $M \models_{g,w} A$ para indicar que g satisfaz a fbf **A**, no mundo **w** e no frame modal **M**. Isso pode ser definido recursivamente como se segue:

(1) $M \models_{w,g} C(t_0,\ldots,t_{n-1}) \equiv \langle V(t_0,w,g),\ldots,V(t_{n-1},w,g)\rangle \in F(w,g)$
onde $V(t,w,g) = g(t)$ se t é variavel ou $V(t,w,g) = F(w,f)(V(t'_0,w,g),\ldots,V(t'_{m-1},w,g))$ se $t = f(t'_0,\ldots,t'_{m-1})$

(2) $M \models_{w,g} t_1 = t_2 \quad \equiv \quad V(t_1,w,g) = V(t_2,w,g)$

(3) $M \models_{w,g} A \wedge B \quad \equiv \quad M \models_{w,g} A$ e $M \models_{w,g} B$

(4) $M \models_{w,g} \neg A \quad \equiv \quad M \not\models_{w,g} A$

(5) $M \models_{w,g} \forall X\ A \quad \equiv \quad \forall d \in D\ M \models_{w,g(d/X)} A$

(6) $M \models_{w,g} \Diamond A \quad \equiv \quad \exists w' \in W \mid wRw' \wedge M \models_{w',g} A$

A interpretação dos demais conetivos lógicos é dada pela definição na forma convencional. Adicionalmente definiremos $\Box A =_{df} \neg\Diamond \neg A$. As propriedades formais de **R** determinam a lógica modal (axiomas e regras de inferência). Diremos que uma fbf é válida em relação a uma particular classe de frames **C** se e somente se ela é satisfeita por todas as funções de atribuição e todos os mundos possíveis em cada um dos frames de **C**. Assim, por exemplo, se considerarmos **C** como sendo a classe de todos os frames onde **R** é tão somente reflexiva, obteremos a lógica usualmente conhecida como **T**. Se adicionalmente estabelecermos que **R** deve também ser transitiva, obteremos a lógica modal **S4**. A lógica modal **S5** é obtida para **R** reflexiva, transitiva e simétrica (isto é, uma relação de equivalência). A lógica modal **T** é caracterizada pelo esquema axiomático do cálculo de predicados, adicionado de:

$$\Box A \rightarrow A \quad \text{e} \quad \Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$$

e da regra de inferência

$$\frac{A}{\Box A},$$

conhecida como a *regra de necessitação*. A lógica S4 é obtida pela adição de $\Box A \rightarrow \Box\,\Box A$ a T, enquanto que a S5 é obtida pela adição de $\Diamond A \rightarrow \Box\,\Diamond A$ a S4. Uma vez que empregamos frames no lugar de domínios fixos de indivíduos, a *fórmula de Barcan:*

$$\forall X(\Box A) \rightarrow \Box(\forall X\ A)$$

é válida para todos os frames.

A concepção clássica das modalidades em termos dos conceitos metafísicos de necessidade e possibilidade guarda uma relação apenas indireta com a ciência da computação. Essa última esta baseada em noções como *programa*, *algoritmo* e *computação*. A lógica modal, entretanto, admite uma interpretação na qual tais conceitos representam um papel essencial, [illegible] o significado [illegible] como derivado de seu impacto sobre a memória de alguma máquina abstrata. Mais explicitamente, vamos considerar um simples comando de atribuição, "X := 5". É natural entender o significado desse comando como uma função que, dado um certo estado de memória de uma alguma máquina abstrata, modifica esse estado ligando o valor 5 à variável X. Em geral, todo comando desse tipo em uma linguagem de programação determinística pode ser visto como uma aplicação da função descrita acima. Essa visão é empregada, por exemplo, na semântica denotacional das linguagens de programação e pode ser generalizada para o caso das linguagens não determinísticas, onde o comando é interpretado não mais como uma função e sim como uma relação binária sobre o estado de memória da máquina abstrata considerada.

Sob tal interpretação, cada comando do tipo G em uma linguagem de programação, induz os operadores modais □G e ◇G. Além disso, nessa perspectiva, o papel dos *mundos possíveis* é agora desempenhado sobre os estados de memória da máquina subjacente, de forma que □GA será verdadeira em um estado s se existe um estado final s', que pode ser atingido a partir de s por meio de G, em que A é verdadeira.

Diversos estudos tem sido realizados sobre a lógica modal visando o seu emprego em sistemas particulares de programação em lógica cuja expressividade é claramente superior ao cálculo de predicados de primeira ordem, entre os quais citamos **[MOO 84]**, **[CER 86]**, **[SAK 86]** e **[IWA 88]**. O sistema MOLOG, por exemplo, proposto em **[CER 86]**, é uma extensão do Prolog com operadores modais com um semântica definida em termos de mundos possíveis. Em **[IWA 88]** é proposto um modelo para representação de conhecimento e inferência baseado na lógica modal de primeira ordem.

4.2 LÓGICA EM NÍVEL META

Uma das características mais interessantes e de grande potencialidade da lógica é a sua capacidade de representar conceitos em nível *meta*. Um exemplo simples de tal capacidade, no âmbito da programação em lógica é a noção do que pode ser computado (inferido) a partir de um conjunto de procedimentos. Consideremos por exemplo a seguinte sentença descrevendo a negação como regra de falha:

infer ¬X de Y se ¬infer X de Y

Aqui **Y**, **X** e **¬X** se referem a elementos constituintes de um sistema a nível objeto, neste caso a declarações de programas na *linguagem objeto* (LO) das cláusulas de Horn. Por outro lado, os símbolos **infer**, **¬infer** e se, são constituintes da *metalinguagem* (ML), na qual descrevemos as características do sistema em seu nível objeto.

Se considerarmos a LO como a linguagem na qual programas concretos são compostos e executados, então a ML se torna uma linguagem para raciocinar sobre a composição e execução de programas, isto é, uma linguagem na qual podemos construir ferramentas para a manipulação de programas tais como interpretadores, verificadores, editores ou sistemas operacionais. O que se apresenta como de especial interesse é o fato de que a ML pode também ser construída sobre a lógica das cláusulas de Horn. Esse duplo papel da lógica foi reconhecido e explorado já nos primeiros dias da programação em lógica. O primeiro esforço bem sucedido de incorporar metateoremas sobre a noção formal de prova e de consequência lógica através do mesmo formalismo empregado para a LO deve-se a Gödel (1931), conforme citado em **[PER 88]**.

O emprego de um único formalismo para desempenhar os papéis de LO e ML denomina-se *amalgamação* e, para ser efetivo, necessita fazer uso de alguma definição PR da *relação de provabilidade*. A construção e análise de PR para a lógica das cláusulas de Horn foi estudada por Kowalski em **[KOW 79]**:

> *"Para definir provabilidade é necessário nomear sentenças e outras expressões por meio de termos. Isso pode ser obtido de diversas maneiras (...). Dada uma representação de sentenças por meio de termos, uma definição PR em uma linguagem L2* **representa corretamente** *a relação de provabilidade, denominada* **demonstra**, *de uma linguagem L1 se e somente se: Sempre que X e Y são sentenças de L1, nomeadas respectivamente*

pelos termos X' e Y' de L2, a conclusão Y pode ser derivada a partir de X em L1 se e somente se a conclusão **demonstra(X',Y')** *pode ser derivada a partir de PR em L2."*

Note-se entretanto que a noção de representabilidade correta não requer que PR implique em **¬demonstra(X',Y')** em L2 quando X não implica Y em L1. Um estudo mais recente das aplicações da amalgamação de linguagens deve-se a Bowen e Kowalski **[BOW 82]**, notadamente na solução de certos pontos críticos da lógica clássica: sua monotonicidade e sua suscetibilidade ao *problema do frame*.

A lógica clássica é dita *monotônica* no sentido em que se aumentarmos qualquer conjunto de sentenças S com uma outra sentença s, nunca reduziremos e possivelmente aumentaremos o conjunto das consequências que podem ser extraídas. Simbolicamente podemos escrever:

$$\forall S\ \forall s\ \forall c\ (S \cup \{s\} \vdash c \text{ se } S \vdash c)$$

de onde se tira que a monotonicidade é uma propriedade da relação de provabilidade $\vdash$. As principais objeções à monotonicidade surgem usualmente em conexão com aplicações envolvendo modificações dinâmicas sobre BCs. Como um exemplo trivial, suponhamos que o estado corrente de uma BC, S, permita a inferência de alguma conclusão c. Dependendo do sistema de inferência utilizado, pode ser que c seja uma consequência lógica necessária de S ou que c seja tão somente inferida como plausível, na falta de de conhecimento contrário. Suponhamos que o sistema seja então informado que ¬c é verdadeira, por exemplo através da adição da sentença s = ¬c em S. Nesse novo estado da BC poderíamos desejar que c fosse agora não-inferível (isto é, desejar um comportamento não-monotônico), entretanto a lógica clássica não se comporta dessa maneira, uma vez que sua

monotonicidade insiste em inferir c a partir de $S \cup \{\neg c\}$. Na verdade, se $S \vdash c$, então a BC aumentada $S \cup \{\neg c\}$ é necessariamente inconsistente. Kowalski **[KOW 79]** propõe que as objeções à monotonicidade da lógica clássica poderiam ser superadas por meio do tratamento de eventuais inconsistências como consequências naturais e mesmo úteis de certos tipos de transições na evolução de uma BC. Em **[BOW 82]**, fortalecendo tal proposta, encontramos uma interessante ilustração do uso de linguagens amalgamadas no contexto de um sistema simplificado de gerenciamento de BDs.

Deve-se notar entretanto que os interpretadores existentes podem, em certos casos, adotar um comportamento não-monotônico. A negação por falha, por exemplo, é não-monotônica porque sempre é possível aumentar um conjunto de sentenças de forma a obter alguma conclusão c que originalmente não podia ser inferida. Neste caso a inferência de $\neg c$ deixa de ser obtida, isto é, $\neg c$ não é consequência de $S \cup \{c\}$, apesar de ser consequência de S.

O *problema do frame* pode também ser considerado como associado à representação e à evolução dinâmica das BCs. Para sua melhor compreensão utilizaremos um exemplo proposto em **[HOG 84]**: Vamos supor que desejamos representar em uma BC uma lista modificável dinamicamente, a qual em um dado instante poderia ser L = [a,b,c], representada pelo seguinte conjunto de assertivas:

```
item(a,1).
item(b,2).
item(c,3).      tamanho(3).
```

Suponhamos que agora desejamos adicionar à lista L um quarto elemento **d**, de forma que o próximo estado de L seja representado por:

```
item(a,1).
item(b,2).
item(c,3).
item(d,4).        tamanho(4).
```

O problema do frame surge na tentativa de descrever esse processo em lógica, devido à necessidade de nomear e relacionar o estado original e o novo estado. Assim, ao invés de escrever **item(U,I)**, escrevemos **item(U,I,S)** para expressar que S é o estado em que **U** é o I-ésimo elemento. Da mesma forma escrevemos **tamanho(S,N)** para expressar que o tamanho da lista é **N** no estado S. Se o estado resultante da adição de um elemento **V** à lista L for representado por **ext(L,V)**, este novo estado pode ser relacionado com o estado original por meio das seguintes sentenças:

item(U,I,ext(L,V)) se item(U,I,L).
item(V,I,ext(L,V)) se tamanho(L,I-1).
tamanho(ext(L,V),I) se tamanho(L,I-1).

A primeira das sentenças acima é um exemplo de um *axioma de frame:* ela serve para identificar os fatos da BC que são preservados durante a transição de estado. Problemas cujas formulações demandam grandes espaços de estados podem requerer a preservação de muitas classes de fatos. No exemplo dado declara-se que, se **U** é o I-ésimo elemento, então ele continua a sê-lo se a lista **L** for estendida para o novo estado.

Kowalski, em [KOW 79] identifica dois aspectos do problema do frame: primeiro, a multiplicidade de axiomas necessários em aplicações reais e, em segundo lugar, a dificuldade em controlá-los de modo eficiente. Para contornar tais obstáculos, sugere respectivamente o uso de um único axioma de frame capaz de acessar uma BC separada identificando todas as propriedades que devem ser preservadas sob os diversos tipos de transição de estados e a proposta de que tal axioma seja executado de forma top-down, tornando sua utilização imune à explosão combinatória de fatos que não se relacionam com objetivos que caracterizam raciocínio bottom-up.

A relação de provabilidade para a linguagem Prolog é dada de forma simplificada e em seu nível mais alto por meio do predicado **demo/2** definido pelas duas cláusulas abaixo [KOW 79]:

```
demo(Prog,Objetivos) :-
        vazio(Objetivos).
demo(Prog,Objetivos) :-
        seleciona(Objetivos,Obj,Resto),
        membro(Proc,Prog),
        renomeia(Proc,Objetivos,Proc1),
        partes(Proc1,Cab,Corpo),
        unifica(Obj,Cab,Sub),
        adiciona(Corpo,Resto,Inter),
        aplica(Inter,Sub,NovosObjs),
        demo(Prog,NovosObjs).
```

A primeira cláusula do predicado **demo/2** estabelece que *qualquer* programa demonstra a solubilidade de uma coleção *vazia* de objetivos. A segunda cláusula, interpretada de forma top-down declara que:

> "*Para* **demonstrar** *a solubilidade de uma coleção de objetivos, (1)* **selecione** *um objetivo, (2)* **encontre** *um procedimento apropriado no programa, (3)* **renomeie** *as variáveis presentes no procedimento de forma que estas sejam distintas das variáveis na coleção de objetivos, (4)* **unifique** *o objetivo selecionado com a cabeça do procedimento, (5)* **adicione** *o corpo do procedimento aos objetivos restantes, (6)* **aplique** *a substituição unificadora de forma a obter uma nova coleção de objetivos, e (7)* **demonstre** *que o programa soluciona a nova coleção de objetivos.*"

O procedimento de prova apresentado acima foi definido por meio de uma relação de dois argumentos entre suposições e conclusões. Na prática os procedimentos de prova tendem a ser mais complexos, envolvendo especificações de controle e a determinação da saída desejada. Dependendo portanto da

aplicação, pode ser preferível definir um procedimento de prova por meio de uma relação de quatro argumentos:

demo(X, Y, U, Z)

que será verdadeira quando, dado um conjunto de axiomas **X**, uma conclusão **Y** e uma especificação de controle **U**, o procedimento de prova produzir uma saída **Z**. O parâmetro de controle **U** pode especificar, por exemplo: (1) o método de prova a ser aplicado, (2) se é desejada uma, todas ou a "*melhor*" solução, ou (3) se em **Z** é desejada uma prova, o *trace* da pesquisa, a substituição para as variáveis na conclusão ou tão somente uma resposta do tipo sim/não.

O *trace* de um procedimento de prova consiste na sequência de sentenças e outras expressões geradas pelo procedimento de prova durante a pesquisa por uma solução. Assim, o procedimento de prova pode ser bem sucedido oferecendo como saída o *trace* de uma pesquisa, mesmo que esta tenha falhado. A relação **demo/4** é adequada à obtenção e ao processamento de [illegible] de uma [illegible] consulta, permitindo a saída de todas as respostas encontradas sob a forma de uma tabela **[KOW 79]**.

4.3 REFLEXÃO COMPUTACIONAL E METACONHECIMENTO

Reflexão Computacional é definida como sendo o comportamento exibido por um sistema reflexivo, onde *sistema reflexivo* é um sistema computacional que é executado sobre si próprio em *conexão causal*. Para substanciar tal definição passaremos a discutir alguns conceitos que com ela se relacionam, assim um *sistema computacional* (de agora em diante denominado

simplesmente *sistema*) é qualquer sistema capaz de ser executado em computadores cujo propósito é responder questões e/ou executar ações em algum domínio. Dizemos então que um sistema é *sobre* o seu domínio, e é constituído por estruturas representando entidades e relações sobre elas e por um programa que descreve como estas devem ser manipuladas. As computações ocorrem quando um processador (CPU ou um interpretador) se encontra executando tal programa, assim, qualquer programa em execução é um exemplo de um sistema computacional **[MAE 88]**. Um sistema é dito estar em *conexão causal* com seu domínio se as estruturas internas e o domínio que elas representam estão ligados de maneira tal que se uma mudança ocorrer em algum deles, isso conduz a um efeito correspondente sobre o outro. Um sistema controlador de um braço-robô, por exemplo, incorpora estruturas representando a posição do braço de maneira tal que:

(i) Se o braço-robô é movido por alguma força externa, tais estruturas são correspondentemente modificadas, e

(ii) Se alguma das estruturas é modificada (por computação), o braço-robô se movimenta para a posição correspondente.

Assim um sistema em conexão causal com seu domínio possui dele uma representação acurada e pode ocasionar modificações sobre ele como efeito de suas computações.

Um *sistema reflexivo* é um sistema que incorpora estruturas que representam o próprio sistema ou alguns de seus aspectos. O conjunto de tais representações é denominado *auto-representação*. Esta torna possível ao sistema responder questões e a executar ações sobre si próprio. Uma vez que a auto representação está em conexão causal com aspectos do sistema que representa, podemos dizer que:

(i) O sistema sempre possui uma representação precisa de si próprio, e

(ii) O estado e a computação do sistema estão sempre em correspondência com essa representação. (Isto significa que um sistema reflexivo pode originar modificações sobre si mesmo em virtude de suas próprias computações).

Muitos dos sistemas atuais exibem, além de sua *computação objeto*, isto é, computações sobre seu domínio exterior, algum tipo de *computação reflexiva*, ou seja, computações sobre si próprio. Como exemplos citamos a manutenção de estatísticas de desempenho, de informações para fins de depuração, mecanismos de *tracing*, interfaceamento (saídas gráficas entrada através de *mouse*, etc.), computação sobre a computação a ser executada a seguir (também denominada *raciocínio sobre o controle*), auto-otimização, auto-modificação (por exemplo, em sistemas de aprendizado) e auto-ativação (por exemplo em monitores e *demons*). A computação reflexiva em geral não contribui diretamente para o processo de solução de problemas sobre o domínio externo, ao invés disso contribui para a organização interna do sistema ou ao seu interfaceamento com o mundo exterior. Seu objetivo é portanto garantir a efetividade e o funcionamento adequado da computação objeto.

Uma linguagem de programação que possua uma *arquitetura reflexiva* toma a *reflexão* como um conceito fundamental e oferece mecanismos para a manipulação explícita de computações reflexivas. Concretamente isso significa que:

(i) O interpretador de tal linguagem deve oferecer a qualquer sistema em execução, acesso à sua própria representação. Os sistemas implementados em tal linguagem tem assim a possibilidade de executar computações reflexivas através da inclusão de código descrevendo como essa representação deve ser manipulada, e

(ii) O interpretador deve também garantir a conexão causal entre a representação e os aspectos do sistema que ela representa. Consequentemente, as modificações que a execução do sistema originam sobre sua auto-representação se refletem no seu próprio estado e computação.

Segundo **[MAE 88]** as arquiteturas reflexivas oferecem uma nova abordagem ao estudo dos sistemas computacionais. Em uma arquitetura reflexiva, um sistema computacional é visto como composto por uma parte objeto e uma parte reflexiva. A tarefa das computações objeto é solucionar problemas e oferecer informações sobre algum domínio externo, enquanto que as computações a nível reflexivo solucionam problemas e oferecem informações sobre as computações objeto. Sob diversos aspectos o conceito de arquitetura reflexiva corre em paralelo com o conceito de arquitetura em nível meta. Uma *meta-arquitetura* é projetada para suportar meta-computações, isto é, raciocínio ou ação executados por um sistema computacional (meta) sobre outro (objeto), entretanto esse conceito nem sempre engloba o de computação reflexiva, por exemplo, quando o meta-sistema e o sistema objeto são implementados em linguagens diferentes ou quando o meta sistema possui somente acesso estático sobre o sistema objeto.

Segundo [COS 88], a noção de *metaconhecimento* é reconhecidamente um conceito essencial para a engenharia de conhecimento. Definimos metaconhecimento como sendo conhecimento

sobre conhecimento, representado através de *metaprogramas* (fatos e regras) que operam sobre domínios tais como teorias, regras e provas. Isso é essencialmente obtido pela representação de programas como dados. A *amalgamação* da linguagem objeto com a metalinguagem requer a representação de todas as entidades sintáticas da linguagem (objeto) como estruturas (termos) da (meta) linguagem e a definição de um interpretador para a linguagem objeto (demo) na metalinguagem (meta-interpretador). O meta-interpretador é uma representação em nível meta da relação de provabilidade da linguagem objeto. A amalgamação é uma extensão conservativa, isto é, tudo o que pode ser feito na linguagem amalgamada pode também ser feito somente na metalinguagem ou na linguagem objeto, entretanto:

(i) A amalgamação é mais expressiva do que a linguagem objeto, uma vez que permite a definição de cláusulas que combinam relações em nível objeto com relações em nível meta, e

(ii) O meta-interpretador pode ser estendido com novos mecanismos e novas regras de inferência. Isso permite uma definição mais simples dos procedimentos desejados e oferece suporte a novos mecanismos da linguagem com as correspondentes regras de inferência.

No caso das linguagens lógicas, as cláusulas são representadas como termos e a lógica das cláusulas de Horn é empregada simultaneamente como linguagem objeto e metalinguagem. Conceitos de metaprogramação são parcialmente suportados em algumas implementações Prolog através de um conjunto adequado de (meta) predicados primitivos. A potencialidade da metaprogramação nas linguagens lógicas é demonstrada pela estrutura do meta-interpretador de três linhas:

```
demo(true) :- !.
demo((Q1,Q2)) :- !, demo(Q1), demo(Q2).
demo(Q) :- clause(Q,Corpo), demo(Corpo).
```

O meta-interpretador pode ser definido de maneira tão simples e curta porque a maior parte do trabalho é realizada pelo interpretador objeto subjacente. O meta-interpretador não define explicitamente os dois procedimentos computacionais mais complexos, isto é, o *backtracking* e a unificação. O mecanismo de backtracking está na realidade implícito na terceira cláusula, enquanto que a unificação é totalmente realizada a nível objeto. Meta-interpretadores mais elaborados se tornam interessantes uma vez que permitem a definição de novos mecanismos sem modificar o programa (conhecimento a nível objeto) e o interpretador subjacente (a máquina de inferências). Podemos por exemplo definir novas estratégias de controle **[PER 82]**, estender a linguagem lógica com novas construções (por exemplo, estruturar o conhecimento **[FUR 84]** ou lidar com incerteza **[SHA 83]**), etc.

Uma das principais características da metaprogramação é a capacidade de estender a linguagem, o mecanismo de inferência e o próprio ambiente sem modificar o interpretador ou compilador Prolog subjacente. As extensões (metaprogramas Prolog) são de fácil definição e elevada portabilidade, ainda que seu desempenho possa resultar algo inferior ao que seria apresentado por uma implementação completa da nova linguagem ou ambiente.

Extensões a nível meta do paradigma básico da programação em lógica de primeira ordem foram pesquisadas entre outros por Bowen **[BOW 82]**, **[BOW 85]** e **[BOW 86]**, e Monteiro e Porto **[MON 88]**. Em **[BOW 85]** é proposta uma extensão a nível meta da linguagem Prolog que inclui o tratamento de *teorias* (BDs) e *nomes* em nível meta como *objetos de primeira classe*, isto é, capazes de assumir valores de variáveis. Um modelo semelhante,

porém com uma semântica declarativa de mundos possíveis é proposto por Monteiro e Porto **[MON 88]** em um sistema denominado pelos autores *programação em lógica contextual.* Nas sessões seguintes examinaremos as principais características desses dois sistemas.

4.4 META PROLOG

O emprego de primitivas como **assert/retract** em Prolog reflete uma necessidade real da programação em lógica em muitas aplicações da inteligência artificial, como por exemplo a necessidade de segmentar ou modularizar a BC expressa em cláusulas para os mais diversos propósitos. Por outro lado, a introdução de tais primitivas não obedece a semântica pura da programação em lógica. Em muitas aplicações, uma segmentação estática da base de dados seria inadequada, uma vez que os segmentos da base de dados podem necessitar modificações, ou mesmo surgir e desaparecer ao longo da execução de um programa. O sistema **metaProlog** proposto por Bowen em **[BOW 85]** adota a abordagem de ascender a um ponto de vista a *nível meta.* Simplificadamente isto consiste em uma axiomatização de primeira ordem em nível meta do nível objeto da teoria da prova. Isto é, o sistema de programação axiomatizaria as noções de teoria e prova para alguma linguagem a nível objeto, permitindo teorias (conjuntos de fórmulas) existirem como objetos de primeira classe, no sentido de poderem ser valores de variáveis e assim serem transmitidas como valores de procedimentos. Em particular, poderiam ser criadas ou destruídas dinamicamente e novas (modificadas) teorias poderiam ser geradas a partir de teorias previamente existentes. A conexão básica entre teorias, fórmulas e provas é fornecida pelo predicado de prova:

demo(Teoria, Fórmula, Prova)

que vale quando **Prova** é uma prova (no sistema lógico axiomatizado) de **Fórmula**, baseada nos axiomas de **Teoria**.

Na visão de Bowen, para explorar o potencial completo da abordagem em nível meta, devemos evitar a estratificação em níveis (objeto, meta, meta-meta, etc) que uma visão mais simplista poderia acarretar. Para conseguir isso devemos efetivamente amalgamar a linguagem objeto básica com o nível meta mais elevado. Uma abordagem que permite isso é estender (conservativamente) uma linguagem do cálculo de predicados de primeira ordem para outra na qual conjuntos finitos de fórmulas são representados por termos e onde cada item sintático (de variáveis a conjuntos de fórmulas) é nomeado por uma constante da linguagem. Note-se que essas constantes participam em outros termos e fórmulas, e estas são tambem, posteriormente nomeadas por outras constantes. A relação

nome_de(Nome,Item)

é incorporada como parte do mecanismo de prova básico de primeira ordem da linguagem estendida. Isso é obtido por meio do processo de estender uma linguagem pela adição de constantes para nomear todos os seus itens. Adicionalmente todas as assertivas **nome_de** são adicionadas a cada iteração. A extensão é conservativa no sentido de que, se A é uma fórmula expressa na linguagem original e A é vista ser provável na extensão, então, segue que A é provável no sistema original.

4.4.1 A Natureza do metaProlog

Numa primeira axiomatização (definicional), mostrada na **Figura 4.1**, a estrutura compartilhada é expressa tornando explícitas as substituições necessárias como argumentos aos predicados apropriados.

```
demo(Teoria, Objetivo, [], Subst, Subst) :-
   vazio(Objetivo).

demo(Teoria, Objetivo, [Premissa | Resto_da_Prova],
   In_Subst, Out_Subst) :-
     select(Objetivo, Sub_Objetivo, Resto_Objetivos),
     react(Teoria, Sub_Objetivo, Premissa,
        In_Subst, Inter_Subst, Cont_Objetivos),
     merge(Cont_Objetivos, Resto_Objetivos, Novo_Objetivo),
     demo(Teoria, Novo_Objetivo, Resto_da_Prova,
        Inter_Subst, Out_Subst).

react(Teoria, demo(Nova_Teoria, Subsid_Objetivo, Subsid_Prova),
   sbs(Subsid_Prova), In_Subst, Out_Subst, true) :-
     demo(Nova_Teoria, Subsid_Objetivo, Subsid_Prova,
        In_Subst, Out_Subst).

react(Teoria, corrente(Teoria), corrente(Teoria), In_Subst,
   Out_Subst, true).

react(Teoria, Sub_Objetivo, s(Sub_Objetivo, Regra),
   In_Subst, Out_Subst, Corpo_da_Regra) :-
     find(Sub_Objetivo, Teoria, Regra),
     parts(Regra, Cabeça_da_Regra, Corpo_da_Regra),
     match(Sub_Objetivo, Cabeça_da_Regra, In_Subst, Out_Subst).
```

Figura 4.1

Axiomatização explícita do metaProlog

O predicado

match(Esquerda, Direita, In_Subst, Out_Subst)

é o unificador. Ele tenta estender a substituição de entrada para uma substituição que unifique os dois primeiros argumentos. São fornecidas duas formas especiais (predicados distinguidos) reconhecidos pelo sistema: **demo/3** e **corrente/1**. A primeira cláusula no predicado **react/6** permite chamadas recursivas ao predicado **demo**, enquanto que a segunda permite a uma dada cláusula determinar a teoria sob a qual está sendo executada.

Se assumirmos que as metavariáveis do sistema metaProlog sendo axiomatizado devem ser identificadas com as variáveis do Prolog subjacente, então podemos apresentar uma axiomatização mais compacta, como mostrado na **Figura 4.2**.

```
demo(Teoria, Objetivo, []) :- vazio(Objetivo).

demo(Teoria, Objetivo, [Premissa | Resto_da_Prova]) :-
   select(Objetivo, Sub_Objetivo, Resto_Objetivos),
   react(Teoria, Sub_Objetivo, Premissa, Cont_Objetivos),
   merge(Cont_Objetivos, Resto_Objetivos, Novo_Objetivo),
   demo(Teoria, Novo_Objetivo, Resto_da_Prova).

react(Teoria, demo(Nova_Teoria, Subsid_Objetivo, Subsid_Prova),
   sbs(Subsid_Prova), true) :-
      demo(Nova_Teoria, Subsid_Objetivo, Subsid_Prova).

react(Teoria, corrente(Teoria), corrente(Teoria), true).

react(Teoria,Sub_Objetivo,s(Sub_Objetivo,Regra),Corpo_da_Regra) :-
   find(Sub_Objetivo, Teoria, Regra),
   parts(Regra, Cabeça_da_Regra, Corpo_da_Regra),
   match(Sub_Objetivo, Cabeça_da_Regra).
```

Figura 4.2

Axiomatização implícita do metaProlog

O papel dos predicados assert/retract no Prolog ordinário "impuro" na modificação da única base de dados global é substituído pela construção de novas teorias, distintas das antigas, através do uso de duas relaçõec primitivas embutidas, como mostrado na **Figura 4.3**.

```
add_to(<Teoria_Anterior>, <Assertiva>, <Nova_Teoria>)

drop_from(<Teoria_Anterior>, <Assertiva>, <Nova_Teoria>)
```

Figura 4.3

Relações **add_to** e **dro'_from**

De um ponto de vista lógico, <Nova_Teoria> é obtida inicialmente através de uma cópia de <Teoria_Anterior> que é então modificada de maneira apropriada. Do ponto de vista da linguagem de programação, entretanto, uma cópia real é geralmente um preço muito alto a pagar (apesar de ser necessária em certas circunstâncias). Ao invés disso, a implementação real deve fazer parecer que tal cópia foi realizada, quando, na verdade, tal não acontece. Isso pode ser conseguido descbevendo-se a nova teoria em função da anterior e vice-versa. Como é o caso do clássico problema do frame, o acesso aos axiomas da teoria representada como uma descrição torna-se mais lento a medida que o número de modificações aumenta. Uma vez que na aplicação típica de tais recursos (add_to e drop_from) é a nova tuoria que deverá ser mais pesadamente acessada, a decisão de projeto para o metaProlog tem sido a de garantir acesso mais rápido às novas teorias. Isso é obtido fazendo a <Teoria_Anterior> ser preferencialmente descrita em termos de <Nova_Teoria> ao invés do contrário.

A interpretação pretendida para a extensão proposta em nível meta do Prolog é de uma axiomatização de primeira ordem de teorias e provas em linguagem amalgamada. No uso comum da lógica, uma axiomatização formal de primeira ordem tem uma interpretação pretendida *fora* da lingeagem. Por exemplo, a interpredação pretendida da axiomatização da geometria de Hilbert era um mundo de figuras geométricas, enquanto que a interpretação pretendida da aritmética de Peano era um mundo de números. Nestes casos, é um efeito posterior quase acidental do processo formal que interpretações (de Herbrand) estruturadas por elementos linguísticos possam ser construídas. Em contraste a interpretação pretendida de uma extensão formal do Prolog em nível meta está em um mundo de linguagens: pretende-se que o meta sistema discorra acerca de teorias e provas de uma linguagem.

Uma possibilidade semântica alternativa é fornecer ao sistema a semântica modal de Hintikka-Kripke. Em ambos os casos a interpretação dos nomes é a de nomes próprios funcionando como

designadores rígidos, isto é, se <n> nomeia uma certa teoria <t>, ele irá apontar para alguma manifestação física de <t>. Se uma operação **add_to(<t>,<a>,<t2>)** é executada, <n> subsequentemente apontará para a manifestação física de <t2>. Por exemplo, assumindo que 'João da Silva' se refere a uma certa pessoa (física), então, após uma cirurgia para a remoção da vesícula biliar de João da Silva, o nome 'João da Silva' se referirá a pessoa física resultante da operação.

4.4.2 Sintaxe do metaProlog

A sintaxe superficial do metaProlog é bastante similar a do Prolog de Edimburgo, com ':-' substituído por '<-' e a conjunção indicada por '&' ao invés da vírgula. A diferença mais significativa é a necessidade de se tornar explícita toda quantificação, ao invés de deixá-la implícita por meio de uma convenção. Tal requisito vem de 2 pontos:

- Do desejo de permitir a manipulação de teorias parcialmente instanciadas, que seguiria do princípio de que as mesmas devem ser tratadas como objetos de primeira classe completos, e
- Do desejo de fornecer uma semântica correta para **add_to** e **drop_from**.

A dificuldade com relação ao segundo ponto pode ser visualizada se considerarmos as duas cláusulas do Prolog de Edimburgo mostradas na **Figura 4.4**.

```
h :- X = a, assert(p(X)), p(b).
h :- assert(p(X)), X = a, p(b).
```

Figura 4.4

Uma contradição da semântica declarativa

No Prolog convencional, a primeira cláusula falha, enquanto que a segunda é bem sucedida, uma vez que o Prolog assume que **assert(p(X))** significa que "*se X não estiver instanciada, adicione 'para_todo[X]:p(X)' à base de dados*". Entretanto, a leitura lógica das cláusulas apresentadas na **Figura 4.4** trata a virgula como uma conjunção conectando os literais, e a conjunção lógica é comutativa, logo as duas cláusulas deveriam produzir o mesmo resultado. No metaProlog, se o programador desejar atingir o efeito ocasionado pelo Prolog convencional, ele deve indicar explicitamente a quantificação:

...& add_to(t1, all[x]:p(x), t2) &...

Se, por outro lado, 'x' é quantificado em algum lugar em uma cláusula, mas em tempo de execução a (meta)variável do interpretador que substituirá x ainda não foi instanciada, o efeito da chamada:

...& add_to(t1, p(x), t2) &...

seria o de adicionar a acsertiva parcialmente instanciada **p(x)** a **t1**, construindo **t2** como uma teoria parcialmente instanciada. Processamento posterior causaria a instanciação de **x**, resultando em uma especificação mais precisa da teoria **t2**. Uma vej que a quantificação deve ser explicitamente representada em metaProlog e, consequentemente, nenhuma convenção para a quantificação de variáveis é utilizada, todos os identificadores começando com uma letra (maiúscula ou minúscula) são agora constantes.

4.4.3 Teorias e Nomec

Uma vez que teorias são objetos de primeira classe, elas podem participar em cláusulas da mesma forma que constantes e outros termos. Assim, uma teoria **T1** pode conder uma assertiva como **útil(T2)** sobre outra teoria **T2**. Considere por exemplo o problema de definir um sistema de gerenciamento para uma base de dados. Uma forma simplificada de tal sistema poderia rodar como mostrado na **Figura 4.5.** O gerenciador da base de dados é representado pelo predicado **dbm.** O primeiro argemento de **dbm** é uma teoria representando o estado corrente da base de dados, enquanto que o segundo argumento é uma lista de comandos a ser processada contra a base de dados. Como se pode notar, **dbm** é um procedimento recursivo simples, onde todo o trabalho é na realidade realizado pelo predicado **process**. O primeiro argumento de **process** é o comando a ser processado contra a base de dados corrente, que é fornecida como o segundo argumento de **process**, enquanto que o estado resultante da base de dados após o processamento do comando é representado pela teoria dada no terceiro argumento de **process**. A primeira cláusula de **process** lida com simples recuperação de informações (que inclui informação implícita, dedutível da base de dados), enquanto que as tres últimas se encarregam da adição de novas assertivas e, dependendo da natureza das restrições de inserção e regras de revisão usadas na terceira cláusula, tambem podem tratar atualizações.

A segunda cláusula de **process** reflete uma filosofia parcimoniosa da parte deste particular gerenciador: nada é adicionado à base de dados se já está explícito ou se pode ser deduzido a partir da informação ali existente. Se esta é uma filosofia apropriada ou exatamente que quantidade de recursos deve ser consumida na tentativa de determinar se alguma coisa está implícita, é critério de cada particular gerenciador de bases de dados.

```
all[CurDB]:
    dbm(CurDB,[]).

all[CurDB,req,reqs,NewDB]:
    dbm(CurDB,[req|reqs]) ←
        process(req,CurDB,NewDB) &
        dbm(NewDB,reqs).

all[assertion,proof,CurDB]:
    process(retrieve(assertion,proof),CurDB,CurDB) ←
        demo(CurDB,assertion,proof).

all[Assertion,Proof,CurDB]:
    process(insert(Assertion,present(Proof)),CurDB,CurDB) ←
        demo(CurDB,Assertion,Proof).

all[Assertion,Response,CurDB,NewDB,IC,UnAcProof,RR,RevProof]:
    process(insert(Assertion,revision(Response)),CurDB,NewDB) ←
        demo(CurDB,insert_constraints(IC),_) &
        demo(IC,unacceptable(Assertion,CurDB),UnAcProof) &
        demo(CurDB,revision_rules(RR),_) &
        demo(RR,revise(UnAcProof,CurDB,NewDB),RevProof) &
        Response = [Assertion,UnAcProof,BevProof].

all[assertion,CurDB,NewDB]:
    process(insert(assertion,done),CurDB,NewDB) ←
        add_to(CurDB,assertion,NewDB).
```

Figura 4.5

O predicado **dbm**

A quarta cláusula é um *default*: se nenhuma das outras cláusulac for aplicável a uma adição proposta, então a assertiva deve ser adicionada à base de dados. A terceira cláusula incorpora a possibilidade de gerenciamento de restrições de integridade e manutenção de raciocínio. Ela tambem ilustra a flexibilidade potencial do uso de teorias. Neste caso a base de dados possui um conjunto de restrições de integridade na forma de uma assertiva:

insert_constraints(IC)

onde IC pretende ser outra teoria abrangendo as retrições de integridade a ser utilizadas no gerenciamento da base de dados. É assumido que IC contem cláusulas definindo o predicado **unacceptable**. Se a adição proposta pode ser provada inaceitável

sob as regras embutidas em IC, então a base de dados tenta ainda outra teoria, **RR**, através da assertiva:

revision_rules(RR)

Estas cláusulac definem o predicado **revise**, o qual usando a informação obtida da prova de inaceitabilidade da assertiva proposta, revisa a base de dados. Isso pode variar de uma simples rejeição da atualização proposta a complexas atividades de manutenção de raciocínio. Os registros necessários à manutenção do raciocínio podem ser representados como teorias e armazenados na base de dados da mesma fobma que as assertivas. Restrições de inserção muito "naturais" podem ser expressas, conforme mostrado na **Figura 4.6**.

```
all[pessoa,quant,quant1]:
     unacceptable(salario(pessoa,quant),DB) ←
          demo(DB,salario(pessoa,quant1),_) &
          quant =/= quant1.
ou:
all[pessoa,quant,quant1,Lim,perc_inc]:
     unacceptable(salario(pessoa,quant),DB) ←
          demo(DB,salario(pessoa,quant1),_) &
          demo(DB,limite(Lim),_) &
          perc_inc(quant1,quant,perc_inc) &
          perc_inc > Lim.
```

Figura 4.6

Restrições de inserção

Podemos implementar **demons** na base de dados de maneira similar, descrevendo-os em uma subteoria de **CurDB** e adicionando condições apropriadas em cláusulas apropriadas do predicado **process**. Note que uma vez que cada base de dados possui suas próprias restrições, regras de revisão e **demons**, estes tambem podem ser modificados, se apropriado, como sugerido na **Figura 4.7**.

```
... &
demo(CurDB,insert_constraints(IC),_) &
modify(IC, ...,NewIC) &
drop_from(CurDB,insert_constraints(IC),InterDB) &
add_to(InterDB,insert_constraints(NewIC),NewDB) & ...
```

Figura 4.7

Modificando restrições

Outra aplicação interessante mostra que teorias podem ser organizadas em estruturas de árvores, de maneira similar a de muitos sistemas operacionais. Suponha que **'subteoria(...)'** seja tomado como um predicado distinguido. Sejam **raiz, t1, t2** e **t3** teorias contendo pelo menos as cláusulas indicadas abaixo:

```
raiz:
        subteoria(t1).
        subteoria(t2).
        superteoria(nil).
        ...
t1:
        subteoria(t3).
        superteoria(raiz).
        ...
t2:     ...
t3:     ...
```

Além disso, assuma que as cláusulas para o predicado **find** (da definição de **demo/react**) contem as seguintes cláusulas entre suas primeiras entradas:

```
find(Obj,U/V,Regra) :-
        demo(U,subteoria(V),_),
        find(Obj,V,Regra).

find(Obj,'..'/V,Regra) :-
        corrente(Teoria),
        demo(Teoria,superteoria(U),_),
        demo(U,subteoria(V),_),
        find(Obj,V,Regra).
```

Estas cláusulas permitiriam chamadas tais como **demo(raiz/t1/t3,Obj,Pr)** serem efetivamente equivalentes a **demo(t3,Obj,Pr)**.

Conforme anteriormente indicado, *nomes* referem-se ou apontam para o (item sintático) que nomeiam. Uma vez que eles são constantes em nosso sistema, eles são tambem objetos de primeira classe e portanto podem aparecer em assertivas. Assim, um dos usos mais comuns dos nomes é formular assertivas sobre cláusulas. Isso é talvez mais naturalmente obtido pelo armazenamento de cláusulas de uma teoria, digamos **t1**, em outra teoria, digamos t2. Assim, se quisermos associar fatores de confiança às assertivas de **t1**, podemos fazê-lo conforme indicado na **Figura 4.8.**

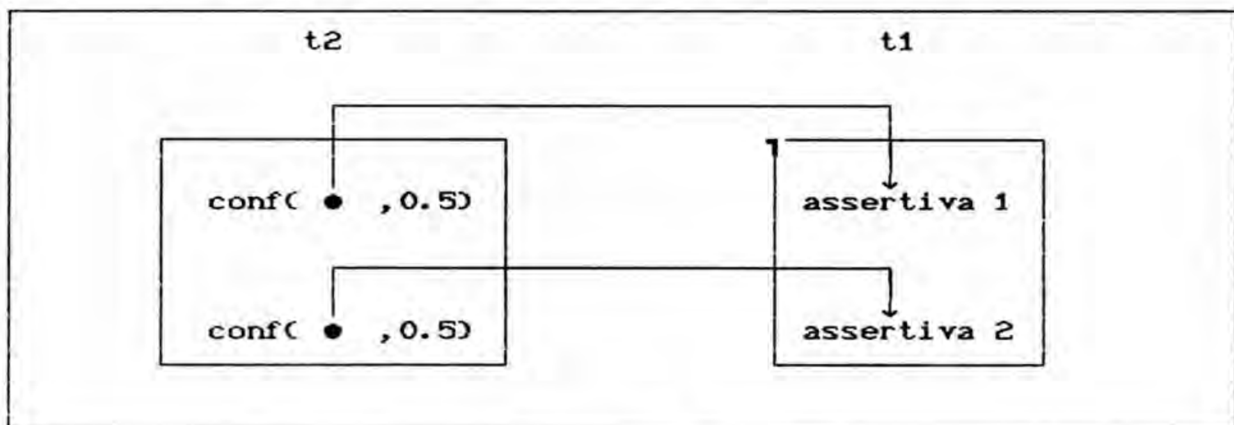


Figura 4.8
Associando fatores de confiança às assertivas de uma teoria

Entre os possíveis benefícios de tal abordagem, relacionamos a habilidade de associar simultaneamente diferentes fatores de confiança (usando **t2, t3,** etc) ao mesmo conjunto de assertivas (t1), e a habilidade de modificar os fatores de confiança associados às assertivas de **t1** (usando **drop_from** e **add_to** para mudar t2 paba t2', etc). Outra aplicação desta técnica é a representação de informação em um sistema de manutenção de raciocínio. Por exemplo, no contexto do gerenciador de bases de dados simplificado apresentado anteriormente, a base de dados básica **db** poderia sempre conter uma assertiva:

justificativas(raciocínios)

onde **raciocínios** é uma teoria que registra as justificativas para cada item adicionado à base de dados. As entradas em **raciocínios** seriam assertivas sobre as cláusulas em **db** e sobre deduções (ou justificativas *default*) justificando a presença da assertiva na base de dados. As reais assertivas em **raciocínios** seriam fatos cujos argumentos seriam nomes de cláusulas na base de dados, nomes de provas, termos0 representando a aplicação de regras *default*, etc. Tal informação é na realidade conhecimento em nível meta sobre a base de dados e as facilidades de um sistema em nível meta, tal como o metaProlog oferecem meios poderosos e flexíveis para a expressão e manipulação de tal conhecimento.

4.5 PROGRAMAÇÃO EM LÓGICA CONTEXTUAL

4.5.1 Teoria Básica da Programação em Lógica Contextual

Além dos conjuntos **Pred, Fun** e **Var** de nomes de predicados, funções e variáveis, é necessário definir um conjunto **Un** de nomes de unidades. Tais conjuntos são assumidos como finitos ou enumeráveis. **A cada u ∈ Un** está associado um subconjunto finito de **Pred,** denominado o sort **de u** e denotado por ▯**u**▯.

Termos, fórmulas atômicas e cláusulas são definidos no modo usual, exceto que cláusulas podem conter uma *fórmula extensional*. A sintaxe para fórmulas extensionais é **u >> G** onde **u** é o nome de uma unidade e **G** é um conjunto finito de fórmulas

atômicas ou extensionais, interpretadas como a *conjunção* de seus elementos. Se g e G são respectivamente uma fórmula atômica ou extensional e uma conjunção, escrevemos (g,G) ao invés de {g} ∪ G. A fórmula *nula* é denotada por Δ, representando *true*. Se g é uma fórmula atômica **p(t1,...,tn)**, escreveremos algumas vezes **nome(g)** para o nome do predicado de **g**, isto é, **p**. Por uma **p-cláusula** deve ser entendida uma cláusula **h** ← G tal que **nome(h)=p**. Diremos que em conjunto de cláusulas define **p** se o conjunto contém alguma **p-cláusula**.

Uma **unidade** é uma fórmula do tipo **u:U**, onde **u** ∈ **Un** e **U** é um conjunto finito de cláusulas tal que o conjunto de predicados definidos em **U** é ⟦u⟧. Denominamos **u** o *nome* da unidade e **U** o seu *corpo*. Um *sistema de unidades* é um conjunto $\mathcal{U}$ de unidades tal que duas unidades distintas em $\mathcal{U}$ nunca possuem o mesmo nome. Para uma unidade em $\mathcal{U}$ com o nome **u**, denotaremos seu corpo por $|u|_{\mathcal{U}}$, ou simplesmente $|u|$ no caso de $\mathcal{U}$ ser subentendido. Em consequência, frequentemente abusaremos da linguagem e nos referiremos a **u** como uma unidade em $\mathcal{U}$, quando na realidade queremos nos referir a unidade **u:**$|u|$.

Um *programa* é visto sintaticamente como um conjunto finito de unidades, como capturado pelo conceito de um sistema de unidades> A visão dinâmica de um programa corresponde à derivação de fórmulas em contextos, para cuja definição nos voltamos agora. *Nomes de contextos* são definidos como uma sequência arbitrária de nomes de unidades, que pretendem registrar a história de formação dos contextos. Assim, o conjunto de nomes de contextos é $\mathbf{Cn}=\mathbf{Un}^{*}$. Representamos nomes de contextos por justaposição. Por exemplo, se **u,v** ∈ **Un**, **uv** ∈ **Cn**. A sequência vazia, λ, é o nome do *contexto vazio*. O contexto resultante da extensão do contexto **c** com a unidade **u** é **uc**.

4.5.2 Derivação Top-Down

Para todo nome de contexto c e fórmula G, denotamos por $c \vdash_{\mathcal{U}} G[\theta]$ o fato de que há uma *derivação top-down* de G em c a partir de $\mathcal{U}$ com a substituição θ. O símbolo ε é reservado para denotar a substituição vazia (identidade). O resultado da aplicação de θ em **G** é escrito $G\theta$. Em consequência podemos escrever $\vdash$ ao invés de $\vdash_{\mathcal{U}}$ se $\mathcal{U}$ estiver subentendido. A *relação de derivação top-down* $\vdash$ é definida por regras no formato:

$$\frac{\textbf{Hipóteses}}{\textbf{Conclusão}} \quad \textbf{Condições} \ ,$$

assegurando a **Conclusão,** sempre que forem válidas as **Hipóteses** e as **Condições**. Assim, $\vdash_{\mathcal{U}}$ é a **menor relação** que satisfaz as seguintes regras:

- **Fórmula Nula:**

$$\frac{}{c \vdash \Delta[\varepsilon]}$$

A fórmula nula é derivável de qualquer contexto com substituição vazia. ∎

- **Conjunção:**

$$\frac{c \vdash g[\theta] \qquad c \vdash G\theta[\sigma]}{c \vdash (g, G)[\theta\sigma]}$$

Para derivar uma conjunção não-vazia, derivamos um elemento de cada vez. ∎

▸ Fórmula Atômica (I):

$$\frac{uc \vdash G\theta[\sigma]}{uc \vdash g[\theta\sigma]} \qquad h \leftarrow G \text{ in } |u|,\ \theta = \text{umg}(g,h)$$

Para derivar uma fórmula atômica em um contexto cuja unidade mais recente define o respectivo predicado, reduzimos a fórmula na unidade e derivamos o corpo da cláusula usado na redução. (Naturalmente, a cláusula h ← G é uma variante de uma cláusula em |u| com nenhuma variável em comum com g.) ■

▸ **Fórmula Atômica (II):**

$$\frac{c \vdash g[\theta]}{uc \vdash g[\theta]} \qquad \text{nome}(g) \notin \|u\|$$

Se o nome de uma fórmula atômica não possui definições na mais recente unidade, derivamos a fórmula no contexto imediatamente precedente. ■

▸ **Fórmula de Extensão:**

$$\frac{uc \vdash G[\theta]}{c \vdash u \gg G[\theta]}$$

Para derivar uma fórmula extensional, derivamos a fórmula *interior* no contexto extendido com a unidade mencionada na fórmula extensional. ■

Dado um sistema de unidades $\mathcal{U}$, a *semântica operacional* de uma fórmula g, conforme determinado pela relação de derivação *top-down* é o conjunto de todas as substituições θ tais que $\lambda \vdash_{\mathcal{U}} g[\theta]$, onde λ é o nome do contexto vazio. Note que esse conjunto é vazio a menos que **g** tenha a forma **u >> G** para alguma unidade em $\mathcal{U}$ com o nome **u**.

4.5.3 Derivação Bottom-Up

A semântica operacional da programação em lógica em geral e da proposta de extensão contextual em particular, está baseada na noção de derivação *top-down*. A noção de derivação *bottom-up* é mais útil em conexão com a semântica declarativa. O símbolo $\vdash\!\!\!\urcorner_{\mathcal{U}}$ denota derivação *bottom-up* a partir de $\mathcal{U}$ e, assim como para a derivação *top-down*, escrevemos simplesmente $\vdash\!\!\!\urcorner$, no caso de $\mathcal{U}$ estar subentendido. Somente necessitamos definir c $\vdash\!\!\!\urcorner$ g para um contexto c e uma fórmula atômica ou extensional g, uma vez que temos c $\vdash\!\!\!\urcorner$ G se e somente se c $\vdash\!\!\!\urcorner$ g para todo elemento g $\in$ G. Para um dado conjunto **U** de cláusulas, seja **Inst(U)** a denotação do conjunto de todas as instâncias de **U** (possivelmente não-básicas). A relação $\vdash\!\!\!\urcorner$ é a menor relação que satisfaz as seguintes condições:

(i) **Se uc $\vdash\!\!\!\urcorner$ g para todo g$\in$G e (h$\leftarrow$G)$\in$Inst(|u|), então uc $\vdash\!\!\!\urcorner$ h.**

(ii) **Se c $\vdash\!\!\!\urcorner$ g e nome(g) $\notin$ [[u]], então uc $\vdash\!\!\!\urcorner$ g.**

(iii) **Se uc $\vdash\!\!\!\urcorner$ g para todo g$\in$G, então c $\vdash\!\!\!\urcorner$ u >> G.**

O relacionamento entre as derivações *top-down* e *bottom-up* é descrito pelo seguinte resultado:

▸ **Teorema 4.1**

1) *Se* c $\vdash\!\!\!\urcorner$ g_0 *e* g_0 *é uma instância de* g, *então há uma substituição* θ *tal que* c $\vdash$ g[θ] *e* g_0 *é instância de* $g\theta$.

2) *Se* c $\vdash$ g[θ], *então* c $\vdash\!\!\!\urcorner$ $g\theta$. ■

UFRGS
INSTITUTO DE INFORMÁTIC
BIBLIOTECA

4.5.4 Semântica Declarativa

Uma interpretação de uma linguagem de primeira ordem consiste em um conjunto não vazio D, denominado o *domínio* da interpretação, juntamente com uma atribuição para cada símbolo funcional n-ário de uma função de D^n para D, e uma atribuição para cada símbolo predicativo de um subconjunto de D^n. Extenderemos essa noção fornecendo uma interpretação adequada para nomes de unidades. Para simplificar, restringiremos nossa discussão às consequências das interpretações de Herbrand, onde o domínio e a interpretação de símbolos funcionais são definitivamente determinados. O domínio de uma interpretação de Herbrand é o universo de Herbrand **H**, o conjunto de todos os termos **t** construídos com os símbolos funcionais em **Fun**. Assumimos que há pelo menos um símbolo funcional nulário, para tornar **H** não-vazio. Cada símbolo funcional **f**, de aridade **n**, é interpretado como uma função de H^n em **H**, que mapeia qualquer sequência de **n** termos **(t1,...,tn)** em um termo **f(t1,...,tn)**. Uma interpretação de Herbrand é então uma atribuição de um subconjunto de H^n a cada símbolo predicativo de aridade **n**. Costuma-se identificar tal interpretação com o conjunto de todas as fórmulas atômicas básicas **p(t1,...,tn)** tal que **(t1,...,tn)** está no subconjunto de H^n atribuído a **p** pela interpretação. Assim, uma interpretação de Herbrand é essencialmente um subconjunto **I** da base de Herbrand **B**, a qual é o conjunto de todas as fórmulas atômicas básicas. O conjunto de todas as interpretações de Herbrand é $\wp$(B), o conjunto potência de B.

Em um sistema de programação em lógica contextual necessitamos considerar diversos subconjuntos da base de Herbrand simultaneamente, aqui denominados **situações**. A razão é que é preciso interpretar, em uma situação, declarações que se referem a outras situações. Por exemplo, uma fórmula extensional **u >> G** é verdadeira em uma situação S, se toda fórmula em G é verdadeira na situação obtida de S de acordo com as especificações contidas em |**u**|.

Assim, a um símbolo predicativo não é, em geral, atribuído um único predicado, uma vez que isso depende da situação sob consideração. Os comentários anteriores tambem sugerem que nomes de unidades deveriam denotar operações de transformação de situações, isto é, funções de $\wp(B)$ em $\wp(B)$. Como uma primeira abordagem, então, uma interpretação de um sistema de unidades é uma atribuição, para cada nome de unidade, de uma transformação de $\wp(B)$. Podemos interpretar esta semântica em termos de um modelo de mundos possíveis, onde situações são vistas como mundos possíveis e unidades como especificações de transformações entre mundos.

A situação S', na qual S é transformada pela denotação de **u** é determinada inicialmente encontrando-se os predicados definidos por **u** na situação S, e então atualizando S com estes predicados. A denotação de **u** em uma interpretação I é então uma função u_I que, quando aplicada a uma situação S, fornece os predicados definidos por **u** naquela situação. A situação resultante S' é a atualização de S por $u_I(S)$.

Para $P \subseteq$ **Pred** e $S \subseteq$ **B**, a *restrição* de S para P é o conjunto:

$$S\lceil P = \{p(t1,\ldots,tn) \in S:\ p \in P\}$$

Escrevemos $S\lceil -P$ ao invés de $S\lceil(\mathbf{Pred}-P)$, e, para simplificar a notação, abreviamos $S\lceil P$ para S_P e $S\lceil -P$ para S_{-P}. Se **U** é um conjunto de cláusulas, **base(U)** é o conjunto de todas as instâncias básicas de cláusulas em **U**. Uma função $\mathbf{f}:\wp(X)\rightarrow\wp(Y)$ é contínua se, para todo $W\subseteq X$ e $y\in Y$, $y\in f(W)$ se e somente se há um subconjunto finito W_0 de W tal que $y\in f(W_0)$. O conjunto de todas as funções contínuas de $\wp(X)$ em $\wp(Y)$ é denotado por $[\wp(X)\rightarrow\wp(Y)]$.

Uma *interpretação* I é uma atribuição, para cada **u**∈**Un**, de uma função contínua:

$$u_I: \wp(B_{-[\![u]\!]}) \rightarrow \wp(B_{[\![u]\!]}).$$

A *atualização* de S ⊆ B por **u** ∈ **Un**, de acordo com I é:

$$S[u_I] = S_{-[\![u]\!]} \cup u_I(S_{[\![u]\!]}).$$

Assim a situação atualizada consiste nos predicados não definidos por **u**, juntamente com os predicados redefinidos por **u**. Note que a *situação atualizada por* **u** é uma função contínua de $\wp(B)$ em $\wp(B)$.

Dada uma situação S, uma fórmula **f** e uma interpretação I, denotamos por:

$$S \models_I \mathbf{f},$$

o fato que f *é verdadeira em* S *com respeito a* I. A relação $\models_I$ é definida pelas seguintes cláusulas dependendo da forma de **f**:

- S $\models_I$ **F**, onde **F** é um conjunto de fórmulas, se e somente se S $\models_I$ **f**, para toda **f** ∈ **F**.

- S $\models_I$ **u:U** se e somente se $S[u_I] \models_I$ **U**.

Uma unidade é verdadeira, em uma dada situação, se todas as cláusulas no corpo da unidade são verdadeiras na situação atualizada pela denotação do nome da unidade. ■

- $S \models_I g \leftarrow G$ se e somente se $S \models_I g_0 \leftarrow G_0$ para todo $g_0 \leftarrow G_0 \in$ **base**$(g \leftarrow G)$.

Uma cláusula é verdadeira se todas as suas instâncias básicas são verdadeiras. ■

- $S \models_I g \leftarrow G$, onde $g \leftarrow G$ é **básica**, se e somente se $S \models g$ quando $S \models G$.

Uma cláusula **básica** é verdadeira se a cabeça é verdadeira quando o corpo é verdadeiro. ■

- $S \vdash u \gg G$, se e somente se $S[u] \vdash G$.

Uma fórmula extensional é verdadeira se a conjunção interior é verdadeira na situação atualizada pela denotação do nome da unidade. ■

- $S \models_I g$, onde g é uma fórmula atômica básica, se e somente se $g \in S$.

Uma fórmula atômica básica é verdadeira em uma dada situação, exatamente no caso em que pertence à situação. ■

Uma interpretação I é um *modelo* de um sistema de unidades $\mathcal{U}$, se toda unidade em $\mathcal{U}$ é verdadeira em toda situação com respeito a I. O conjunto:

$$\Im = \prod_{u \in Un} [\wp(B_{-[\![u]\!]}) \rightarrow \wp(B_{[\![u]\!]})]$$

de todas as interpretações, é parcialmente ordenado por:

$I \subseteq J \equiv u_I(S) \subseteq u_J(S)$, para todo $u \in Un$ e $S \in \wp(B_{-|u|})$.

$\mathfrak{I}$ é um reticulado algébrico cujo maior e menor elementos, $\bot$ e $\top$ são dados respectivamente por $u_\bot(S) = \emptyset$ e $u_\top(S) = B_{|u|}$ para todo $u \in Un$ e $S \in \wp(B_{-|u|})$. O **supremo** de um conjunto não vazio $\{I_\alpha : \alpha \in A\}$ de interpretações é a interpretação I tal que

$$u_I(S) = \bigcup \{u_{I_\alpha}(S) : \alpha \in A\} \text{ para todo } u \in Un \text{ e } S \in \wp(B_{-|u|}).$$

▸ **Teorema 4.2**

Todo sistema de unidades $\mathcal{U}$ tem um modelo minimal $\mathbf{M} = \mathbf{M}_{\mathcal{U}}$. ∎

4.5.5 Semântica de Ponto Fixo

A transformação $T = T_{\mathcal{U}} : \mathfrak{I} \longrightarrow \mathfrak{I}$ associada a um sistema de unidades $\mathcal{U}$, mapeia uma interpretação I para a interpretação T(I), tal que:

$$u_{T(I)}(S) = \{g : \exists g \leftarrow G \in \text{base}(|u|),\ S[u_I] \models_I G\}$$

▸ **Teorema 4.3**

I *é um modelo de* $\mathcal{U}$ *se e somente se* $T(I) \subseteq I$. ∎

O modelo minimal de um sistema de unidades é então o **seu menor ponto** fixo. O seguinte resultado estabelece o relacionamento entre a semântica declarativa baseada na relação $\models_{\mathbf{M}}$ com respeito ao modelo minimal **M** e a relação $\vdash_{\mathcal{U}}$ para a derivação *bottom-up*. Para estabelecer o teorema, necessitamos de uma nova definição. A situação $S_{c,I}$ associada com um nome de contexto c na interpretacão I é definida indutivamente como se segue:

$$S_{\lambda,I} = \emptyset$$

$$S_{uc,I} = S_{c,I[u_I]}$$

▸ **Teorema 4.4:**

Dado um sistema de unidades $\mathcal{U}$ com modelo minimal **M**,

$c \vdash_{\mathcal{U}} g$ *se e somente se* $S_{c,\mathbf{M}} \models_{\mathbf{M}} g$

para todo nome de contexto c *e fórmula básica atômica ou extensional* g. ■

A relação entre as semânticas declarativa e operacional conforme descrito pela relação de derivação é dada por:

▸ **Teorema 4.5:**

Para toda fórmula extensional u>>G *e substituição* θ,

$\lambda \vdash_{\mathcal{U}} u{>>}G[\theta]$ *se e somente se* $u_{\mathbf{M}}(\emptyset) \models_{\mathbf{M}} G_0$

para toda instância básica G_0 *de* $G\theta$, *onde* **M** *é o modelo minimal de* $\mathcal{U}$. *(Note que* $u_{\mathbf{M}}(\emptyset)=\emptyset[u_{\mathbf{M}}]$.*)* ■

5 PROGRAMAÇÃO EM LÓGICA E BASES DE DADOS

Os conceitos derivados da teoria da Programação em Lógica tem se revelado de grande utilidade tanto para a descrição de BDs estáticas (esquemas de BDs) como também para especificar o processamento de BDs passíveis de atualização. Por outro lado, a evolução da pesquisa sobre BDs em geral e particularmente sobre o modelo relacional, tem conduzido cada vez mais à utilização de tais conceitos. Segundo **[KOW 78]**, uma das principais causas que conduzem a isso reside na característica particular dos sistemas de programação em lógica que permite o tratamento indistinto entre BDs e programas: estratégias aplicáveis à execução de programas são igualmente aplicáveis à recuperação de informações a partir de consultas formuladas à BD. Além disso, métodos para provar propriedades dos programas aplicam-se também à verificação de restrições de integridade e procedimentos para a manutenção dinâmica de bases de dados podem ser convenientemente aplicados ao processo de aquisição de conhecimento.

5.1 BASES DE CONHECIMENTO E BASES DE DADOS

É conveniente neste ponto estabelecer uma distinção entre *bases de conhecimento* (BCs) e *bases de dados* (BDs). Na visão de Wiederhold, em [BRO 86], tal distinção está baseada na diferença entre *dados* e *conhecimento* que, para os propósitos do presente trabalho, é apresentada na **Figura 5.1** a seguir:

CONHECIMENTO	DADOS
• Pequenos volumes • Natureza Intensional • Impreciso • Declarações aplicáveis a grandes grupos • Mantido por especialistas	• Grandes volumes • Natureza Extensional • Verificáveis • Declarações atômicas ou individuais • Mantidos por operadores

Figura 5.1

Distinção entre Conhecimento e Dados

As diferenças entre *conhecimento* e *dados* apresentadas na **Figura 5.1** conduzem à seguinte definição prática:

> *"Se a responsabilidade pela atualização está a cargo de operadores, então trata-se de dados. Se os responsáveis são especialistas, então trata-se de conhecimento."*

Assim a declaração *"Pedro tem 45 anos"* pode ser verificada e atualizada por operadores, sendo portanto um *dado*. Já declarações do tipo *"uma pessoa de meia idade"* e *"pessoas de meia idade são cuidadosas"* requerem o tratamento por especialistas, e portanto são *conhecimento*. Sob tal enfoque é possível afirmar que, a grosso modo, as BCs contém informações equivalentes a declarações universalmente quantificadas (*"pessoas de meia idade são cuidadosas"*), enquanto que as BDs contém apenas informações equivalentes a assertivas atômicas básicas (*"Pedro tem 45 anos"*). Para Brachman e Levesque em **[BRA 86]** *"...BDs são (ou pelo menos podem ser proveitosamente vistas como sendo) BCs de uma certa classe"*. Os autores fundamentam essa idéia na hipótese do *Nível de Conhecimento* proposta por Newell **[NEW 82]**, e esse parece ser o pensamento corrente da maioria dos pesquisadores na área. Para os propósitos do presente trabalho parece-nos portanto ser conveniente adotar essa mesma premissa e considerar as BDs como sendo uma certa classe, limitada, de BCs.

Programação em Lógica e BDs possuem diferenças fundamentais no que diz respeito ao processo de tratamento da informação, ocasionadas principalmente pelos diferentes enfoques semânticos adotados sob cada um dos pontos de vista. A lógica oferece suporte adequado para BDs relacionais e para a teoria das BDs em geral. Entretanto, a visão tradicional das BDs é *modelo-teorética* [GAL 78]: uma definição estática de um esquema de BDs (uma teoria) é fornecida por meio da definição de estruturas de dados e restrições de integridade sobre tais estruturas. Um *estado* de uma BD em um determinado momento é uma interpretação (que deve ser um modelo) dessa teoria. Atualizações sobre a BD modificam o modelo mas não a teoria subjacente (o esquema de BDs), consequentemente as operações de atualização preservam a integridade original, isto é, o novo estado da BD deve ser um modelo para a teoria.

Por outro lado, a Programação em Lógica prefere adotar o ponto de vista *prova-teorético* [GAL 78] para o estudo de BDs. Fatos e regras de dedução (que são mais gerais do que a definição de visões na teoria das BDs) constituem *a própria teoria*, ao invés de um modelo de uma teoria mais geral. As consultas são encaradas como teoremas a serem provados com base na teoria a partir de um pequeno número de técnicas de prova bem fundamentadas. Não existe um *esquema estático*, no sentido da semântica modelo-teorética da teoria das BDs, consequentemente as operações de atualização *modificam* a teoria. Assim o estabelecimento de restrições de integridade *ultrapassa* o nível do sistema, controlando as atualizações como uma meta-teoria para a qual o estado da BD constitui um modelo. Dito de outra forma: o modelo da teoria é a realidade e as restrições de integridade exigem comparação da teoria com a realidade.

5.2 A LÓGICA COMO ESQUEMA DE REPRESENTAÇÃO DE CONHECIMENTO

Na lógica de primeira ordem, fatos e relações entre fatos são representados em uma determinada BC por meio de fórmulas. Segundo **[REI 84]**, as principais vantagens oferecidas pelos ERs baseados em lógica de primeira ordem são as seguintes:

- **Semântica Bem Definida:**
  A semântica da lógica de primeira ordem é formalmente bem definida.

- **Notação Simplificada:**
  A sintaxe da lógica de primeira ordem é simples e bem compreendida.

- **Economia Conceitual:**
  Cada fórmula (isto é, cada fato e cada regra de dedução) é representada uma única vez na BC, independentemente de suas diferentes aplicações.

- **Uniformidade de Representação:**
  Fatos, hipóteses, implicações, consultas e visões são todos expressos na mesma linguagem de primeira ordem. Mesmo assertivas a nível meta (por exemplo, restrições de integridade) podem ser programadas em lógica.

- **Uniformidade Operacional:**
  A teoria de prova de primeira ordem é o único mecanismo necessário para o processamento de consultas. Em decorrência da uniformidade de representação e da uniformidade operacional, as regras de inferência podem ser aplicadas diretamente aos fatos.

- **Esquema Padrão de Representação:**
  A lógica oferece uma base comum para a definição e comparação entre outros ERs.

- **Procedimentos Gerais de Inferência e Prova:**
  Os procedimentos de inferência e prova podem ser utilizados para consultas, dedução de novos fatos, recuperação solução de problemas e prova de teoremas.

- **Definição de Extensões:**
  A lógica de primeira ordem pode ser empregada para definir extensões de si própria, desde que a semântica de tais extensões esteja bem definida.

Há, entretanto, tres principais desvantagens que devem ser consideradas na utilização da lógica como ER [BRO 86a]. Em primeiro lugar, apesar do elevado grau de expressividade que apresenta, a lógica de primeira ordem não consegue representar facilmente determinado tipo de conhecimento sobre o mundo real, como crenças, *defaults*, conhecimento incompleto e auto-conhecimento, se bem que determinados aspectos de tal conhecimento possam ser representado. Por exemplo, é possível expressar conhecimento sobre o conteúdo de uma BC (correspondendo ao dicionário de dados de um SGBD), entretanto, mesmo que um determinado tipo de conhecimento possa ser representado em lógica, o processo de formalização requerido pode ser de difícil execução e podem existir formas mais naturais para a representação de tal conhecimento.

Em segundo lugar, conhecimento heurístico e conhecimento procedimental são de difícil representação em lógica. O conhecimento procedimental é crítico para qualquer integração entre conhecimento e dados, particularmente para as atividades de aquisição e manipulação de conhecimento. A lógica de primeira ordem tende a lidar com com coleções estáticas de fatos, facilitando a resposta de consultas sobre um determinado conjunto de cláusulas. Por outro lado, não possui uma semântica "pura" para a atualização de coleções de fatos ao mesmo tempo em que preserva sua integridade. A dificuldade em representar conhecimento procedimental é uma limitação não apenas para a representação de conhecimento mas para a própria programação em lógica.

A terceira deficiência da lógica como ER é refletida pela falta de princípios organizacionais essenciais para o projeto lógico, desempenho e engenharia de software quando se trata de BCs de grande porte ou complexidade. Em uma BC baseada em lógica, todas as fórmulas são independentes e qualquer estrutura ou relacionamento entre elas é estabelecida operacionalmente através de procedimentos de inferência. Essa extrema modularidade é uma vantagem em sistemas de pequeno ou médio porte, uma vez que permite múltiplas utilizações para os fatos. Entretanto, não há conceitos de nível mais alto que ofereçam modularidade para peças de conhecimento de maior porte, o que constitui uma carga substancial para, por exemplo, a organização de conhecimento orientado a objetos, a manutenção de nomes únicos para predicados, o entendimento da organização do conhecimento, depuração, etc.

5.3 LÓGICA E BASES DE DADOS

As linguagens da lógica de primeira ordem podem ser usadas sobre BDs para a definição, análise e processamento de consultas, visões e restrições de integridade. Em particular, o modelo relacional e a teoria das BDs estão baseados em uma visão modelo-teorética da lógica de primeira ordem, com extensões que conduzem a algumas das desvantagens da lógica como ER estabelecidas na seção anterior.

A programação em lógica normalmente adota uma visão prova-teorética para BDs. Essa abordagem enfoca um esquema de BDs e um estado da BD como constituindo uma teoria de primeira ordem. A teoria da prova oferece a possibilidade de se raciocinar sobre essa teoria de primeira ordem, enquanto que a teoria dos modelos permite lidar sobre um estado único de uma BD. As vantagens potenciais da abordagem prova-teorética sobre BDs incluem uma forma de tratamento uniforme para:

- Aplicar regras de dedução e recuperar fatos. A lógica pode ser usada para derivar todos os fatos que podem ser deduzidos a partir dos fatos armazenados. Para esse propósito, esquemas convencionais de BDs necessitam empregar recursos especiais para o processamento de visões.

- Definir formalmente a semântica de modelos não-lógicos de dados.

- Comparar modelos de dados (expressos em lógica).

- Investigar, definir e suportar a semântica de modelos de dados especiais ou extendidos (por exemplo: informação disjuntiva, valores nulos ou conceitos semânticos para a modelagem de dados).

- Investigar e analisar a semântica de aplicações (expressas em lógica).

- Processar consultas sobre informações incompletas.

- Expressar e analisar a satisfatibilidade de restrições de integridade.

- Provar a correção de algoritmos não prova-teoréticos para o processamento de consultas, com respeito à sua semântica lógica.

- Provar a correção de algoritmos de manutenção de integridade com respeito à definição prova-teorética da satisfação de restrições.

Devido às questões de eficiência envolvidas na implementação de provadores de teoremas, a abordagem prova-teorética pode não ser apropriada para representar algoritmos de recuperação sobre BDs de grande porte. Ao invés disso essa abordagem poderia ser utilizada como uma teoria para a definição e análise da semântica de modelos de dados, bem como para ampliar o processamento de consultas extra-SGBD **[BRO 86a]**.

Uma questão de grande importância neste contexto é até que ponto a implementação de um sistema de programação em lógica pode oferecer simultaneamente correção e eficiência. O método de prova oferecido pelo teorema da Resolução (a unificação, no caso da linguagem Prolog) pode conduzir à *explosão inferencial* devido a duas razões principais. Em primeiro lugar, todas as cláusulas e fatos são armazenados em uma BD homogênea não-estruturada. Depois, todas as cláusulas são tratadas pelo provador de teoremas sem qualquer consideração semântica. Para reduzir tais problemas, alguns sistemas de programação em lógica oferecem mecanismos para expressar controle ou conhecimento procedimental, como, por exemplo o *cut* [CLA 82].

A programação em lógica oferece simultaneamente um esquema potencial para a representação de conhecimento e um modelo computacional para BDs dedutivas ou BCs. Mecanismos de consulta equivalentes aos empregados na programação em lógica podem ser adicionados à álgebra relacional por meio da introdução de operadores de ponto fixo. Pode ser demonstrado que o subconjunto das cláusulas de programa da programação em lógica é equivalente a uma álgebra relacional constituída pelas operações de seleção, junção e projeção (SJP) mais um operador para o *menor ponto fixo* [CHA 82]. Tal subconjunto pode ser implementado através da álgebra relacional acrescida de algoritmos interativos para as aplicações envolvendo o operador de menor ponto fixo [AHO 79].

Outras desvantagens da programação em lógica e de suas implementações, no que diz respeito às BDs são o seu potencial limitado de representação, a impossibilidade de aplicar o paradigma da programação em lógica a certas aplicações envolvendo atualizações e conhecimento procedimental, o processamento eficiente de fórmulas lógicas e um suporte eficiente ao gerenciamento da BC.

5.4 A LINGUAGEM PROLOG E SISTEMAS DE BASES DE DADOS

A linguagem Prolog representa uma classe de implementações do subconjunto da programação em lógica formado pelas cláusulas de programa. Os programas Prolog correspondem a hipóteses, enquanto que as consultas equivalem a teoremas a ser provados por meio de um provador baseado no princípio da unificação. As diferentes implementações da linguagem Prolog baseiam-se em uma interpretação procedimental das cláusulas de programa, onde a implicação:

B_1 e B_2 e ... e B_n implica em A

é interpretada como um procedimento que reduz os problemas da forma **A** para subproblemas da forma **B_1 e B_2 e ... e B_n**. Cada subproblema, por sua vez, é interpretado como um chamada a procedimentos que representam outras implicações. Assertivas da forma **P** são interpretadas como "**P** se *verdadeiro*", com uma coleção de vazia de subproblemas.

A linguagem Prolog convencional difere da programação em lógica "pura" em diversos pontos. Em primeiro lugar, os sistemas Prolog introduzem quatro tipos de mecanismos extra-lógicos: Para a entrada e saída são oferecidos predicados embutidos que permitem a leitura e escrita de cláusulas em terminais e na BD. Para o controle do mecanismo de pesquisa (ou seja, pesquisa em profundidade com *backtracking*) são oferecidos predicados embutidos e outros mecanismos (*cut*, *fail*, a ordem dos predicados na BD, a ordem dos termos nos predicados, etc.). Para os dados a linguagem Prolog oferece estruturas de dados limitadas (listas, árvores, etc.), meios para lidar com variáveis (*var*, *nonvar*, *real*, *integer*, etc.), e aritmética limitada. Finalmente, para suporte ao desenvolvimento e depuração de programas, são oferecidos alguns utilitários para *debug* e *trace*. Ainda que alguns desses mecanismos possam ser expressos em lógica de primeira ordem (se bem que com menor eficiência), a maioria não o permite.

Em segundo lugar, a resolução prova-teorética oferecida pela linguagem Prolog implementa uma forma de unificação que difere em dois pontos da unificação empregada na resolução lógica, primeiro porque a resolução implementada no Prolog confere uma *ordenação* ao processo de unificação (da esquerda para a direita e de cima para baixo) e depois porque a resolução não permite que uma variável seja instanciada para algum termo que contenha a própria variável, como no seguinte conjunto de expressões Prolog:

```
igual(X,X).
?- igual(f(Y),Y).
```

Para identificar e evitar tais casos (que produzem termos "infinitos"), uma "*verificação de ocorrência*" deveria ser executada antes que a instanciação fosse permitida. Essa verificação frequentemente poderia ser necessária, entretanto, se fosse implementada, tornaria o processo de resolução muito menos eficiente, de forma que é oferecida apenas opcionalmente em algumas implementações.

A grosso modo, uma analogia entre BDs e conceitos Prolog é apresentada na **Figura 5.2**, abaixo:

BASES DE DADOS	PROLOG
• Tuplas	• Cláusulas básicas (fatos)
• Atributos	• Argumentos de predicados
• Definição de visões	• Cláusulas de Horn
• Consultas	• Teoremas
• Programas	• Hipóteses
• Insert/Delete	• Assert/Retract
• Assertivas	• Consultas
• Gatilhos	• Predicados
• Conjuntos de valores	• Conjuntos de fatos

Figura 5.2

Uma analogia entre BDs e conceitos Prolog

A correspondência apresentada na **Figura 5.2** não é exata, uma vez que a semântica dos conceitos ali relacionados difere como indicam as seguintes comparações:

- **Tuplas x Fatos:**
  Em Prolog não há o conceito de um tipo de relação que possua uma estrutura de registro definida e restrições de integridade associadas. Uma relação básica Prolog é definida através de um conjunto de cláusulas básicas (fatos) que possuem o mesmo nome de predicado e um número idêntico de argumentos. O mesmo nome de predicado pode aparecer com um número diferente de argumentos, representando outra relação. Além disso, uma relação pode ser definida parcialmente através de cláusulas básicas e parcialmente através de cláusulas de Horn com o mesmo predicado como cabeça, de modo que não há distinção entre relações básicas e relações derivadas (visões concretas) em Prolog.

- **Visões x Semântica de Predicados:**
  Uma visão de uma BD pode ser definida como sendo o valor de uma expressão relacional sobre relações básicas e outras visões já definidas. Normalmente as visões sobre uma BD são atualizadas somente quando a atualização satisfaz a definição da visão e há um mapeamento bem definido entre a visão e as relações básicas subjacentes. Em Prolog, uma BC contém simultaneamente as definições de relações e de visões. Por exemplo, as cláusulas: **avô(pedro, jorge)** e **avô(X, Y) :- pai(X, Z), pai(Z, Y)** podem ser armazenadas simultaneamente na BC. Isso significa que o Prolog não distingue o esquema (teoria) da BD (modelo). Relações, visões e definições de visões podem ser modificadas diretamente na BC através de operações *assert/retract* e o Prolog não garante que tais modificações satisfaçam quaisquer condições. Em termos de BDs isso corresponde à atualizar uma visão sem considerar sua definição.

- **Consultas x Teoremas:**

  Em BDs as consultas são consideradas sob uma interpretação fixa. Em Prolog, consultas correspondem a teoremas que são avaliados conforme a teoria da prova. Por um lado, a abordagem uniforme empregada pelo Prolog para a avaliação de consultas torna o seu algoritmo básico muito simples, cuja eficiência pode ser melhorada através de *cuts* e reordenação de cláusulas. Por outro lado, tal uniformidade também pode ser uma desvantagem: o processamento de consultas orientadas a conjuntos (em especial o múltiplo uso de resultados intermediários) é de difícil suporte, a menos que predicados especiais sejam introduzidos, o que resulta na destruição da uniformidade.

- **Insert/Delete x Assert/Retract:**

  Em BDs a semântica das operações *insert* e *delete* constituem parte da semântica do modelo de dados, que tipicamente incluem um conceito de consistência que é mantido por tais operações. Em Prolog as operações *assert* e *retract* são extra-lógicas e se aplicam tanto a fatos como a predicados, não garantindo a manutenção da consistência dos fatos com respeito aos predicados existentes (a teoria).

- **Assertivas x Consultas:**

  As assertivas em uma BD são normalmente predicados de primeira ordem que exigem que qualquer modificação na BD deva satisfazer as restrições sob pena de ser rejeitada. Tais assertivas não existem em Prolog, ainda que possam ser implementadas [KIT 84]. Por outro lado, a implementação de tais operações básicas em Prolog, ao invés de no sistema subjacente, tem se mostrado ineficientes e necessitam contar com a disciplina do programador.

- **Gatilhos x Predicados:**
  Os "gatilhos" das BDs se assemelham às regras de dedução do Prolog, no sentido em que um determinado padrão (ou condição) ocasiona a execução de um programa previamente definido. Os gatilhos são disparados em BDs sempre que esse padrão é detectado, de modo semelhante ao conceito de "demons" em IA, porém com parcimônia, devido ao seu elevado custo. Em Prolog toda a programação é excecutada por meio da unificação de padrões (termos e literais).

Podemos ainda relacionar diferenças adicionais existentes entre BDs e Prolog. Discutiremos a seguir uma série de recursos normalmente disponíveis em BDs e que não são oferecidos pela linguagem Prolog básica, ainda que possam ser implementados em alguns casos.

- **Estruturas de Controle:**
  Uma das maiores diferenças entre o Prolog e as linguagens de consulta relacionais diz respeito à expressão declarativa das consultas. As linguagens de consulta relacionais são linguagens declarativas orientadas para conjuntos e não requerem estruturas de controle explícitas. Em programas Prolog, a execução deve ser considerada e em alguns casos, por razões de eficiência, o controle deve ser explicitamente introduzido.

- **Backtracking sobre a Base de Dados:**
  É frequentemente desejável a construção de um modelo na BC durante a execução de um programa Prolog, por exemplo, na tradução de uma sentença em linguagem natural. Isso é obtido em Prolog por meio dos predicados *assert* e *retract*, entretanto, as modificações introduzidas pela construção do modelo não são automaticamente desfeitas quando os predicados que as produziram são reconsiderados através de

backtracking. Isso reporta em parte o conceito de *transação* sobre uma BD que assegura que as alterações somente ocorrerão se a transação como um todo for aceita pela BD. Não há um conceito correspondente em Prolog.

- **Tipos de Dados:**
  Os sistemas gerenciadores de bases de dados (SGBDs) empregam extensivamente o conceito de tipos de dados, para assegurar a integridade semântica e a validade dos dados, havendo consideráveis esforços nessa área, notadamente na implementação de mecanismos para a eficiente identificação e validação de tipos. Já o Prolog é uma linguagem que não suporta diretamente o conceito de tipos de dados, ainda que estes possam ser definidos por meio de predicados adequados. Tais predicados, entretanto, são muito menos eficientes que os mecanismos usados nas linguagens de programação em geral e particularmente em BDs.

- **Consistência:**
  Os SGBDs tipicamente suportam algum mecanismo para a manutenção da consistência das BDs com respeito ao esquema adotado. Por exemplo, as tuplas em uma BD relacional são únicas, isto é, não pode haver mais de um conjunto de valores de atributos para cada entidade, e as visões devem ser atualizadas consistentemente com suas definições. Já a linguagem Prolog não possui o conceito de esquema nem tampouco uma semântica de atualização que inclua a consistência.

- **Facilidades oferecidas por SGBDs que não estão disponíveis em Prolog:**
  As implementações Prolog convencionais não possuem os mecanismos oferecidos pelos SGBDs para o gerenciamento

de suas BDs. O armazenamento de dados em dispositivos auxiliares é normalmente gerenciado por meio de sistemas de arquivos. Relacionaremos a seguir as facilidades normalmente oferecidas por SGDBs que não estão disponíveis em Prolog e que seriam necessárias para sistemas gerenciadores de BCs (SGBCs) baseadas em lógica:

- **Acesso eficiente a grandes bases de fatos.** Este é o domínio tradicional dos SGBDs que operam sobre um reduzido número de tipos com um grande número de instâncias.

- **Multiplicidade de usuários.** Normalmente os sistemas Prolog são monousuários. Para o compartilhamento de uma BD Prolog por múltiplos usuários é necessário introduzir um controle de concorrência.

- **Estrutura da base de dados.** O gerenciamento e a programação eficientes de grandes BDs se beneficiam de sua capacidade de estruturação (por exemplo, entidades e relacionamentos, hierarquias, etc.).

- **Orientação à conjuntos.** A otimização de consultas em BDs é orientada à conjuntos, com o gerenciamento de caminhos de acesso permanente e resultados intermediários.

- **Gerenciamento dos dados.** O gerenciamento dos dados em BDs inclui a sua segurança e recuperação.

- **Transações.** Uma transação sobre BDs pode incluir qualquer número de ações sobre os objetos na BD, garantindo a sua consistência. Em Prolog há somente um predicado objetivo por regra de dedução. Os sistemas Prolog não oferecem o conceito de atualização e de consistência de transações.

- **Facilidades oferecidas pelo Prolog que não estão disponíveis em SGBDs:**

  Por outro lado, os sistemas Prolog oferecem diversas facilidades que normalmente não estão disponíveis em SGBDs convencionais:

  - **Definição dinâmica de tipos.** Normalmente os SGBDs não oferecem a definição dinâmica de tipos, entretanto, alguns SGBDs permitem a definição dinâmica de visões. O Prolog não distingue entre a declaração de novos tipos de predicados e os já existentes (isto é, não possui o conceito de *tipo*), de modo que dados de novos tipos podem ser inseridos a qualquer momento na BC.

  - **Conhecimento incompleto, axiomas e unificação parcial.** O Prolog permite a expressão de fórmulas não-básicas e fatos com conhecimento incompleto, além de permitir a formulação de consultas com informações parciais, o que não é completamente suportado pela álgebra relacional.

  - **Recursão.** O Prolog permite a definição recursiva de predicados, o que não é em geral suportado por SGBDs mas que teria larga aplicação na estruturação de dados e de consultas.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

- **Backtracking e tracing.** O Prolog aciona um mecanismo de backtracking sempre que ocorre uma falha no processo de unificação e mantém o registro de todos os passos que ocorreram na sua execução.

- **"Functores".** O Prolog permite o uso de *functores*, isto é, símbolos funcionais como argumentos de predicados, o que não é suportado em SGBDs. BDs relacionais na primeira forma normal proíbem estritamente o emprego de relações dentro de relações.

- **Ordenação da base de dados.** Em Prolog, é importante a ordem das cláusulas na BC e dos termos nas cláusulas, enquanto que em BDs relacionais isso não é significativo.

BCs de grande porte irão certamente requerer um processamento mais inteligente do que o atualmente oferecido pelos SGDBs **[BRO 86a]**, assim como um acesso mais eficiente do que os disponíveis nos atuais sistemas de IA. A integração da programação em lógica com as técnicas desenvolvidas para SGBDs pode alcançar os requisitos desejados para SGBCs (no sentido em que são atualmente entendidas). Para as BDs, os benefícios de tal integração incluem principalmente as extensões que o potencial de expressividade da lógica oferece à representacão de conhecimento, à abordagem prova-teorética da programação em lógica e aos recursos oferecidos pela linguagem Prolog. Para a programação em lógica os benefícios incluem o acesso eficiente e o gerenciamento de dados em BDs de grande porte, o compartilhamento dos dados, a ampliação da capacidade de representação (por exemplo, a semântica procedimental e de atualização) e organização do conhecimento. Uma questão em aberto é a natureza de tal integração: *de que modo devem ser integrados os recursos da programação em lógica com os oferecidos pelos SGBDs?*

Quatro arquiteturas básicas devem ser consideradas para a integração da programação em lógica com BDs **[GAL 83], [JAR 84]**: O livre acoplamento entre sistemas independentes, extensões do Prolog para o gerenciamento de BDs lógicas, o incremento da capacidade dedutiva dos SGBDs e a integração total de um sistema de programação em lógica com um SGBD. Ainda que as três primeiras abordagens possuam a vantagem da pureza conceitual, a abordagem integrada reflete a idéia de que nem a programação em lógica nem os SGBDs possuem todas as vantangens do seu lado.

A idéia do livre acoplamento sugere um sistema de programação em lógica para consultas com a capacidade de atualização de um DBMs. Por um lado, a abordagem prova-teorética do Prolog é particularmente adequada para a formulação inteligente de consultas. Conceitos a nível meta relativamente simples poderiam ser facilmente empregados na otimização de consultas, tornando o Prolog uma linguagem de consultas declarativa de grande eficiência. Por outro lado, a atualização de BDs parece requerer a abordagem modelo-teorética para assegurar a consistência entre relações e visões, entretanto os SGBCs podem requerer um mecanismo de manutenção de integridade mais complexo do que o normalmente oferecido pelos SGBDs. Nesse caso podemos justificar o esforço adicional na programação em lógica a nível meta necessária na abordagem prova-teorérica.

Estender o Prolog com recursos de SGBDs ou dotar um SGBD de um *shell* dedutivo constituem somente soluções parciais. Nestas abordagens surgem normalmente os mesmos problemas que aparecem na integração total oferecendo como resultado sistemas potencialmente menos eficientes. Por exemplo, muitos recursos ausentes em Prolog, para os quais a semântica é bem conhecida, podem ser expressos através de predicados Prolog (por exemplo, restrições de integridade, tipos de dados, negação lógica, etc.). Entretanto isso requer a disciplina do programador e aparentemente é muito menos eficiente do que se dispor da semântica implementada diretamente no sistema subjacente.

Para as necessidades de representação de conhecimento, engenharia de software e desempenho, é desejável um sistema totalmente integrado. Isso fica claro ao considerarmos a noção de linguagem e de representação de conhecimento. Um determinado paradigma de programação não é ideal para todas as aplicações. Ao contrário, paradigmas especiais como a programação em lógica são mais adequados para determinados tipos de problemas. O desafio é integrar os paradigmas desejados em um único recurso de programação que demonstre ser o mais adequado para a classe particular de aplicações que interessam. Uma linguagem de programação para BCs de grande porte deve englobar conceitos de lógica, de BDs e de linguagens de programação.

Todos os problemas apresentados acima devem ser considerados na abordagem da integração total. Uma classe adicional de questões diz respeito à divisão ideal do trabalho entre SGBDs e programação em lógica. Em certos casos, essa divisão é óbvia, há entretanto algumas áreas em que a distribuição de tarefas precisa ser melhor investigada, incluindo as situações abaixo:

- Como dividir a inferência entre o SGBD e o provador de teoremas? Deveria o primeiro realizar toda a inferência modelo-teorética deixando toda a inferência prova-teorética ao provador de teoremas? Poderia este utilizar o processador de consultas e visões e o dicionário de dados do SGBD? Em que momentos a teoria da prova é mais eficiente do que a teoria de modelos (por exemplo, na avaliação da satisfabilidade das consultas)?

- Que padrões de utilização irão requerer um *shell* dedutivo? Um ambiente simples de transações baseado em menus pode não requerer os recursos sofisticados de processamento de visões de um interface baseado em programação em lógica, nem as estratégias de otimização lógica oferecidas por tais sistemas.

- De que modo o processamento recursivo de consultas deve ser implementado para máxima eficiência?

- Para que operações é necessário o conceito de tipos de dados? Deve-se extender a programação em lógica com tipos, deixar a verificação de tipos a critério do SGBD ou transmitir ao programador essa responsabilidade?

- Quais as vantagens relativas em se dispor de um *shell* em programação em lógica que permita o compartilhamento por múltiplos usuários? Ou diversos interfaces compartilhando uma mesma BD? Os interfaces devem ser projetados para propósito geral ou visar uma particular aplicação?

6 REPRESENTAÇÃO DE CONHECIMENTO EM HIPERREDES

No presente capítulo estudaremos a representação de conhecimento em hiperredes, modelo originalmente proposto por Georgescu em [GEO 85], que se destaca pelo grande potencial de expressividade que apresenta associado a uma semântica simplificada baseada fundamentalmente em definições recursivas para o tratamento de redes estruturadas. O modelo em si aproxima-se bastante dos sistemas de frames, contando entretanto com maior liberdade de expressão e favorecendo a estruturação e composição de *peças de conhecimento* (PCs) recursivamente definidas, o que parece ser bastante adequado ao tratamento de domínios complexos. Um estudo anterior de tal modelo foi realizado em [PAL 89], onde foi proposta uma implementação experimental do mesmo em Prolog.

6.1 O MODELO DAS HIPERREDES

Uma **hiperrede** é uma estrutura recursivamente definida, representada por uma tripla, **H=(N, R, p)**, onde:

- **N** - é um conjunto de objetos definidos recursivamente denominados **hipernodos**,
- **R** - é um conjunto de entidades relacionais definidas recursivamente, denominadas **hiperrelações**, e
- **p** - é um conjunto de pares *atributo-valor* arranjados como uma lista de **propriedades**.

O conceito de hipernodo está relacionado ao conceito de objeto abstrato. Um **hipernodo** é uma quíntupla **N=(h, s, I, E, p)**, onde:

h - é o **identificador** do hipernodo,

s - é o sort do hipernodo,

I - é uma hiperrede de nível mais baixo que descreve a **estrutura interna** do hipernodo, em termos de hipernodos e hiperrelações componentes,

E - é um conjunto de fbf's denominadas **padrões de ligação,** que descreve a seleção restritiva de hipernodos componentes que podem participar como pontos de ligação em combinação com outras peças de conhecimento, e

p - é uma lista de **propriedades**.

Por exemplo, o conceito **de cidade** pode ser representado por um hipernodo como um objeto composto cujos componentes são **distritos,** relacionados entre si por meio de **estradas**. Todos os distritos componentes são representados na **estrutura interna** da correspondente peça de conhecimento, enquanto que os demais distritos são *vistos* por meio de sua representação na **estrutura externa**, como pode ser visualizado na **Figura 6.1**.

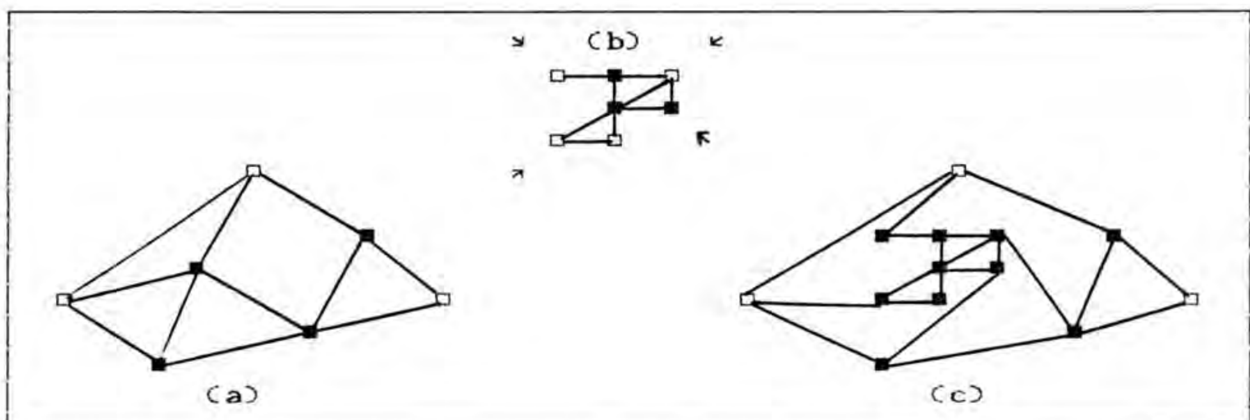


Figura 6.1

O refinamento (c) de um hipernodo (a) é obtido pela substituição de um dos componentes pela sua representação detalhada (b).

Por definição, o objeto componente de uma estrutura interna é também um hipernodo. Podemos visualizar um **distrito** como uma instância do conceito de **lugar**, que também deriva o conceito de **cidade**. Um **distrito** possui componentes (**logradouros** - a instância básica do conceito de **lugar**) que também se relacionam entre si por meio de **estradas**. Quando refinamos nosso conhecimento sobre a **cidade**, usualmente substituímos um hipernodo primitivo (por exemplo **distrito**) por uma descrição mais detalhada deles (a representação dos reais componentes e estrutura de um dado distrito - ver **Figura 6.1b**). A estrutura externa de um distrito deve fornecer todas as conexões (estradas) que se ligam aos demais **distritos** das **cidades** existentes na base de dados.

O conceito de **hiperrelação** está relacionado com o conceito de gerador de relações e captura o mecanismo que representa como as entidades relacionais com similaridade estrutural são produzidas, a partir de um conceito unificado. A base para tal mecanismo é o conceito de **protótipo,** que é um padrão estruturado não-instanciado, que pode ser preenchido por um hipernodo, desempenhando, no método das hiperredes um papel similar ao conceito de **lugar** em expressões relacionais ou funcionais primitivas. Uma **hiperrelação** é então uma quíntupla **R=(h,s,I,E,p)**, onde:

h - é o **identificador** da hiperrelação,

s - é o **sort** da hiperrelação, que identifica a classe de hiperrelações que compartilham as mesmas características estruturais,

I - é a **estrutura interna** da hiperrelação, em termos de **protótipos** componentes e relações de nível mais baixo,

E - é a **estrutura externa** da hiperrelação, definida em termos de padrões de ligação, especificando os **protótipos** da hiperrelação que participam em operações de composição, e

p - é uma lista de **propriedades.**

Se tomarmos, por exemplo, o conceito abstrato **caminho**, podemos considerá-lo como uma relação entre **lugares**. Uma instância de tal relação pode ser um conjunto de pares que listam todas as estradas existentes entre os lugares de uma dada região. Não há problema em representar o conhecimento acerca de uma estrada particular ou mesmo sobre um mapa rodoviário completo. A representação se torna difícil quando temos que trabalhar com um conceito abstrato como **caminho**, como uma ligação entre **lugares**. Um caminho em um grafo, ou um caminho em uma prova de consistência, ou mesmo um caminho em um mapa rodoviário, todos podem ser gerados a partir do mesmo conceito abstrato unificado, que tentaremos representar agora. A relação binária:

caminho(_____lugar', _____lugar)

vale entre objetos que satisfazem as restrições estruturais definidas para os protótipos que possuem o sort **lugar**. Isso é verdadeiro para o conceito de **cidade**, e o conceito abstrato **caminho** se torna então uma hiperrelação, que irá gerar todos os tipos de estradas entre cidades, como a estrada mostrada na **Figura 6.2** para o **mapa-rodoviário** entre **cidade-A** e **cidade-B**. A relação binária inicial é portanto refinada, mostrando sua aridade verdadeira e sua complexidade estrutural.

A estrutura básica de um **protótipo** é similar às estruturas das entidades anteriormente definidas (hipernodos e hiperrelações). Um **protótipo** é representado por uma quintupla: **B=(h,s,I,E,P)**, onde:

h - é um **identificador**, um símbolo que relaciona a posição de uma entidade não-especificada com a descrição estrutural do **protótipo**. Quando possui atribuição de valor, este valor se torna uma variável formal que participa em processos de interpretação posteriores,

s - é o **sort** aceito pelo protótipo,

I - é a **estrutura interna** do protótipo (um diagrama para os hipernodos passíveis de preencher o protótipo), especificado em termos de protótipos e hiperrelações componentes,

E - é a **estrutura externa** do protótipo, com padrões de ligação especificados em termos de protótipos componentes, e

p - é uma lista de **propriedades genéricas**.

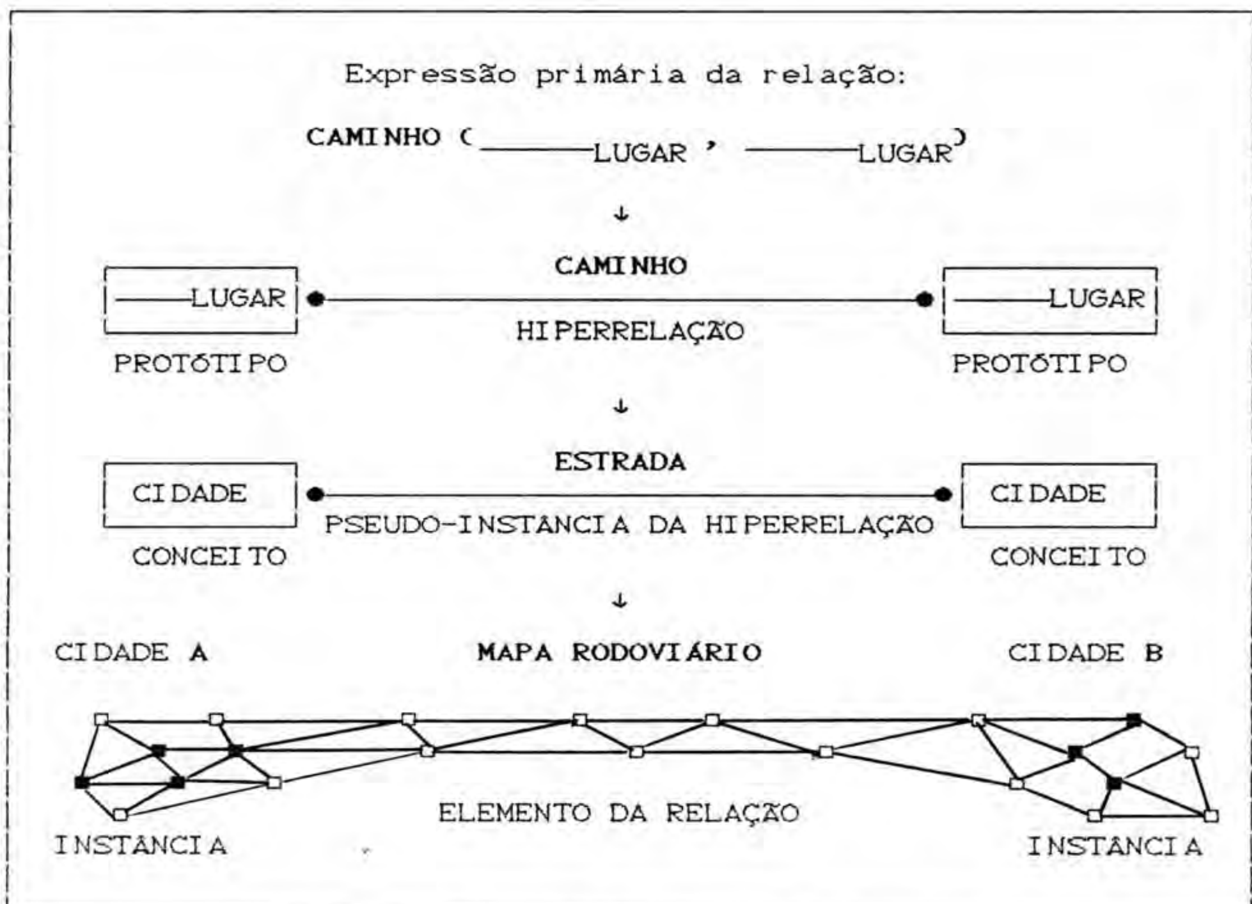


Figura 6.2
A instância **ESTRADA** da hiperrelação **CAMINHO** pode ser refinada, fornecendo a relação **MAPA RODOVIÁRIO**.

Por exemplo, os protótipos na **Figura 6.2** possuem pelo menos um ponto de ligação cada um em sua estrutura externa, e

pertencem ao sort **lugar**. Quaisquer outras restrições estruturais podem ser estabelecidas nas peças de conhecimento que representam protótipos. No nível básico do sistema há hipernodos primitivos, protótipos e hiperrelações que não possuem componentes.

6.2 A LINGUAGEM HYPER

Para o tratamento das entidades representadas em hiperredes, Georgescu propõe uma linguagem de representação denominada **Hyper**, cuja sintaxe apresentaremos a seguir. O formalismo utilizado para as definições sintáticas é o BNF-IN, uma extensão da Backus-Naur Form com a facilidade de indexação, que permite o uso de símbolos não-terminais como índices para outros símbolos não-terminais.

a) Entidades Básicas.

<base_de_conhecimento> ⟶
<peça_de_conhecimento>$_{\langle tipo \rangle}$ |
<base_de_conhecimento>, <peça_de_conhecimento>$_{\langle tipo \rangle}$

<peça_de_conhecimento>$_{\langle tipo \rangle}$ ⟶
(<tipo> (<identificador>$_{\langle tipo \rangle}$
((<sort>$_{\langle tipo \rangle}$)
(<estrutura_interna>$_{\langle tipo \rangle}$)
(<estrutura_externa>$_{\langle tipo \rangle}$)
(<propriedades>$_{\langle tipo \rangle}$))))

<tipo> ⟶ **hipernodo** | **protótipo** | **hiperrelação**

<identificador>$_{\textbf{hipernodo}}$ ⟶ <nome>

<identificador>$_{\textbf{hiperrelação}}$ ⟶ <nome>

<identificador>$_{\textbf{protótipo}}$ ⟶ <variável_formal>

<sort>$_{\textbf{hipernodo}}$ ⟶ <nome>

<sort>$_{\textbf{hiperrelação}}$ ⟶ <nome>

<sort>$_{\textbf{protótipo}}$ ⟶ <nome> | <variável_formal>

b) Estruturas Básicas.

<estrutura_interna>$_{\langle tipo \rangle}$ ⟶
    estrutura_interna (<lista_de_estruturas>$_{\langle tipo \rangle}$)

<lista_de_estruturas>$_{\langle tipo \rangle}$ ⟶
    <sort>$_{\langle tipo \rangle}$ (<descrição_estrutural>$_{\langle tipo \rangle}$) |
    <lista_de_estruturas>$_{\langle tipo \rangle}$;
    <sort>$_{\langle tipo \rangle}$ (<descrição_estrutural>$_{\langle tipo \rangle}$)

<descrição_estrutural>$_{\langle tipo \rangle}$ ⟶
    (<estrutura>$_{\langle tipo \rangle}$) |
    <descrição_estrutural>$_{\langle tipo \rangle}$(<estrutura>$_{\langle tipo \rangle}$)

<estrutura>$_{\langle tipo \rangle}$ ⟶
    <estrutura_não_orientada>$_{\langle tipo \rangle}$ |
    <estrutura_precedente>$_{\langle tipo \rangle}$ |
    <estrutura_sucedente>$_{\langle tipo \rangle}$

<estrutura_não_orientada>$_{\langle tipo \rangle}$ ⟶
    conecta (<lista_de_ramificações>$_{\langle tipo \rangle}$)

<estrutura_precedente>$_{\langle tipo \rangle}$ ⟶
    precede (<lista_de_ramificações>$_{\langle tipo \rangle}$)

```
<estrutura_sucedente> <tipo> ⟶
     sucede (<lista_de_ramificações> <tipo>)

<lista_de_ramificações> <tipo> ⟶
     (<ramificação> <tipo>) |
     <lista_de_ramificações> (<ramificação> <tipo>)

<ramificação> <tipo> ⟶
     (<nodo_raiz> <tipo>, <lista_de_nodos_folha> <tipo>)

<nodo_raiz> <tipo> ⟶ <nodo> <tipo>

<lista_de_nodos_folha> <tipo> ⟶
     <nodo> <tipo> | <lista_de_nodos_folha> <tipo>, <nodo> <tipo>

<nodo> hipernodo ⟶ <identificador> hipernodo

<nodo> protótipo ⟶ <variável_formal>

<nodo> hiperrelação ⟶ <nodo> hipernodo | <nodo> protótipo

<estrutura_externa> <tipo> ⟶
     estrutura_externa (<lista_de_padrões> <tipo>)

<lista_de_padrões> <tipo> ⟶
     <sort> <tipo> (<descrição_de_padrão> <tipo>) |
     <lista_de_padrões> <tipo> ;
     <sort> <tipo> (<descrição_de_padrão> <tipo>)

<descrição_de_padrão> <tipo> ⟶
     <conetivo> (<lista> <conetivo>,<tipo>) |
     <descrição_de_padrão> <tipo> ;
     <conetivo> (<lista> <conetivo>,<tipo>          )

<conetivo> ⟶ <conetivo_lógico> | cond

<conetivo_lógico> ⟶ or | xor | and
```

$\langle\text{lista}\rangle_{\langle\text{conetivo_logico}\rangle,\langle\text{tipo}\rangle} \longrightarrow$

$\quad\langle\text{componente}\rangle_{\langle\text{tipo}\rangle} \mid$

$\quad\langle\text{lista}\rangle_{\langle\text{conetivo_logico}\rangle,\langle\text{tipo}\rangle}$,

$\quad\langle\text{componente}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{componente}\rangle_{\langle\text{tipo}\rangle} \longrightarrow$

$\quad\langle\text{nodo}\rangle_{\langle\text{tipo}\rangle} \mid$

$\quad\langle\text{descrição_de_padrão}\rangle_{\langle\text{tipo}\rangle} \mid$

$\quad\langle\text{componente}\rangle_{\langle\text{tipo}\rangle}$, $\langle\text{nodo}\rangle_{\langle\text{tipo}\rangle} \mid$

$\quad\langle\text{componente}\rangle_{\langle\text{tipo}\rangle}$, $\langle\text{descrição_de_padrão}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{lista}\rangle_{\textbf{cond},\langle\text{tipo}\rangle} \longrightarrow$

$\quad\langle\text{condição}\rangle_{\langle\text{tipo}\rangle}$, $\langle\text{conexão}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{condição}\rangle_{\langle\text{tipo}\rangle} \longrightarrow$

$\quad\langle\text{variável_booleana}\rangle \mid \langle\text{descrição_de_padrão}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{conexão}\rangle_{\langle\text{tipo}\rangle} \longrightarrow$

$\quad\langle\text{nodo}\rangle_{\langle\text{tipo}\rangle} \mid \langle\text{descrição_de_padrão}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{propriedades}\rangle_{\langle\text{tipo}\rangle} \longrightarrow$

$\quad\textbf{propriedades}\ (\langle\text{lista_de_atributos}\rangle_{\langle\text{tipo}\rangle}$

$\langle\text{lista_de_atributos}\rangle_{\langle\text{tipo}\rangle} \longrightarrow$

$\quad(\langle\text{atributo}\rangle_{\langle\text{tipo}\rangle}$, $\langle\text{valor}\rangle) \mid$

$\quad\langle\text{lista_de_atributos}\rangle_{\langle\text{tipo}\rangle}(\langle\text{atributo}\rangle_{\langle\text{tipo}\rangle}$, $\langle\text{valor}\rangle)$

$\langle\text{atributo}\rangle_{\textbf{hipernodo}} \longrightarrow \langle\text{nome}\rangle$

<atributo>$_{\textbf{hiperrelação}}$ ⟶ <nome>

<atributo>$_{\textbf{protótipo}}$ ⟶ <nome> | <variável_formal>

<valor> ⟶ <número> | <símbolo>

6.3 OPERAÇÕES BÁSICAS

<operação> ⟶ <instanciação> |
<composição> |
<decomposição> |
<abstração> |
<modificação> |
<deleção> |
<renomeação>

- <instanciação> ⟶ **instância** (<identificador_de_instância>, <identificador_de_conceito>, <lista>)

Semântica: Uma nova peça de conhecimento denominada <identificador_de_instância> é criada em concordância com o protótipo <identificador_de_conceito>, preenchendo os símbolos e valores associados (se houver) a partir da <lista>.

- <composição> ⟶ <comp_inclusiva> | <comp_exclusiva>

<comp_inclusiva> ⟶ **comp_inc**
(<identificador>,
<peça_de_conhecimento-1>,
<peça_de_conhecimento-2>)

<comp_exclusiva> ⟶ **comp_exc**
(<identificador>,
<peça_de_conhecimento-1>,
<peça_de_conhecimento-2>)

Semântica: Uma nova peça de conhecimento K é criada como o resultado da operação de composição aplicada às peças de conhecimento 1 e 2. O símbolo especificado por <identificador> se torna o nome da nova peça de conhecimento **K**. O **sort** da nova peça de conhecimento é sort(K) = (sort(K1),sort(K2)), onde K1 e K2 são respectivamente os nomes de <peça_de_conhecimento-1> e <peça_de_conhecimento-2> Se **I1** e **I2** são os símbolos para as estruturas internas de **K1** e **K2**, e **E1** e **E2** o são respectivamente para as estruturas externas. O resultado composto **K** terá:

- Para a **composição inclusiva:**

$$I = I1 \cup I2$$
$$E = E1 \cup E2$$

- Para a **composição exclusiva:**

$$I = I1 \cup I2$$
$$E = (E1 \cup E2) \setminus (E1 \cap E2)$$

A lista de propriedades é imprevisível, devendo ser especificada pelo usuário.

- <decomposição> ⟶ **decomp**
  (<peça_de_conhecimento-1>,
  <peça_de_conhecimento-2>)

Semântica: O primeiro operando, <peça_de_conhecimento-1>, identifica a peça de conhecimento original **K1**, que é decomposta pela remoção de **K2**, a peça especificada como segundo operando, <peça_de_conhecimento-2>. Os resultados da operação de decomposição são:

a) A descrição de K2=(h2,s2,I2,E2,p2), se não houver sido especificada anteriormente, onde:

- h2 e s2 são respectivamente o identificador e o sort de K2, conforme especificado e contido na descrição de K1;

- I2 é uma estrutura interna gerada a partir do pré-requisito de ser K2 um componente de K1. Se $\Im K2$ é o conjunto de índices para as ligações de K2 em I1, há uma estrutura minimal, I2=(**link**(K2',...,K2i,...)), $i \in \Im K2$, com K2' sendo um hipernodo intermediário que combina todas as configurações possíveis de K2;

- E2 é uma estrutura externa para K2, gerada a partir das condições de ligação de K2 dentro da estrutura interna de K1, onde:

  \- Se $K2 \notin E1$, então E2=(**and**(...,K2i,...)), $i \in \Im K2$, ou

  \- Se $K2 \in E1$, e E12 é o subconjunto dos padrões de ligação de E1 onde K2 aparece, então E2=(E12 $\cup$ (**and**(...,K2i,...))), $i \in \Im K2$;

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

- A lista de propriedades é imprevisível.

b) A descrição do hipernodo Kr=(hr,sr,Ir,Er,pr), onde:

- hr é um identificador gerado a partir dos nomes de K1 e K2 por meio de um procedimento padrão do sistema;

- sr = s1 \ s2;

- Ir = I1 \ K2;

- Er é a estrutura externa obtida de:

  - Se K2 ∉ E1, então Er = E1 ∪ E2, ou
  - Se K2 ∈ E1, então Er = (E1 ∪ E2) \ E12;

- A lista de propriedades é imprevisível.

c) A descrição da hiperrelação H12=(h12,s12,I12,E12,p12) onde:

- h12 é o identificador gerado para a hiperrelação;

- s12=s1, porque o sort da hiperrelação é o mesmo da tupla geradora;

- I12=(**link**(B2,...,B2i,...)), i∈ℑk2, onde B2 é um protótipo correspondendo à estrutura de K2 e B2i é definido pelos requisitos de construção das estruturas K2i;

- E12=(**and**(B2,...,B2i,...)), i∈ℑK2;

- p12=p1, porque as propriedades de uma hiperrelação são as mesmas da tupla geradora.

- <abstração> ⟶ <abstração_conjuntiva> | <abstração_disjuntiva>

<abstração_conjuntiva> ⟶ **abstr_conj** (<lista_de_peças_de_conhec.>)

Semântica: Sejam K1,...,Kn peças de conhecimento. Então a abstração conjuntiva gera um conceito K=(h,s,I,E,p) que abstrai todas as características dos argumentos dados como se segue:

- h é um identificador internamente gerado;

- s = ∩si, 1≤i≤n, é o sort resultante da intersecção de todas as expressões formais das classes individuais;

- I = ∩Ii, 1≤i≤n, é a estrutura interna abstraída, que representa a estrutura comum a todas as peças de conhecimento individuais;

- E = ∩Ei, 1≤i≤n, é a estrutura externa similarmente abstraída;

- p = ∩pi, 1≤i≤n, é a lista de propriedades comuns.

A operação de abstração conjuntiva abstrai uma classe para peças de conhecimento previamente sem classificação, a nível de protótipo.

- <abstração_disjuntiva> ⟶ **abstr_disj**
  (<lista_de_peças_de_conhec.>)

Semântica: Essa operação abstrai uma nova peça de conhecimento a partir de suas descrições parciais. Todas as intersecções das descrições semânticas anteriores, especificadas para a abstração conjuntiva, são aqui substituídas por uniões.

- <modificação> ⟶ **modifica**
  (<peça_de_conhecimento>,
  <lista_de_modificações>)

<lista_de_modificações> ⟶ <modificação_elementar> |
<lista_de_modificações> ,
<modificação_elementar>

<modificação_elementar> ⟶ <inserção> |
<substituição> |
<eliminação>

<inserção> ⟶ **insere**
(<descrição>)

<descrição> ⟶ <sort> $_{\text{<tipo>}}$ |
<estrutura_interna> $_{\text{<tipo>}}$ |
<estrutura_externa> $_{\text{<tipo>}}$ |
<propriedades> $_{\text{<tipo>}}$

<substituição> ⟶ **substitui**
(<descrição-1> , <descrição-2>)

<eliminação> ⟶ **elimina**
(<descrição>)

Semântica: A inserção incorpora a descrição especificada ao corpo de uma peça de conhecimento. A substituição especifica que uma parte da peça de conhecimento, especificada por <descrição-1> será substituída pela estrutura descrita por <descrição-2>. A eliminação produz, alem da deleção das estruturas especificadas, novos padrões de ligação em concordância com a estrutura resultante. Todas as modificações estão sujeitas a provas de consistência implícitas.

- <deleção> ⟶ **deleta**
  (<peça_de_conhecimento>)

Semântica: Apaga a peça de conhecimento especificada da base de conhecimento.

- <renomeação> ⟶ **renomeia**
  (<identificador_anterior> ,
  <identificador_atual>)

<identificador_anterior> ⟶ $\langle\text{identificador}\rangle_{\langle\text{tipo}\rangle}$

<identificador_atual> ⟶ $\langle\text{identificador}\rangle_{\langle\text{tipo}\rangle}$

Semântica: A peça de conhecimento denominada <identificador_anterior> passa a ser conhecida pelo nome de <identificador_atual>.

6.4 COMANDOS

Além das operacões básicas apresentadas nas seções anteriores, a linguagem **Hyper** para a representação de conhecimento oferece declarações para trabalhar com bases de conhecimento, ou para aquelas operações que envolvem mais do que um único acesso a uma peça de conhecimento. Apesar das estruturas sintáticas de tais declarações serem altamente dependentes das opções de implementação, apresentaremos a seguir uma base informal para sua semântica.

- **Declaração CREATE:** É destinada à criação de uma nova base de conhecimento. O trabalho com diversas bases de conhecimento é típico em sistemas de crenças. Argumentos típicos para essa declaração são o nome da base de conhecimento e o nome do especialista de quem é adquirido o conhecimento. Argumentos opcionais podem também ser fornecidos para especificar *passwords*, política de classificação, graus de credibilidade associados às fontes de conhecimento e símbolos para a conexão de procedimentos usados para repassar interpretações do sistema às operações de acesso da base de conhecimento.

- **Declaração DESTROY:** Aplica-se à destruição parcial ou total de uma base de conhecimento. Argumentos regulares para essa declaração são o nome da base de conhecimento e o nome de um protótipo. A execução desta declaração resulta na deleção da sub-árvore que possui o seu nodo raiz no protótipo especificado e inclui todas as suas instâncias descendentes. A opção *default* para o nome do protótipo produz a deleção do conteúdo completo da base de conhecimento, sem destruir entretanto o registro *header*, que contém a informação

global especificada pela declaração **CREATE**. Isso permite recomeçar o processo de aquisição de conhecimento a partir do zero, dentro da perspectiva original da criação da base de conhecimento. Quando há uma requisição para a destruição total da base de conhecimento, incluindo sua informação global e, naturalmente, o seu nome, utiliza-se um argumento **KILL** na posição do nome do protótipo.

- **Declaração DECLARE:** Registra, na base de conhecimento especificada, uma nova peça de conhecimento do tipo **protótipo**. Os protótipos são usados para definições de classes controladas por padrões. Argumentos usuais para essa declaração são o nome da base de conhecimento e a descrição do protótipo de acordo com a sintaxe apresentada nas seções anteriores. Para bases de dados especiais, protegidas ou classificadas, *passwords* e chaves de classificação são também necessárias como argumentos. A declaração é permitida unicamente para protótipos independentes, que correspondem a raízes em árvores taxonômicas. Protótipos subsequentes que descrevem ramos das árvores taxonômicas são introduzidos por meio da operação básica **instância**, pois são instâncias conceituais do protótipo raiz independente.

- **Declaração REFINE:** Expande os componentes especificados de uma dada peça de conhecimento, pesquisando a base de conhecimento em busca de descrições mais detalhadas das mesmas. Os argumentos usuais para essa declaração são o nome, **K**, da peça de conhecimento a ser refinada, e uma lista de componentes a serem pesquisados na base de conhecimento e expandidos na estrutura da peça de conhecimento **K** por

meio de um processo de refinamento. Para os componentes não encontrados na base de conhecimento especificada, mensagens apropriadas devem ser fornecidas. Erros detetados na tentativa de combinar padrões de ligação no processo de refinamento devem invalidar a expansão das estruturas para os componentes onde o erro é encontrado. O resultado de uma operação de refinamento deve conter somente os componentes expandidos que são consistentes com os requisitos estruturais da peça de conhecimento **K** na base de conhecimento. A especificação *default* para a lista de componentes expandíveis deve ser considerado um requisito para a expansão de todos os componentes.

- **Declaração SELECT:** Seleciona a base de conhecimento especificada para ser operada (com operações básicas). Uma nova declaração **SELECT** desabilita a base de conhecimento anteriormente selecionada.

- **Declaração TRANSFER:** Permite a transferência de peças de conhecimento entre diferentes bases de conhecimento. Argumentos usuais para essa declaração são o identificador da peça de conhecimento a ser transferida e os identificadores das bases de conhecimento fonte e destino. Modificadores apropriados podem especificar as características do processo de transferência: com deleção da peça transferida na base de conhecimento fonte, com transferência de uma cópia da peça de conhecimento, qual o meio em que se processará a transferência, etc.

Outras declarações devem ser fornecidas pela linguagem para operações de manutenção da base de conhecimento, como:

- Operações de salvamento em volumes físicos,

- Extração de dicionários conceituais e árvores taxonômicas,

- Reorganização da base de conhecimento,

- Criação de novas bases de conhecimento por cópia seletiva de classes de peças de conhecimento e peças de conhecimento individuais de acordo com requisitos taxonômicos especificados, e

- Manutenção on-line.

6.5 PADRÕES DE LIGAÇÃO EM ESTRATÉGIAS DE CONTROLE

De acordo com o modelo de representação de conhecimento em hiperredes, a estrutura composta por entidades de conhecimento é representada pelo grafo dos relacionamentos entre seus componentes, que é definido como a estrutura interna **I** da entidade. Uma restrição **E** é também definida sobre os componentes da estrutura interna, de forma que $\mathbf{E} \subseteq \mathbf{I}$ representa somente aqueles componentes da estrutura interna que possuem a propriedade de poderem ser utilizados como **pontos de ligação** quando a dada entidade é composta ou combinada com outras entidades, resultando em peças de conhecimento com ordem superior de complexidade. A variedade dos padrões de ligação resulta da regra:

$$\langle \text{sort} \rangle_{\textbf{padrão_de_ligação}} \longrightarrow \textbf{free} \mid \textbf{disj} \mid \textbf{conj} \mid \textbf{cond}$$

e o aspecto geral da definição de um padrão pode ser obtido pela reunião das expressões elementares conectadas pelos mesmos conetivos lógicos. Por exemplo:

$$E_{pl} = (or(p_1, \ldots, p_j), xor(p_{j+1}, \ldots, p_k), and(p_{k+1}, \ldots, p_n))$$

com $j<k<n$, onde **pl** é a abreviatura de **padrão de ligação**, **xor** é a abreviatura de **ou exclusivo** e **pi**, $1 \le i \le n$, são pontos de ligação ou padrões de ligação de nível mais baixo. Consideremos a expressão acima como representando um argumento de um padrão de ligação:

$$pl = sort_{pl}(E_{pl})$$

Então a cada ponto de ligação é associada uma variável booleana cujo valor **é true** somente se o ponto correspondente está de fato envolvido em uma composição. Consequentemente a validade da composição controlada por este padrão é dada pelo valor-verdade de Epl, onde as sub-expressões em **or, xor** e **and**, são conectadas por disjunção para padrões livres e disjuntivos e por conjunção em padrões conjuntivos.

- **Padrões de Ligação Livres.**

Os padrões de ligação livres são definidos para aquelas composições onde somente algumas das capacidades de ligação dos padrões são efetivamente usadas. Então a propriedade **free** de um padrão, assume, para todos os pontos de ligação não utilizados em uma composição, que eles aparecerão nas estruturas externas da peça de conhecimento composta gerada e, portanto, poderão ser utilizados em processos de composição subsequentes.

Seja **free**(**or**(p1,..,pj),**xor**(pj+1,..,pk),**and**(pk+1,..,pn)) um padrão de ligação livre, e {pj1,...,pjq} ⊆ {p1,...,pj}, {pk1,...,pkr} ⊆ {pj+1,...,pk},{pk+1,...,pn} os pontos de ligação efetivamente usados em uma composição. O padrão de ligação que será transferido para a estrutura externa da entidade composta gerada será:

free(**or**((p1..pj)\(pj1..pjq)),**xor**((pj+1..pk)\(pk1..pkr))).

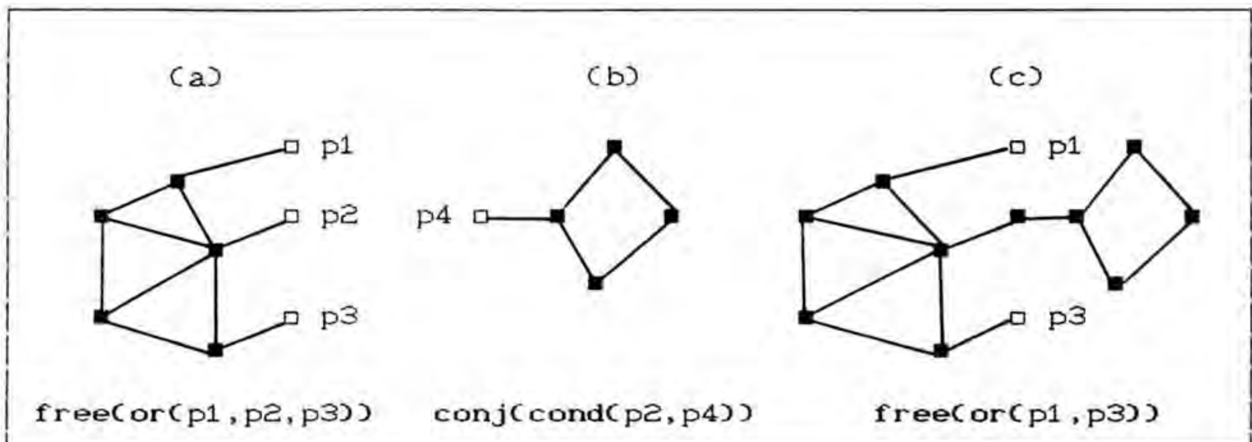


Figura 6.3

As peças de conhecimento (a) e (b) que participam de um processo de composição envolvendo um padrão de ligação livre e a peça resultante (c).

Note que todos os pontos "**and**" foram utilizados na composição. Entretanto, em padrões livres, quando subexpressões-**and** não são efetivamente usadas, elas aparecerão no padrão de ligação resultante que será transferido para a estrutura externa da entidade composta gerada. Essas propriedades são mostradas na **Figura 6.3.**

- **Padrões de Ligação Disjuntivos.**

Os padrões de ligação disjuntivos são similares aos padrões de ligação livres no que se refere a validade da composição por eles controlada. Diferem, entretanto, no que padrões de ligação disjuntivos não permitem que pontos de ligação não utilizados participem em composições subsequentes. Tais propriedades são mostradas na **Figura 6.4**.

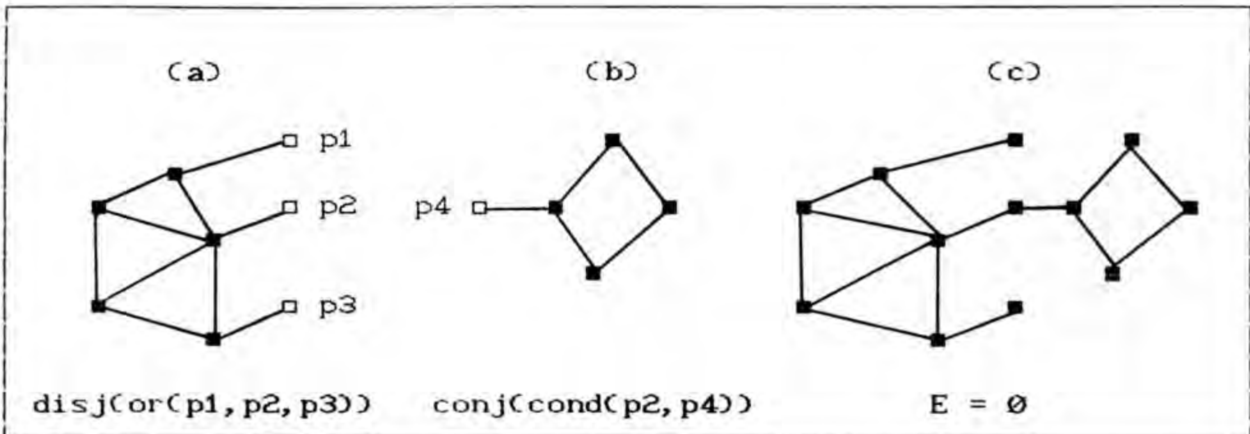


Figura 6.4

Padrões disjuntivos conectados são removidos da estrutura externa da entidade composta gerada.

- **Padrões de Ligação Conjuntivos.**

A composição sob o controle de um padrão de ligação conjuntivo é válida somente se todos os padrões em subexpressões envolvidas são verdadeiros. Não há padrão de ligação resultante a ser transferido para a estrutura externa da entidade composta gerada. O padrão de ligação conjuntiva fornece o mais restritivo controle ao processo de composição. As propriedades dos padrões de ligação conjuntivos são mostradas na **Figura 6.5**, onde é possível verificar as consequências de sua aplicação, conforme exposto acima.

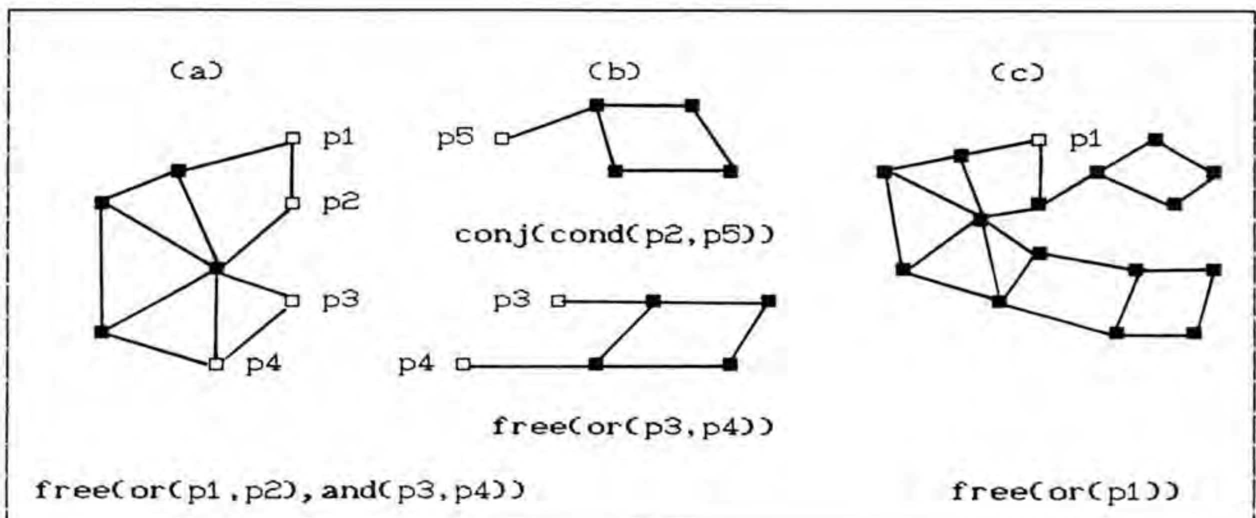


Figura 6.5

Todas as subexpressões de um padrão de ligação conjuntiva devem ser verdadeiros quando conectados num processo de composição.

- **Padrões de Ligação Condicionais**

De acordo com a sintaxe apresentada anteriormente, padrões de ligação condicionais são utilizados para validar um padrão de ligação como componente em uma estrutura externa, quando uma determinada condição é satisfeita. Tais padrões atuam como regras **se... então...**. Não há restrições com respeito aos termos e a complexidade da condição de um padrão de ligação condicional. Assim:

- A condição pode ser estabelecida por meio de outro padrão de ligação, cujo envolvimento na composição corrente atribui o valor-verdade da condição, assim validando o padrão para uso em processos de composição subsequentes,

- A condição pode ser estabelecida por uma expressão em termos de hiperrelações, assim validando o padrão para uso somente no contexto de tais hiperrelações,

- A condição pode ser estabelecida por uma expressão em termos de hipernodos, que selecionam diretamente quais os hipernodos são permitidos na composição sob o controle do padrão condicional, e

- A condição pode ser estabelecida por uma expressão em termos de outras entidades não diretamente envolvidas no processo de composição, mas que desempenham um papel *catalisador* em tal processo.

7 REPRESENTAÇÃO DE HIPERREDES EM LÓGICA

O modelo das hiperredes, apresentado no capítulo anterior, pode ser adequadamente formulado por meio de linguagens da lógica de primeira ordem. Entretanto, para explorar completamente seu potencial de expressividade, em determinados momentos torna-se interessante o emprego de extensões a nível meta. No presente capítulo iremos estudar a representação de hiperredes em Prolog objetivando a estruturação de programas e derivações de forma coordenada. Nosso propósito é duplo: Por um lado pretendemos atingir os conceitos de engenharia de software de modularidade, flexibilidade, reusabilidade, etc. Por outro lado desejamos obter um modelo computacional para o raciocínio em hiperredes. As noções de *teoria*, *contexto*, os conceitos da programação em lógica contextual **[MON 88]** e em nível meta **[BOW 85]** estabelecidos no capítulo 4 serão empregados com vistas à proposição de tal modelo.

7.1 REPRESENTANDO HIPERREDES EM PROLOG

Usaremos aqui a linguagem Prolog como formalismo para a representação de BCs estruturadas segundo o modelo das hiperredes. Sob tal abordagem uma BC é vista como um conjunto (possivelmente vazio) de objetos estruturados ou *peças de conhecimento* (PCs) que por sua vez são classificadas como hipernodos, hiperrelações ou protótipos, podendo ainda ser outra BC. Na **Figura 7.1** apresentamos o código Prolog correspondente.

```
baseDeConhecimento(BC) :-
      BC =.. [Nome|PCs],
      (nome(Nome);variavelFormal(Nome)),
      pecasDeConhecimento(PCs).

nome(N) :-
      atom(N).

variavelFormal(V) :-
      list_text([@|_],V).

pecasDeConhecimento([]).
pecasDeConhecimento([PC|PCs]) :-
      (baseDeConhecimento(PC);
       pecaDeConhecimento(PC)),
      pecasDeConhecimento(PCs).

pecaDeConhecimento(Id) :-
      X =.. [Id,Tipo,Sort,Int,Ext,Prop],
      call(X),
      validos(Id,Tipo,Sort,Int,Ext,Prop).

validos(Tipo,Id,Sort,Int,Ext,Prop) :-
      tipo(Tipo),
      identificador(Tipo,Id),
      sort(Tipo,Sort),
      estruturaInterna(Tipo,Int),
      estruturaExterna(Tipo,Ext),
      propriedades(Tipo,Prop).

tipo(hipernodo).
tipo(hiperrelacao).
tipo(prototipo).
```

Figura 7.1

Especificando BCs como Hiperredes

Uma BC é então representada como uma hiperrede. Esta, por sua vez, é definida sintaticamente como um termo Prolog complexo, cujo "functor" principal é o nome da BC e cujos argumentos correspondem a uma lista de entidades ou peças de conhecimento (PCs) que constituem a BC. As PCs são também representadas sintaticamente como termos Prolog cujo "functor" principal é o seu *identificador* e cujos argumentos são: um *tipo* (Tipo), um *sort* (Sort), uma estrutura interna (Int), uma

estrutura externa (Ext) e um conjunto de propriedades (Prop). A especificação sintática de um identificador é apresentada na **Figura 7.2**, onde as relações **nome/1** e **variavelFormal/1** são as mesmas definidas na **Figura 7.1**.

```
identificador(hipernodo,Id) :-
     nome(Id).
identificador(hiperrelacao,Id) :-
     nome(Id).
identificador(prototipo,Id) :-
     variavelFormal(Id).
```

Figura 7.2

Especificação sintática de um identificador

O *sort* de uma PC identifica uma classe de PCs que compartilham as mesmas características estruturais. Sua especificação sintática é dada na **Figura 7.3**, abaixo.

```
sort(hipernodo,Sort) :-
     nome(Sort).
sort(hiperrelacao,Sort) :-
     nome(Sort).
sort(prototipo,Sort) :-
     nome(Sort);
     variavelFormal(Sort).
```

Figura 7.3

Especificação sintática de um sort

A estrutura interna de uma PC é, por sua vez, uma hiperrede de nível mais baixo representando o *conteúdo de informação* presente na PC. Se esta for atômica, então sua estrutura interna corresponderá a um conjunto de termos ou cláusulas Prolog descrevendo o objeto ou procedimento que se deseja representar. Por outro lado, se a PC for complexa, a especificação de sua estrutura interna será fornecida através de

uma adequada combinação de estruturas cujo detalhamento conduzirá à construção da hiperrede correspondente. Hipernodos, hiperrelações e protótipos possuem, naturalmente, diferentes restrições no que diz respeito à construção de suas estruturas internas, uma vez que seus elementos constituintes obedecem a diferentes regras. Na **Figura 7.4** apresentamos a especificação completa para a construção da estrutura interna de PCs de qualquer tipo.

```
estruturaInterna(Tipo,int(I)) :-
      listaDeEstruturas(Tipo,I).

listaDeEstruturas(_,[]).
listaDeEstruturas(Tipo,[A|B]) :-
      A =.. [S|D],
      sort(Tipo,S),
      subEstruturas(Tipo,D),
      listaDeEstruturas(Tipo,B).

subEstruturas(_,[]).
subEstruturas(Tipo,[S|Ss]) :-
      subEstrutura(Tipo,S),
      subEstruturas(Tipo,Ss).

subEstrutura(Tipo,X) :-
      naoOrientada(Tipo,X);
      precedente(Tipo,X);
      sucedente(Tipo,X).

naoOrientada(Tipo,conecta(X)) :-
      listaDeRamificacoes(Tipo,X).

precedente(Tipo,precede(X)) :-
      listaDeRamificacoes(Tipo,X).

sucedente(Tipo,sucede(X)) :-
      listaDeRamificacoes(Tipo,X).

listaDeRamificacoes(_,[]).
listaDeRamificacoes(Tipo,[R|Rs]) :-
      ramificacao(Tipo,R),
      listaDeRamificacoes(Tipo,Rs).

ramificacao(_,[]).
ramificacao(Tipo,[N|Ns]) :-
      nodoRaiz(Tipo,N),
      nodosDescendentes(Tipo,Ns).
```

Figura 7.4a

Especificação de estruturas internas (Continua...)

```
nodoRaiz(Tipo,N) :-
     nodo(Tipo,N).

nodosDescendentes(_,[]).
nodosDescendentes(Tipo,[N|Ns]) :-
     nodo(Tipo,N),
     nodosDescendentes(Tipo,Ns).

nodo(hipernodo,Id) :-
     X =.. [Id,hipernodo,Sort,Int,Ext,Prop],
     call(X),
     validos(Id,hipernodo,Sort,Int,Ext,Prop).
nodo(prototipo,Id) :-
     X =.. [Id,prototipo,Sort,Int,Ext,Prop],
     call(X),
     validos(Id,prototipo,Sort,Int,Ext,Prop).
nodo(hiperrelacao,Id) :-
     pecaDeConhecimento(Id).
nodo(Tipo,Id) :-
     tipo(Tipo),
     codigo(Id,_).
```

Figura 7.4b

Especificação de estruturas internas (Final).

A estrutura externa de uma PC corresponde a um conjunto de restrições definidas sobre os componentes da estrutura interna de modo a especificar quais os que podem ser utilizados como pontos de ligação quando a dada entidade é composta ou combinada com outras, resultando em PCs com ordem de complexidade superior (ver seção 6.5). Os padrões de ligação definidos pela estrutura externa destinam-se portanto a impedir a geração de combinações inválidas ou absurdas, por exemplo, em operações de abstração. A cada ponto de ligação possível é associada uma variável booleana cujo valor é *true* somente se o ponto correspondente está de fato envolvido na composição considerada. O código Prolog correspondente é apresentado na **Figura 7.5**.

```
estruturaExterna(Tipo,ext(E)) :-
      listaDePadroes(Tipo,E).

listaDePadroes(_,[]).
listaDePadroes(Tipo,[A|B]) :-
      A =.. [S|D],
      sort(Tipo,S),
      descricaoDePadrao(Tipo,D),
      listaDePadroes(Tipo,B).

descricaoDePadrao(_,[]).
descricaoDePadrao(Tipo,[A|B]) :-
      A =.. [C|D],
      conetivo(C),
      elementos(Tipo,C,D),
      descricaoDePadrao(Tipo,B).

conetivo(cond).
conetivo(C) :-
      conetivoLogico(C).

conetivoLogico(or).
conetivoLogico(xor).
conetivoLogico(and).

elementos(Tipo,cond,[A,B]) :-
      condicao(Tipo,A),
      conexao(Tipo,B).
elementos(Tipo,C,[A,B]) :-
      conetivoLogico(C),
      componente(Tipo,A),
      componente(Tipo,B).
elementos(Tipo,C,[A|B]) :-
      conetivoLogico(C),
      componente(Tipo,A),
      elementos(Tipo,C,B).

condicao(Tipo,X) :-
      variavelBooleana(X);
      descricaoDePadrao(Tipo,X).

conexao(Tipo,X) :-
      nodo(Tipo,X);
      descricaoDePadrao(Tipo,X).

componente(_,[]).
componente(Tipo,[A|B]) :-
      (nodo(Tipo,A);descricaoDePadrao(Tipo,A)),
      componente(Tipo,B).
```

Figura 7.5

Especificação de estruturas externas

O último parâmetro necessário à especificação de uma PC é uma lista de propriedades genéricas representadas por pares *atributo-valor*, cuja especificação é dada na **Figura 7.6**, a seguir. Em operações de composição essa lista nem sempre pode ser obtida de modo automático, devendo ser fornecida pelo usuário.

```
propriedades(Tipo,prop(P)) :-
      listaDeAtributos(Tipo,P).

listaDeAtributos(_,[]).
listaDeAtributos(Tipo,[(A,V)|B]) :-
      atributo(Tipo,A),
      valor(V),
      listaDeAtributos(Tipo,B).

atributo(hipernodo,A) :-
      nome(A).
atributo(hiperrelacao,A) :-
      nome(A).
atributo(prototipo,A) :-
      nome(A);
      variavelFormal(A).

valor(V) :-
      number(V);
      atom(V).
```

Figura 7.6

Especificando listas de propriedades

Os predicados apresentados nas **Figuras 7.1** a **7.6** constituem conjuntamente uma linguagem para a definição de hiperredes como termos Prolog similar à linguagem Hyper, cuja sintaxe foi dada na seção 6.2. Na **Figura 7.7**, a seguir, apresentamos como exemplo a especificação estrutural de uma base de conhecimento geográfica, descrevendo a Zona Sul do estado do Rio Grande do Sul.

```
zonaSul(
      @municipio,
      vizinhoDe,
      pelotas,
      rioGrande,
      saoLourencoDoSul,
      cangucu,
      pedroOsorio,
      arroioGrande,
      piratini,
      pinheiroMachado,
      bage,
      jaguarao,
      saoJoseDoNorte,
      santaVitoriaDoPalmar,
      erval,
      capaoDoLeao,
      morroRedondo).

@municipio(
      prototipo,
      municipio,
      int(conecta([[Distrito:X]:Y])),
      ext(or([Distrito:Z])),
      prop([(area,A),(populacao,P),(icms,V)])).

vizinhoDe(
      hiperrelacao,
      relacaoBinaria,
      int(conecta([municipio,municipio])),
      ext(and([municipio,municipio])),
      prop([])).

pelotas(
      hipernodo,
      municipio,
      int(conecta([
            [sedePel,disPel2,disPel9],
            [disPel2,disPel3,disPel4,disPel9],
            [disPel3,disPel4,disPel5,disPel6,disPel7],
            [disPel4,disPel6],
            [disPel5,disPel7,disPel8,disPel9],
            [disPel6,disPel10],
            [disPel7,disPel10]])),
      ext(or([sedePel,disPel4,disPel5,
            disPel6,disPel7,disPel8,disPel10])),
      prop([(area,2205),(populacao,300000),(icms,5700)])).

...
```

Figura 7.7

Estrutura de uma BC geográfica

A representação de BCs como hiperredes formuladas em Prolog oferece portanto uma estrutura ao mesmo tempo de elevado potencial de expressividade e de grande flexibilidade, cujas principais características podem ser resumidas nos seguintes pontos:

- Uma BC representada como uma hiperrede assume sintaticamente a forma de um termo Prolog cujo "functor" principal é o *nome* da BC e cujos argumentos são *identificadores* de PCs.

- Uma PC pode ser um hipernodo, uma hiperrelação, um protótipo ou ainda outra BC. Os três tipos básicos partilham as mesmas características estruturais exteriores, sendo especificados por meio de cinco parâmetros (um tipo, um sort, uma estrutura interna, uma estrutura externa e uma lista de propriedades) e referenciados por um identificador.

- O identificador de um hipernodo ou uma hiperrelação é um *nome*, representado por um atomo Prolog. O identificador de um protótipo é uma *variável formal*, representada por uma sequência de caracteres iniciada por "@". Um protótipo pode ser instanciado assumindo a representação um hipernodo ou uma hiperrelação.

- O sort de uma PC identifica uma classe de PCs que compartilham as mesmas características estruturais. Em geral um sort é representado por um nome, entretanto o sort de um protótipo pode ser representado também por uma variável formal.

- A estrutura interna de uma PC é uma hiperrede de nível mais baixo representando o *conteúdo de informação* que esta contém. Se a PC for *atômica*, sua estrutura interna abrangerá um conjunto de

termos Prolog descrevendo a entidade que se deseja representar. Se a PC for complexa, sua estrutura interna é fornecida por meio de uma adequada combinação de estruturas (possivelmente sobrepostas) cujo detalhamento conduzirá a construção da hiperrede correspondente.

- A estrutura externa de uma PC corresponde a um conjunto de restrições definidas sobre os componentes da estrutura interna especificando quais deles podem participar como pontos de ligação em operações de composição. O objetivo é impedir a geração de combinações inválidas ou absurdas.

- As PCs podem ainda apresentar uma lista de propriedades particulares, representadas por pares *atributo-valor*. Nem sempre é possível prever de antemão as propriedades de uma PC resultante de uma operação de composição, de modo que estas devem eventualmente ser fornecidas pelo usuário.

7.2 RACIOCÍNIO SOBRE HIPERREDES

Na seção anterior procurou-se estabelecer uma forma de representação de hiperredes em Prolog. Tentaremos agora formular um mecanismo para a produção de inferências sobre hiperredes, essencialmente baseado nas idéias de Bowen **[BOW 85]** e Monteiro e Porto **[MON 88]**, apresentadas no capítulo 4, que contemplam, sob diferentes pontos de vista, o emprego de extensões da linguagem objeto a nível meta na obtenção de derivações sobre teorias e contextos.

Como deve ter ficado claro na seção anterior, a representação de conhecimento em hiperredes por meio de hiperrelações e protótipos produz na verdade um *esquema de aspectos*, onde cada aspecto corresponde a uma estrutura completamente instanciada capaz de satisfazer as restrições estabelecidas pelas entidades que compõem a hiperrede. Empregaremos aqui então o termo *aspecto* para designar uma hiperrede derivada cujos componentes são todos hipernodos, isto é, que apresenta uma estrutura estática.

Assim a derivação (top-down) de certo objetivo sobre uma hiperrede subentende uma sequência de três passos distintos: (1) A geração de um aspecto, (2) a *contextualização* desse aspecto, isto é, a tradução do mesmo em um espaço de prova, e (3) a derivação propriamente dita. No primeiro passo obtem-se uma instância do esquema de aspectos representados pela hiperrede. No segundo obtem-se o *sistema de contextos* determinado pelo aspecto, isto é, extrai-se do aspecto o conjunto estruturado de termos ou cláusulas Prolog que representa o conteúdo de informação ali presente e que será empregado como espaço de prova. Finalmente, aciona-se o interpretador Prolog subjacente para a execução da derivação desejada. É importante notar que, dependendo da implementação, esses tres passos podem se sobrepor dinamicamente. Na **Figura 7.8** apresentamos o código Prolog correspondente ao nível mais alto desse mecanismo.

```
demo(BC,Objetivos,[]) :-
     vazio(Objetivos).
demo(BC,Objetivos,[Premissa|RestoDaProva]) :-
     seleciona(Objetivos,Obj,RestoObj),
     aspecto(BC,Asp),
     contextualiza(Asp,SisCont),
     demonstra(SisCont,Obj,Premissa,ContObj),
     adiciona(ContObj,RestoObj,NovosObj),
     demo(BC,NovosObj,RestoDaProva).
```

Figura 7.8

Derivação top-down em hiperredes

O predicado **demo/3** apresentado na **Figura 7.8** é então uma relação de 3 argumentos representada por **demo(BC,Objetivos,Prova)** que será verdadeira se **Prova** é uma sequência de prova para os **Objetivos** em **BC**. A primeira cláusula da relação estabelece que qualquer BC demonstra a solubilidade de uma coleção vazia de objetivos. A segunda cláusula, tomada como um procedimento, estabelece que:

> *"Para demonstrar a solubilidade de uma coleção de objetivos, (1)* **selecione** *um dos objetivos formulados em* **Objetivos**, *(2)* **estabeleça** *um aspecto de* BC *que ainda não tenha sido considerado para esse objetivo, (3)* **contextualize** *o aspecto obtido no passo anterior sob a forma de um conjunto estruturado de termos ou cláusulas, (4)* **demonstre** *o objetivo selecionado nesse contexto gerando* **Premissa** *e (possivelmente) uma nova coleção de objetivos, (5)* **adicione** *esses novos objetivos aos objetivos remanescentes da seleção executada no primeiro passo, e (6)* **demonstre** *que a* **BC** *soluciona a coleção de objetivos assim formada."*

Observe que a demonstração executada no passo (4), que corresponde de certa forma à relação de demonstrabilidade proposta em [KOW 79] e comentada na seção 4.2, pode ser apenas parcial, necessitando considerar outros aspectos da BC. Nesse caso uma nova coleção de objetivos é produzida e adicionada aos objetivos remanescentes. Essa situação ocorrerá, por exemplo, quando o objetivo selecionado referenciar componentes da BC que não se encontram presentes no aspecto considerado. Nas seções seguintes analisamos em maior profundidade cada um dos passos do procedimento **demo/3**, apresentado na **Figura 7.8**, procurando consubstanciar a proposta ali formulada.

7.2.1 Seleção de Objetivos

O primeiro passo do procedimento **demo/3** é selecionar um dos objetivos presentes na lista **Objetivos**. Diversos critérios podem ser empregados para tal seleção, dentre os quais o mais direto consiste simplesmente em tomar o objetivo que encabeça a lista. Uma outra possibilidade seria selecionar o objetivo *mais complexo*, isto é, o que aparentemente é o mais difícil de ser satisfeito, seja por envolver um maior número de variáveis, seja por apresentar o maior número de sub-objetivos. Este último critério, entretanto, necessita partir do princípio que os objetivos presentes na lista são totalmente independentes entre si e que portanto a ordem em se apresentam não possui qualquer significado. Isso nem sempre é verdadeiro e além disso a adoção de tal critério demandaria consumo adicional de processamento para a determinação do objetivo de maior complexidade. Assim, iremos preferir a solução mais direta de selecionar o objetivo que se encontra mais à esquerda. A seleção de objetivos proposta em **seleciona(Objetivos,Obj,RestoObj)** pode então ser especificada simplesmente por:

```
seleciona([X|Y],X,Y).
```

A adoção desse critério, ainda que simplificado, possui uma vantagem adicional: se os novos objetivos gerados no decorrer do processo forem inseridos encabeçando a lista dos objetivos remanescentes da seleção inicial, então o mecanismo de seleção/reformulação da lista de objetivos pode ser tomado como operações sobre uma estrutura de pilha, de reconhecida economia operacional, que também é o modelo adotado pelo Prolog subjacente.

7.2.2 Geração de Aspectos

Um *aspecto* de uma **BC** foi anteriormente definido como uma instância capaz de satisfazer a todas as restrições

especificadas pelas entidades que compõem a **BC**. Esta, por sua vez, é vista como um *esquema de aspectos*. O conceito de aspecto não deve ser confundido com o de *visão* (assunto que abordaremos na seção 7.4), uma vez que estas podem também apresentar múltiplos aspectos. A hiperrede representativa de um aspecto se caracteriza por não possuir hiperrelações ou protótipos, sendo formada exclusivamente por hipernodos. Derivar um aspecto de uma **BC** corresponde então a extrair dela uma hiperrede atendendo aos seguintes requisitos:

(1) Todos os hipernodos especificados na **BC** pertencem automaticamente ao aspecto,

(2) Todos os protótipos especificados na **BC** são instanciados com hipernodos ou hiperrelações, conforme o caso,

(3) Todas as hiperrelações originais e as produzidas em (2) são acionadas produzindo hipernodos, e

(4) Todos os hipernodos obtidos em (3) pertencem ao aspecto.

Podemos então pensar nos aspectos determinados por uma **BC** como sendo constituídos por uma parte fixa (os hipernodos originais da **BC**) e uma parte variável (formada por hipernodos gerados a partir das hiperrelações originais ou derivadas dos protótipos existentes na **BC**).

A união de todos os aspectos que uma **BC** pode produzir é o seu *conteúdo de informação*, correspondendo à totalidade do conhecimento nela representada. O código Prolog para extrair um aspecto de uma **BC** é apresentado em seu nível mais alto na **Figura 7.9**.

```
aspecto(BC,Aspecto):-
      separa(BC,Nodos,Relacoes,Prototipos),
      instancia(Nodos,Relacoes,Prototipos,PNodos,PRelacoes),
      uniao(Relacoes,PRelacoes,TotRelacoes),
      aplica(TotRelacoes,Nodos,RNodos),
      uniao(Nodos,RNodos,Aspecto).
```

Figura 7.9

Extraindo aspectos de uma BC

No procedimento **aspecto/2**, acima, a relação **separa/4** tem a função de classificar as entidades especificadas na BC, segundo o seu tipo, em tres listas: uma para hipernodos, uma para hiperrelações e uma para protótipos. Sua execução escapa ao trivial porque convencionamos que uma BC pode conter outras BCs, e nesse caso a separação das entidades presentes em BCs componentes deve preservar sua integridade, ou seja, de algum modo deve ser possível reconstruir a estrutura original da BC após a sua classificação. Uma solução pode ser a manutenção do nome das BCs componentes aplicados às suas entidades particulares identificando sua origem. Por exemplo, considere a especificação abaixo, onde nomes de hipernodos são representados por consoantes,nomes de hiperrelações por vogais e nomes de protótipos são variáveis formais (iniciando com o caracter "@"):

bc1(@1,a,b,c,bc2(@2,d,e,bc3(@3,f,g)),h,i)

A aplicação da relação **separa/4** sobre **bc1** irá originar as seguintes listas de entidades:

Nodos:	**[b,c,bc2(d),bc2(bc3(f)),bc2(bc3(g)),h]**
Relações:	**[a,bc2(e),i]**
Protótipos:	**[@1,bc2(@2),bc2(bc3(@3))]**

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

O nome **bc1** pode ser omitido, uma vez que se aplica indistintamente a todos os componentes. Essa solução, ainda que simples, resolve eficientemente o problema de manter a especificação da estrutura original da BC, após a classificação de seus componentes.

A relação **instancia/5** promove uma instanciação global para todos os protótipos especificados na BC com hipernodos ou hiperrelações, conforme o caso, obtendo as listas **PNodos** e **PRelacoes**. A instanciação de um protótipo é possível quando um hipernodo (ou uma hiperrelação) possui uma estrutura interna e externa capaz de satisfazer as restrições estabelecidas pelo protótipo. Nesse caso o protótipo é substituído pela entidade correspondente. O objetivo principal aqui é obter as hiperrelações derivadas, que participarão juntamente com as já especificadas na extração de um aspecto da BC. Observe-se que a estruturação original da BC deve ser respeitada, isto é, na instanciação de um protótipo especificado em uma BC componente, somente poderão participar entidades a ela pertencentes. Assim, no exemplo acima, **bc2(bc3(@3))** somente poderá ser instanciado com **bc2(bc3(f))** ou **bc2(bc3(g))**.

Nos passos seguintes as hiperrelações originais e as derivadas são reunidas em uma única lista e aplicadas, obtendo-se um conjunto de hipernodos derivados que, juntamente com os originais, constituem um dos aspectos possíveis da BC.

Uma última consideração deve ser feita com respeito à geração de aspectos: Um mesmo aspecto não deve ser empregado mais do que uma única vez sobre um mesmo objetivo. Esta restrição, é automaticamente respeitada pelo Prolog subjacente no backtracking sobre a relação **aspecto/2**, quando a relação **demonstra/4** falhar, significando que nenhuma derivação foi possível para o objetivo selecionado sobre o aspecto corrente e que um novo aspecto deve ser tentado. Se, por outro lado, foi

obtida uma derivação *parcial*, isto é, que ainda contém variáveis, então em **demonstra/4** a variável **ContObj** conterá uma lista de objetivos derivados que ainda não foram completamente satisfeitos e que certamente não o serão no aspecto corrente. Esses objetivos derivados são adicionados encabeçando a lista dos objetivos remanescentes em **NovosObj** e uma chamada recursiva a **demo/3** será realizada, sendo necessário evitar o emprego do mesmo aspecto anteriormente considerado. Uma solução que nos parece apropriada seria a manutenção do registro dos aspectos empregados na solução de cada objetivo, por meio de um mecanismo automático de geração de *nomes de aspectos*, produzindo na BC cláusulas do tipo:

tentado(NomeDeAspecto,Objetivo).

Tal mecanismo não se encontra explicitamente formulado no procedimento **demo/3** por razões de clareza, fica entretanto registrada aqui a necessidade do seu emprego (ou de algum outro dispositivo que atenda à mesma finalidade).

7.2.3 Contextualização de Aspectos

Antes que um determinado aspecto possa ser empregado para a derivação de um certo objetivo, é necessário *contextualizá-lo*, ou seja, extrair dele o conjunto estruturado de cláusulas Prolog que corresponde ao conhecimento nele representado. Como já vimos, um aspecto de uma BC é uma hiperrede derivada, constituída exclusivamente de hipernodos, satisfazendo a todas as restrições especificadas pelas entidades que compõem a BC original. O nível mais alto de qualquer entidade em uma hiperrede apresenta a seguinte formulação:

<identificador>(Tipo,Sort,Int,Ext,Prop)

No caso particular de um hipernodo, **<identificador>** é o *nome* do hipernodo (representado por uma constante da linguagem), **Tipo** é a constante "hipernodo", **Sort** especifica uma classe de hipernodos que compartilham as mesmas características estruturais, **Int** é sua estrutura interna, **Ext** a sua estrutura externa e **Prop** um conjunto de propriedades.

A partir da configuração de um aspecto, os argumentos **Tipo** e **Sort** não são mais necessários. O primeiro porque todas as entidades em um aspecto são do tipo hipernodo e o segundo porque seu emprego se destina simplesmente a facilitar a instanciação de protótipos. Podemos então adotar, nessa fase do processo, uma formulação reduzida para os hipernodos constituintes de um aspecto:

<identificador>(Int, Ext, Prop)

Um hipernodo em um aspecto pode se apresentar como *atômico* ou *complexo*, em função do conteúdo de sua estrutura interna. Hipernodos atômicos são hipernodos do mais baixo nível, cuja estrutura interna é formada por um conjunto de termos ou cláusulas Prolog descrevendo um objeto ou procedimento. Já a estrutura interna de hipernodos complexos será constituída por fórmulas especificando combinações de hipernodos de níveis inferiores. Por *contextualização* de um aspecto queremos então significar a representação desse aspecto sob a forma de um sistema de *teorias* **[BOW 85]** ou *contextos* **[MON 88]** que possuam, além do conhecimento a nível objeto, extraído das estruturas internas e das propriedades dos hipernodos que as originaram, também conhecimento a nível meta estabelecendo relações de acessibilidade entre si, fornecido pelas estruturas externas dos mesmos hipernodos. Ao sistema de contextos obtido de um aspecto pode então ser associada uma semântica declarativa de mundos possíveis, onde o *conteúdo de informação* de um contexto pode ser visto como um mundo possível e conhecimento a nível meta representando relações de acessibilidade entre contextos como especificações de transição entre mundos.

Na **Figura 7.8** a relação contextualiza(Asp,SisCont) será satisfeita se SisCont é o sistema de contextos representado pelo aspecto Asp. Em seu nível mais alto essa relação é apresentada na **Figura 7.10**.

```
contextualiza(Asp,SisCont) :-
     geraSisCont(Asp,Asp,SisCont).

geraSisCont(Asp,[],[]).
geraSisCont(Asp,[H|Hs],[C|Cs]) :-
     contexto(Asp,H,C),
     geraSisCont(Asp,Hs,Cs).

contexto(_,H,C) :-
     H =.. [Id,Int,Ext,Prop],
     Int =.. [int,codigo(Cod)],
     relAcess(Ext,RA),
     uniao([Cod,Prop,RA],TotCod),
     C =.. [Id,TotCod].
contexto(Asp,H,C) :-
     H =.. [Id,Int,Ext,Prop],
     Int =.. [int|ListaDeEstruturas],
     decompoe(Asp,H,ListaDeEstruturas,[E|Es]),
     geraSisCont(Asp,[E|Es],C).
```

Figura 7.10
Contextualização de Aspectos

O sistema de contextos obtidos pela contextualização de um aspecto pode então ser empregado como espaço de prova para a derivação do objetivo selecionado. Pode ser interessante, para fins de eficiência e orientação do processo de derivação, construir parelalemente ao processo de contextualização de aspectos uma rede taxonômica onde cada contexto é representado pelo seu identificador e as relações de acesso entre contextos por meio de fatos Prolog especificando o tipo de conexão entre dois contextos. Para exemplificar, consideremos o sistema de contextos representado na **Figura 7.11**:

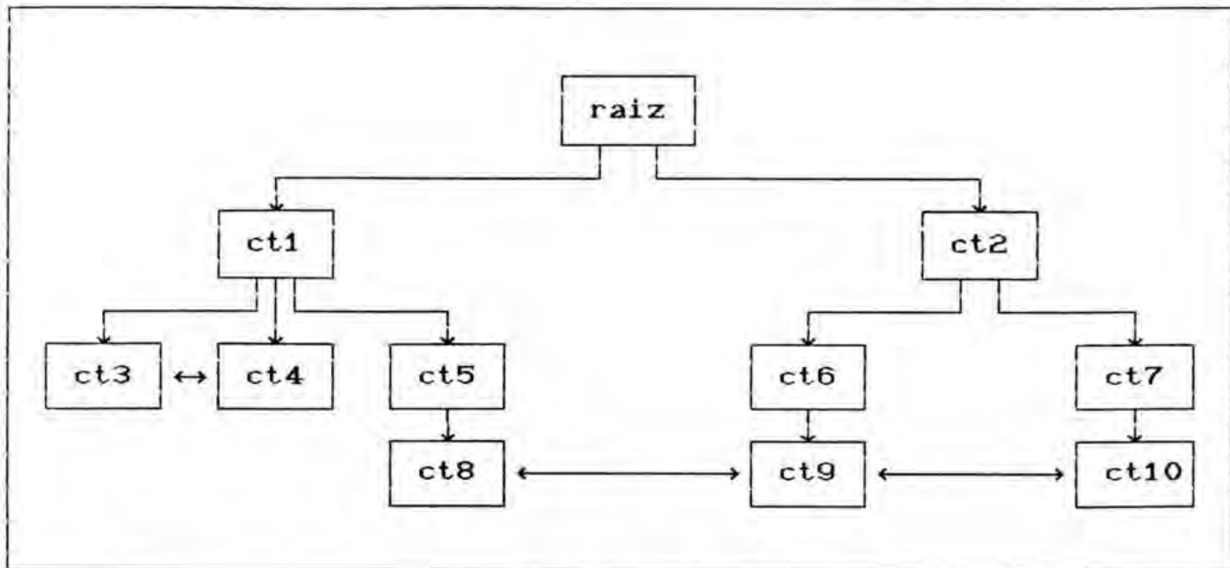


Figura 7.11

Um sistema de contextos

Vamos assumir que, na figura acima, **raiz,ct1,...,ct10** são nomes de contextos e que toda a rede ali representada corresponde a um sistema de contextos extraído de um aspecto. Dois tipos de ligações entre contextos podem ser ali encontradas: as *ligações orientadas*, representadas por setas direcionadas em um único sentido, e as *ligações não-orientadas*, representadas por setas com duplo sentido, como por exemplo entre **ct3** e **ct4**. As ligações orientadas traduzem a idéia de hierarquia entre os contextos. Assim, por exemplo, a partir de **ct1** é possível acessar diretamente os contextos **ct3, ct4** e **ct5**, entretanto não é possível acessar **ct1** a partir de nenhum deles. Já as ligações não-orientadas permitem o acesso nos dois sentidos, por exemplo, a partir de **ct3** é possível acessar **ct4** e também é possível acessar **ct3** a partir de **ct4**.

As ligações orientadas são representadas em contextos como conhecimento em nível meta por meio das relações **precede/1** e **sucede/1**, ao passo que ligações não-orientadas são indicadas através da relação **conecta/1**. Essas relações, tomadas separadamente, formam a rede taxonômica representativa do sistema de contextos, que pode atender a múltiplas finalidades no

processo de derivação de objetivos. A rede taxonômica correspondente a um sistema de contextos pode ser então construída de maneira similar à proposta em [BOW 85] (ver seção 4.4.3), como é exemplificado na **Figura 7.12** para o sistema de contextos apresentado na **Figura 7.11**.

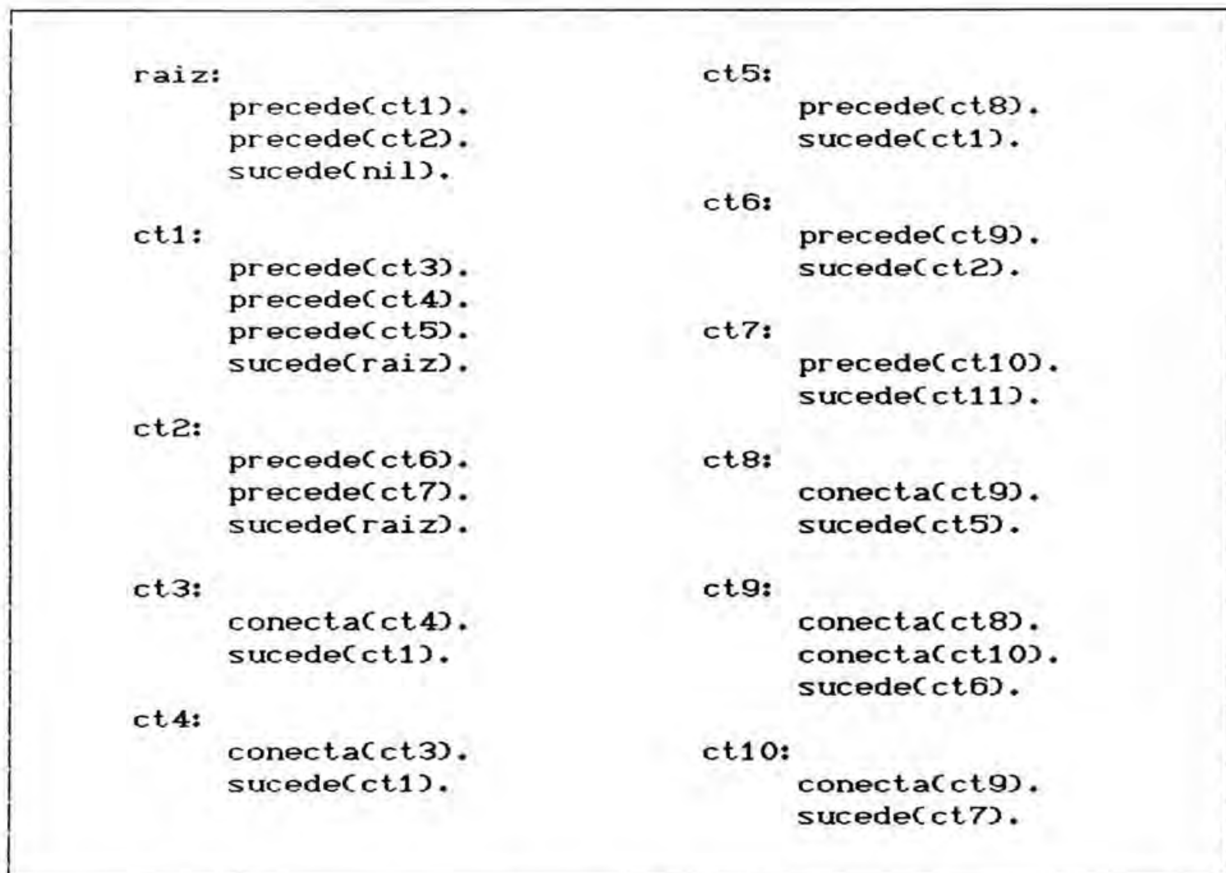


Figura 7.12

Representando a rede taxonômica de um sistema de contextos

7.2.4 Derivando Objetivos em Sistemas de Contextos

A derivação de um objetivo (possivelmente constituído por uma conjunção de sub-objetivos), em um sistema de contextos, é obtida em nível meta através da relação **demonstra/4**, que será verdadeira quando for possível demonstrar que o objetivo pode ser total ou parcialmente derivado no espaço de prova oferecido por esse particular sistema de contextos. A relação **demonstra/4** será

satisfeita quando o sistema de contextos for capaz de produzir um conjunto de substituições que, aplicado às variáveis presentes no objetivo, permita a dedução de todos os sub-objetivos que o constituem como consequências lógicas do sistema de contextos. Quando o conjunto de substituições encontrado permitir a ligação de todas as variáveis presentes no objetivo, dizemos que a derivação é *total*, caso contrário, isto é, se alguma variável permanecer livre, temos uma derivação *parcial*. As ligações produzidas são coletadas na sequência de prova do objetivo, enquanto que os sub-objetivos que ainda contiverem variáveis livres são reunidos em uma coleção que posteriormente será adicionada encabeçando a lista de objetivos remanescentes da seleção original em **demo/3** para se tentar a sua derivação em outro aspecto da BC. O nível mais alto da relação **demostra/4** é apresentado na **Figura 7.13**.

```
demonstra(SisCont,Objs,[],[]) :-
     vazio(Objs).
demonstra(SisCont,Objs,[Prem|RProva],[CObj|CObjs]) :-
     seleciona(Objs,Obj,RObjs),
     mostra(SisCont,Obj,Prem,CObj),
     demonstra(SisCont,RObjs,RProva,CObjs).
```

Figura 7.13

Demonstrando um conjunto de objetivos em um sistema de contextos

Se a relação **demonstra/4** falhar em **demo/3** (ver a **Figura 7.8**), um novo aspecto de BC será gerado por backtracking para a dedução de **Objs**. Caso contrário, duas possibilidades devem ser consideradas: Ou a derivação procurada possui uma solução total, e neste caso não há objetivos pendentes e um novo objetivo será selecionado para derivação em **demo/3**, ou a solução é apenas parcial, e então a lista [CObj|CObjs] conterá uma coleção de objetivos parcialmente derivados que devem ser adicionados aos remanescentes da seleção original para se tentar a sua derivação sobre um outro aspecto da BC. Lembramos que,

para esses objetivos pendentes, o mesmo aspecto tentado anteriormente não deve ser reconsiderado, sendo necessário manter no espaço de prova o registro de que essa tentativa está destinada a falhar, como foi visto na seção 7.2.2. O processo então continua por meio de uma chamada recursiva de **demo/3** visando a solução dos objetivos em **NovosObj**.

O mecanismo de derivação top-down em hiperredes aqui apresentado é, sem dúvida, bastante complexo, entretanto permite claramente a redução do espaço inferencial sobre um esquema de representação de conhecimento de elevada expressividade. Embora não abordados aqui, mecanismos para a herança e síntese de atributos em hiperredes parecem ser de fácil construção, assim como derivações bottom-up. Não se pode deixar de notar ainda o forte apelo que o modelo faz ao emprego de sistemas de processamento paralelo ou concorrente, dadas as suas características estruturais.

A concepção modelo-teorética nos permite entender uma hiperrede como uma representação estática de um esquema de aspectos - isto é, uma teoria - fornecida por meio de um conjunto de entidades estruturadas (hipernodos, hiperrelações e protótipos), contendo restrições de integridade representadas em nível meta através de suas estruturas externas. Um aspecto de uma **BC** representada sobre hiperredes é pois uma interpretação (que deve ser um modelo) dessa teoria. Atualizações executadas sobre um aspecto modificam o modelo, mas não a teoria subjacente, consequentemente as operações de transformação de aspectos preservam a integridade original da **BC**.

Por outro lado, o mecanismo de derivação top-down em sistemas de contextos, parece possuir maior afinidade com o ponto de vista prova-teorético: Os fatos e regras de dedução representados em contextos constituem a própria teoria, ao invés de um modelo de uma teoria mais geral. Os objetivos são vistos como teoremas a ser provados. Não existe um esquema estático, no sentido da semântica modelo-teorética, consequentemente as operações de atualização modificam a teoria. Assim as restrições

de integridade ultrapassam o nível do sistema, controlando as atualizações como uma meta-teoria para a qual o estado do sistema de contextos deve ser um modelo. Na **Figura 7.14** esquematizamos uma representação de tais idéias:

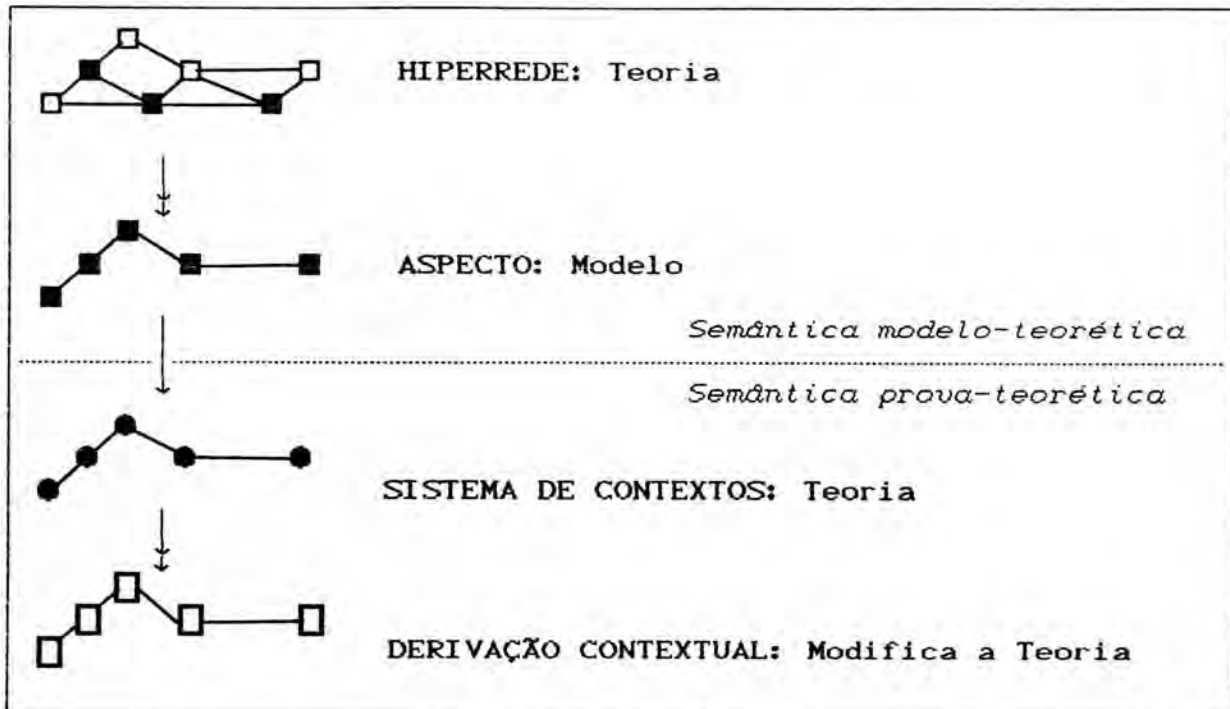


Figura 7.14

Visões semânticas do modelo das hiperredes

Uma possibilidade semântica alternativa seria fornecer aos sistemas de contextos a semântica modal de Hintikka-Kripke [BOW 85], onde contextos seriam vistos como mundos possíveis que conteriam conhecimento a nível meta especificando transições entre si.

7.3 OPERAÇÕES SOBRE HIPERREDES

No capítulo 6, um conjunto de operações básicas foi proposto para o tratamento de hiperredes, a saber:

- instanciação,
- composição,
- decomposição,
- abstração,
- modificação,
- deleção, e
- renomeação.

Na presente seção tentaremos estabelecer o código Prolog correspondente a tais operações, com base no mecanismo de derivação introduzido na seção anterior.

7.3.1 Instanciação

A operação de instanciação corresponde à criação de uma nova PC em concordância com um determinado conceito especificado através de um **protótipo** previamente especificado. O código Prolog correspondente é apresentado na **Figura 7.15**.

```
instancia([Id,prototipo,Sort,Int,Ext,Prop],
          [Id1,Tipo,Sort1,Int1,Ext1,Prop1]) :-
     validos(Id,prototipo,Sort,Int,Ext,Prop),
     validos(Id1,Tipo,Sort1,Int1,Ext1,Prop1),
     coerentes([prototipo,Sort,Int,Ext,Prop],
               [Tipo,Sort1,Int1,Ext1,Prop1]),
     criaPC(Id1,Tipo,Sort1,Int1,Ext1,Prop1).
```

Figura 7.15

A operação de instanciação

Na **Figura 7.15**, o predicado **validos/6** garante a correção sintática das entidades envolvidas na operação de instanciação, enquanto que o predicado **coerentes/2** se encarrega dos aspectos semânticos. Se ambos forem satisfeitos, a nova PC é efetivamente criada através de **criaPC/6**. A operação de

instanciação é portanto empregada para a criação de indivíduos, representados por meio de hipernodos ou hiperrelações, a partir de conceitos formulados através de protótipos.

7.3.2 Composição

Por meio da operação de composição pretende-se criar uma nova PC a partir de duas outras previamente existentes. Dois tipos de composição são possíveis: a composição **inclusiva** e a composição **exclusiva**. O código Prolog a elas correspondente é dado na **Figura 7.16**.

```
compInclusiva([Id1,Tipo,Sort1,Int1,Ext1,_],
              [Id2,Tipo,Sort2,Int2,Ext2,_],
              [Id,Tipo,Sort,Int,Ext,Prop]) :-
    identificador(Tipo,Id),
    Sort = (Sort1,Sort2),
    uniao(Int1,Int2,Int),
    uniao(Ext1,Ext2,Ext),
    geraProp(Prop).

compExclusiva([Id1,Tipo,Sort1,Int1,Ext1,_],
              [Id2,Tipo,Sort2,Int2,Ext2,_],
              [Id,Tipo,Sort,Int,Ext,Prop]) :-
    identificador(Tipo,Id),
    Sort = (Sort1,Sort2),
    uniao(Int1,Int2,Int),
    uniao(Ext1,Ext2,A),
    interseccao(Ext1,Ext2,B),
    diferenca(A,B,Ext),
    geraProp(Prop).
```

Figura 7.16

Composição inclusiva e composição exclusiva

A principal diferença entre as duas formas de composição reside na formulação da estrutura externa da PC resultante. Na composição inclusiva, esta é obtida pela união

das estruturas externas das PCs componentes, enquanto que na composição exclusiva a estrutura externa resultante é fornecida pela diferença entre a união e a intersecção das estruturas externas das PCs componentes. As propriedades da PC resultante são imprevisíveis, sendo necessária a intervenção do usuário na sua especificação, o que ocorre atraves do predicado **geraProp/1**.

7.3.3 Decomposição

A operação de decomposição proporciona um mecanismo para a remoção de um determinado componente da especificação de uma PC. O código Prolog correspondente é dado na **Figura 7.17**.

```
decomposicao([Id1,Sort1,Int1,Ext1,_],
             [Id2,Sort2,Int2,Ext2,_],
             [Id,Sort,Int,Ext,Prop]) :-
     geraId(Id1,Id2,Id),
     diferenca(Sort1,Sort2,Sort),
     diferenca(Int1,Id2,Int),
     diferenca(Ext1,Id2,Ext),
     geraProp(Prop).
```

Figura 7.17

Decomposição de PCs

7.3.4 Abstração

Essa operação abstrai uma classe para um conjunto de PCs, a nível de protótipo. Dois tipos de abstração são possíveis: a abstração **conjuntiva** e a abstração **disjuntiva**. No primeiro uma nova PC é abstraída a partir da intersecção das características das PCs apresentadas como argumentos, no segundo a abstração é obtida através da união de tais argumentos. Em ambos os casos, o resultado obtido será um **protótipo** capaz de gerar, por instanciação, qualquer uma das PCs que participaram na sua criação. O código Prolog destinado à produzir os dois tipos de abstração citados é apresentado na **Figura 7.18**.

```
abstrConj(ListaDePCs,[Id,Sort,Int,Ext,Prop]) :-
     geraId(ListaDePCs,Id),
     intSort(ListaDePCs,Sort),
     intInt(ListaDePCs,Int),
     intExt(ListaDePCs,Ext),
     intProp(ListaDePCs,Prop).

abstrDisj(ListaDePCs,[Id,Sort,Int,Ext,Prop]) :-
     geraId(ListaDePCs,Id),
     uniaoSort(ListaDePCs,Sort),
     uniaoInt(ListaDePCs,Int),
     uniaoExt(ListaDePCs,Ext),
     uniaoProp(ListaDePCs,Prop).
```

Figura 7.18

Abstração conjuntiva e abstração disjuntiva

7.3.5 Modificação

A operação de modificação é obtida por **inserção** de um determinado componente no corpo de uma PC, por **substituição** de um componente por outro ou por **eliminação** de um componente. Todas as modificações estarão sujeitas a provas de consistência implícitas. O código Prolog é dado na **Figura 7.19**.

```
modificacao(insercao(PC1,PC2)) :-
     copia(PC1,Aux),insere(Aux,PC2),
     consistente(Aux),insere(PC1,PC2).
modificacao(substituicao(PC,PC1,PC2)) :-
     copia(PC,Aux),substitui(Aux,PC1,PC2),
     consistente(Aux),substitui(PC,PC1,PC2).
modificacao(eliminacao(PC1,PC2)) :-
     copia(PC1,Aux),elimina(Aux,PC2),
     consistente(Aux),elimina(PC1,PC2).
```

Figura 7.19

Modificação de uma PC

7.3.6 Deleção

Essa operação apaga a PC especificada da base de conhecimento. Note-se que a deleção se dá a nível de BC, isto é, aplica-se apenas a PCs de nível mais alto, sendo portanto diversa da operação de eliminação vista na seção anterior. O código Prolog é apresentado na figura abaixo.

```
delecao(BC, PC, NovaBC) :-
        BC =.. [Nome|PCs],
        membro(PC, PCs),
        remove(PC, PCs, NovasPCs),
        NovaBC =.. [Nome|NovasPCs].
```

Figura 7.20

Deletando uma PC de uma BC

7.3.7 Renomeação

A operação de renomeação subentende a mudança do identificador de uma PC. Essa modificação deve ser executada na descrição da PC e sobre a árvore taxonômica descritiva da BC à qual ela pertence. O código Prolog é dado abaixo:

```
renomeacao(BC, PC, Id,) :-
        BC =.. [Nome|PCs],
        membro(PC, PCs),
        renomeia(PC, PCs, Id, NovasPCs),
        taxonomia(Nome, Tax),
        ren(Tax, PC, Id).
```

Figura 7.21

Renomeando PCs

7.4 VISÕES EM HIPERREDES

O modelo das hiperredes parece adequado principalmente à implementações de grande porte, sendo especialmente favorecido em sistemas de processamento distribuído, concorrente ou paralelo. Tal fato sugere a necessidade do emprego do conceito de *visão*, que pode ser indentificado em uma hiperrede com um conjunto de restrições à formação de aspectos que irá encapsular determinadas partes da hiperrede original. Esse enfoque apresenta elevada flexibilidade, prestando-se adequadamente às operações vistas na seção anterior, uma vez que uma visão pode ser encarada como uma sub-rede da da hiperrede representativa da BC original. Assim o problema da integridade da BC fica reduzido ao problema da integridade das visões que podem ser dela obtidas. Abordaremos agora a representação de visões em sistemas de hiperredes e tentaremos estabelecer uma interpretação semântica informal para a noção de visão sobre esse tipo de estrutura.

7.4.1 Preliminares

O *problema das visões* em bases de dados relacionais tem sido amplamente estudado entre outros por **[VIA 88]**, que enfoca a classe de visões formadas por objetos obtidos por projeção a partir da base de dados original e denominadas *visões de projeção de objetos*. Em tais visões, cada tupla representa um objeto e conjuntos de atributos são caracterizados por meio de dependências funcionais dinâmicas que devem ser satisfeitas pela base de dados. Em nosso estudo, visões são identificadas com conjuntos de fórmulas especificando restrições à formação de aspectos a partir da hiperrede original. Assim, somente é permitida a obtenção de aspectos cuja estrutura satisfizer as restrições impostas pela visão. Na presente seção estabeleceremos alguns conceitos básicos necessários ao estudo de visões sobre o modelo das hiperredes.

Sejam **N**, **R** e **P** respectivamente os conjuntos de hipernodos, hiperrelações e propriedades de uma hiperrede **H**. Uma *visão* sobre **H** é uma tripla **V(H) = (N',R',P')** onde **N'**$\subseteq$ **N**, **R'**$\subseteq$ **R** e **P'**$\subseteq$ **P**. Denotaremos tal visão simplesmente por **V** quando **H** estiver subentendida. O conjunto de todas as visões possíveis sobre **H** é denotado por $V^{[H]}$. Uma visão **V** sobre **H** é dita ser *própria* se $V \in V^{[H]}$ e é possível a extração de pelo menos um aspecto de **V**, isto é, se existe pelo menos uma instância dos elementos constituintes de **V** capaz de produzir um aspecto (de **H**) sobre **V** (ver seção 7.2). Segundo tal conceito, uma visão própria deve encapsular sub-redes da hiperrede original com restrições à produção de entidades que se adicionam às restrições de integridade descritas pela estrutura externa de cada entidade constituinte, garantindo que pelo menos um aspecto (um modelo) da hiperrede (teoria) original será produzido. Tal idéia é representada na **Figura 7.22**, abaixo.

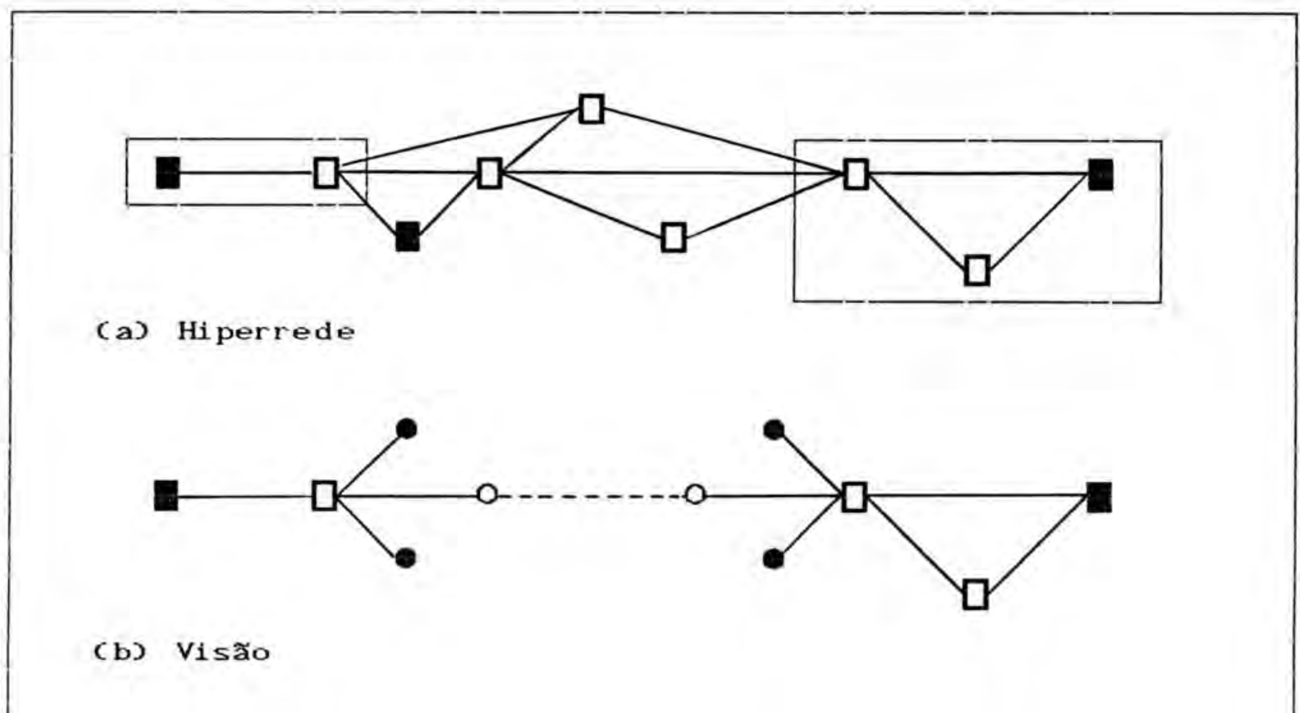


Figura 7.22

Visões em Hiperredes

Sob tal enfoque, visões correspondem a uma certa classe de modelos para a teoria subentendida pela hiperrede, que denominamos *modelos parciais*, onde parte da informação contida na hiperrede original é suprimida ou por não possuir interesse semântico para a aplicação subjacente ou por razões de segurança, eficiência, etc. Isso nos permite modelar uma visão como uma hiperrede derivada, que é um modelo parcial da hiperrede original, apropriada à solução de uma determinada classe de problemas com base no seu conteúdo de informação. É claro que, nesse nível conceitual, uma visão pode ser propriamente representada como uma hiperrede genérica da qual pelo menos um aspecto (não vazio) pode ser extraído, podendo ser trabalhada, para todos os efeitos, de modo semelhante ao apresentado no capítulo anterior para hiperredes representativas de BCs.

7.4.2 Representação de Visões em Hiperredes

A especificação de visões próprias sobre hiperredes irá corresponder à definição de uma PC complexa representada como uma hiperrede capaz de satisfazer aos seguintes requisitos:

Seja **V** uma visão própria sobre uma hiperrede **H**, então vale que:

(1) Se **E** é uma entidade genérica (hipernodo, hiperrelação, protótipo ou BC) e $\mathbf{E} \in \mathbf{V}$, então $\mathbf{H} \vdash \mathbf{E}$,

(2) Se **A** é um aspecto e $\mathbf{V} \vdash \mathbf{A}$, então $\mathbf{H} \vdash \mathbf{A}$, e

(3) Existe pelo menos um aspecto $\mathbf{A} \neq \emptyset$ tal que $\mathbf{V} \vdash \mathbf{A}$.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Na **Figura 7.23**, abaixo, apresentamos o código Prolog correspondente:

```
visaoPropria(BC,V) :-
    V =.. [N|LPC],
    demo(BC,LPC,_),
    aspecto(V,Asp),
    aspecto(BC,Asp).
```

Figura 7.23
Definindo visões próprias em hiperredes

Esse enfoque garante que uma visão própria sempre será estruturalmente consistente com a **BC** que a originou. O problema da consistência da **BC** fica assim reduzido ao problema da consistência das visões que podem ser dela obtidas. O modelo das hiperredes permite também a definição recursiva de visões e confere independência lógica à informação presente na **BC**. Visões podem ser definidas sobre visões de forma estruturada e visões atômicas são identificadas com aspectos da BD. O emprego de visões sobre **BCs** representadas em hiperredes permite claramente a redução do espaço de derivação, contribuindo para a solução do problema do controle. As idéias aqui levantadas são sintetizadas pela **Figura 7.24**

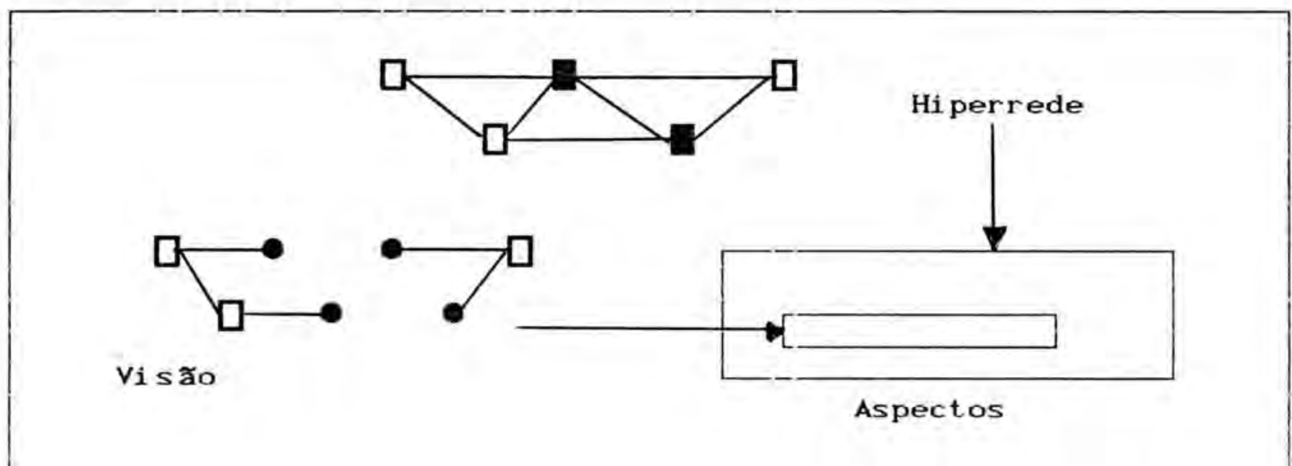


Figura 7.24
Visões como Hiperredes

A coerência estrutural entre a hiperrede representativa de uma visão própria de uma BC e a hiperrede correspondente à BC original preserva a teoria. Uma visão própria é então uma interpretação parcial da BC que é um modelo para o conjunto das sub-redes que interpreta.

Uma vez que as visões assumem a estrutura de uma hiperrede, podemos empregar sobre elas as operações apresentadas na seção 7.3. Duas formas de construção de visões devem ser consideradas: (1) *interativa*, e (2) *derivada*. No primeiro caso a visão é obtida interativamente, sob a supervisão do usuário, porém respeitando as restrições de integridade presentes nas entidades componentes. No segundo, a visão é produzida dinâmicamente em resposta à modificações experimentadas sobre o ambiente da BC. Nosso principal resultado aqui é a concepção de visão como um objeto do modelo, o que permite o emprego, na sua modelagem, dos mesmos mecanismos empregados na construção das demais entidades. Tais mecanismos serão discutidos com maior profundidade nos próximos capítulos.

8 SISTEMAS GERENCIADORES DE BASES DE CONHECIMENTO

Nos capítulos precedentes, discorremos sobre as características de alguns sistemas de programação em lógica, discutimos o uso da lógica na construção de BDs, propomos o emprego de hiperredes para a representação estruturada de conhecimento em lógica e especificamos alguns mecanismos de controle em nível meta capazes de reduzir o espaço de derivação, contribuindo assim para minimizar o problema da explosão inferencial, fator limitante do emprego da lógica como formalismo na representação de BCs de grande porte. Analisaremos agora os requisitos gerais que um SGBC orientado a objetos deve apresentar segundo **[MAT 89]** para suportar efetivamente os conceitos e abstrações associados a esse modelo, tentando definir precisamente quais os mecanismos que devem ser oferecidos em cada um de seus componentes para atender aos requisitos especificados.

8.1 ARQUITETURA

Segundo **[MAT 89]**, SGBCs devem integrar três diferentes classes de funções, oferecendo mecanismos apropriados para:

- Modelar o conhecimento envolvido em uma particular aplicação (isto é, *construir* uma BC),

- Manipular o conhecimento assim modelado (isto é, *aplicar* o conhecimento contido na BC na solução de problemas), e

- Processar a manutenção da BC (garantindo sua integridade, eficiência, etc).

Deve ficar claro que tais sistemas podem somente incumbir-se de tarefas cuja execução não dependa de qualquer informação sobre uma particular aplicação. Como tal, suportam apenas operações que são independentes da aplicação e da classe de problemas considerada, deixando as que apresentam algum tipo de dependência para serem solucionadas pelo particular SC a que se destinam. Sob tal enfoque, as funções que um determinado SGBC deve suportar são as que se relacionam com a BC e com os mecanismos de raciocínio que implementa, isto é, aquelas associadas ao ER correspondente. Estratégias para a solução de problemas, assim como interfaces para SCs, estão relacionados muito de perto com com aplicações do mundo real e suas particulares classes de problemas, devendo, por essa razão, ser mantidas juntamente com o SC correspondente. Assim, as tarefas atribuídas ao componente de solução de problemas ficam divididas entre o SC e o SGBD.

Considerando inicialmente as funções de suporte à construção da BC, podemos dizer que as mesmas são ditadas pelo modelo de desenvolvimento de SCs. Assim, sistemas para o gerenciamento de conhecimento devem, antes de tudo, oferecer meios para suportar o processo de *construção incremental* de SCs. Uma vez que a modelagem de conhecimento é o coração de tal processo, expressividade e exatidão na representação do conhecimento desempenham um papel de suma importância nesse contexto, isto é, os SGBCs devem (1) oferecer ERs que permitam a representação de todos os tipos de conhecimento, e (2) permitir descrever o conhecimento representado de forma independente dos programas de aplicação.

Na análise do ambiente de aplicação de SGBCs, observa-se que há três diferentes aspectos a considerar: as necessidades dos usuários, o suporte oferecido à engenharia de conhecimento e os recursos de hardware e software disponíveis. Assim, para atingir seus objetivos, um SGBC deve oferecer diversos mecanismos que se classificam sob três diferentes pontos de vista: o da aplicação, o da engenharia de conhecimento e o da implementação, como é expresso na **Figura 8.1**.

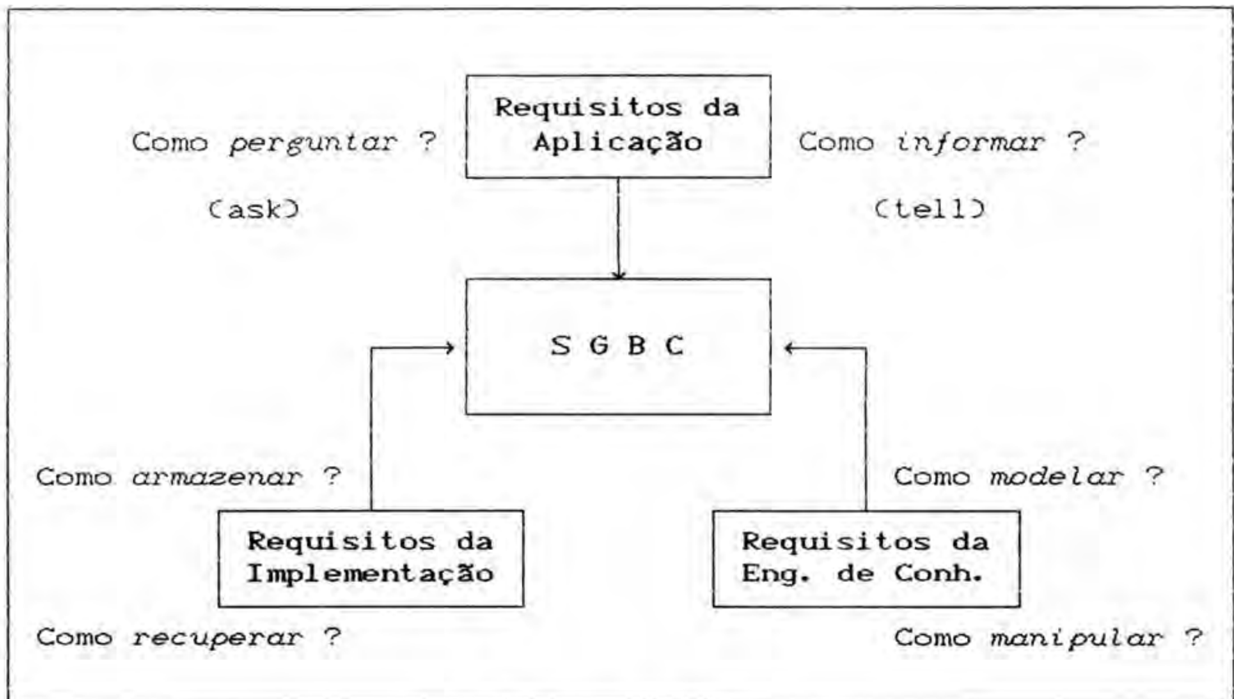


Figura 8.1
Requisitos da Arquitetura de SGBCs

Esses diferentes pontos de vista conduzem a uma divisão natural da arquitetura de SGBCs em tres diferentes enfoques: o da implementação, o da engenharia de conhecimento e o da aplicação. A nível de implementação procura-se definir estruturas de dados adequadas, algoritmos eficientes e outras construções necessárias ao armazenamento e recuperação do conhecimento. O enfoque da engenharia de conhecimento visualiza o SGBC em termos de como o conhecimento necessário a uma aplicação pode ser modelado e manipulado. Em outras palavras este enfoque lida com os aspectos *descritivo*, *operacional* e *organizacional* do modelo adotado para a representação de conhecimento. Finalmente, a nível de aplicação, o conhecimento é tratado de forma abstrata, independente dos aspectos de eficiência considerados a nível de implementação e dos aspectos de modelagem enfocados pela engenharia de conhecimento. Aqui o conhecimento é visto funcionalmente, em termos do que o SGBD "sabe" sobre o domínio de aplicação. Na **Figura 8.2**, abaixo, tentamos organizar melhor tais idéias.

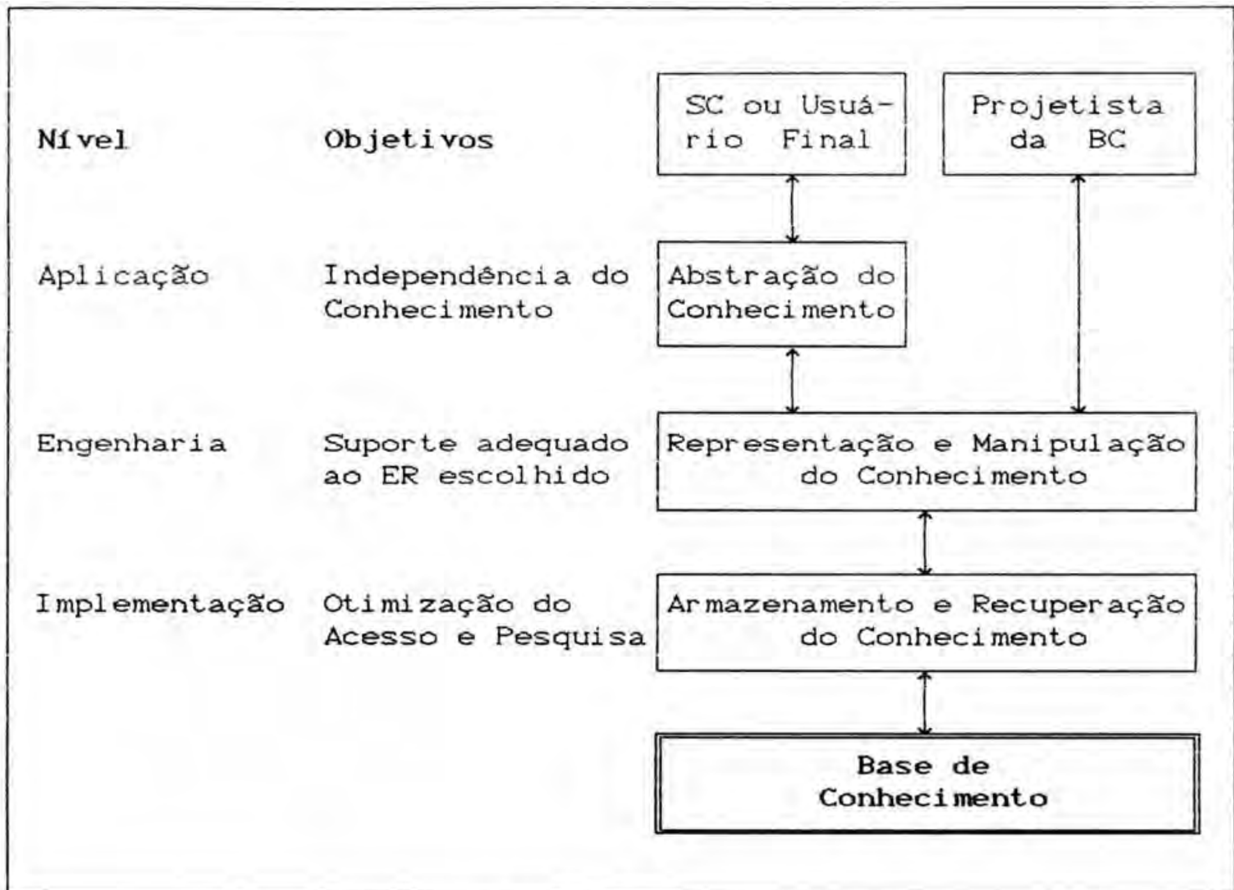


Figura 8.2

Arquitetura Geral de um SGBC

Tradicionalmente os ERs tem sido projetados apenas para suportar o enfoque da engenharia de conhecimento, isto é, preocupando-se apenas com o oferecimento de mecanismos flexíveis para a modelagem e manipulação das estruturas simbólicas que constituem a BC, embutindo em seus programas a forma com que o conhecimento se encontra organizado e as possibilidades de recuperação do mesmo. Uma mudança na estrutura do conhecimento armazenado na BC conduziria a modificações correspondentes nos programas de aplicação. Tal situação, entretanto, não corresponde à desejada *independência do conhecimento*, isto é, os SGBCs seriam continuamente aprimorados por meio de:

- Extensões nas construções de modelagem,
- Introdução de novas operações,

- Adaptações nas estruturas de armazenamento,
- Modificações nas vias de acesso ao conhecimento,
- Reorganização da BC subjacente, ou
- Aplicação de técnicas de acesso mais eficientes,

sem que isso afete os programas de aplicação. Vista dessa forma, a independência do conhecimento pode ser definida como "a imunidade dos SCs em relação a possíveis modificações no correspondente SGBC". Assim, um SGBC deve ser caracterizado funcionalmente em termos do conhecimento que possibilita obter sobre um determinado domínio e não em função dos esquemas de representação que emprega, devendo ser projetado segundo o *princípio de ocultação* do modelo de representação, que garante o confinamento local de todos os tipos de modificações e aprimoramentos realizados sobre o SGBC, isolando dessa forma os SCs da representação adotada.

8.2 O Nível da Aplicação

Do ponto de vista da aplicação, o conhecimento é tratado de modo abstrato, independentemente dos aspectos internos envolvidos em sua representação e implementação, sendo visto funcionalmente em termos do que é possível perguntar ou informar ao SGBC com respeito a um determinado domínio. Assim, a nível de aplicação a BC pode ser comparada a um tipo abstrato de dados que interage com o usuário final ou com SCs através de um conjunto apropriado de operações que permitem formular questões sobre o conhecimento armazenado ou informar ao SGBC novos conhecimentos acerca do domínio de aplicação que devem ser introduzidos na BC.

Uma vez que o conhecimento é tratado como um tipo abstrato de dados, a forma com que o mesmo é aquisitado ou modificado é totalmente transparente do ponto de vista da aplicação. Assim, nenhuma distinção pode ser feita neste nível

entre o conhecimento armazenado extensionalmente e o que deve ser derivado (intensionalmente) da BC. Em consequência, a nível de aplicação, a manipulação do conhecimento se reduz a operações de armazenagem e recuperação. Por outro lado é atribuida a esse nível a responsabilidade de selecionar as estruturas simbólicas adequadas à representação de conhecimento e os mecanismos de recuperação e raciocínio necessários para responder perguntas e assimilar novos conhecimentos.

Consideraremos portanto, nesse nível, a existência de apenas dois tipos básicos de operações: o que permite interrogar (ask) o SGBC com base no conhecimento armazenado na BC e o que permite informar (tell) ao SGBC novos conhecimentos que deverão ser introduzidos na BC. Meios adequados e flexíveis para a execução de tais operações devem ser oferecidos, em correspondência com as muitas formas possíveis de acesso e modificação do conhecimento armazenado na BC. A linguagem de consulta deve portanto ser projetada de forma a suportar as seguintes características:

- **Completeza:** A linguagem de consulta deve ser completa, isto é, deve permitir o armazenamento e a recuperação de todo e qualquer conhecimento capaz de ser representado pelo ER adotado.

- **Extensibilidade:** A qualificação de PCs não deve se restringir aos predicados de qualificação oferecidos pela linguagem de consulta, que deve permitir extensões através de predicados definidos pelo usuário.

- **Projeção:** Ao usuário deve ser facilitada a especificação da forma que deseja para a apresentação dos resultados de uma consulta, permitindo a definição de projeções que suprimam informações desnecessárias quando for o caso.

- **Orientação a conjuntos:** A linguagem de consulta deve permitir ao usuário a obtenção ou modificação de diversas PCs em uma única operação. Esse requisito permite a redução de operações de comunicação entre a aplicação e o SGBC, otimizando o funcionamento dos níveis mais internos.

- **Recursão:** A linguagem de consulta deve suportar processos recursivos sob pena de inviabilizar a desejada independência do conhecimento. O processamento de consultas recursivas através do SGBC potencializa a otimização do sistema como um todo.

- **Modificações orientadas por estados:** Na especificação de modificações sobre a BC, o usuário deve descrever somente o novo estado desejado, correspondente às novas circustâncias do seu domínio de aplicação, sem se preocupar com a forma através da qual tal estado será atingido.

8.3 O Nível da Engenharia de Conhecimento

Um sistema computacional, como é o caso dos SCs, constitui um modelo do mundo real sobre o qual contem informações específicas relacionadas a uma particular aplicação. Assim o processo de especificação de software pode ser visto como o processo de construir um modelo preciso de algum empreendimento. Tal visão corresponde ao enfoque da engenharia de conhecimento, sob o qual um SGBC é interpretado do ponto de vista do projetista da BC ou engenheiro de conhecimento, que considera o SGBC em termos de como o conhecimento correspondente a uma determinada aplicação pode ser modelado. Assim o ponto de partida para o projeto nesse nível corresponde à seleção de um particular ER

para a modelagem do conhecimento envolvido com a aplicação desejada. Tal ER deve permitir uma representação precisa desse conhecimento, refletindo diretamente e de forma natural a concepção que o usuário faz do universo de discurso.

A principal diferença entre esse enfoque e os demais recai na ênfase que é dada aos diferentes tipos de conhecimento que se deseja modelar. Como foi anteriormente colocado, há basicamente três tipos de conhecimento envolvidos na aplicação de SCs. O primeiro tipo é o denominado *conhecimento declarativo*, que corresponde à parte passiva do domínio de aplicação e representa a informação sobre os objetos existentes, seus atributos, restrições, etc. O segundo tipo define as características procedimentais do domínio de aplicacão, sendo denominado *conhecimento procedimental*. Finalmente, o terceiro tipo denomina-se *conhecimento estrutural* e determina a forma através da qual se organizam os dois tipos anteriores. É claro que esses três tipos de conhecimento se refletem em diferentes construções sobre o ER, definindo três diferentes aspectos que denominaremos *descritivo*, *operacional* e *organizacional*, os quais passamos a discutir.

8.3.1 Aspectos Descritivos

Os *aspectos descritivos* de um ER estão relacionados com a parte do domínio de aplicação capaz de ser representada declarativamente, dadas suas características inativas, e correspondem aos *modelos de dados* do estudo de BDs. A grosso modo tais modelos se fundamentam na noção de "registro" e nas ligações existentes entre eles, entretanto essa abordagem parece ser mais apropriada à descrição da forma sob a qual os dados são armazenados e acessados em computadores do que à modelagem de conceitos do mundo real e dos relacionamentos existentes entre eles [BOR 86].

Em contraste com os modelos de dados, o ER oferecido a nível de engenharia de conhecimento deve representar o mundo refletindo exatamente o que ele aparenta ser e não sob uma forma que possa ser ótima para a execução de um determinado programa. Assim, nesse nível, a porção declarativa do domínio de aplicação é vista em termos de *objetos conceituais*, também denominados *entidades*, que possuem *descrições* (isto é, atributos ou propriedades) associadas e permitem estabelecer *relacionamentos* entre si. Além disso, *atividades* (isto é, operações sobre tais objetos) ocorrem ao longo do tempo, resultando em modificações nos objetos e em seus interrelacionamentos. Tanto as descrições de objetos quanto as de atividades estão sujeitas a *restrições*, que definem *conceitos* e permitem distinguir a "realidade" de outros mundos possíveis.

Usamos aqui o termo objeto para referenciar uma estrutura de dados na BC que pretende denotar uma entidade do universo de discurso. Por exemplo, uma BC pode conter um objeto representando a pessoa *José*, além disso empregamos também objetos para representar conceitos abstratos como *pessoa*, *carro*, etc. Essa observação oferece a motivação necessária para o seguinte axioma da modelagem de conhecimento:

> *"Tudo o que existe no mundo real, inclusive as entidades empregadas na descrição de outras entidades, são objetos do modelo"* **[MAT 89]**

Essa visão do mundo torna claro que *tipos* de objetos (isto é, *classes*) são também objetos do modelo, correspondendo a conceitos abstratos do mundo real mencionado acima. Assim, não deve haver nenhuma distinção entre tipos de objetos e suas instâncias, sendo ambos tratados como objetos do modelo. Semanticamente, entretanto, pode haver objetos representando tipos ou objetos representando instâncias. É possivel mesmo que um determinado objeto represente simultaneamente um tipo em um determinado contexto e uma instância em outro. O modelo deve entretanto tratar todos os objetos da mesma maneira até

determinar exatamente a semântica de cada um em um dado contexto. Para a engenharia de conhecimento é então perfeitamente possível aplicar as mesmas operações sobre objetos denotando tipos e objetos denotando instâncias.

8.3.2 Aspectos Operacionais

Na seção anterior, discorremos sobre o lado passivo do domínio de aplicação, definindo-o como uma coleção de objetos inativos possuindo apenas características declarativas. As entidades do mundo real, entretanto, possuem também características procedimentais, de modo que o domínio de aplicação deveria, na verdade, ser definido como uma coleção de *objetos ativos* ou *atores*. A primeira característica procedimental importante de uma entidade é o seu *comportamento*. Por exemplo, as ações que ocorrem quando um carro (objeto) é posto em movimento definem um aspecto particular do seu comportamento.

Na modelagem do domínio de aplicação, tais *comportamentos*, assim como outras características procedimentais das entidades do mundo real, revestem-se de grande importância uma vez que se relacionam intrinsecamente com as propriedades declarativas de tais entidades. Comportamentos, quando ativados, geralmente originam transformações nas propriedades declarativas das entidades do domínio. Por exemplo, o comportamento de um carro em movimento reduz a quantidade de combustível no tanque enquanto aumenta a quilometragem percorrida, exatamente como ocorre no mundo real.

A ativação de comportamentos ocorre através da interação entre objetos. Por exemplo, quando deseja por seu carro em movimento, José interage com ele, ligando a ignição, engrenando a primeira marcha e conduzindo-o. O componente ativado da interação, isto é, o *receptor*, especifica o tipo de

operações a serem executadas. Como uma propriedade de si próprio, o receptor "sabe" quais as ações associadas ao seu comportamento. É importante observar que José não precisa saber nada sobre as ações mecânicas, elétricas e químicas que permitem que seu carro ande. A única coisa que ele precisa saber é como interagir com o carro para colocá-lo em movimento. Uma vez ativado o movimento, o carro se desloca obedecendo a direção de José. Note-se ainda que o comportamento de um objeto não modifica apenas suas propriedades, podendo ativar ainda o comportamento de outros objetos. Por exemplo, uma vez posto em movimento, o carro de José irá certamente ativar diversos outros comportamentos em seus componentes (motor, rodas, sistema elétrico, etc.).

O segundo tipo de conhecimento procedimental encontrado no domínio de aplicação corresponde às *reações* das entidades do mundo real quando uma determinada situação ocorre. Por exemplo, qual será a reação de José quando o carro à frente do seu frear bruscamente? Na descrição de tal conhecimento não estamos interessados em caracterizar o comportamento rotineiro dos objetos, mas sim as consequências de eventos que ocorrem no mundo real. Tais consequências, entretanto, podem somente ter efeito sobre as entidades do mundo real, sendo portanto traduzidas ou por meio da ativação de certos comportamentos dos objetos envolvidos, ou através de modificações sobre o seu conhecimento declarativo. No exemplo acima, uma possível consequência do evento descrito poderia ser a ativação do comportamento de José de também acionar os freios de seu carro, de modo que nada mais aconteça. Outra consequência poderia ser a ocorrência de um acidente, que ocasionaria uma série de modificações na descrição de cada um dos objetos envolvidos (os dois carros, José, o outro motorista, etc.). Outro exemplo desse tipo de conhecimento seria a reação de José ao perceber que o combustível de seu carro está acabando. Neste caso a reação de abastecer o carro (um comportamento de José) deveria ocorrer.

É importante observar que a descrição das situações do mundo real constitui conhecimento declarativo, de forma que as correspondentes reações se relacionam muito de perto com os aspectos descritivos da modelagem do conhecimento, devendo consequentemente ocorrer em conexão com outros objetos e suas propriedades. As reações são, portanto, descrições adicionais de objetos e de seus atributos, devendo ser tratadas exatamente como outra descrição qualquer.

Até aqui as construções analisadas se acham envolvidas com a descrição do domínio de aplicação, expressa por meio de objetos, atributos, ações, relacionamentos e restrições. O modelo de representação deve ainda oferecer construções para descrever o conhecimento necessário à solução de problemas. Parece que a maior parte de tal conhecimento pode ser representado por meio de um conjunto de regras do tipo *situação-ação*. As situações estão normalmente associadas com algum estado do mundo real, enquanto que as ações determinam algum tipo de conclusão com base nesse estado.

Sob esse enfoque, tanto situações como ações possuem uma natureza passiva, definindo aspectos descritivos do domínio de aplicação e, por essa razão, um certo tipo de conhecimento declarativo. Apesar disso, o aspecto mais importante de tais situações e ações não é o seu conteúdo declarativo, mas sim o efeito procedimental resultante da combinação de ambos, ou seja: "Se a BC se encontra na situação **X**, então a ação **Y** deverá ocorrer". Esse aspecto procedimental se relaciona muito de perto com o processo de raciocínio e deve ser suportado pelo ER. Isso significa que o ER deve oferecer construções que permitam a representação de tais regras e, também, mecanismos capazes de interpretá-las, isto é, capazes de executar derivações sobre elas. Outro importante aspecto de tais regras é que elas determinam um tipo de meta-conhecimento, uma vez que definem informações sobre o conhecimento do domínio.

8.3.3 Aspectos Organizacionais

Em nosso cotidiano, raramente descrevemos o mundo real na forma discutida nas seções anteriores, em termos de informação factual específica sobre um domínio de aplicação. Usualmente empregamos construções que tem suas raízes em métodos epistemológicos para a organização do conhecimento. Em outras palavras, na tentativa de modelar o universo do discurso normalmente aplicamos, involuntariamente, algum tipo de abstração para organizar nosso conhecimento na forma desejada. Tais abstrações nos permitem suprimir alguns detalhes específicos de determinados objetos, enfatizando os que são importantes para a solução do problema no contexto da visão adotada. O nível organizacional é portanto a ferramenta fundamental empregada na descrição de conhecimento e por essa razão a mais importante construção a ser suportada pelo ER.

Segundo o conceito de abstração, os objetos do mundo real podem ser *simples*, isto é, definidos em si próprios, ou *compostos*, quando são definidos como uma abstração de outros objetos. Uma abstração é então expressa por meio de relacionamentos entre objetos visando, de alguma forma a organização de tais objetos. Dois tipos de relacionamentos voltados à abstração podem ser encontrados no mundo real. O primeiro envolvendo objetos simples, visando a construção de objetos compostos e o segundo envolvendo objetos compostos na formação de objetos de maior complexidade. Abstrações *do primeiro tipo* são portanto relacionamentos de um único nível, enquanto que as abstrações *do segundo tipo* correspondem a relacionamentos de "n" níveis, uma vez que podem ser aplicados n vezes, possivelmente de forma recursiva, permitindo a representação de objetos de elevado grau de complexidade. A diferenciação entre estes dois tipos de relacionamento é importante devido à particular semântica a eles associada.

8.3.3.1 Classificação

A *classificação* é considerada a mais importante e melhor compreendida forma de abstração, sendo obtida pelo agrupamento de objetos que possuem propriedades comuns em um novo objeto representando todos os demais. Em outras palavras, classificação é uma forma de abstração na qual um objeto composto denominado *tipo* ou *classe* é definido como um conjunto de objetos simples que possuem as mesmas propriedades e são denominados *instâncias*. Isso estabelece um relacionamento *instância_de* entre alguns objetos simples (as instâncias) e um objeto composto (a classe a que pertencem). Assim a classificação pode ser considerado um relacionamento de um único nivel, isto é, uma abstração do primeiro tipo.

A descrição de um modelo em termos de informação factual específica é entretanto pouco satisfatória. Frequentemente é importante extrair os detalhes de cada objeto e tratá-los de maneira genérica, sem considerar os valores específicos de cada propriedade. Na verdade a informação requerida é de natureza genérica, isto é, o que se deseja é uma forma de se referir a todos os objetos pertencentes a uma mesma classe abstraindo-se os detalhes particulares de cada um. A classificação oferece meios para a representação dessa informacão genérica permitindo interpretar uma classe como um objeto representativo ou um *protótipo* de suas instâncias.

A *instanciação* é o inverso da classificação e se propõe a obter objetos que respeitem as restrições associadas às propriedades especificadas por uma certa classe. Uma vez que um objeto pode ser instância de mais uma classe, a instanciação pode ser usada para obter objetos que possuam as propriedades de ambas as classes.

8.3.3.2 Generalização

O conceito de *generalização* complementa o de classificação. Seu objetivo é oferecer uma forma de extrair, de uma ou mais classes dadas, a descrição de uma classe mais geral que captura as propriedades comuns a essas classes suprimindo os conflitos que porventura ocorrerem. A generalização permite então a composição de objetos denominados *superclasses*, definidos como uma coleção de objetos de menor complexidade denominados *subclasses*, estabelecendo o relacionamento *é_um* ou *subclasse_de* entre um objeto e sua superclasse.

Uma vez que a generalização pode ser empregada recursivamente na especificação de superclasses de ordem mais alta, constitui um relacionamento a *n* níveis, que organiza classes em uma certa *generalização*. Em consequência obtem-se uma grande homogeneidade entre certas classes, uma vez que as que possuem superclasses comuns diferem entre si apenas em pequenos detalhes. Isso não é comum em aplicações típicas de BDs, onde tipos de dados (isto é, classes) são normalmente heterogêneos, havendo entretanto grande homogeneidade entre os dados de um mesmo tipo (instâncias).

A metodologia que normalmente empregamos para modelar o mundo real é entretanto fundamentada em um processo que possui um efeito exatamente oposto ao de generalização, isto é, a *especialização*. De acordo com essa metodologia, um modelo é construido inicialmente pela descrição em termos de classes de objetos mais gerais, passando-se depois a lidar com subclasses de tais objetos, especializando-os. Esse processo emprega uma das mais importantes propriedades da generalização, que corresponde ao conceito de *herança*.

8.3.3.3 Herança

Uma vez que as propriedades apresentadas por uma superclasse são normalmente também válidas para suas subclasses, não é necessário repetir a descrição contida na superclasse em cada uma de suas subclasses. As propriedades da superclasse são então *herdadas* por suas subclasses. Como resultado, podemos abreviar descrições sem perder a flexibilidade necessária para especificar os atributos dos objetos do domínio de aplicação. Há duas formas básicas de interpretar a descrição associada a uma classe e, consequentemente, de tratar a herança de atributos. A primeira, que é a forma normalmente adotada pelos ERs para a visualização de classes, é que classes caracterizam simplesmente instâncias prototípicas. Em consequência, propriedades herdadas poderiam ser negadas por subclasses de uma classe ou mesmo por uma particular instância desta. A segunda, que representa a abordagem seguida pelos modelos semânticos de dados, vê a descrição associada a uma classe como constituindo as condições necessárias (propriedades e restrições) que cada instância ou subclasse deve satisfazer. As consequências dessas duas interpretações são as seguintes:

- Se as classes são tratadas de forma prototípica, a herança ocorre por *default*, isto é, as propriedades herdadas são válidas somente se não forem contraditas pela definição da subclasse ou instância. Por exemplo, se há uma propriedade na descrição da classe dos pássaros que diz que todos os pássaros voam e os pinguins são uma subclasse dos pássaros, eles não precisam necessariamente voar.

- Se as classe são tratadas como *padrões* ou *gabaritos*, a herança passa a ser *estrita*, isto é, todas as propriedades herdadas devem existir e ser válidas em cada subclasse e consequentemente em todas as suas instâncias. Nesse caso não seria possivel dizer que todos os pássaros voam, na classe dos pássaros, uma vez que isso não é verdadeiro para todas as subclasses.

À primeira vista a visão prototípica das classes parece ser mais apropriada, uma vez que permite a representação direta de exceções, entretanto essa interpretação termina por enfraquecer o potencial assercional associado à definição de classe. De fato, permitir que propriedades herdadas sejam alteradas em uma determinada subclasse, prejudica a noção de classificação entre as instâncias dessa subclasse e suas superclasses. Por exemplo, não é possível declarar que as instâncias da classe *pinguim* (por exemplo, um certo pinguim), são também instâncias da superclasse *pássaros*, uma vez que não podem satisfazer a todas as propriedades ou restrições nela especificadas.

Por outro lado, analisando mais profundamente a semântica da herança estrita, verificamos que a mesma não somente suporta mecanismos para a representação de exceções, mas também fortifica a semântica da noção de classificação. Aqui as exceções podem ser expressas através de refinamentos sobre a definição de propriedade na especialização de uma classe. Por exemplo, na classe dos pássaros a propriedade *locomoção* poderia expressar que os pássaros voam, nadam ou ambos. Na subclasse dos pinguins essa propriedade seria refinada, de modo que a propriedade locomoção se reduziria a "*nadar*". Consequentemente, todas as restrições especificadas nas superclasses são válidas em suas subclasses e, portanto, todas as instâncias de subclasses são tambem instâncias de superclasses, o que fortalece o conceito de classificação.

8.3.3.4 Associação de Elementos

Frequentemente é necessário agrupar objetos não por causa da existência de propriedades comuns (classificação), mas porque é importante descrever as propriedades desse determinado grupo de objetos como um todo. Digamos que uma certa classe de *veículos* (por exemplo: automóveis, navios e caminhões) possui uma propriedade para expressar seus custos de manutenção. Vamos assumir também que se esteja interessado em representar o

conjunto dos veículos possuídos por uma certa companhia de transportes para expressar propriedades como, por exemplo, o custo médio de manutenção dos veículos da frota. Uma vez que tais propriedades somente podem existir em associação com objetos do domínio, é necessário definir um novo objeto que possua tal propriedade.

Uma primeira idéia seria definir uma nova classe contendo os veículos possuídos pela companhia como instâncias, entretanto, se analisarmos mais uma vez o conceito de classificação, concluiremos que essa modelagem não seria consistente. Uma classe deve abranger a definição de propriedades e restrições associadas que cada uma de suas instâncias deve respectivamente possuir e satisfazer. Consequentemente cada instância dos veículos pertencentes à companhia deveria possuir a propriedade *custo médio de manutenção*. Esta é entretanto uma propriedade que nenhum desses objetos possui ou pode satisfazer. Fica claro então que esta não é uma propriedade dos objetos em si, mas do grupo como um todo.

Para eliminar tal problema empregamos o conceito de *associação de elementos*, de acordo com o qual um novo tipo de objeto denominado *conjunto* é introduzido para representar tais propriedades. Em outras palavras, a associação de elementos é uma forma de abstração na qual alguns objetos, denominados *elementos* são considerados do ponto de vista de um objeto de nível mais alto denominado *conjunto*. Os detalhes dos elementos são suprimidos e as propriedades do grupo são enfatizadas no objeto conjunto, estabelecendo o relacionamento *elemento_de* entre os elementos e o conjunto.

8.3.3.5 Associação de Conjuntos

Como se pode ver a associação de elementos, também denominada *agrupamento*, corresponde a uma parte da teoria dos conjuntos. Portanto, em adição ao relacionamento de um só nível, que constroi objetos compostos (conjuntos) a partir de um grupo

de objetos simples (elementos), é necessário se dispor também de uma forma de expresssar relacionamentos entre objetos compostos (conjuntos), para suportar totalmente a teoria matemática dos conjuntos. Tais relacionamentos estão englobados no conceito denominado *associação de conjuntos*, com o nome *subconjunto_de*. Assim o relacionamento associação de elementos corresponde à relação matemática "∈" (pertence a), ao passo que a associação de conjuntos representa a relação "⊆" (está contido em). Visto dessa forma o relacionamento *subconjunto_de* possui *n* níveis, uma vez que pode ser aplicado recursivamente, definindo, juntamente com a relação *elemento_de*, uma hierarquia entre os objetos do domínio, denominada *hierarquia de associação*. Da mesma forma que na hierarquia é_um, onde cada instância de uma classe é também instância de suas superclasses, os elementos de um conjunto são também elementos de seus superconjuntos. Da mesma forma, um objeto pode ser elemento de muitos conjuntos e conjuntos podem ser subconjuntos de diversos superconjuntos.

Não há herança nas associações envolvendo conjuntos uma vez que as propriedades neles descritas não são características de seus elementos, mas sim do conjunto como um todo. Os conjuntos, entretanto, contém certas propriedades que são válidas para todos os seus elementos. Por exemplo, todos os elementos do conjunto *veículos_da_companhia_de_transporte* possuem a companhia como valor da propriedade *proprietário*. Tais propriedades são denominadas *estipulações de pertinência*. Um conjunto pode tornar-se subconjunto de outro através da restrição de alguma estipulação de pertinência. Por exemplo, os elementos do conjunto *veículos_da _companhia_de_transporte* podem possuir como em relação à propriedade *capacidade_de_carga* os valores *alta*, *média* ou *baixa*, enquanto que o valor dessa mesma propriedade para os elementos do subconjunto *grandes_veículos* se restringe a *alta*. Restringindo as estipulações de pertinência, a determinação das propriedades de um conjunto ficarão restritas a um subgrupo dos elementos do superconjunto. Por exemplo, a determinação da propriedade *custo_médio_de_manutenção* do conjunto *grandes_veículos* se restringirá apenas aos veículos de grande porte, não abrangendo a frota inteira.

8.3.3.6 Agregação

Frequentemente é necessário tratar os objetos de um determinado domínio de aplicação como uma *composição* de outros objetos e não como entidades atômicas como ocorre na classificação, generalização e associação. Por exemplo, pode ser importante em uma certa aplicação descrever o objeto *automóvel* como sendo constituído por um motor, rodas e carroceria, suprimindo entretanto, num primeiro momento, os detalhes específicos dos objetos constituintes. Isso determina um conceito de abstração denominado *agregação*, segundo o qual uma coleção de objetos é vista como um único objeto de nível mais alto.

Da mesma forma que os demais conceito, a agregação envolve objetos simples e compostos. Os objetos simples, denominados *elementos*, expressam entidades atômicas na agregação, isto é, aqueles objetos que não podem ser decompostos. Juntos eles representam os objetos compostos de mais baixo nível, isto é, os *componentes* mais simples. Estes, por sua vez, podem ser empregados na construção de componentes de nível mais alto, de forma que uma *agregação*, ou hierarquia *parte_de* é produzida. Sobre os objetos componentes há um relacionamento *subcomponente_de*, enquanto que entre componentes e seus elementos define-se o relacionamento *parte_de*. O relacionamento *subcomponente_de* estabelece o conceito elementar denominado *agregação de componentes*, em concordância com a associação de conjuntos e a generalização. Por outro lado o relacionamento *parte_de* estabelece o conceito de agregação de elementos, em concordância com a associação de elementos e a classificação.

O conceito de agregação corresponde à noção de propriedade, no sentido de composição, uma vez que expressa, por exemplo, a idéia que os automóveis são constituídos por um motor, rodas e uma carroceria. Mais estritamente podemos dizer que o conceito de agregação descreve as propriedades necessárias que um objeto deve possuir para assegurar sua consistência. Em outras palavras, os objetos componentes não podem existir sem

seus elementos ou partes. Esse requisito torna o conceito de agregação diferente dos demais. Classes, por exemplo, podem existir sem suas instâncias e conjuntos vazios podem ocorrer na associação. As propriedades da agregação são portanto diferentes das dos demais conceitos, não devendo ser interpretadas como características dos objetos, mas sim como outros objetos representando suas partes. Usualmente essas propriedades não mudam ao longo do tempo, uma vez que os objetos dificilmente modificam suas partes. Os automóveis, por exemplo, são sempre construídos a partir dos mesmos componentes. Além disso, objetos com componentes diferentes são, provavelmente, objetos diferentes.

8.3.3.7 Integração de Abstrações

Até aqui, discutimos separadamente cada um dos conceito de abstração. Vimos, por exemplo que a associação de objetos determina uma hierarquia que, até o presente momento, era construida independentemente da hierarquia obtida pela classificação dos mesmos objetos. O mesmo pode ser dito com relação à agregação. Naturalmente, no mundo real, essa independência não existe. Os objetos do domínio de aplicacão são simultaneamente instâncias de classes, elementos de conjuntos e componentes de agregações. Verifica-se então que se considerarmos todos os conceitos de abstração ao mesmo tempo, as descrições de um determinado conceito podem empregar as descrições de outros. Isso ocasiona muito maior expressividade e precisão na descrição das entidades do domínio, que devem ser obtidas segundo o desenvolvimento de modelos que reflitam mais direta e naturalmente a conceituação do usuário sobre o mundo real. Outra vantagem é que os objetos podem ser definidos com maior concisão sem perder a necessária flexibilidade e alguns erros clássicos provocados pela introdução de redundâncias podem ser evitados pela redução do volume de repetições nas descrições.

Portanto os objetos do modelo podem apresentar seis tipos básicos de relacionamentos entre si. A saber:

- instância_de,
- subclasse_de,
- elemento_de,
- subconjunto_de,
- parte_de, e
- subcomponente_de,

correspondendo aos diferentes papéis que representam em cada hierarquia. Consequentemente, o significado semântico dos objetos pode variar consideravelmente entre os diversos contextos de que fazem parte. Isso quer dizer que esse significado semântico pode somente ser determinado em relação a um contexto particular, isto é, juntamente com os correspondentes relacionamentos de abstração. Na **Figura 8.3** apresentamos um resumo esquemático de tais idéias.

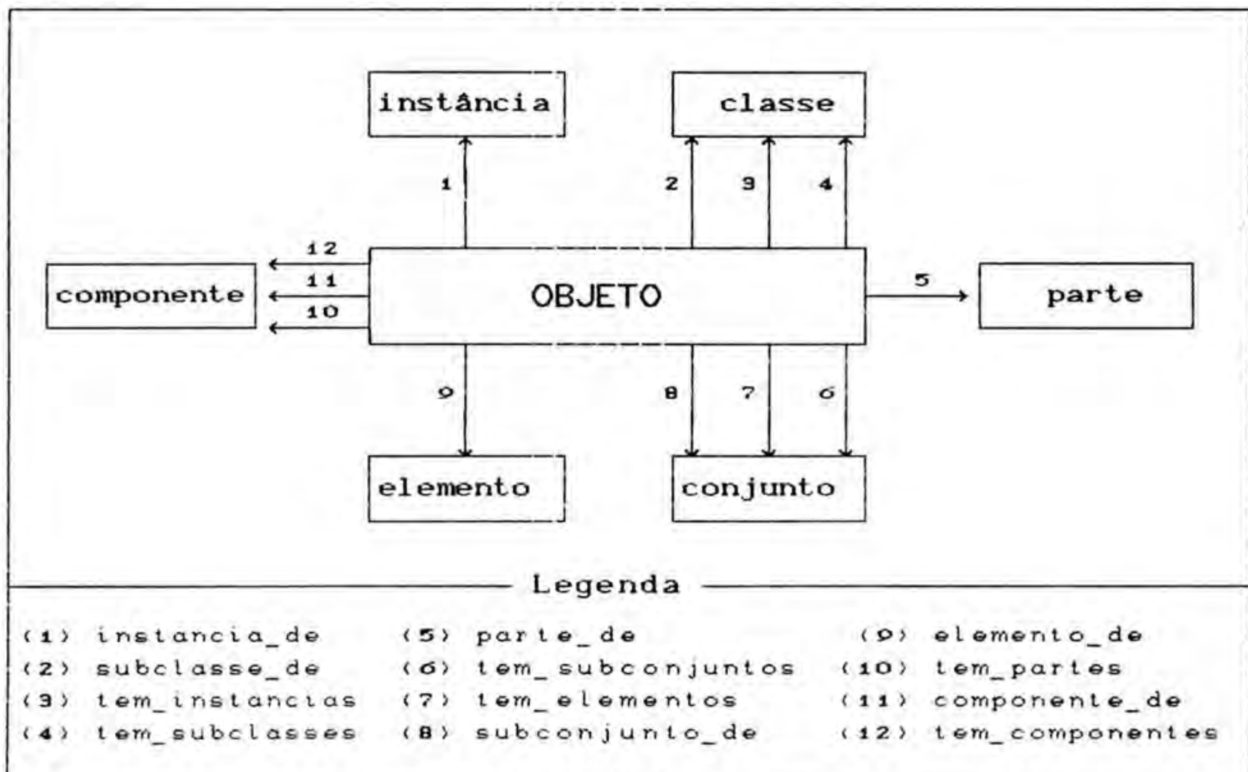


Figura 8.3

Esquema semântico dos conceitos de abstração

A descrição dos objetos do universo de discurso é então passível de refinamento, isto é, vai sendo enriquecida passo a passo na medida em que novos relacionamentos vão sendo considerados. Consideremos inicialmente as propriedades do conceito de classificação/generalização em conjunto com a associação. A concentração de propriedades de classe juntamente com propriedades de conjunto em um único objeto conduz à herança das propriedades de conjunto juntamente com as propriedades de classe para todas as instâncias desse objeto, o que corresponde a uma situação indesejável. Para evitar tal problema é importante diferenciar propriedades de classe de propriedades de conjuntos, de modo que somente as primeiras sejam herdadas. Isso pode ser obtido pela inclusão de um "tipo" associado a cada uma das propriedades do objeto. Assim, propriedades do tipo *instância* seriam as relacionadas com classificação/generalização, enquanto que propriedades do tipo *pertinência* se relacionariam com o conceito de associação. As primeiras seriam herdadas, uma vez que descrevem características das instâncias de uma classe, enquanto que as do tipo pertinência não o seriam, já que descrevem características do próprio conjunto de objetos.

Naturalmente, no caso em que um objeto representa o papel de uma classe em um certo contexto e o de uma instância em outro, é também importante evitar a herança das propriedades do tipo instância. As propriedades herdadas pelo objeto no segundo contexto, onde ele representa uma instância, não devem ser herdadas por suas instâncias no contexto em que ele representa uma classe, uma vez que descrevem características do objeto relacionadas tão somente à sua existência como instância. Por essa razão, iremos generalizar o significado das propriedades do tipo pertinência, não restringindo o mesmo ao conceito de associação. Assim, interpretaremos essas propriedades como expressando características particulares do objeto, seja ele uma instância ou um conjunto. Dessa maneira a herança ocorrerá sobre os relacionamentos de classificação/generalização de acordo com as seguintes regras:

- As propriedades do tipo instância são herdadas como propriedades do mesmo tipo por uma subclasse de uma classe e como propriedades do tipo pertinência por suas instâncias, e

- As propriedades do tipo pertinência não são herdadas em nenhuma hipótese.

Na combinação do conceito de agregação com os demais é também importante diferenciar as propriedades do tipo agregação das demais propriedades que um certo objeto possua, uma vez que os valores de tais propriedades devem ser interpretados como outros objetos do modelo. Deve-se notar portanto que a semântica de tais propriedades difere completamente das relacionadas aos conceitos de classe e conjunto. Enquanto que as propriedades de agregação representam *partes* de objetos, as propriedades de classe e conjunto representam *características* desses objetos. Consequentemente devemos introduzir uma indicação para informar se determinada propriedade descreve um relacionamento de agregação ou não, na descrição de um certo objeto, em adição à sua tipagem como instância ou pertinência. Indicaremos propriedades do tipo *parte_de* como *não terminais*, uma vez que seus valores correspondem a outros objetos do modelo, ao passo que as demais propriedades serão interpretadas como *terminais*.

Por exemplo, considerando novamente o objeto automóvel, verificamos que este possui propriedades terminais tais como *cor*, *proprietário*, *preço*, etc, e também propriedades não terminais como *motor*, *rodas* e *carroceria*. Uma vez que todas as instâncias de automóvel também possuem as partes *motor*, *rodas* e *carroceria*, usamos o conceito de classificação e definimos adicionalmente motor, rodas e carroceria como propriedades de instanciação, de modo que elas são transmitidas a todas as instâncias de automóvel. Neste caso integramos os conceitos de abstração e usamos as características de alguns deles (no exemplo a herança) para a descrição de outros (no caso a agregação). O resultado é

uma descrição *abreviada*. Como não foi empregado o conceito de classificação, as propriedades motor, rodas e carroceria devem ser repetidas em cada instância.

Como foi discutido anteriormente, os relacionamentos de abstração devem ser explicitamente representados no modelo. Uma vez que as propriedades de classe e conjunto não contém a informacão necessária para a representar os correspondentes relacionamentos de abstração, propriedades pré-definidas são introduzidas em cada objeto para expressá-los diretamente. Além disso, combinamos tais propriedades em pares, de forma a obter um modelo simétrico. A classificação é representada pelas propriedades *tem_instâncias*, especificada para uma classe, e o seu "retorno", *instância_de*, especificada nas correspondentes instâncias. A propriedade *tem_subclasses*, definida para uma superclasse, corresponde à propriedade *subclasse_de* em suas correspondentes subclasses. Tais propriedades representam conjuntamente o conceito de generalização. Do mesmo modo a associação de elementos é representada pelas propriedades *tem_elementos* e *elemento_de*, enquanto que a associação de conjuntos é suportada por *tem_subconjuntos* e *subconjunto_de* (ver **Figura 8.3**).

Na representação dos relacionamentos envolvidos no conceito de agregação, poderíamos também empregar propriedades pré-definidas (*parte_de*, *tem_partes*, *componente_de* e *tem_componentes*), entretanto isso não é necessário uma vez que as propriedades da agregação já especificam exatamente o significado de tais relacionamentos. Além disso, expressando cada relacionamento de agregação como uma propriedade, podemos melhor combinar o conceito de agregação com os demais, permitindo a associação de restrições específicas com cada particular relacionamento. Em nosso exemplo, não seria viável a especificação de valores possíveis e cardinalidade para cada subcomponente da classe automóveis, se tais componentes fossem representados juntos através de uma única propriedade pré-definida (*tem_componentes*), como ocorre no caso dos outros

conceitos de abstração. Uma modelagem simetrica é também desejável aqui, de modo que é necessario a especificação de uma propriedade representando referências "para trás", como uma propriedade não terminal a nível de subcomponente ou elemento.

8.4 O Nível da Implementação

O objetivo aqui é lidar de maneira eficiente com o armazenamento e recuperação de conhecimento, em suporte aos níveis de engenharia e aplicação. Assim, muitas das questões levantadas neste nível encontram-se intimamente relacionadas com os problemas tradicionalmente enfrentados no estudo de BDs, aplicados agora a grandes BCs possivelmente compartilhadas por diversos usuários: estruturas de armazenamento, técnicas de pesquisa, eficiência, controle do acesso concorrente, mecanismos de registro e recuperação, etc. Na presente seção discutiremos alguns desses requisitos.

8.4.1 Características de Acesso

O acesso à informacão executado a nível de implementação em ERs orientados a objetos se caracteriza por apresentar reduzida granularidade, isto é, na maior parte das vezes são referenciados atributos particulares, ao invés de objetos completos. É típica também a execução de sequências de acessos sobre um mesmo objeto. Isso ocorre porque normalmente as máquinas de inferência são projetadas de modo a processar uma PC de cada vez, resultando em acessos a pequenos volumes de conhecimento de cada vez. Assim, é importante se dispor de mecanismos capazes de reduzir o caminho de acesso aos conteúdos da BC, permitindo à aplicação referenciar objetos o mais

diretamente possível, o que significa que o projeto de implementação deve priorizar a fácil localização do conhecimento contido na BC.

8.4.2 Tipos de Acessos

Geralmente a maior parte do acesso que uma aplicação faz a uma BC se reduz a operações de leitura e escrita. Operações de inserção e remoção de componentes podem também ocorrer, entretanto sua frequência é menor, uma vez que tais operações estão associadas à modificações estruturais na BC. Em outras palavras, durante uma consulta, modificações na estrutura da BC como, por exemplo, a mudança de tipos de objetos ou a adição de novas instâncias a um tipo, são raras. Por outro lado, durante o processo de construção da BC a situação se inverte. Essa grande *estabilidade* da BC deve ser explorada na escolha das estruturas que orientarão o seu armazenamento. Uma vez que a construção da BC é um processo interativo e incremental e, consequentemente, não requer um desempenho tão elevado quanto o processamento de consultas, devemos dar prioridade à otimização de operações de recuperação.

8.4.3 Frequências de Acesso

As frequências de acesso aos atributos de um objeto podem também apresentar grandes disparidades. Atributos dinâmicos, isto é, os que representam o conhecimento inferido durante uma consulta ou informação específica sobre o caso que está sendo analisado, possuem uma frequência de acesso muito alta. Por outro lado, os atributos estáticos, que representam o conhecimento especializado de um particular SC, possuem uma frequência de acesso menor. Isso ocorre principalmente em função do modo com o que o mecanismo de raciocínio emprega o conhecimento. Tipicamente , cada peça de conhecimento estático,

por exemplo, a *ação* associada a uma regra, é empregada uma única vez durante o processo de raciocínio, quando o SC necessita inferir um particular conhecimento (ou seja, o conhecimento dinâmico que a regra permite inferir). Entretanto, cada peça de conhecimento dinâmico, por exemplo a informação de que um determinado paciente tem febre, é necessária em diversas ocasiões. Uma vez que essa informação pode ser usada para inferir diversas outras PCs (a informação "febre" pode ocorrer como condição em diversas regras), ela é usada e portanto acessada muito frequentemente.

Para bem empregar tal característica de acesso no armazenamento de objetos na BC, a implementação deve oferecer meios para armazenar os atributos dinâmicos em separado dos atributos estáticos. Estruturas de acesso projetadas dessa forma contribuiriam, sem dúvida para otimizar a localização de conhecimento, reduzindo simultaneamente a frequencia de acesso e o *overhead* de transferência de informações que nem sempre são necessárias.

8.4.4 Processamento de Conhecimento

Tem sido observado que, em cada fase do processo de solução de problemas, os acessos se concentram nos conteúdos de um grupo restrito de objetos da BC, uma vez que os SCs necessitam de diferentes partes da BC para solucionar diferentes problemas. Os objetos acessados representam, portanto, o conhecimento necessário para satisfazer um objetivo específico em uma determinada fase do processamento. Como foi estabelecido no capítulo 6, tais objetos constituem um *aspecto* da BC, que permite derivar um sistema de *contextos*. O emprego desse sistema de contextos no processamento de conhecimento depende, portanto, da particular estratégia de solução empregada. Por exemplo, esquemas que exploram a propagação de restrições possuem contextos correspondendo a diversas restrições definidas na BC. Da mesma forma uma estratégia voltada à redução de problemas determina contextos para o processamento de cada subproblema.

8.4.5 Manutenção Temporária do Conhecimento Dinâmico

Uma vez que o conhecimento dinâmico é produzido no decorrer de uma consulta e empregado na obtenção de uma resposta particular, ao final da consulta ele deixa de ser significativo. Por outro lado, o conhecimento estático apresenta-se como relevante para todas as consultas. Assim a implementação deve considerar que os atributos dinâmicos representam conhecimento temporário, enquanto que os atributos estáticos devem ser armazenados em caráter permanente na BC.

8.4.6 Ambientes Multiusuários

Podemos imaginar que, em ambientes multiusuários os atributos estáticos devam ser acessados concorrentemente por cada usuário para a inferência do conhecimento dinâmico correspondente à sua particular consulta. Assim é razoável propor que uma única cópia do conhecimento estático seja mantida, enquanto que o conhecimento dinâmico deve ser mantido em uma versão para cada usuário, constituindo cada uma um certo tipo de conhecimento privado associado a uma aplicação específica, como pode ser visto na **Figura 8.4**.

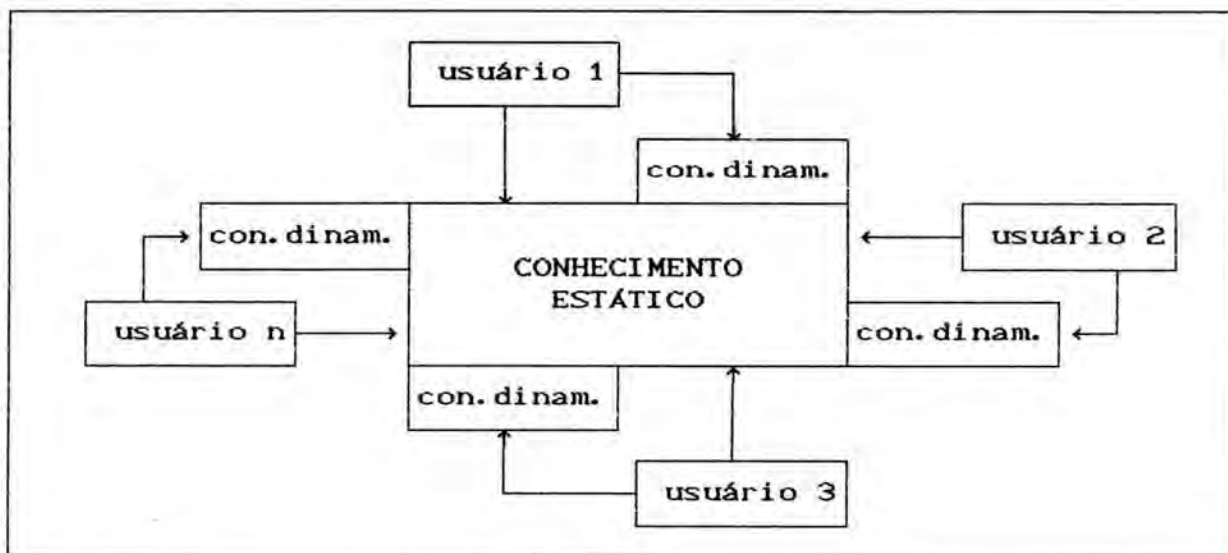


Figura 8.4

Organização do conhecimento em ambientes multiusuários

8.4.7 Estruturas de Conhecimento

As estruturas sob as quais o conhecimento é organizado são normalmente muito complexas. Frequentemente os objetos na BC são construídos como uma composição de outros objetos já existentes, de modo que a implementação deve oferecer mecanismos eficientes para representar tais estruturas minimizando o emprego de redundâncias e flexibilizando o tratamento de objetos complexos. Além disso, os atributos associados a um objeto podem apresentar características de multiplicidade, assim como um significado particular para cada tipo de relacionamento. Em geral, não é possível estabelecer um limite para o conhecimento contido em um atributo e, além disso, este pode ser dinamicamente modificado no processamento de uma consulta. Em consequência, o SGBC deve empregar, em sua implementação, estruturas de armazenamento e métodos de acesso mais sofisticados do que os usualmente empregados em SGBDs.

9 RHESUS: UM SISTEMA EXPERIMENTAL

Com base nas idéias discutidas no capítulo anterior, apresentaremos agora uma proposta para o desenvolvimento de um SGBC que emprega o modelo das hiperredes como formalismo básico para a especificação de objetos e seus relacionamentos a nível de engenharia de conhecimento. Denominamos tal sistema **Rhesus** em alusão à sua finalidade experimental. Os níveis de aplicação e implementação escapam da abrangência pretendida para o presente trabalho, entretanto, serão igualmente abordados conquanto necessários para a perfeita caracterização e satisfação dos requisitos de viabilidade de nossa proposta. Inicialmente proporemos uma estrutura global para o sistema Rhesus, tentando depois definir precisamente quais os mecanismos que devem ser oferecidos em cada um de seus componentes.

Sob a ótica da pesquisa corrente em BDs, o emprego de hiperredes na representação de conhecimento caracteriza-se por ser *orientado a objetos*, possuindo como tal tres principais vantagens sobre os tradicionais modelos hierárquico e relacional. Em primeiro lugar, permite visualizar uma BD como uma coleção de objetos abstratos, ao invés de um conjunto de tabelas pré-definidas possivelmente interrelacionadas. Depois, suporta construções explícitas para a representação das abstrações estudadas no capítulo 8 (classificação/generalização, associação e agregação). Finalmente captura com maior facilidade a noção de restrições de integridade, favorecendo sua propagação e o encapsulamento de entidades. Em particular os atributos de objetos abstratos podem ser vistos como funções, permitindo, segundo **[KIN 86]**, muito maior precisão na especificação de tais restrições.

Devido a essa expressividade "extra" a modelagem orientada a objetos é também denominada *modelagem semântica*, uma vez que oferece mecanismos capazes de capturar mais da semântica de um particular domínio de aplicação do que é possível através dos modelos tradicionais. Deve ter ficado claro também, a partir da leitura do capítulo 8, o quanto o modelo das hiperredes é adequado à representação de objetos, possuindo mesmo soluções naturais, inerentes à sua própria estrutura, para muitas das questões ali levantadas.

9.1 A ARQUITETURA DO SISTEMA RHESUS

A proposta de adicionar lógica ao modelo das hiperredes em um sistema híbrido evolui naturalmente a partir da percepção de que ambos os modelos componentes possuem características que se complementam em muitos pontos. Por exemplo, a facilidade em se definir contextos de prova em hiperredes constitui um mecanismo que permite o controle do crescimento potencial das inferências do componente lógico. Além disso a estrutura das hiperredes oferece um suporte adequado aos raciocínios *top-down* e *bottom-up* que podem ser implementados em lógica com relativa facilidade. A arquitetura básica do sistema Rhesus, apresentada na **Figura 9.1**, baseia-se na utilizada pelo sistema Intexp [GEO 85a] e se caracteriza por ser independente de domínio, permitindo a coexistência de pelo menos tres linhas de raciocínio:

- Raciocínio orientado à recuperação, baseado em herança e síntese de atributos,
- Raciocínio dedutivo com backtracking automático, fornecido pelo interpretador Prolog subjacente, e
- Raciocínio circunstancial, suportado por encadeamento para frente.

A essas três formas básicas de raciocínio podemos facilmente acrescentar outras diretamente associadas aos conceitos de abstração estudados na seção anterior e que serão discutidas mais adiante. Caracterizamos o sistema Rhesus como *experimental* uma vez que a complexidade resultante do modelo empregado dificulta a previsão à priori de como seu comportamento pode ser afetado pela liberdade estrutural que apresenta. É nosso intuito utilizá-lo, num primeiro momento, para melhor compreender as regras gerais que determinam a estruturação do conhecimento face aos métodos de raciocínio empregados na solução de problemas em diversas classes de domínios.

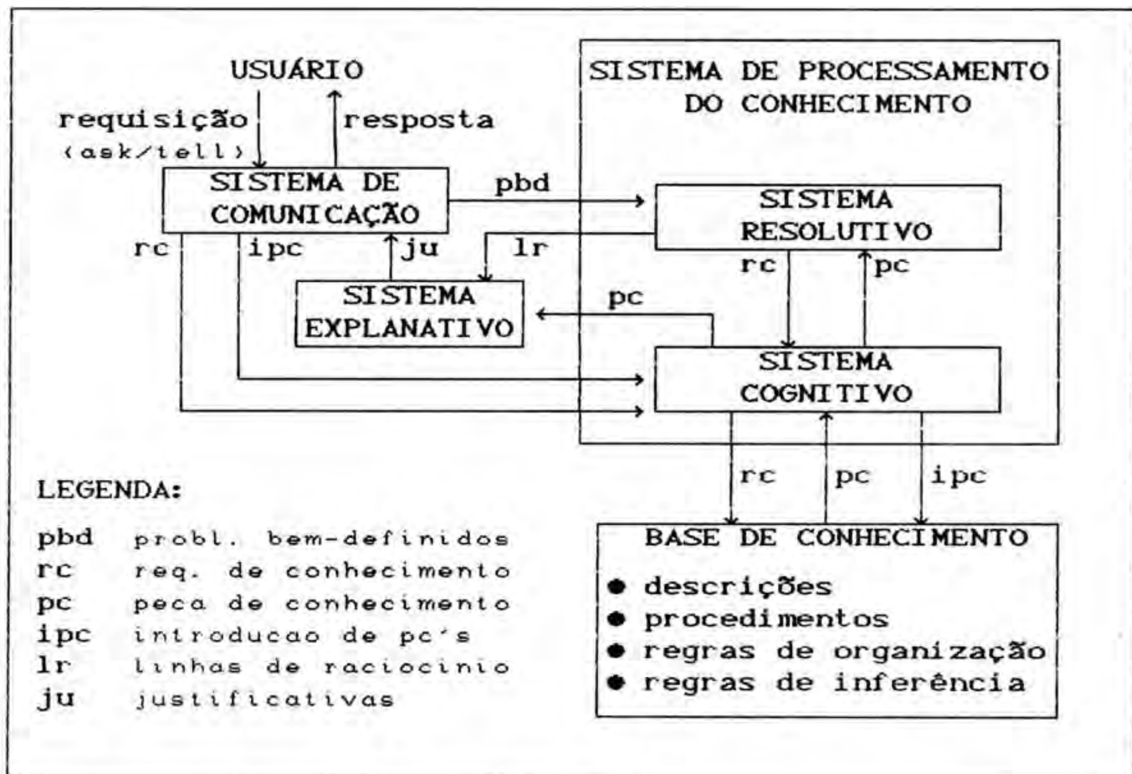


Figura 9.1

Arquitetura proposta para o sistema Rhesus

O sistema Rhesus foi concebido arquiteturalmente de modo a implementar as quatro atividades constituintes do *ciclo cognitivo*, a saber: aquisição, memorização, aplicação e explicação (ver seção 2.1). A aquisição de conhecimento é suportada pelo sistema de comunicação que estabelece, através de

uma linguagem de consulta, um interface entre o usuário e o sistema capaz de suportar operações do tipo *ask* e *tell*, conforme requisição do usuário, bem como traduzir, para uma forma adequada à sua compreensão, os resultados internamente produzidos. O componente de memorização é representado pela BC, modelada sob a forma de um esquema de hiperredes, que se comunica com os demais componentes por meio do sistema cognitivo, devotado ao armazenamento e recuperação de conhecimento estático. Este último componente, juntamente com o sistema resolutivo, integra o que denominamos sistema de processamento do conhecimento, responsável pela aplicação do mesmo na solução de problemas e produção dinâmica de conhecimento. Finalmente a atividade de explicação, que justifica ao usuário as decisões tomadas na solução de um determinado problema, é executada pelo sistema explanativo, completando assim o ciclo. Nas seções seguintes abordamos cada um dos módulos componentes do sistema Rhesus do ponto de vista da aplicação, da engenharia de conhecimento e da implementação, procurando definir suas funções e detalhar os mecanismos necessários ao seu funcionamento.

9.2 A BASE DE CONHECIMENTO

Do ponto de vista do presente trabalho, a BC constitui o principal componente do sistema Rhesus e pode ser identificada com uma coleção de objetos abstratos, construídos a partir das três classes primitivas de entidades: hipernodos, hiperrelações e protótipos. A própria BC pode ser vista como um objeto abstrato e o modelo adotado permite que uma BC contenha outras BCs, visões, aspectos, etc. A liberdade estrutural inerente ao modelo das hiperredes favorece a especificação de múltiplas combinações das entidades contidas na BC, respeitadas as restrições de integridade formuladas em suas estruturas externas. Na **Figura 9.2** representamos uma classificação geral para as entidades capazes de ser representadas em BCs segundo o modelo proposto.

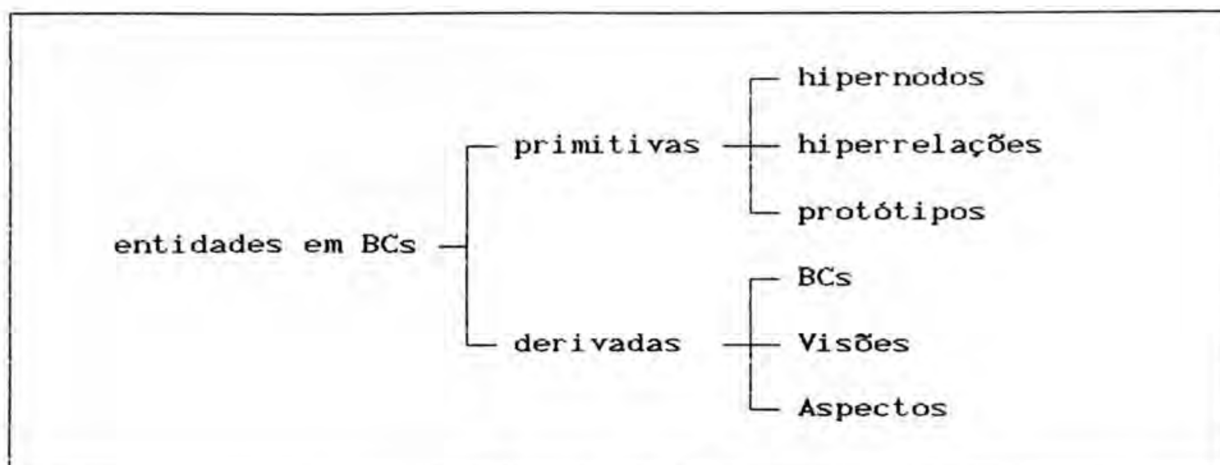


Figura 9.2
Entidades em BCs

É conveniente neste ponto traçarmos um paralelo entre as entidades primitivas do modelo que adotamos e as três classes de conhecimento que devem ser consideradas a nível de engenharia, a saber, conhecimento declarativo, procedimental e organizacional. O conhecimento capaz de ser representado declarativamente corresponde ao conhecimento passivo da BC, e tem nos hipernodos o seu veículo natural de representação. Hiperredes (objetos complexos) constituídas somente de hipernodos formam as entidades que denominamos *Aspectos* (ver capítulo 6). O conhecimento procedimental encontra sua representação natural nas hiperrelações e protótipos, onde as primeiras podem ser vistas vantajosamente como procedimentos que recebem como entrada hipernodos que são instâncias de objetos conceituais representados por alguns dos protótipos nelas descritos, fornecendo como saída instâncias (totais ou parciais) de outros. Finalmente, o conhecimento organizacional é representado em todos os objetos do modelo por meio de seu *sort*, sua estrutura externa e suas propriedades.

Na modelagem de conhecimento que adotamos, tanto as entidades primitivas como as derivadas são representadas na BC por meio de *descritores* que, em todos os casos, assumem a seguinte formulação sintática:

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

<identificador>(Tipo, Sort, Int, Ext, Prop)

onde, resumidamente:

- **<identificador>** é o *nome* da entidade. Toda entidade especificada em uma BC possui um nome único representado por uma constante da linguagem lógica subjacente. O identificador de um protótipo é uma variável formal, representada por uma sequência de caracteres iniciada por "@". Às entidades produzidas internamente o sistema atribui um nome no formato

  <tipo>nnnnn

  onde **<tipo>** é o tipo da entidade e **nnnnn** é um número entre 00000 e 99999. Protótipos gerados internamente recebem, por exemplo, nomes como "@prototipo00003".

- **Tipo** é o *tipo* da entidade, representado em seu descritor por uma constante conforme a classificação apresentada na **Figura 9.2**. Assim, segundo o seu *tipo*, uma entidade pode ser:

  hipernodo
  hiperrelacao
  prototipo
  bc
  visao
  aspecto

  Ainda que determinadas entidades possam ser sintaticamente tomadas indistintamente como BCs ou visões, a especificação do tipo é necessária para a manutenção da integridade semântica do sistema.

- **Sort** identifica uma classe de entidades que compartilham as mesmas características estruturais. Em geral um sort é representado por uma constante da linguagem, entretanto, o sort de um protótipo pode ser representado também por uma variável formal.

- **Int** é a *estrutura interna* da entidade: uma hiperrede de nível mais baixo representando o seu *conteúdo de informação*. Se a entidade for *atômica*, sua estrutura interna será constituída por um conjunto de cláusulas descrevendo um objeto ou procedimento. Se for complexa, sua estrutura interna corresponderá a uma adequada combinação de entidades cujo detalhamento conduzirá à construção da hiperrede correspondente.

  Não se descarta a existência de entidades atômicas, do tipo hiperrelação, cuja estrutura interna corresponda a um procedimento em linguagem executável, principalmente em se tratando de primitivas de E/S ou cálculos matemáticos. O esquema de representação deve ser flexível o bastante para tratar os objetos assim representados, integrando-os no mesmo ambiente das demais entidades. Isso não apenas é possível, como altamente desejável, uma vez que dotaria o modelo das hiperredes de abrangência suficiente para ser empregado como um formalismo de propósito geral na construção de SCs da mais elevada complexidade.

- **Ext** é a *estrutura externa* da entidade: um conjunto de restrições encapsulando os componentes da estrutura interna e as propriedades da entidade e especificando quais deles podem participar em que tipos de relacionamentos e com que tipo de objeto. A principal finalidade da estrutura externa é, além de oferecer uma forma para o tratamento de exceções, impedir a geração de combinações inválidas ou absurdas. Por meio da

estrutura externa pode-se, por exemplo, bloquear determinados relacionamentos que de outra forma poderiam ocorrer e restringir qualquer relacionamento especificado pelas propriedades da entidade.

- **Prop** é uma lista de propriedades genéricas representadas por meio de pares *atributo-valor*, onde *atributo* representa o identificador da propriedade e *valor* corresponde a uma lista. Tais propriedades podem se propagar, por herança, aos objetos que são instância de uma determinada classe. Além disso, devem ser especificadas conceitualmente na forma integrada discutida no capítulo anterior, satisfazendo completamente e de maneira simétrica aos conceitos de classificação/generalização, associação e agregação.

A idéia de empregar um único descritor para a especificação de entidades de qualquer tipo garante elevada flexibilidade ao modelo e permite um tratamento uniforme às diferentes categorias de conhecimento representadas em hiperredes. Por exemplo, procedimentos podem ser tratados como dados na obtenção de sistemas de contextos. Essa liberdade no estabelecimento de associações deve ser apropriadamente cerceada por meio das restrições de integridade contidas na estrutura externa de cada entidade, que irão garantir que somente as combinações desejadas serão produzidas. Por outro lado, essa característica do modelo das hiperredes torna-o bastante indicado à pesquisa congnitiva sobre tipos polimórficos.

Também em decorrência da decisão de implementar um único tipo de descritor, todas as entidades presentes em uma BC podem ser objeto das operações sobre hiperredes definidas na seção 7.3, a saber:

- **Instanciação:** Uma nova entidade é criada em concordância com um determinado conceito especificado através de um protótipo.

- **Composição:** Uma nova entidade é criada a partir de outras já existentes, por classificação, generalização, associação ou agregação.

- **Decomposição:** Um determinado componente é removido da especificação de uma entidade.

- **Prototipação:** Um novo conceito é produzido, a nível de protótipo, a partir de um conjunto de entidades.

- **Modificação:** A modificação de uma entidade é produzida por *inserção* de um determinado componente, por *substituição* de um componente por outro ou por *eliminação* de um componente da entidade.

- **Deleção:** A entidade especificada é removida da BC.

- **Renomeação:** O identificador da entidade é substituído por outro especificado pelo usuário.

Além das operações citadas acima, o sistema deve oferecer mecanismos para a manutenção da BC, tais como salvamento e recuperação em volumes físicos, extração de dicionários conceituais e árvores taxonômicas, reorganização da BC, criação de novas BCs (ou visões) por cópia seletiva de classes de entidades e entidades individuais de acordo com requisitos taxonômicos especificados, refinamento de entidades e manutenção on-line.

9.3 O SISTEMA DE PROCESSAMENTO DO CONHECIMENTO

Como foi dito anteriormente, o sistema Rhesus procura implementar as quatro atividades constituintes do ciclo de conhecimento (ver **Figura 2.1**): aquisição, memorização, aplicação e explicação. O *Sistema de Processamento do Conhecimento*, que denominaremos SP, é responsável pela atividade de *aplicação* do conhecimento disponível na BC, atendendo às solicitações do usuário.

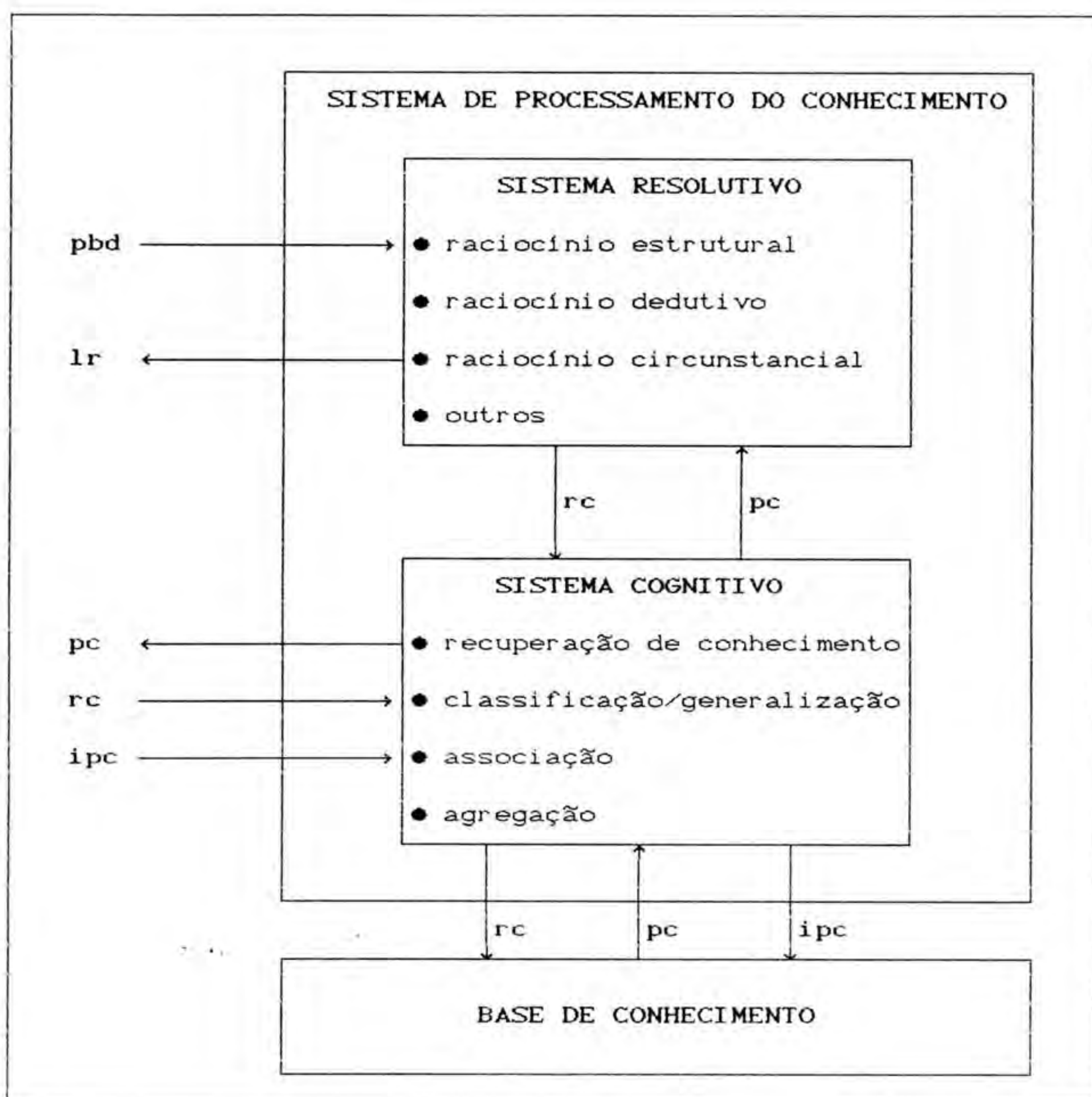


Figura 9.3

O Sistema de Processamento de Conhecimento

Julgamos conveniente, para bem determinar as ações que o SP deve executar, dividi-lo em dois principais componentes: O *Sistema Resolutivo* (SRes) e o *Sistema Cognitivo* (SCog), este último, subordinado ao primeiro, é o único componente que possui acesso direto à BC, servindo de interface entre esta e os demais componentes do sistema Rhesus (ver **Figura 9.1**). Uma visão mais detalhada da estrutura do SP é dada na **Figura 9.3**, onde especificamos a divisão das atividades entre os seus componentes, assim como o fluxo de informação entre estes e os demais componentes do sistema.

Assim, dois *níveis* de raciocínio estão previstos no SP. O primeiro, suportado pelo SRes, é mais elaborado e atua sobre os objetos (PCs) fornecidos pelo SCog. Este, por sua vez, tem sua atuação limitada aos relacionamentos de abstração apresentados no capítulo 8 e é fortemente orientado à recuperação direta do conhecimento representado na BC. Nas seções seguintes detalharemos as atividades desempenhadas por esses dois componentes.

9.3.1 O Sistema Resolutivo

O Sistema Resolutivo (SRes) é acionado pelo Sistema de Comunicação (SCom) para solucionar *problemas bem-definidos* (pbds) formulados pelo usuário ou aplicação de nível mais elevado. Sua primeira tarefa é obter, por meio de raciocínio estrutural sobre hiperredes, a visão particular ou modelo da BC que irá empregar para esse fim, conforme foi explicado no capítulo 7. Uma vez que isso esteja determinado, o SRes emprega para a solução do problema proposto: ou o raciocínio dedutivo, suportado pelo Prolog subjacente para a obtenção de derivações *top-down*, ou uma forma de raciocínio circunstancial, suportada por encadeamento

para frente, ou ainda combina essas duas formas de raciocínio, atendendo a especificações do usuário ou a regras heurísticas formuladas sobre o tipo de conhecimento disponível e a inferir. Outras formas de raciocínio poderiam ser implementadas a nível de SRes, como o raciocínio analógico, ou mesmo o tratamento de modalidades sobre objetos representados como hiperredes. Essa variedade potencial de estratégias de raciocínio suportadas reforça a condição experimental do sistema Rhesus, que oferece um *framework* muito adequado à pesquisa aplicada nesse campo. Analisaremos, na presente seção, as tres formas principais de raciocínio propostas para esse componente do SP.

9.3.1.1 Raciocínio Estrutural

Essa forma de raciocínio baseia-se nas leis de formação de hiperredes, interpretando a BC como um esquema e dele extraindo uma particular visão ou modelo para a solução de um determinado problema proposto. As atividades executadas com esse fim, conforme especificado na seção 7.2, são as seguintes:

(1) Obtenção de um modelo da BC,

(2) A geração de um aspecto desse modelo, e

(3) A contextualização desse aspecto.

O produto final do raciocínio estrutural é então um sistema de contextos, estruturalmente consistente com uma visão particular da BC, sobre o qual outras formas de raciocínio passam a ser empregadas para a solução do problema proposto. Essa estratégia de raciocínio, voltada à obtenção de contextos, foi estudada em profundidade no capítulo 7.

9.3.1.2 Raciocínio Dedutivo

Essa abordagem, também denominada *top-down* ou raciocínio orientado a objetivos, é suportada no SRes pelo Prolog subjacente. A idéia aqui é focalizar somente os objetos que são relevantes para a solução do problema, a partir da especificação de um determinado objetivo. O mecanismo de dedução tenta encontrar, no sistema de contextos produzidos através do raciocínio estrutural, quais as regras (ou predicados) capazes de satisfazer (total ou parcialmente) o objetivo proposto. Após encontrá-las, formula novos objetivos visando a satisfação das condições nelas estabelecidas. É claro que, se há diversas regras potencialmente capazes de satisfazer o objetivo corrente, o sistema deve ter condições de escolher qual delas será tentada em primeiro lugar, por meio de alguma estratégia de resolução de conflitos. Uma vez que uma regra frequentemente possui diversas conclusões, o processo de satisfação de um objetivo costuma normalmente ser transformado no problema de solucionar diversos sub-objetivos. O objetivo original é satisfeito quandos todos os seus sub-objetivos forem solucionados. Na seção 7.2.4 propomos um mecanismo, formulado através do predicado **demonstra/4** para representar esse tipo de raciocínio.

9.3.1.3 Raciocínio Circunstancial

Nesse tipo de raciocínio, também conhecido como *bottom-up* ou encadeamento para frente, a aplicabilidade de uma determinada regra ou predicado é determinada pelo *estado corrente* do conhecimento disponível. Assim o processo de inferência parte de um estado inicial, em um certo contexto, e produz passos intermediários (novos estados) por meio da aplicação de regras que modificam o estado anterior. Quando um desses estados intermediários pode ser unificado com o objetivo inicial o problema está solucionado.

Como é fácil concluir, essa estratégia tende a produzir uma grande quantidade de estados intermediários, uma vez que algumas das regras aplicadas não possuem qualquer relação com o problema focalizado. Essa estratégia, entretanto, apesar de sua potencial ineficiência, é muito apropriada em situações em que o objetivo não pode ser formulado precisamente (como no xadrez, por exemplo). O raciocinio circunstancial é recomendado ainda para explicitar as consequências de um estado particular, isto é, quando se deseja saber o que pode ser derivado a partir de uma determinada assertiva. Este é, por exemplo, o caso do diagnóstico médico, onde é necessário analisar diversas possibilidades de doenças com base em um conjunto de sintomas.

O raciocínio circunstancial é facilmente implementado em Prolog, conforme **[BRA 86]** e [STE 86], entre outros. No sistema Rhesus essa forma de raciocínio é especialmente interessante na construção de objetos prototípicos, podendo servir de suporte ao raciocínio estrutural. Além disso, sob o controle de um mecanismo apropriado, pode ser combinado com o raciocínio dedutivo, interagindo com este de forma concorrente.

9.3.2 O Sistema Cognitivo

O Sistema Cognitivo (SCog), como já foi dito, é o único componente do Sistema Rhesus capaz de acessar diretamente a BC. Além disso, é também capaz de processar diretamente entradas do tipo *ask/tell*, oriundas do Sistema de Comunicação, por meio dos relacionamentos de abstração que incorpora. Visto isoladamente, o SCog pode ser tomado como um sistema para o gerenciamento de BCs orientadas à objetos, uma vez que dispõe de todo o instrumental necessário ao tratamento de tais entidades. Sua concepção da BC é, portanto, a de uma coleção de objetos, modelados como hiperredes, e organizados segundo três hierarquias distintas: a hierarquia *é_um*, a hierarquia associativa e a

hierarquia de agregação. A grande vantagem deste enfoque reside na possibilidade de acionamento automático de três modalidades de raciocínio: por *classificação/generalização*, por *associação* e por *agregação*, o que é suportado pelos relacionamentos de abstração implementados nos objetos do modelo. Na presente seção iremos discorrer sobre essas três formas de raciocínio e suas principais propriedades.

9.3.2.1 Raciocínio por Classificação/Generalização

A mais importante e natural das três estratégias de raciocínio utilizadas pelo SCog é a que se baseia na hierarquia é_um, isto é, a herança de atributos. Seu emprego permite que múltiplos objetos compartilhem o mesmo conhecimento descritivo, determinando a estrutura de instâncias e oferecendo meios para a manutenção automática da integridade semântica da BC. Essa forma de raciocínio baseia-se na estrutura de classes. Assim, quando um objeto é inserido como instância de uma determinada classe, o sistema é capaz de estabelecer automaticamente a sua estrutura. Por exemplo, se declaramos que um determinado objeto é uma instância da classe dos *automóveis*, o SCog pode recuperar a "crença" que esse objeto possui uma *cor*, um *proprietário* e um *preço*. Da mesma forma, se a classe dos *meios_de_transporte* possui *veículos* como subclasse e esta é uma superclasse de *automóveis*, o SCog deve ser capaz de concluir que *automóveis* é uma subclasse de *meios_de_transporte*, ainda que essa informação não esteja explicitamente representada. Em consequência o sistema é também capaz de considerar que todas as instâncias de *automóveis* possuem as propriedades especificadas em *meios_de_transporte* e satisfazem as restrições de integridade associadas a essa classe.

É importante notar aqui que essa concepção de classificação/generalização não corresponde à suportada pelos modelagem convencional de sistemas de BDs, onde a *crença* de que

um objeto é uma instância de uma classe (ou seja, uma tupla de uma relação) não é representada explicitamente no modelo e, por isso, não pode ser oferecida como resposta a uma consulta. Na verdade, a inserção de um objeto fica condicionada à especificação de seu *tipo*, isto é, à especificação de um único relacionamento *instância_de*. Em outras palavras, um objeto somente pode existir se possui um tipo (classe) implicitamente associado a ele. Além disso, não é possível mudar o tipo de um objeto. Por exemplo, não é possível expressar no modelo relacional a crença de que o objeto *carro_2* não é uma instância de *automóveis* e sim de *caminhões*. A única maneira de fazer isso seria por meio da remoção de *carro_2* da relação *automóveis* seguida da inserção de um novo objeto na relação *caminhões*, o qual, todavia, não será interpretado como o mesmo objeto *carro_2*. Concluindo: o relacionamento *instância_de é estático* em sistemas de BDs relacionais.

Em nosso modelo, entretanto, os objetos podem existir por si próprios. Assim é importante permitir ao usuário a inserção de um objeto particular na BC, mesmo que ele não saiba dizer ao certo a que classe o objeto pertence. Por exemplo, o usuário pode não ter certeza no momento da inserção se o objeto *carro_2* é um automóvel, caminhão ou pertence ainda a outra classe qualquer. Em consequência, os objetos podem existir no modelo independentemente de relacionamentos *instância_de*. Pode mesmo ocorrer com frequência a inserção de objetos no modelo sem qualquer especificação do tipo *instância_de*. O usuário pode então refletir sobre o tipo de objeto inserido, ou então pode desejar descobrir qual seria a "aparência" do objeto como instância de uma determinada classe (por exemplo, ele "acredita", a princípio que *carro_2* seja um *automóvel*) e define um relacionamento *instância_de* para verificar o seu palpite. O SCog então deduzirá a estrutura de *carro_2*, partindo do princípio que o mesmo é efetivamente uma instância de *automóveis*, gerando assim uma descrição mais detalhada desse objeto. Com base nessa nova descrição o usuário pode então perceber que o objeto construído não corresponde a uma entidade do mundo real, retornando à atividade de determinar o seu tipo.

Portanto os relacionamentos *instância_de* correspondem na realidade às crenças do usuário ou do construtor da BC, e estas estão sujeitas a modificações sem que isso afete a existência dos objetos. Por essa razão esses relacionamentos devem ser explicitamente representados no modelo de forma que possam ser alvo de interrogação a qualquer tempo. Assim, especificações do tipo *instância_de* representam aqui, ao contrário do que é usualmente considerado em sistemas de BDs, a *causa* e não consequência do processo como um todo. Devido a isso, descrições de classes não podem ser usadas para evitar erros na inserção de objetos como acontece em BDs convencionais, mas sim para a dedução da estrutura desses objetos em conformidade com as crenças particulares do usuário.

Essa mesma estratégia de raciocínio pode empregar a estrutura de um objeto para determinar sua posição em uma hierarquia *é_um*. Baseado nas características dos relacionamentos *subclasse_de* e *instância_de* aprentados por um determinado objeto, o SCog pode deduzir quais os objetos da BC que representam suas superclasse, subclasses e instâncias. Além disso, através do uso de restrições é possível determinar se um determinado item deve ser um valor ou uma propriedade, de forma que as leis que regem os relacionamentos de classificação e/ou generalização não são violadas.

9.3.2.2 Raciocínio por Associação

Assim como acontece no raciocínio por classificação/ generalização, o potencial do conceito de associação recai sobre as "crenças" representadas no modelo quando dois objetos são considerados pelo correspondente conceito de abstração. Por exemplo, se declararmos que *caminhão_3* é uma instância de caminhões, o SCog pode concluir que ele é também um veículo e tem uma *cor*, um *proprietário*, um *preço* e uma *capacidade_de_carga*. O raciocínio por associação funciona da mesma maneira. Por exemplo, se há um conjunto de *veículos_velozes* em nosso modelo

com a estipulação de pertinência de que todo elemento desse conjunto é um veículo com a propriedade de ser *veloz*, dizer que o caminhão *caminhão_3* pertence a esse conjunto corresponde, para o sistema à declaração de que *caminhão_3* é um veículo veloz.

A diferença entre classificação/generalização e associação é que as propriedades de *classe*, isto é, as especificadas pelas classes na definição da estrutura das instâncias, constroem a estrutura dos objetos, enquanto que as estipulações de pertinência não. Na verdade, se restringirmos estas últimas a serem expressas unicamente pela *estrutura* de seus elementos ocasionaríamos a perda completa do potencial de raciocínio do conceito de associação.

Vamos inicialmente examinar o significado associado às estipulações de pertinência. As consequências da declaração de que *caminhão_3* é um elemento de *veículos_velozes*, isto é, que se movimenta velozmente, podem ser também obtidas pela definição de uma propriedade diretamente em *caminhão_3* descrevendo essa sua característica. Se o conjunto *veículos_velozes* possui muitos elementos, a mesma situação poderia ser obtida pela definição de uma nova classe onde a propriedade de ser veloz seria especificada, sendo então herdada por todas as suas instâncias, neste caso os elementos do conjunto *veículos_velozes*. Assim as estipulações de pertinência correspondem a propriedades comuns com valores comuns compartilhadas por objetos heterogêneos, isto é, que não possuem superclasses em comum. O conceito de associação possui então, em um certo sentido, um tipo de herança inerente aos relacionamentos que define.

Vamos assumir agora que é necessário modelar as crenças de um segundo usuário do sistema, que deseja definir o conjunto *veículos_lentos*. Suponhamos que esse segundo usuário seja da opinião que *caminhão_3* é um veículo lento, e o inclua nesse conjunto. Apesar de parecer à primeira vista que isso torna nosso modelo inconsistente, já que *caminhão_3* é ao mesmo tempo um elemento de *veículos_velozes* e *veículos_lentos*, não há absolutamente qualquer inconsistência, uma vez que cada um desses

dois conjuntos representa as crenças de duas pessoas diferentes. O modelo estaria entretanto num estado inconsistente se essas duas propriedades fossem descritas pelo conceito de generalização/classificação (e consequentemente herdadas por *caminhão_3*), porque estariam determinando duas descrições estruturais diferentes para o mesmo objeto. Esse problema é denominado *conflito por herança múltipla*.

Como o conceito de associação não aplica diretamente a herança, ele não define descrições estruturais diferentes para *caminhão_3*, mas sim diferentes *visões* desse objeto. Isto é, *caminhão_3* é um veículo veloz, do ponto de vista do conjunto *veículos_velozes*, e um veículo lento na visão determinada por *veículos lentos*.

Então as estipulações de pertinência são na verdade propriedades que todo objeto deve satisfazer para ser elemento de um determinado conjunto. A associação portanto, ao contrário da classificação/generalização, permite ao sistema estabelecer crenças independentes da estrutura dos objetos. Em consequência diversos *mundos* diferentes possivelmente contraditórios podem ser facilmente modelados. Por essa razão esse conceito de abstração tem encontrado ampla aplicação na modelagem de planos e projeto de atividades, uma vez que é extremamente apropriado à representação de mundos hipotéticos e espaços de crenças.

Naturalmente, especificando as estipulações de pertinência com base nas estruturas dos objetos elementos, o sistema pode determinar se uma certa modificação poderia ser efetuada sobre um determinado objeto sem violar os relacionamentos *elemento_de* e/ou *subconjunto_de* a *que* o objeto pertence, sendo também capaz de decidir se essa modificação deve ser aceita ou rejeitada. No primeiro caso o relacionamento violado seria desfeito e novos relacionamentos, agora satisfazendo o novo objeto, seriam criados.

Além dessa linha de raciocínio, o conceito de associação oferece outra, agora sem empregar o mecanismo de herança. Enquanto que o raciocínio por classificação/generalização depende da estrutura das classes, vista de forma *top-down*, o raciocínio por associação é independente da estrutura dos conjuntos, sendo, ao invés disso, determinado pela estrutura dos elementos que os compõem, isto é, construido de forma *bottom-up*. Assim, enquanto que modificações sobre a estrutura das classes (por exemplo, a remoção de uma propriedade) implicam em modificações *top-down* sobre os objetos do modelo, as modificações sobre as propriedades de um conjunto não ocasionam qualquer modificação em seus elementos. Por outro lado, uma modificação nos elementos de um conjunto pode conduzir a alterações *bottom-up* nas suas propriedades. Por exemplo, mudando o valor da propriedade *custos_de_manutenção* de um veículo da *companhia_de_transportes*, o SCog deve correspondentemente obter o novo valor da propriedade *média_dos_custos_de_manutenção* do conjunto *veículos_da_companhia_de_transportes*. Essa mesma forma de raciocínio é necessária quando novos objetos se tornam elementos de um conjunto, para manter uma visão consistente do mundo representado.

9.3.2.3 Raciocínio por Agregação

Como acontece na associação, o raciocínio por agregação independe da estrutura dos objetos, suportando a noção de objetos complexos, o que permite ao usuário, em um momento qualquer, tomar um objeto como uma única entidade e em seguida ver somente as suas partes como objetos independentes. Por meio da agregação podemos então interpretar os objetos como sendo entidades isoladas em um determinado contexto e componentes de outros objetos em outro.

Entretanto, ao contrário da associação, o conceito de agregação baseia-se na estrutura dos objetos e, além disso, em propriedades comuns a componentes, subcomponentes e elementos. O

raciocínio por agregação é suportado no SCog por meio de predicados definidos sobre a hierarquia de agregação e se baseia nessas propriedades comuns. Uma propriedade cujo valor decresce de forma *top-down* na hierarquia de agregação é dita ser *monotonicamente decrescente*. Por exemplo o *peso* de um subcomponente é sempre menor ou igual ao peso de seus supercomponentes. Paralelamente as propriedades cujos valores crescem conforme se vai percorrendo a hierarquia de agregação de maneira descendente são ditas *monotonicamente crescentes*.

Com base nestas propriedades, os predicados são definidos de forma que o raciocínio possa fluir. Um predicado é implicado de forma *ascendente* se a sua satisfação por um objeto implica em que todos os supercomponentes desse objeto o satisfaçam. Por exemplo, o predicado *peso* $\geq$ *500 kg* é implicado de forma ascendente, uma vez que todos os seus supercomponentes possuirão um peso maior ou igual a 500 kg. Da mesma forma um predicado é implicado de forma *descendente* quando a sua satisfação por um componente inplica em sua satisfação por todos os subcomponentes.

Os relacionamentos do conceito de agregação estabelecem as conclusões que o sistema pode obter com base nos predicados implicados quando dois objetos se relacionam por meio de *subcomponente_de* ou *parte_de*. Por exemplo, conclusões sobre o peso de um componente podem ser obtidas sobre o correspondente predicado ascendente. Da mesma forma o sistema pode usar esses predicados implicados como restrições na determinação de se um certo valor pode ser ser atribuído a uma propriedade do objeto de modo que a monotonicidade dessa propriedade não seja violada.

Uma outra facilidade de raciocínio oferecida pela agregação está relacionada ao axioma que diz que os objetos não podem existir sem os seus subcomponentes. Devido a isso, a remoção de um objeto pode significar para o sistema que todos os seus componentes devam ser removidos. Por exemplo, quando um particular automóvel é removido da BC, o sistema conclui que o seu motor, suas rodas e sua carroceria devem ser correspondentemente removidos.

9.4 O SISTEMA DE COMUNICAÇÃO

Na arquitetura proposta para o Sistema Rhesus (ver **Figura 9.1**) o Sistema de Comunicação (SCom) tem a função de suportar as consultas do usuário, estabelecendo o necessário interface entre este e os outros módulos do sistema. Sua construção deve portanto atender aos requisitos estabelecidos do ponto de vista do nível de aplicação (ver seção 8.2), garantindo a independência do conhecimento e tomando a si a responsabilidade de selecionar as estruturas simbólicas adequadas a interpretação pretendida. O SCom deve portanto oferecer ao usuário uma visão funcional do sistema, concebida em termos do conhecimento que pode ser dele extraído sobre determinado domíninio sem considerar os esquemas de representação empregados pelos demais componentes.

Dois tipos básicos de operações estão associados ao SCom: *ask*, cujo objetivo é interrogar o sistema com base no conhecimento armazenado na BC e *tell*, que permite lidar com a introdução de novos conhecimentos no sistema. Evidentemente, poderosos mecanismos de representação deverão ser oferecidos para apoiar a construção e evolução da BC, tais como um interface bidirecional entre o SCom e os demais módulos do sistema além de ferramentas como um editor de (hiper)textos, um editor gráfico orientado à representação de redes de objetos, um interpretador para a linguagem lógica subjacente, etc. Na verdade esse conjunto de facilidades configura um *ambiente* ou *shell* cuja estrutura é apresentada na **Figura 9.4**.

O SCom oferece então ao usuário um ambiente dotado de todo o instrumental periférico capaz de auxiliá-lo em sua interação com o sistema Rhesus. A atividade de processamento de textos, por exemplo, pode se valer da própria modelagem em hiperredes, ativando assim um sistema de hipertextos para a produção de código reutilizável ou para a associação de uma descrição textual a um objeto ou hierarquia do modelo.

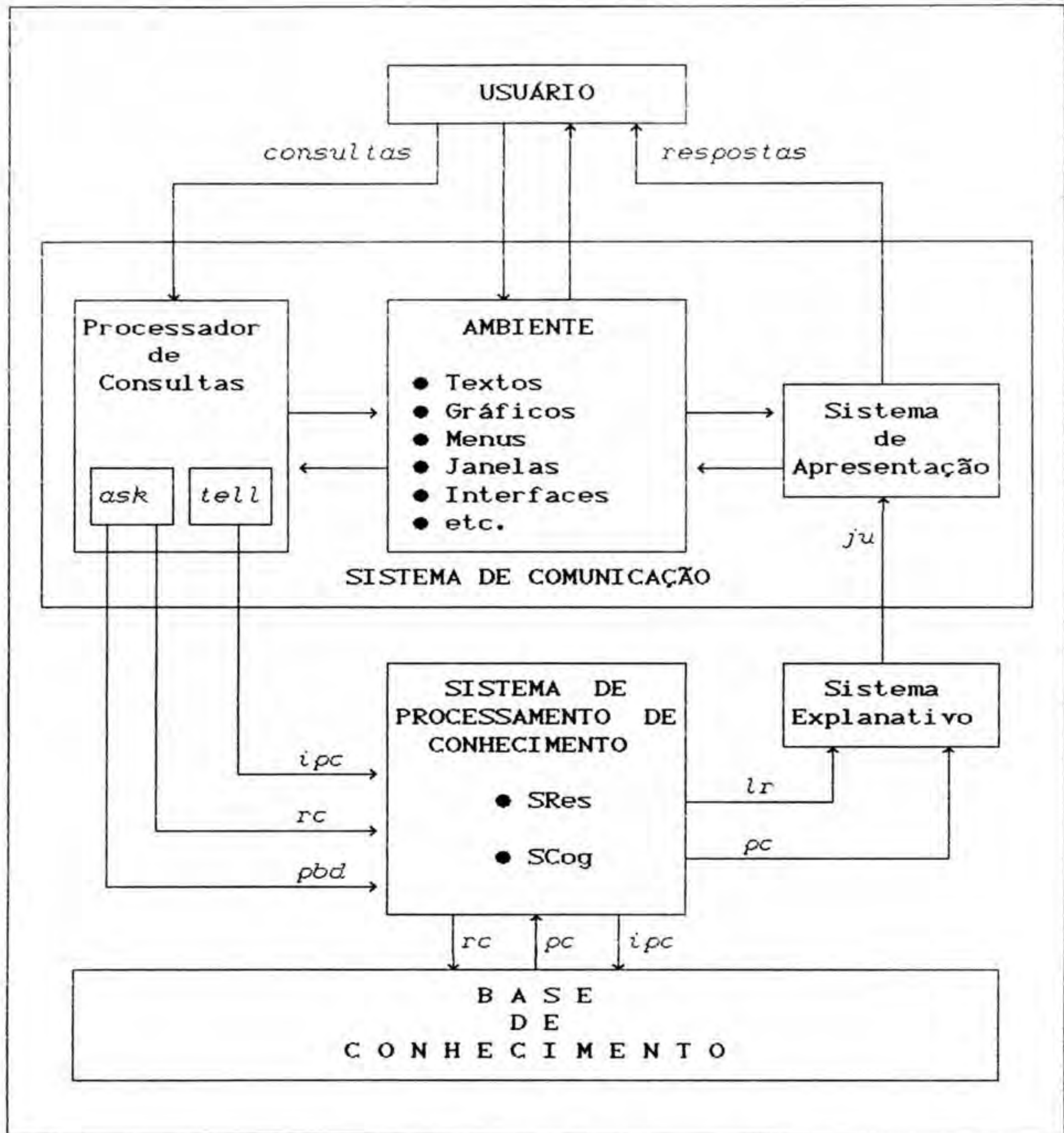


Figura 9.4

Estrutura Funcional do Sistema de Comunicação

A linguagem de consulta é o meio pelo qual o usuário se comunica com o ambiente oferecido pelo SCom. Claramente tal linguagem deve possuir características que permitam não somente interagir com o processo de raciocínio em execução, como também com as demais facilidades suportadas pelo ambiente. Assim, numa

aplicação típica, o usuário poderia utilizar o processador de textos do ambiente para construir uma aplicação ou um objeto qualquer, representar graficamente sob diversos pontos de vista o objeto especificado, submeter esse objeto à uma determinada visão da BC, inserir, remover ou modificar objetos nessa visão ou no próprio esquema representado pela BC, especificar estratégias de raciocínio, etc. Todas essas solicitações de atividades devem ser inicialmente analisadas pelo Processador de Consultas (PCon), que as converte em conjuntos de operações *ask/tell*, para serem então submetidas ao Sistema de Processamento de Conhecimento (SP) sob a forma de um *problema bem-definido* (*pbd*), de uma *requisição de conhecimento* (*rc*) ou solicitando a introdução de uma nova peça de conhecimento na BC (*ipc*).

O SP interage então com a BC na tentativa de atender à consulta realizada. As linhas de raciocínio (*lr*) empregadas e as peças de conhecimento (*pc*) produzidas são coletadas ordenadamente pelo Sistema Explanativo (SExp), produzindo uma *justificativa* (*ju*). Essa justificativa é então utilizada juntamente com as facilidades oferecidas no ambiente pelo Sistema de Apresentação (SApr), para fornecer a resposta do sistema à consulta do usuário.

Como vimos no capítulo 8, a Linguagem de Consulta (LC) suportada pelo PCon deve apresentar completeza e extensibilidade de representação, permitir a projeção de objetos e a sua modificação, ser recursiva e orientada a conjuntos. Além disso, para satisfazer mais um requisitos da engenharia de software, deve oferecer transparência e legibilidade ao usuário, isto é, apresentar a propriedade que denominamos *conveniência notacional*. Isto significa que a LC deve dispor de mecanismos que descrevam a complexidade das construções do modelo de forma simples, objetivas, fáceis de ler e entender, permitindo o acesso imediato e sob diversos enfoques a qualquer objeto ou hierarquia do modelo. Por outro lado o SCon deve possibilitar uma visão com a mesma ordem de clareza das respostas fornecidas, o que é obtido através do Sistema de Apresentação, com base nas justificativas oferecidas pelo Sistema Explanativo traduzidas através das facilidades proporcionadas pelo ambiente.

9.5 O SISTEMA EXPLANATIVO

O Sistema Explanativo (SExp) possui uma arquitetura interna muito simples, nem por isso deixando de ser importante no contexto do sistema Rhesus. Sua principal finalidade é coletar ordenadamente os passos executados pelo SP no processamento de uma determinada consulta, originando uma árvore de prova (que denominamos *justificativa*) onde são representadas as linhas de raciocínio utilizadas, objetos produzidos ou considerados, contextos e derivações realizadas pelo sistema. Uma das possibilidades que apresenta é a habilidade de demonstrar as causas de falhas ocorridas durante o processo de solução, mencionando os (sub-)objetivos insolúveis acompanhados de sugestões correspondentes a uma possível solução. Isso torna as ações executadas pelo SP mais confiáveis, ao mesmo tempo em que coloca o usuário em uma posição mais *cooperativa* em relação ao processo de solução. O SExp pode ainda ser adequadamente empregado em atividades como:

- **Educação**, uma vez que é facultado ao usuário formular suas próprias questões e compará-las com as oferecidas pelo sistema, e

- **Ajustamento da BC**, uma vez que conhecimento omitido ou contraditório se reflete no *trace* construído pelo SExp durante o processo de solução.

Para atingir essas expectativas, comandos do tipo **COMO**, **PORQUÊ** e **QUANDO** devem ser implementados na LC, permitindo ao sistema apresentar explicações atendendo às seguintes especificações:

- O usuário deve ser capaz de definir, independentemente da estratégia de raciocínio, o grau de *profundidade* em que deseja receber a árvore de prova,

- O usuário deve ser capaz de definir o grau de *concisão* no qual deseja que a árvore de prova seja apresentada, omitindo os detalhes que não lhe são significativos, e

- O SExp deve ser capaz de justificar todos os passos do SP, mesmo nos casos em que haja efeitos colaterais associados. Para o tratamento de tais casos o sistema deve manter um registro de todas as modificações ocorridas sobre a BC.

9.6 DIRETRIZES DE IMPLEMENTAÇÃO

Na presente seção iremos discutir algumas questões relacionadas com a implementação do sistema Rhesus, tentando definir uma forma eficiente de tratar o armazenamento e recuperação do conhecimento de modo a apoiar efetivamente os níveis de engenharia e aplicação. Com essa finalidade, abordaremos cada um dos módulos componentes do sistema, considerando aspectos específicos de sua implementação e dos mecanismos que devem ser oferecidos para atingir os propósitos formulados anteriormente. É de se notar que, a partir de um determinado grau de detalhamento, os requisitos de implementação confundem-se com os que devem ser preenchidos por BDs convencionais: estruturas de armazenamento, técnicas de pesquisa, controle de acesso, registro e recuperação, etc. Assim, iremos nos concentrar principalmente nos aspectos relacionados com o *nível de conhecimento* do sistema, no sentido proposto em [BRA 86].

9.6.1 A Modelagem do Conhecimento

O modelo das hiperredes oferece um mecanismo uniforme para a representação de objetos e meta-objetos na BC. Assim, toda entidade representada na BC, assume a forma de um *objeto Rhesus*, que se caracteriza por apresentar a estrutura:

<identificador>(Tipo,Sort,Int,Ext,Prop)

cujas características sintáticas já foram discutidas anteriormente. Iremos considerar agora como o conhecimento pode ser representado em tais objetos, considerando seus aspectos descritivos, operacionais e organizacionais.

O conhecimento declarativo é visto aqui em termos de objetos conceituais correspondendo aos modelos de dados estudados na pesquisa de BDs. A estrutura do modelo das hiperredes naturalmente associada a esse tipo de conhecimento é o *hipernodo*, cujas leis de construção se adaptam perfeitamente aos conceitos de classificação/generalização, associação e agregação e, por conseguinte, à extração das correspondentes hierarquias. Um hipernodo fica perfeitamente definido se:

(1) Possui um *identificador* ou *nome* único,

(2) É declarado do *tipo* hipernodo, possuindo um *sort* que o classifica como uma particular sub-estrutura do tipo hipernodo,

(3) Possui uma *estrutura interna*, descrita em termos de hipernodos componentes,

(4) Possui uma *estrutura externa*, constituída por restrições à sua participação em operações com outros objetos do modelo (visando distinguir o mundo *real* de outros mundos *possíveis*), e

(5) Possui uma *lista de propriedades*, estabelecendo suas características particulares assim como os relacionamentos e hierarquias de que participa.

Ressaltamos, mais uma vez, que *tipos* ou *sorts* de objetos (isto é, classes, conjuntos e supercomponentes) são também objetos do modelo, de modo que a representação dos objetos e de suas instâncias, elementos ou componentes, pode ser realizada totalmente em termos de hipernodos. Além disso, um hipernodo pode representar simultaneamente uma entidade coletiva para um determinado contexto e uma entidade elementar para outro. Essa grande expressividade do modelo das hiperredes, obtida unicamente através dos hipernodos que permite construir, torna-o equivalente a diversos outros modelos "*completos*" da literatura sobre BDs semânticas, como por exemplo os apresentados em **[KIN 86]** ou **[WAT 86]**.

Os aspectos operacionais do conhecimento são representados primitivamente em hiperredes por meio de hiperrelações e protótipos. Estas duas estruturas permitem a *ativação* do conhecimento passivo representado em hipernodos, onde a primeira atua como a especificação de uma transformação e a segunda como metavariável, capaz de ser instanciada parcial ou totalmente por outros objetos do modelo. Essa interpretação permite multiplicar o potencial expressivo das hiperredes, atingindo níveis dificilmente encontrados com a mesma concisão conceitual em outros modelos. Uma hiperrelação possui os mesmos componentes estruturais que os hipernodos, entretanto sua estrutura interna permite combinar todas as demais entidades do modelo: hipernodos, hiperrelações de ordem mais baixa, protótipos, bcs, visões e aspectos.

Isso é possível em função da noção de *protótipo*, os quais podem ser vistos como especificações de metavariáveis representando os possíveis elementos da hiperrelação. Assim, através de hiperrelações podemos relacionar quaisquer objetos do modelo. Além disso, é necessário, em muitos casos, se dispor de hiperrelações atômicas primitivas cuja estrutura interna corresponda a código diretamente executável. Isso não entra em choque com a semântica do modelo e a possibilidade de lidar com esse tipo de conhecimento reforça ainda mais seu potencial de expressividade.

Por outro lado, o conhecimento organizacional do modelo não é representado por uma particular entidade, estando disperso entre os objetos da BC em suas estruturas externas e propriedades. Essa visão conduz à necessidade de se dispor de mecanismos devotados à construção e modificação dinâmica das três hierarquias associadas aos conceitos de abstração, assim como de uma rede taxonômica capaz de representar o estado instantâneo da BC no decorrer da operação do sistema. Isso é possível por meio de um mecanismo de nomeação para todos os objetos na BC, incluindo os internamente produzidos, mesmo que temporários. A manutenção da integridade dos relacionamentos produzidos pode ser então ser conduzida inicialmente em função de tais hierarquias, sendo complementada pelas restrições particulares de cada objeto, representadas em suas estruturas externas.

9.6.2 O Sistema de Processamento de Conhecimento

Como foi visto anteriormente, o SP se divide estruturalmente em dois componentes principais. O SCog, intimamente relacionado com a construção da BC, é o único que tem acesso direto à ela, responsabilizando-se também pela sua integridade estrutural de acordo com os conceitos de abstração que modela. O segundo componente, que denominamos SRes, executa formas de raciocínio mais elaboradas sobre os contextos produzidos pelo SCog. Examinaremos agora esses dois componentes enfatizando suas características principais sob o ponto de vista da implementação.

9.6.2.1 O Sistema Cognitivo

O SCog é, por excelência, o componente construtor da BC e seu único meio de acesso, devendo portanto incorporar, além dos

mecanismos de modelagem e estruturação do conhecimento, o instrumental necessário para permitir o compartilhamento da BC em ambientes multiusuários, incluindo técnicas de pesquisa, controle do acesso concorrente, mecanismos de registro e recuperação, etc. Assim, sob um determinado ponto de vista, o SCog sozinho corresponde ao núcleo de um completo SGBC orientado ao tratamento de objetos e suas abstrações, devendo ser o primeiro componente a ser implementado dentre os que constituem o sistema Rhesus. Na **Figura 9.5** apresentamos a estrutura do sistema Rhesus enfatizando esse componente.

As funções de acesso à BC devem ser orientadas à recuperação, uma vez que esse mecanismo é normalmente o mais solicitado na operação de SGBCs. Sua presença sob o controle do SCog deve-se ao fato de que este é, como já dissemos, o único meio de acesso à BC, devendo portanto responsabilizar-se pela sua integridade. Para o tratamento de BCs de grande porte, o SCog deve ainda apresentar facilidades de segmentação orientadas a visões, o que irá permitir maior eficiência na alocação e compartilhamento de memória. O controle dos objetos na BC fica então sujeito a hierarquias e dicionários dinâmicos, construidos paralelamente à sua evolução.

O Componente de nível mais alto do SCog é uma máquina de inferências orientada ao processamento das abstrações definidas sobre a BC. Nesse nível de raciocínio, a estrutura interna de objetos atômicos é irrelevante, uma vez que estes são vistos pelo SCog apenas em função de sua estrutura externa e das características ditadas por suas propriedades. O mesmo não acontece entretanto com objetos compostos, uma vez que sua estrutura interna descreve sua construção em função dos objetos participantes.

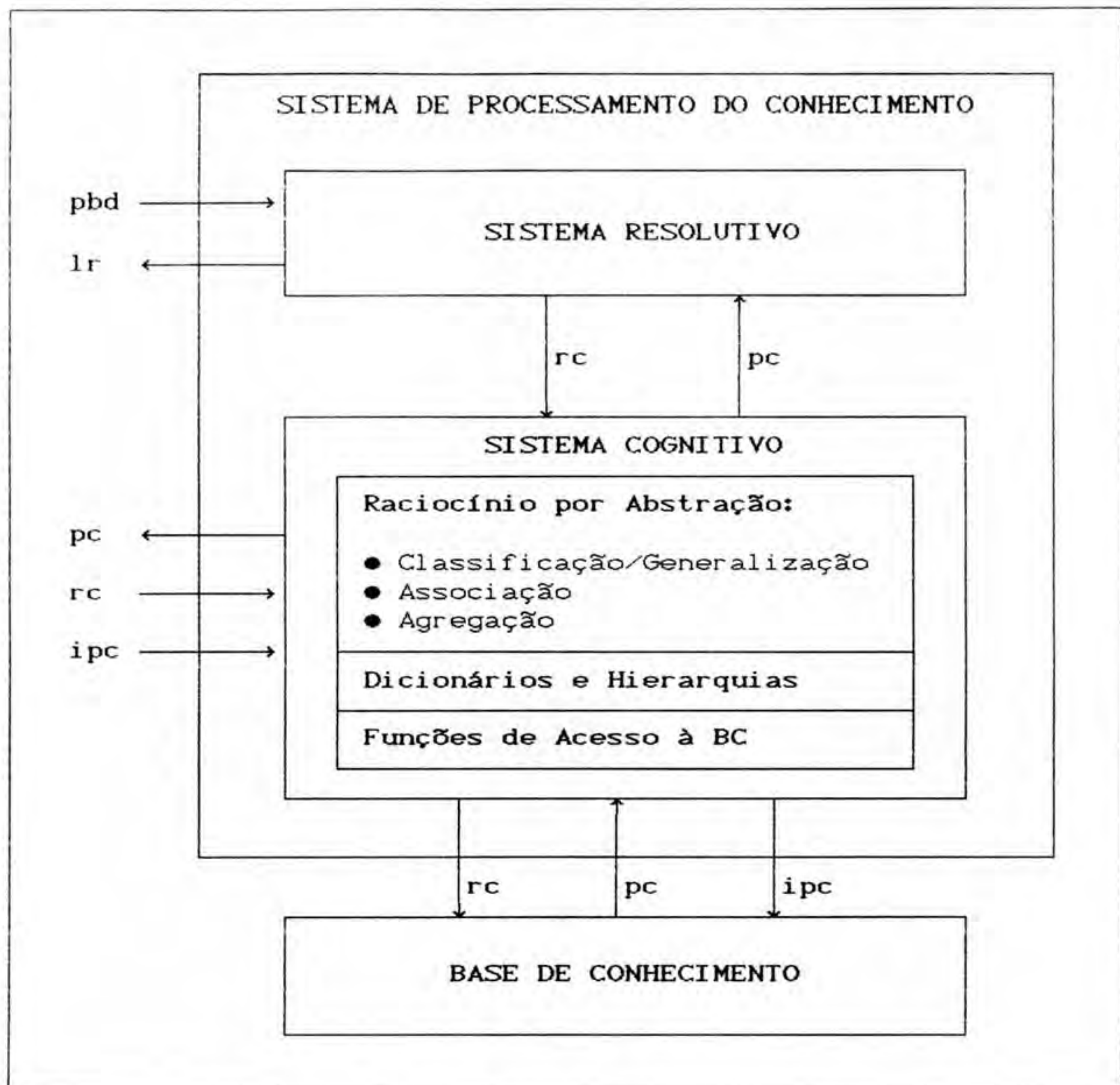


Figura 9.5

O Sistema Cognitivo enfatizado no contexto do SP

9.6.2.2 O Sistema Resolutivo

O SRes destina-se a oferecer as formas de raciocínio que não são suportadas pelo SCog, normalmente executadas sobre um aspecto (um objeto complexo) fornecido por este último. Na **Figura 9.6** apresentamos em destaque a estrutura do SRes no contexto do SP.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

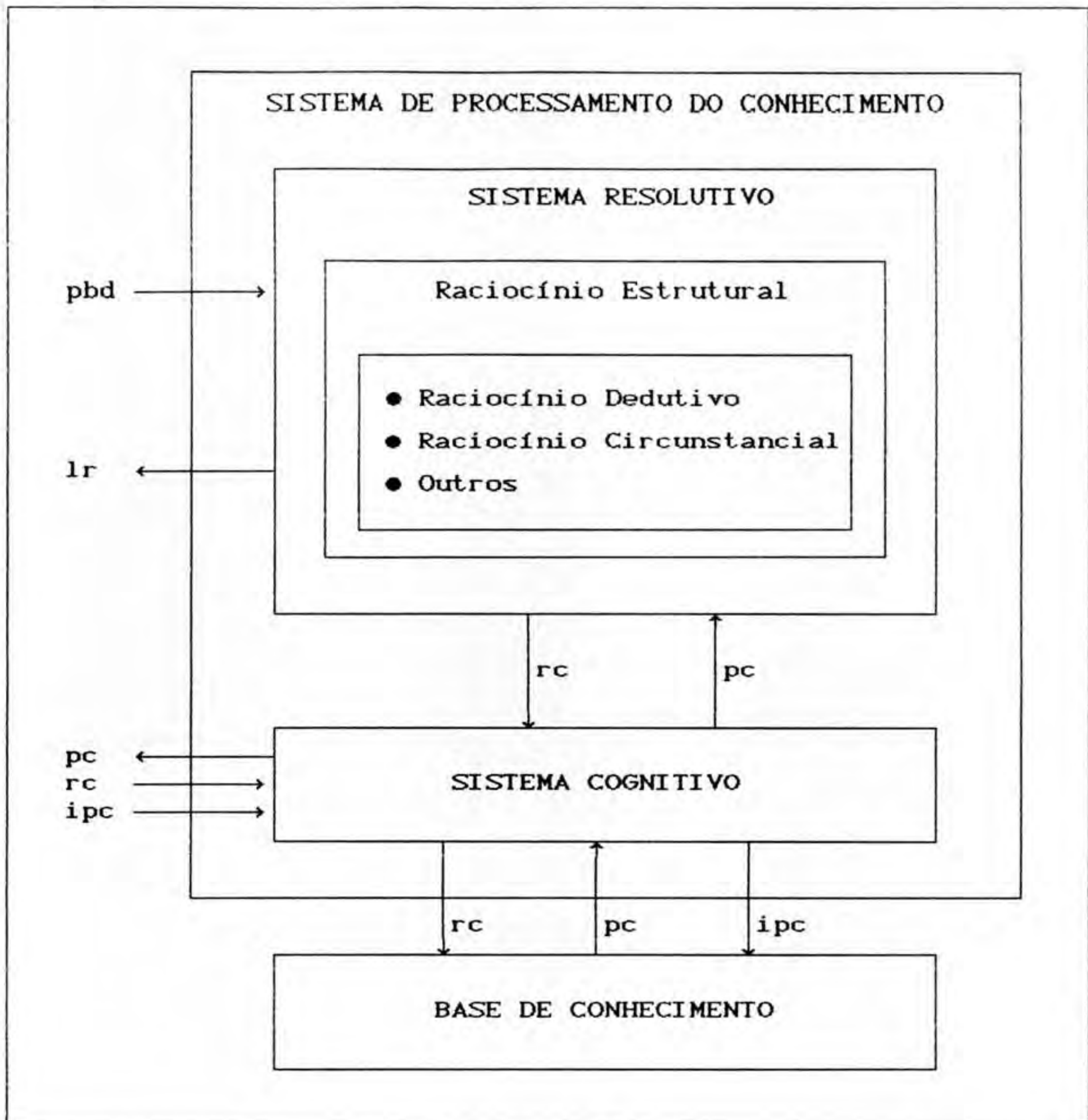


Figura 9.6

O Sistema Resolutivo enfatizado no contexto do SP

Ao contrário do que acontece com o SCog, aqui a estrutura interna das entidades atômicas é importante, uma vez que a partir delas é que o SRes irá determinar o sistema de contextos sobre o qual alguma forma de raciocínio deverá ser executada. Essa atividade é desempenhada pelo módulo de raciocínio estrutural (ver seção 7.2). O raciocínio estrutural exerce então sua atividade sobre um *aspecto* da BC transmitido pelo SCog, visando transformá-lo em um sistema de contextos.

Denominamos essa forma de raciocínio *estrutural* uma vez que a mesma é dirigida a objetos estruturados (os aspectos) e se destina a obter deles outro tipo de estruturas (os sistemas de contextos), que irão configurar o espaço de prova para as inferências desejadas. Sua ação deve portanto ser também voltada à redução desse espaço de prova e à determinação da forma de raciocínio a empregar sobre ele.

Visto dessa forma, ao raciocínio estrutural deve ser permitido o emprego tanto de estratégias dedutivas como circunstanciais, em função do problema a ser solucionado. Sua implementação deve ser suficientemente flexível para permitir sua extensão e/ou modificação sem que isso afete os demais componentes do sistema, dentro do princípio da independência do conhecimento.

As atividades do raciocínio estrutural compreendem também toda transferência de conhecimento entre o SRes e os demais componentes do sistema. Do SCom são recebidos *problemas bem-formados* que devem ser traduzidos em objetos a serem requisitados ao SCog e linhas de raciocínio a aplicar sobre estes objetos. Durante o decorrer do processo de solução, as linhas de raciocínio empregadas serão transmitidas ao SExp para a construção da árvore de prova correspondente. Assim, como pode ser visto na **Figura 9.6**, o raciocínio estrutural encapsula as demais estratégias de raciocínio do SRes, controlando o seu emprego e servindo como elo de ligação entre elas e os demais componentes do sistema.

9.6.3 O Sistema de Comunicação

Na **Figura 9.4** apresentamos a estrutura proposta para o SCom no contexto do sistema Rhesus, onde se pode ver que o mesmo é composto por três principais componentes: o *ambiente*, o

Processador de Consultas (PCon) e o *Sistema de Apresentação* (SApr). O ambiente está intimamente relacionado com o sistema operacional sob o qual o sistema Rhesus atua e deve abranger, além das rotinas de acesso ao hardware, um conjunto de ferramentas auxiliares ao processo de desenvolvimento de aplicações. O PCon recebe a consulta do usuário, formulada através de uma Linguagem de Consulta (LC), cuja implementação deve atender aos requisitos de completeza, extensibilidade, transparência, legibilidade, ser recursiva e orientada a conjuntos e a descrição de visões, além de permitir a projeção de objetos e sua modificação. Por outro lado o SApr deve receber os resultados do processamento das consultas, interpretando-os sob a forma de textos, diagramas e gráficos que serão fornecidos como resposta à consulta do usuário. Na presente seção iremos estudar uma alternativa de implementação para o SCon a partir dos requisitos de seus componentes.

9.6.3.1 O Ambiente

Conforme foi dito, o ambiente apresentado ao usuário por meio do SCom estará fortemente vinculado ao sistema operacional sob o qual ocorrer a implementação. Isso é tomado inicialmente como medida de eficiência uma vez que as rotinas de acesso ao hardware suportadas por este último dificilmente terão seu desempenho superado por mecanismos programados em nível mais alto. Por outro lado o sistema Rhesus deve ser suficientemente flexível para suportar o acesso a outros programas e arquivos do sistema valendo-se dos mesmos no processo de solução de problemas do usuário. Isso corresponde ao desenvolvimento de um *shell* que permita interpretar comandos do usuário como diretivas ao sistema operacional subjacente, garantindo ao usuário a transparência do acesso realizado sem perda de eficiência. É necessário portanto considerar os detalhes do sistema operacional pretendido para

essa implementação. Do nosso ponto de vista, três alternativas se impõem imediatamente em função de sua popularidade atual e fácil acesso nos meios acadêmicos:

- **MS DOS 5.0** ou **OS/2 2.0**, para microcomputadores compatíveis com a linha IBM PC ou PS,

- **Unix-like**, para supermicros, e

- **VM/SP** para computadores de médio a grande porte.

Além disso, um conjunto eficaz de ferramentas de suporte às atividades de modelagem, construção e recuperação de objetos e aplicações deve ser colocado à disposição do usuário. Dentre tais ferramentas, identificamos como indispensáveis as seguintes:

- **Processador de Textos**: Orientado à codificação de conhecimento e aplicações, incorporando o tratamento de hipertextos com base no modelo subjacente,

- **Processador Gráfico**: Orientado a representação e interpretação estrutural dos objetos na BC, e

- **Interpretador Prolog**: Destinado a apoiar o processo de engenharia de conhecimento e algumas estratégias de raciocínio.

A construção do shell deve ainda oferecer um interface amigável ao usuário, permitindo a operação interativa do sistema através de facilidades como menus, janelas, mouse, help on-line, teclas de controle, etc.

9.6.3.2 O Processador de Consultas

A comunicação do usuário com o sistema ocorre através de uma Linguagem de Consulta (LC), capaz de permitir a manipulação interativa ou programada dos objetos na BC. O interpretador da LC recebe a aplicação do usuário, convertendo-a em dois possíveis tipos de operação: *ask*, para interrogar o sistema e *tell*, para a introdução de novos objetos na BC. Essas operações devem se caracterizar por serem independentes do conhecimento, de forma que do ponto de vista do usuário a BC possa ser concebida como um tipo abstrato de dados sob o qual são definidas as operações:

ask : BC x Informação ⟶ Informação

tell : BC x Informação ⟶ BC

A LC oferece então ao usuário uma visão *funcional* da BC, que pode ser descrita por meio da abordagem modelo-teorética, considerando a BC como um modelo da teoria de primeira ordem expressa pela LC. Para o tratamento das declarações da LC é necessário então assumir três hipóteses que expressam uma certa representação implícita para fatos negativos e definem o universo de referência para consultas à BC, a saber:

- **Hipótese do Mundo Fechado**, também denominada *convenção para informação negativa*, que estabelece que se um fato não pode ser reconhecido como verdadeiro então ele deve ser assumido como falso,

- **Hipótese do Nome Único**, que estabelece uma identificação inequívoca para cada um dos objetos representados na BC, e

- **Hipótese do Fecho de Domínio**, que define o universo de referência como sendo formado pelos objetos representados na BC, isto é, não há outras entidades no mundo além das presentes na BC.

A LC deve ainda oferecer ao usuário uma coleção ampla e extensível de comandos e declarações como *create*, *refine*, *insert*, *remove*, *modify*, *kill*, *how*, *why*, *when*, *where*, etc. Além disso deve permitir a formatação das respostas e o redirecionamento dinâmico dos canais de E/S.

9.6.3.3 O Sistema de Apresentação

O SApr atua no SCom a partir das respostas e justificativas obtidas, traduzindo-as para uma forma de apresentação adequada ao entendimento do usuário. Para isso deve empregar as ferramentas oferecidas pelo ambiente, tais como o suporte gráfico, janelas, diagramas, etc. Deve também permitir a projeção das respostas obtidas, omitindo descrições consideradas desnecessárias pelo usuário e suportar a apresentação das respostas em diferentes níveis de detalhamento por meio de um adequado mecanismo de *zooming*. Atua ainda como interface entre o ambiente e o usuário por meio de um sistema de janelas suportando a concorrência de aplicações.

9.6.4 O Sistema Explanativo

Encerrando os componentes do sistema Rhesus o SExp coleta todos os passos do processamento de uma consulta,

construindo uma árvore de prova detalhada abrangendo as linhas de raciocínio empregadas, objetos produzidos ou considerados, contextos e inferências executadas pelo sistema. Sua implementação exige o conceito de *passo de inferência*, que pode ser traduzido como sendo o caminho percorrido pelo sistema entre duas operações distintas, permitindo localizar a ocorrência de falhas e sugerindo ao usuário alternativas de solução. Na base de sua implementação pode ser empregado um mecanismo de *tracing* semelhante ao normalmente oferecido em sistemas Prolog, sobre o qual seriam construídos os demais mecanismos necessários ao seu funcionamento efetivo.

10 CONCLUSÕES

Não obstante a importância da noção de *conhecimento* para a pesquisa em IA, os fundamentos teóricos da RC repousam ainda sobre uma concepção bastante limitada do que significa afirmar que um determinado agente cognitivo *sabe* alguma coisa. A visão correntemente adotada na literatura especiaLizada considera que uma máquina "*sabe*" determinada proposição se o seu estado a representa explicitamente como sentença de uma linguagem formal ou se tal sentença pode ser derivada através de um sistema lógico apropriado a partir de outras sentenças da linguagem explicitamente representadas no referido estado. Um enfoque alternativo foi proposto por Rosenschein, em **[ROS 85]**, buscando analisar o conhecimento em termos das relações entre o estado de uma máquina e o estado de seu ambiente ao longo do tempo, empregando a lógica como metalinguagem.

O **problema da RC**, isto é, como representar efetivamente o conhecimento de senso comum, esbarra na complexidade do mundo real **[HOF 79]**, o qual, segundo sugerem o Teorema da Incompleteza de Gödel, o Teorema da Indecidibilidade de Church, o Teorema da Parada de Turing e o Teorema da Verdade de Tarski, não pode ser representado em um único formalismo. Assim, os ERs correntemente empregados ou conseguem cobrir apenas alguns aspectos de um determinado corpo de conhecimento ou, quando permitem a representação da maior parte de seus aspectos, acabam por tornar-se inviáveis para aplicações em grande escala, como acontece com a lógica de primeira ordem devido ao problema da explosão inferencial ou *problema do controle* **[LLO 84]**.

Apesar de sua inerente indecidibilidade e do problema do controle, extensões da lógica de primeira ordem tem se mostrado capazes de superar a questão da monotonicidade vindo a constituir ERs de expressividade virtualmente universal.

Resta entretanto solucionar ou pelo menos amenizar o problema do controle, que limita o seu emprego a aplicações de pequeno a médio porte. Nesse sentido o trabalho de Bowen [BOW 85], [BOW 86] e de Monteiro e Porto [MON 88] parecem indicar que a chave para superar a explosão inferencial passa obrigatoriamente pela *estruturação* do conhecimento reforçada pela incorporação de metaconhecimento aos objetos assim representados, de modo a permitir o exercício de algum controle sobre as possíveis inferências a serem realizadas.

O modelo das hiperredes [GEO 85], parece atingir plenamente tais objetivos. Em primeiro lugar, pelo seu elevado potencial de estruturação, oferece um *framework* adequado para o representação de hierarquias de objetos, possibilitando a independência de conhecimento e permitindo a coexistência de diversas formas de raciocínio. Depois, pela sua inerente expressividade, facilita o tratamento de construções descritivas, operacionais e organizacionais, de conhecimento incompleto e de exceções, além de permitir o emprego de descrições recursivas. Finalmente, a simplicidade e uniformidade sintática de suas entidades primitivas parece possibilitar uma interpretação semântica bastante clara do modelo, por exemplo, baseada em grafos.

O trabalho aqui apresentado representa uma tentativa de associar a programação em lógica ao formalismo das hiperredes, visando obter um novo modelo capaz de preservar a expressividade da primeira beneficiando-se simultaneamente do potencial heurístico e da capacidade de estruturação do segundo. Acreditamos ter conseguido demonstrar que esta combinação é possível, oferecendo indiscutíveis vantagens sobre os dois modelos tomados isoladamente, permitindo uniformizar o tratamento dispensado às diferentes categorias de conhecimento, que em todas as suas expressões são representadas como objetos do modelo, e oferecendo uma visão funcional das BCs por eles constituídas, garantindo assim a independência entre o conhecimento representado e as aplicações do usuário.

Uma primeira implementação do modelo das hiperredes para a representação do conhecimento foi realizada em Prolog, conforme descrita em [PAL 89]. Ali procurou-se reunir elementos que permitissem estabelecer uma avaliação inicial do seu uso em um substrato baseado na programação em lógica. A experiência obtida com o desenvolvimento do citado protótipo conduziu à proposta de implementação do sistema Rhesus, na forma descrita no capítulo anterior, com o objetivo de melhor explorar as potencialidades da associação entre lógica e hiperredes.

O sistema Rhesus, apresentado no capítulo 9, foi concebido com a finalidade de permitir uma verificação experimental das características do modelo proposto, devendo como tal ser implementado de maneira modular e "aberta", facilitando o isolamento ou diferentes combinações dos mecanismos de raciocínio de modo a permitir a realização de experiências sobre os mesmos. No momento em que escrevemos estas linhas (maio de 1991), dois projetos de implementação encontram-se em desenvolvimento. O primeiro deles teve sua execução iniciada em abril de 1991, junto ao Núcleo de Pesquisas e Desenvolvimento em Informática (NPDI) da Universidade Católica de Pelotas, e contempla a implementação de um protótipo em microcomputadores compatíveis com a linha IBM PC At sob MS-DOS 4.01, devendo estar concluído até março de 1993. O segundo, ainda em fase de concepção, deverá ser executado pela Divisão de Projetos Científicos (DPC) do Centro de Informática da Universidade Federal de Pelotas em um computador IBM 4381 sob VM/SP.

Ambos os projetos citados possuirão características experimentais e se destinam à avaliação de diferentes enfoques do modelo proposto em ambientes computacionais de portes distintos, sendo, de certa forma, complementares. Como resultados esperados das pesquisas a serem realizadas destacamos, entre outros possíveis, os seguintes:

- **Formalização Semântica:** O modelo carece ainda de uma semântica formal que permita uma interpretação clara das construções que oferece à representação e manipulação do conhecimento.

- **Raciocínio sobre o Controle:** As implementações deverão permitir a realização de experiências com o emprego de meta-raciocínio para a produção de inferências, possibilitando a comparação dos resultados obtidos.

- **Representação da Linguagem Natural:** As construções oferecidas pelo modelo parecem ser particularmente adequadas para a representação da linguagem natural.

- **BCs Estruturadas:** O modelo favorece a estruturação, integração e abstração de BCs, que trata indistintamente como objetos semânticos, assim como as demais entidades que permite representar, tais como visões, aspectos, etc.

- **Integração com outros Modelos:** O modelo das hiperredes pode ser empregado, possivelmente em condições vantajosas, para a descrição de outros modelos estruturados, tais como redes semânticas, sistemas de frames, redes neurais, etc.

- **Pesquisa Cognitiva:** Diferentes modelos cognitivos podem ser representados em hiperredes, permitindo investigar e comparar suas características.

Acreditamos haver conseguido mostrar que a associação da lógica ao modelo das hiperredes configura um ER particularmente promissor, cujos resultados práticos podem, por enquanto, ser apenas vislumbrados. A investigação realizada no presente trabalho permite, entretanto, reafirmar a importância da modelagem do conhecimento nos sistemas computacionais que nele se baseiam, priorizando a construção de bases de conhecimento eficientemente modeladas em relação aos métodos de raciocínio que atuarão sobre elas. A continuidade do trabalho iniciado aqui apresenta-se, para nós, de grande importância, e qualquer contribuição ao seu desenvolvimento será, naturalmente, bem recebida. Esperamos que as pesquisas que ora se iniciam, baseadas na presente dissertação, possam permitir um melhor entendimento do problema da representação do conhecimento e contribuir, de alguma forma, para a sua solução.

BIBLIOGRAFIA

[AHO 79] AHO, A. V.; ULMANN, J. D. Universality of Data Retrivial Languages. In: ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 6., Jan. 1979, San Antonio. **Proceedings** ... New York: ACM, 1979. p.110-117.

[AIE 88] AIELLO, L.; LEVI, G. The Uses of Metaknowledge in AI Systems. In: **META LEVEL ARCHITECTURES AND REFLECTION**. Amsterdam: North-Holland, 1988. 355p. p.243-254.

[AMB 87] AMBLE, T. **Logic Programming and Knowledge Engineering**. Reading: Addison-Wesley, 1987. 348p.

[BOR 86] BORGIDA, A. Survey of Conceptual Modelling of Information Systems. In: **READINGS IN KNOWLEDGE REPRESENTATION**. New York: Springer-Verlag, 1986. 425p. p.122-156.

[BOW 82] BOWEN, K. A.; KOWALSKI, R.A. Amalgamating Language and Metalanguage in Logic Programming. In: **LOGIC PROGRAMMING**. London: Academic Press, 1982. 366p. p.153-172.

[BOW 85] BOWEN, K. A. Meta Level Programming and Knowledge Representation. **New Generation Computing**, Tokyo, v.3, n.12, p.359-383, Oct. 1985.

[BOW 86] BOWEN, K. A. Meta Level Techniques in Logic Programming. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND ITS APPLICATIONS, 1986, Singapore. **Proceedings** ... Amsterdam: North Holland, 1986. p.262-271.

[BRA 82] BRACHMAN, R. J. e LEVESQUE H. J. Competence in Knowledge Representation. In: THE AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE '82, 1982, Pittsburgh. **Proceedings** ... Pittsburgh, P.A.: The Association, 1982. p.189-192.

[BRA 83] BRACHMAN, R. J. What IS-A Is and Isn't. **Computer**, Los Alamitos, v.16, n.10, p.30-36, Oct. 1983.

[BRA 83a] BRACHMAN, R. J. et al. KRYPTON: A Functional Approach to Knowledge Representation. **Computer**, Los Alamitos, v.16, n.10, p.67-73, Oct. 1983.

[BRA 86] BRACHMAN, R. J.; LEVESQUE, H. J. What makes a Knowledge Base Knowledgeable? A View of Databases from the Knowledge Level. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.69-78.

[BRI 86] BRATKO, I. **Prolog Programming for Artificial Intelligence**. Englewood Cliffs: Addison-Wesley, 1986. 423p.

[BRO 86] BRODIE, M. L. et al. Knowledge Base Management Systems: Discussions from the Working Group. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.19-34.

[BRO 86a] BRODIE, M. L.; JARKE, M. On Integrating Logic Programming and Databases. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/ Cummings, 1986. 701p. p.191-208.

[CAR 88] CARNOTA, R. J.; TESZKIEWICZ, A. D. **Sistemas Expertos y Representación del Conocimiento.** Buenos Aires: EBAI, 1988.

[CAS 87] CASANOVA, M. A.; GIORNO, F. A. C.; FURTADO, A. L. **Programação em Lógica e a Linguagem Prolog**. São Paulo: Edgard Blücher, 1987. 461p.

[CER 86] CERRO, L. F. D. MOLOG: A System That Extends PROLOG With Modal Logic. **New Generation Computing**, Tokyo, v.4, n.1, p.35-50, 1986.

[CHA 82] CHANDRA, A. K.; HAREL, D. Horn Clauses and Fixpoint Query Hierarchy. In: ACM SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, March 1982, Los Angeles. **Proceedings** ... New York: ACM, 1982. 304p. p.158-163.

[CLA 82] CLARK, K.; TÄRNLUND, S-A. **Logic Programming.** London: Academic Press, 1982.

[COS 88] COSCIA, P. et al. Object Level Reflection of Inference Rules by Partial Evaluation. In: **META LEVEL ARCHITECTURES AND REFLECTION.** Amsterdam: North-Holland, 1988. 355p. p.313-327.

[DAH 83] DAHL, V. Logic Programming as a Representation of Knowledge. **Computer**, Los Alamitos, v.16, n.10, p.106-111, Oct. 1983.

[DAT 83] DATE, C. J. **An Introduction to Database** Systems. 3. ed. Reading: Addison-Wesley, 1983. 513p.

[ELC 83] ELCOCK, E. W. How Complete are Knowledge Representation Systems? **Computer**, Los Alamitos, v.16, n.10, p.114-118, Oct. 1983.

[FEI 77] FEIGENBAUN, E. A. The Art of Artificial Intelligence. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 5., Aug. 1977, Cambridge, Massachusetts. **Proceedings** ... New York: ACM, 1977. p.1014-1029.

[FIS 87] FISCHLER, M.; FIRSCHEIN, O. **The Eye, the Brain and the Computer**. Reading: Addison-Wesley, 1987. 331p.

[FUR 84] FURUKAWA, K. et al. Mandala: A Logic Based Programming System. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1984, Tokyo. **Proceedings** ... Amsterdam: North Holland, 1984. 703p. p.613-622.

[GAL 78] GALLAIRE, H.; MINKER, J. **Logic and Databases**. New York: Plenum Press, 1978.

[GAL 83] GALLAIRE, H. Logic Databases vs Deductive Databases. In: LOGIC PROGRAMMING WORKSHOP '83, 1983, Albufeira, Portugal. **Proceedings** ... Amsterdam: North Holland, 1983.

[GAL 84] GALLAIRE, H.; MINKER, J.; NICOLAS, J.-M. Logic and Databases: A Deductitive Approach. **Computing Surveys**, New York, v.16, n.2, p.153-185, Jun. 1984.

[GEO 85] GEORGESCU, I. The Hipernets Method for Representing Knowledge. In: **ARTIFICIAL INTELLIGENCE: METHODOLOGY, SYSTEMS AND APPLICATIONS**. Amsterdam: North-Holland, 1985. p. 47-58.

[GEO 85a] GEORGESCU, I. et al. INTEXP: A Domain Independent Expert System. In: **ARTIFICIAL INTELLIGENCE: METHODOLOGY, SYSTEMS AND APPLICATIONS**. Amsterdam: North-Holland, 1985. p. 129-135.

[GRE 69] GREEN, C. Theorem Proving by Resolution as a Basis for Question-Answering Systems. In: **MACHINE INTELLIGENCE 4**. Edimburgh: Edimburgh University Press, 1969. p. 183-205.

[HAY 79] HAYES, P. The Logic of Frames. In: **FRAME CONCEPTIONS AND TEXT UNDERSTANDING**. Berlin: Walter de Gruyter, 1979. p.46-61.

[HOF 79] HOFSTADTER, D. **Godel, Escher and Bach.** New York: Basic Books, 1979.

[HOG 84] HOGGER, C. J. **Introduction to Logic Programming.** London: Academic Press, 1984. 278p.

[ISR 83] ISRAEL, D.; BERANEK, B. The Role of Logic in Knowledge Representation. **Computer,** Los Alamitos, v. 16, n. 10, p.37-41, Oct. 1983.

[IWA 88] IWANUMA, K.; HARAO, M. Knowledge Representation and Inference Based on First-Order Modal Logic. In: LOGIC PROGRAMMING 88', **Proceedings** ... Berlin: Springer-Verlag, 1988. p.237-251.

[JAC 86] JACKSON, P. **Introduction to Expert Systems.** Reading: Addison-Wesley, 1986. 292p.

[JAR 84] JARKE, M. External Semantic Query Simplification: A Graph-Theoretic Approach and Its Implementation in Prolog. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.675-692.

[KIN 86] KING, R. A Database Management System Based on an Object-Oriented Model. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.433-468.

[KIT 84] KITAKAMI, H. S.; MIYACHI, T.; FURUKAWA, K. A Methodology for Implementation of a Knowledge Acquisition System. In: INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, Feb. 1984, Atlantic City. **Proceedings** ... New York: ACM, 1984.

[KOW 74] KOWALSKI, R. A. Predicate Logic as a Programming Language. In: THE IFIP CONGRESS '74, Apr. 1974, Stockholm. **Proceedings** ... Amsterdam: North-Holland, 1974. p.569-574.

[KOW 75] KOWALSKI, R. A. A Proof Procedure Using Conection Graphs. **Journal of ACM,** New York, v. 22, n. 4, p.572-595, Apr. 1975.

[KOW 78] KOWALSKI, R. A. Logic for Data Description. In: **LOGIC AND DATABASES.** New York: Plenum Press, 1978.

[KOW 79] KOWALSKI, R. A. Algorithm = Logic + Control. **Communications of ACM,** New York, v.22, n.7, p.424-436, Jul. 1979.

[KOW 79a] KOWALSKI, R. A. **Logic for Problem Solving**. New York: Elsevier, 1979. 287p.

[LAN 90] LANGERAK, R. View Updates in Relational Databases with an Independent Scheme. **ACM Transactions on Database Systems**, New York, v.15, n.1, p.40-86, Mar. 1990.

[LID 84] LI, D. **A Prolog Database System**. Hertfordshire: Research Studies Press, 1984. 207p.

[LLO 84] LLOYD, J. W. **Foundations of Logic Programming**. Berlin: Springer-Verlag, 1984. 124p.

[MAE 88] MAES, P. Issues in Computational Reflection. In: **META LEVEL ARCHITECTURES AND REFLECTION**. Amsterdam: North-Holland, 1988. 355p. p.21-36.

[MAT 89] MATTOS, N. M. **An Approach to Knowledge Base Management**. Kaiserslautern: University of Kaiserslautern, 1989. 255p. PhD Thesis, Department of Computer Science.

[MCC 69] MCCARTHY, J.; HAYES, P. J. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: **MACHINE INTELIGENCE 4**. Edimburgh: Edimburgh University Press, 1969. p.463-502.

[MCC 77] MCCARTHY, J. Epistemological Problems of Artificial Intelligence. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 5., Aug. 1977, Cambridge, Massachusetts. **Proceedings** ... New York: ACM, 1977.

[MCC 80] MCCARTHY, J. Circunscription: A Form of Non-Monotonic Reason. **Artificial Intelligence**, v.13, n.1, p.27-39, 1980.

[MIN 75] MINSKI, M. A Framework for Representing Knowledge. In: **THE PSICHOLOGY OF COMPUTER VISION**. New York: McGraw-Hill, 1975. p.211-280.

[MIN 82] MINSKI, M. Why People Think Computers Can't? **AI Magazine**, v.3, n.1, p.2-8, 1982.

[MON 88] MONTEIRO, L.; PORTO, A. **Contextual Logic Programming**. Lisboa: Departamento de Informática, Universidade Nova de Lisboa, 1988.

[MOO 84] MOORE, R. C. A Formal Theory of Knowledge and Action. In: **FORMAL THEORIES OF THE COMMON SENSE WORLD**. Norwood: Ablex, 1984. p.319-358.

[NEW 82] NEWELL, A. The Knowledge Level. **Artificial Intelligence**, v.18, n.1, p.87-127, 1982.

[NIL 80] NILSSON, N. J. **Principles of Artificial Intelligence.** Palo Alto: Tioga, 1980.

[PAL 89] PALAZZO, L. A. M. **Rhesus: Um Modelo Experimental Para Representação de Conhecimento.** Porto Alegre: CPGCC da UFRGS, 1989. 115p.

[PAR 86] PARKER Jr., D. S. et al. Logic Programming and Databases. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.35-48.

[PEN 83] PENTLAND, A. P.; FISCHLER, M. A. A More Rational View of Logic. **AI Magazine**, v.4, n.4, Winter 1983.

[PER 82] PEREIRA, L. M. Logic Control With Logic. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1., Sept. 1982, Marseille, Fr. **Proceedings** ... Berlin: Springer-Verlag, 1982.

[PER 88] PERLIS, D. Meta in Logic. In: **META LEVEL ARCHITECTURES AND REFLECTION.** Amsterdam: North-Holland, 1988. 355p. p.37-50.

[REI 84] REITER, R. Towards a Logical Reconstruction of Relational Database Theory. In: **CONCEPTUAL MODELLING.** Berlin: Springer-Verlag, 1984. p.191-233.

[ROB 65] ROBINSON, J. A. A Machine-Oriented Logic Based on the Resolution Principle. **Journal of ACM,** New York, v.12, n.1, p.23-41, Jan. 1965.

[ROS 85] ROSENSCHEIN, S. J. Formal Theories of Knowledge in AI and Robotics. **New Generation Computing,** Tokyo, v.3, n.12, p.345-357, Oct. 1985.

[STE 86] STERLING, L.; SHAPIRO, E. **The Art of Prolog.** Cambridge: MIT Press, 1986. 427p.

[SAK 86] SAKAKIBARA, Y. Programming im Modal Logic: An Extension of Prolog Based On Modal Logic. In: LOGIC PROGRAMMING CONFERENCE, 5., 1986, Tokyo. **Proceedings** ... Berlin: Springer-Verlag, c1987.

[SHA 83] SHAPIRO, E. Y. Logic Programming with Uncertainties: A Tool For Implementing Rule-Based Systems. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 8., 1983, Karlsrue. **Proceedings** ... Los Altos, Calif.: Distributed by W. Kaufmann, 1983. p.529-532.

[STI 85] STICKEL, M; TYSON, W. An Analisys of Cosecutively Bounded Depth-First Search With Applications in Automated Deduction. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 9., 1985, Los Angeles. **Proceedings** ... Los Altos, Calif.: Distributed by M. Kaufmann, 1985. p.465-471.

[STO 88] STOREY, V. C.; GOLDSTEIN, R. C. A Methodology for Creating User Views in Database design. **ACM Transactions on Database Systems**, New York, v.13, n.3, p.305-338, Sept. 1988.

[TAR 75] TÄRNLUND, S.-A. An Interpreter for the Programming Language Predicate Logic. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 4., 1975, Tblisi. **Proceedings** ... New York: ACM, 1975. p.601-608.

[TUR 84] TURNER, R. **Logics for Artificial Intelligence**. West Sussex: Ellis Horwood, 1984. 121p.

[VIA 88] VIANU, V. A Dinamic Framework for Object Projection. **ACM Transactions on Database Systems**, New York, v.13, n.1, p.1-22, Mar. 1988.

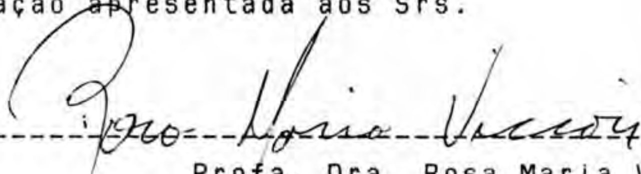
[WAT 86] WATERMAN, D. A. **A Guide to Expert Systems**. Reading: Addison-Wesley, 1986. 471p.

[ZAN 86] ZANIOLO, C. et al. Object Oriented Database Systems. In: INTERNATIONAL WORKSHOP ON EXPERT DATABASE SYSTEMS, 1., Oct. 24-27, 1984, Kiawah Island, South Carolina. **Proceedings** ... Menlo Park: Benjamin/Cummings, 1986. 701p. p.49-66.

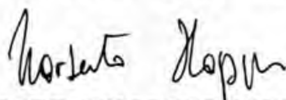
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Representação de conhecimento: programação em lógica e o modelo de hiperredes".

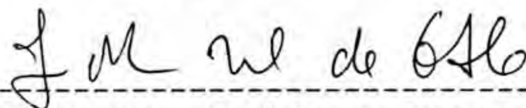
Dissertação apresentada aos Srs.



Profa. Dra. Rosa Maria Viccari



Prof. Dr. Norberto Hoppen



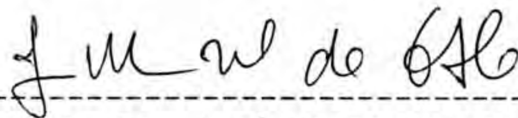
Prof. Dr. José Mauro Volkmer de Castilho

o examinador enviou parecer por escrito.

Prof. Dr. Daniel Schwabe (PUC/RJ)

Visto e permitida a impressão

Porto Alegre, 29/03/93



Prof. Dr. José Mauro Volkmer de Castilho
Orientador

Prof. Dr. Ricardo Augusto da L. Reis
Coordenador do Curso de Pós-Graduação
em Ciência da Computação

ESR/CCB ASSINA.BAN