Um Ambiente para Auxiliar a Construção de Núcleos de Sistemas Especialistas

Solange Rezende Rodrigues

Orientação:

Profa. Dra. Maria Carolina Monard

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos - USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Fevereiro de 1990

Ao Edinho
pelo incentivo, apoio e dedicação.

À Naiara
pela felicidade de sua existência.

Agradeço

à Maria Carolina, pelo seu exemplo de entusiasmo e dedicação ao trabalho e pela amizade demonstrada durante todo o desenvolvimento deste trabalho;

às Profas. Maria do Carmo Nicoletti e Doris Ferraz de Aragon, pelas sugestões e pelas críticas muito construtivas;

às amigas Maritza, Maria Inés, Júnia e Júlia pela amizade e constante participação na realização deste trabalho;

à amiga Helenice de Oliveira Florentino com carinho especial;

ao CNPq pelo apoio financeiro;

àqueles que de um modo ou de outro contribuiram para a realização deste trabalho.

Resumo

A construção do Núcleo de um Sistema Especialista pode ser facilitada se for realizada dentro de um ambiente que permita articular, bem como alterar os diversos subsistemas que o constituem tal que estes possuam características apropriadas para manipular Bases de Conhecimentos com características diferentes.

Neste trabalho é apresentada a implementação de cada um dos subsistemas que constitui este ambiente. As implementações realizadas são abertas, ou seja, é permitido que o projetista do SE se utilize tanto de um subconjunto das facilidades fornecidas, bem como — respeitando algumas condições — que troque algumas estruturas e redefina e/ou incremente o código dos subsistemas. O usuário pode interagir com este ambiente, de maneira a adequar o Núcleo de Sistema Especialista à manipulação da Base de Conhecimento de seu interesse.

Este ambiente, implementado na linguagem de programação lógica Prolog para microcomputadores IBM PC - compatível, leva em consideração a maioria dos problemas encontrados na construção de núcleos específicos e é dirigido a usuários não leigos em Sistemas Especialistas e Prolog.

Abstract

The process of constructing Expert Systems can be simplified if it is developed modularly as an independent knowledge base plus augmented meta-interpreters which can be tailored and combined as required.

In this work we describe a tool — based on this idea — which is part of an environment for developing Expert Systems.

This tool allows the user to choose the facilities needed to manipulate an especific knowledge base from several meta-interpreters that perform different functions. Eventually, when the system is debugged and ready for regular use, the collection of knowledge base and meta-interpreters can be mixed into an efficient program.

We also discuss inherent problems in manipulating this tool and how they may be solved in order to achieve the final environment.

The system has been written in Prolog and is currently running in Arity Prolog version 5.1 on IBM-PC based microcomputers.

Conteúdo

Lista de Figuras

Abreviaturas

BCh - Backward Chaining

EC - Engenheiro de Conhecimento

FCh - Forward Chaining

IA - Inteligência Artificial

MCD - Módulo Coletor de Dados

ME - Módulo de Explicação

MI - Motor de Inferência

NSE - Núcleo de Sistema Especialista

RC - Representação de Conhecimento

SE - Sistemas Especialistas

Capítulo 1

Introdução

1.1 Considerações Iniciais

Um Sistema Especialista -SE- é um programa que se comporta como um especialista em algum *domínio específico* de conhecimento. Assim sendo, um SE tenta imitar o raciocínio humano, na procura de soluções para um determinado tipo de problema.

SE é uma das áreas de pesquisa do grupo de Inteligência Artificial -IA- do Departamento de Computação e Estatística do ICMSC-USP. O tema começou a ser pesquisado em 1984, contando com a orientação do Prof. Antonio Eduardo Costa Pereira. Desde então foram várias as dissertações como por exemplo, [Barros 86], [Hebihara 88] e [Monard 86], e trabalhos, como [Monard 87] e [Nunes 86], desenvolvidos especificamente nesta área. Além do estudo teórico, foram realizadas implementações de protótipos, que diversas vezes foram submetidos a processos de refinamento.

Em geral, os trabalhos citados acima, concentram-se na resolução de problemas referentes à construção de SE, tais como: compilação de regras de conhecimento, raciocínio incerto, raciocínio por "default", etc. O desenvolvimento de tais trabalhos permitiu aos pesquisadores de Inteligência Artificial do ICMSC-USP adquirir experiência com relação a busca e implementação de soluções para aqueles problemas.

Deve ser ressaltado que, na área de Sistemas Especialistas, há uma grande variedade de Software de apoio comercializados, comumente denominados "shell", para auxiliar na construção deste tipo de sistemas. Entretanto, no ambiente da nossa Universidade, e devido em parte à atual política de informática existente no Brasil, temos uma certa dificuldade de importar Software.

A maioria dos "shells", à qual temos acesso, não possui grande flexibilidade: nem sempre é possível modificar sua a estratégia de busca, e tampouco realizar computações que não foram especificamente implementadas pelos projetistas do sistema. Esta falta de flexibilidade se deve, em parte, à ausência de programas que possuem a habilidade de se

auto-explicarem, de representar conhecimento, bem como da impossibilidade de trocar a representação usada por estes programas e suas estruturas de controle.

1.2 Motivação do Trabalho

A experiência adquirida no desenvolvimento dos trabalhos citados anteriormente, bem como a evidente falta de flexibilidade de grande parte dos "shells" aos quais temos acesso, motivaram o direcionamento deste trabalho para a construção de um Ambiente para desenvolver Núcleos de Sistemas Especialistas.

O Ambiente para auxiliar na construção de Núcleos de Sistemas Especialistas é um projeto, que tem como objetivo auxiliar projetistas de Sistemas Especialistas na construção dos Núcleos desses sistemas. O Ambiente gerará, com auxílio do projetista, um Núcleo de Sistema Especialista, com características apropriadas para manipular Bases de Conhecimento específicas.

A implementação do Ambiente é baseada na implementação abrangente de um Núcleo de Sistema Especialista, que leva em consideração a maioria dos problemas encontrados na construção de núcleos específicos, e é dirigido a usuários — projetistas de SE — não leigos em SE e Prolog.

O Núcleo do SE é constituído de três subsistemas principais, os quais serão detalhados em capítulos posteriores. Os subsistemas são:

- Um Motor de Inferência, com *raciocínio Backward Chaining* que também manipula regras imprecisas;
- Um Módulo Coletor de Dados, que interroga o usuário e admite diversas categorias de perguntas;
- Um Módulo de Explicação que explica, entre outros tópicos, a linha de raciocínio do sistema.

As implementações de cada um dos subsistemas que constitui este Ambiente são abertas; no sentido que elas permitem que se utilize um subconjunto das facilidades fornecidas, bem como — respeitando algumas condições — que se troque algumas estruturas, e redefina e/ou incremente o código dos subsistemas. O usuário pode, então, interagir com este Ambiente para construir um Núcleo de SE, mais adequado para manipular a Base de Conhecimento de seu interesse. Desta forma, o tempo de implementação do Núcleo de um Sistema Especialista *personalizado* diminue drasticamente.

1.3 Organização da Dissertação

A dissertação apresentada está organizada da seguinte forma:

No capítulo 1 são apresentadas as bases que motivaram este trabalho, dentro do interesse atual de pesquisas, no desenvolvimento de Núcleos de Sistemas Especialistas.

No Capítulo 2 é apresentada a conceituação e principais características de Sistemas Especialistas. Em seguida, descreve-se a Estrutura Básica de SE utilizada no desenvolvimento deste trabalho.

No Capítulo 3 é descrita a idéia geral usada na construção do ambiente para auxiliar no desenvolvimento de Núcleos específicos. Nesta descrição é destacada as possíveis mudanças a serem realizadas em cada um dos níveis da construção do Núcleo específico. A linguagem de implementação usada no desenvolvimento do trabalho é apresentada, bem como a representação de conhecimento manipulada pelo Núcleo do Sistema Especialista no nível mais alto do Ambiente proposto.

O Capítulo 4 é dedicado à apresentação do Módulo Coletor de Dados, juntamente com os tipos de perguntas implementadas, e exemplos de execução deste subsistema.

No Capítulo 5 é descrito o meta-interpretador usado na implementação do Motor de Inferência com raciocínio *Backward Chaining*, que também manipula regras imprecisas.

O Capítulo 6 é dedicado à descrição do Módulo de Explicação, onde são detalhados os tipos de explicações implementadas neste subsistema.

No Capítulo 7 são descritos os programas responsáveis pela comunicação entre os diversos subsistemas que constituem o Núcleo do Sistema Especialista que contém todas as facilidades implementadas. Também é descrita a comunicação destes subsistemas com a Base de Conhecimento, e com a Base de Dados.

O Capítulo 8 contém exemplos da interação com o Núcleo implementado.

Finalmente, no Capítulo 9 é realizado uma avaliação geral do Ambiente proposto. São apresentadas as conclusões e algumas sugestões de trabalhos futuros.

Capítulo 2

Sistemas Especialistas

2.1 Considerações Iniciais

Sistemas Especialistas constituem um dos segmentos mais explorados das pesquisas de Inteligência Artificial [Alty 84], [Carnota 88], [Hayes Roth 83], [Jackson 86], [Michie 82], [Shell 85], [Waterman 86].

Um SE é um programa que se comporta como um especialista humano em algum domínio específico de conhecimento. Assim sendo, um SE tenta imitar o comportamento humano na procura de soluções para o problema ao qual é aplicado. Por exemplo, programas para jogar xadrez, "bridge" ou "go", poderiam ser considerados como SE bem como os programas para diagnosticar possíveis causas de problemas de saúde na área médica, projetos eletrônicos, etc. Já um pacote estatístico ou matemático que envolve fórmulas complexas não seria normalmente considerado um SE.

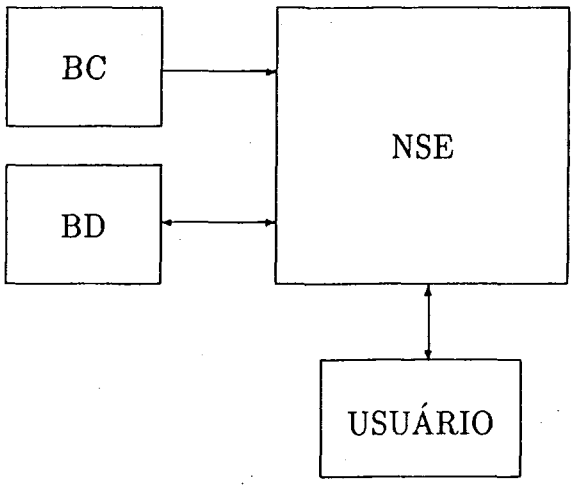
A razão principal é que, da mesma forma que um especialista humano, um SE também deve ser capaz de tratar com domínios cujas informações e métodos estejam incompletos ou são inexatos. A maioria dos problemas em um domínio de especialidade não tem uma solução baseada em algorítmos determinísticos, já que os problemas tem origem em um contexto muito complexo e não possuem descrições ou especificações exatas.

São várias as características que um SE deve ter. Em primeiro lugar, espera-se um bom desempenho na solução de problemas considerados "difíceis", embora não seja razoável exigir que tenha sempre um desempenho superior ao do especialista humano. Outra característica é que ele deve ser capaz de explicar sua linha de raciocínio, ou seja, *como* chega a certas conclusões e *porque* precisa de informações adicionais para chegar a essas conclusões.

Este capítulo é dedicado à apresentação da estrutura básica de um Sistema Especialista.

2.2 Estrutura Básica de um Sistema Especialista

Uma característica dos Sistemas Especialistas que é sempre evidenciada é a sua modularidade, uma vêz que ele é composto, fundamentalmente, pelos seguintes subsistemas — figura 2.1.



BC: Base de Conhecimento
BD: Base de Dados
NSE: Núcleo do Sistema Especialista

Figura 2.1: Estrutura Básica de um Sistema Especialista

- Base de Conhecimento -BC- contém a modelagem do conhecimento e, em alguns casos, heurísticas para manipular este conhecimento;
- Base de Dados -BD- constitui a área de trabalho do sistema;
- Núcleo do SE -NSE- é responsável pelo processamento do conhecimento usando alguma linha de "raciocínio"; pela justificação ou explicação das novas conclusões que obtém baseado neste raciocínio e pela interação com o usuário ou equipamentos externos.

Deve ser ressaltado que a figura 2.1 mostra a estrutura geral do SE do ponto de vista do usuário do sistema, ao qual não é permitido, em princípio, alterar a BC implementada pelo Engenheiro de Conhecimento.

2.2.1 Base de Conhecimento

A BC diz respeito aos fatos relativos à área de especialização; implementa regras que descrevem relações no domínio de conhecimento, e também, se for o caso, métodos heurísticos apropriados para resolver problemas no domínio específico. O responsável pela transferência do conhecimento do especialista humano para a Base de Conhecimento é o Engenheiro de Conhecimento.

Muitas regras estão calcadas na experiência ou intuição do especialista humano, ou seja, são regras empíricas ou aproximadas. Nestes casos, onde não existe teoria capaz de gerar um algoritmo, uma possível solução é trabalhar com dados e regras imprecisas, aos quais está associado um fator de certeza para indicar que nem sempre são dignas de confiança absoluta. Por exemplo, "pássaros voam" é um exemplo de regra imprecisa, enquanto que, "o paciente pode ser alérgico a penicilina" é um exemplo de dado impreciso.

Há vários formalismos para tratar estes fatores de certeza [Monard 89a], mas todos eles têm sido questionados [Pearl 86], [White 85]. Portanto, qualquer método escolhido está aberto a críticas.

São várias as razões que justificam aquele questionamento. Em primeiro lugar, não é sabido até agora, com certeza absoluta, como o ser humano pensa e raciocina e os SE necessitam de um modelo de raciocínio. Em segundo lugar, os métodos para tratamento de incerteza usados até agora, ainda que baseados em teorias supostamente corretas, requerem que algumas suposições sejam aceitas a fim de poder realizar uma implementação do método que evite a explosão combinatória.

Entre os vários modelos de raciocínio propostos, há três que têm tido bastante influência na área de SE; eles são os modelos usados em MYCIN [Shortliffe 76], PROSPECTOR [Duda 79] e Lógica Nebulosa ("Fuzzy-Logic") [Zadeh 65].

Em MYCIN, cada regra é associada a um número entre 0 e 1 que representa a certeza contida na regra. Usando estes números, que são semelhantes à probabilidades, mas não idênticos, MYCIN é capaz de combinar várias informações para chegar a uma conclusão com um certo grau de segurança. Esta é uma forma de raciocinar com pesos para validade.

Neste trabalho o modelo usado é semelhante ao de MYCIN. Deve ser salientado que é simples alterar o código referente a raciocínio com incertezas a fim de implementar os outros métodos, como será mostrado na seção 5.3.

2.2.2 Base de Dados

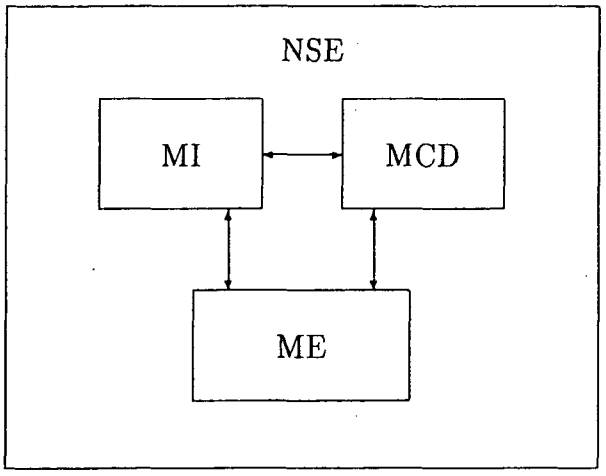
A BD serve para acompanhar o problema através de todas as suas fases. Ela contém os dados de entrada do problema, as respostas fornecidas pelo usuário, as asserções que foram deduzidas da BC, etc. A BD carrega a sequência histórica do que se tem feito, com o objetivo de poder explicar, se solicitado, os passos da linha de raciocínio do sistema.

2.2.3 Núcleo do Sistema Especialista

O NSE é o responsável pelo processamento do conhecimento usando alguma linha de "raciocínio", pela justificação ou explicação das novas conclusões obtidas usando este raciocínio e pela interação com o usuário. O NSE pode, então, ser subdividido em três módulos, como mostra a figura 2.2.

O Módulo Coletor de Dados -MCD- é responsável pela comunicação com o usuário e/ou outros sistemas. Este módulo, ativado pelo Motor de Inferência, fará as perguntas ao usuário, cujas respostas serão usadas pelo MI para inferir novas asserções.

O Motor de Inferência -MI- processa a BC e a BD usando uma linha de "raciocínio" a fim de propor alguma solução ao problema que está sendo analisado.



NSE: Núcleo do Sistema Especialista
MCD: Módulo Coletor de Dados
MI: Motor de Inferência
ME: Módulo de Explicação

Figura 2.2: O Núcleo do Sistema Especialista

Basicamente, quando a Base de Conhecimento usa regras, este processo pode ser executado pelas seguintes estratégias:

- Encadeamento regressivo ou "backward chaining" -BCh-.
- Encadeamento progressivo ou "forward chaining" -FCh-.

Se o raciocínio usado é o BCh, o MI pesquisa uma lista de hipóteses e procura reunir evidências, perguntando ao usuário por exemplo, de maneira a viabilizar a conclusão da validade de alguma(s) da(s) hipótese(s). Esta estratégia corresponde a perguntar:

`É possível provar as hipóteses a partir dos dados que temos ?`

Se o raciocínio usado pelo MI é o FCh, então o MI parte dos dados ou asserções existentes e a seguir, baseando-se nas regras de conhecimento, vai deduzindo outras asserções até chegar à solução do problema. Esta estratégia corresponde a perguntar:

`O que é possível concluir a partir dos dados que temos ?`

O MI implementado neste trabalho usa raciocínio BCh.

Finalmente, o Módulo de Explicação -ME- é responsável pela explicação de *como* o MI chega a certas conclusões, *por que* faz certas perguntas, etc.

2.3 Considerações Finais

Sistemas Especialistas podem ser construídos de maneiras diferentes, envolvendo diversas formas de representação do conhecimento e de suas estruturas de controle, bem como da estrutura na qual ele está baseada.

Neste capítulo foi apresentada uma pequena introdução a Sistemas Especialistas, concentrando-se na definição da estrutura Básica de um Sistema Especialista em que o Ambiente proposto neste trabalho está apoiado.

Capítulo 3

Um Ambiente para a Construção de Núcleos de Sistemas Especialistas

3.1 Considerações Iniciais

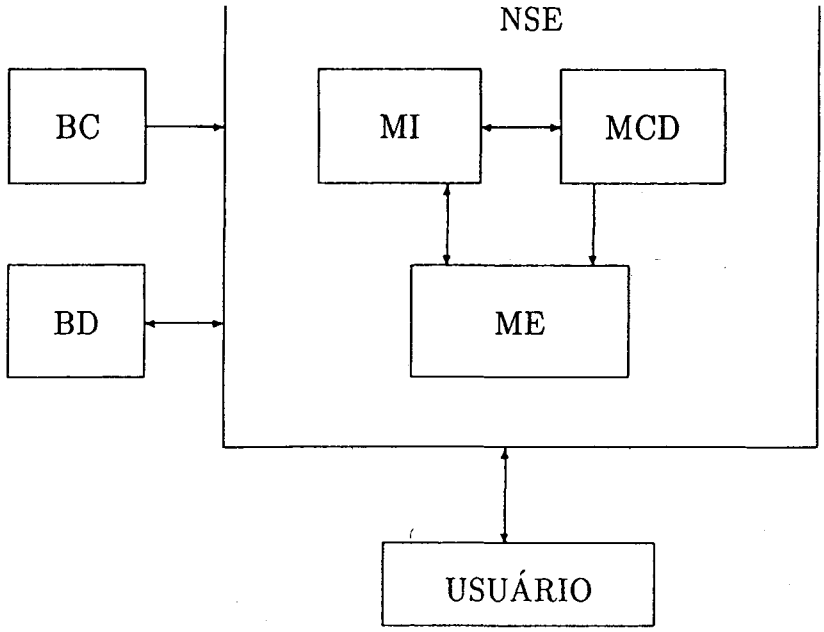
A construção de Núcleos de Sistemas Especilistas envolve decisões com relação ao formalismo a ser usado para representar o conhecimento, à escolha do mecanismo de inferência, às facilidades de interação com o usuário a serem oferecidas, etc. Assim sendo, é interessante desenvolver Núcleos de Sistemas Especialistas em um ambiente que permita conectar, bem como alterar, os subsistemas que o constituem, tornando-os adequados para a manipulação de Bases de Conhecimento com características diferentes.

Neste capítulo é apresentada uma descrição do Ambiente para o desenvolvimento de Núcleos de Sistemas Especialistas, para que o leitor tenha uma idéia mais clara do propósito do ambiente desenvolvido. Na descrição do Ambiente são destacados os procedimentos de mudança possíveis em cada um dos níveis a fim de construir Núcleos de Sistemas Especialistas específicos.

A seguir é apresentada suscintamente a linguagem de programação lógica Prolog, usada na implementação do sistema, bem como algumas diferenças entre um NSE com raciocínio BCh e o Motor de Inferência do Prolog, juntamente com a representação de conhecimento que é manipulada pelo Núcleo do Sistema Especialista implementado.

3.2 Descrição do Ambiente

A figura 3.1, exibe a estrutura do Sistema Especialista em que está apoiado o Ambiente desenvolvido.



BC: Base de Conhecimento
BD: Base de Dados
NSE: Núcleo do Sistema Especialista
MCD: Módulo Coletor de Dados
MI: Motor de Inferência
ME: Módulo de Explicação

Figura 3.1: Estrutura do Sistema Especialista na qual o Ambiente está Apoiado

O ambiente para construir Núcleos específicos de SE pode ser visualizado na figura 3.2. No nível mais alto temos a implementação completa do Núcleo de um SE, que inclue todas as opções implementadas neste Ambiente. O projetista do sistema pode utilizar todas as opções oferecidas neste nível, se este for o caso, o projetista estaria usando o Núcleo do SE completo, mostrado na figura 3.2. Entretanto, se a Base de Conhecimento que está sendo processada não fizer uso de todas as facilidades fornecidas neste nível, o usuário do sistema pode construir um núcleo que utiliza apenas as facilidades necessárias para sua aplicação específica. Este núcleo mais restrito — mostrado no Nível 1 da figura 3.2 — corresponde a um subconjunto do núcleo ilustrado no nível superior. Isto é realizado retirando de cada um dos subsistemas o código não relevante, mas sem realizar nenhuma alteração no código resultante.

Na passagem do Nível superior para o Nível 1, o usuário pode retirar as facilidades de cada um dos subsistemas que não são de seu interesse, reduzindo assim o código e, em muitos casos, melhorando o tempo de resposta deste núcleo específico.

A passagem para o Nível 2 somente é necessária para o usuário que decide alterar, por exemplo, a forma de gravar e acessar a Base de Dados, a estrutura de informação manipulada pelo Módulo de Explicação, o modelo usado pelo Motor de Inferência para processar raciocínio incerto, etc. Portanto, esta segunda fase é dirigida a usuários mais experientes.

A seguir mostramos a idéia geral usada na implementação do Núcleo completo e a representação de conhecimento por ele manipulada. Mostramos também algumas das opções fornecidas para a construção do primeiro e segundo nível do Núcleo específico.

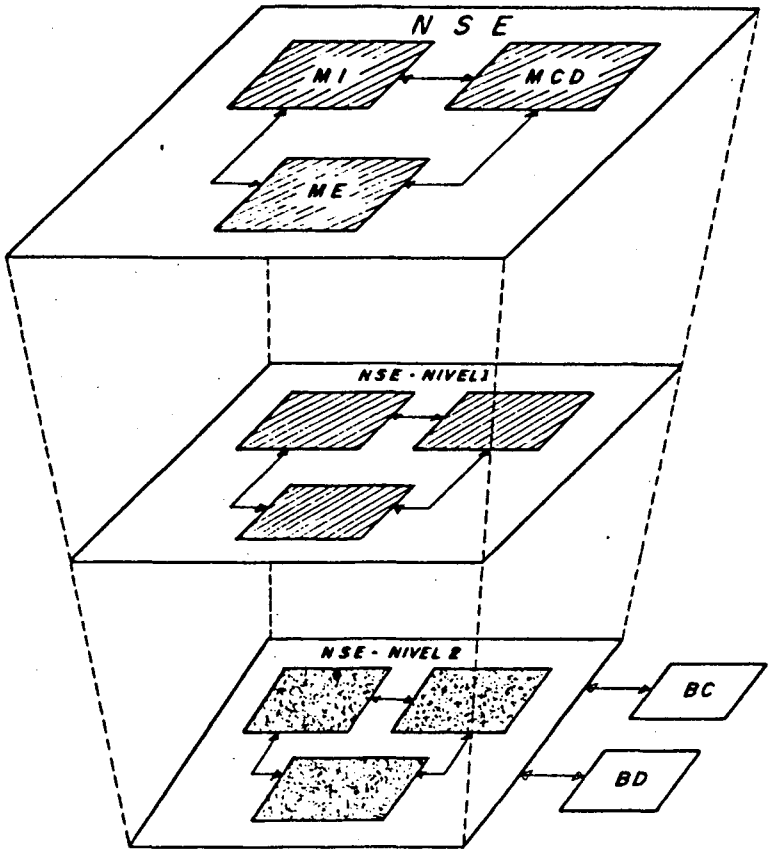


Figura 3.2: Ambiente para Auxiliar a Construção de Núcleos Específicos

3.3 A Linguagem de Implementação

Embora seja possível escrever qualquer tipo de programa em qualquer linguagem de programação, o processo de contrução de sistemas de IA pode ser facilitado consideravelmente se for usada uma linguagem que ofereça suporte para uma variedade de estruturas, tanto de dados quanto de controle. Várias linguagens de programação possuem essas características. Uma dessas linguagens é a linguagem de programação lógica Prolog, que é a linguagem utilizada no desenvolvimento deste trabalho.

A linguagem Prolog foi desenvolvida há quase vinte anos por Alain Colmerauer e seu grupo de IA em Marselha-França e atraiu imediatamente a atenção de vários cientistas de computação. Entretanto, a linguagem começou realmente a se destacar do ponto de vista prático quando David Warren [Warren 77] implementou, na Universidade de Edinburgh, um compilador Prolog muito eficiente. O interesse na linguagem aumentou ainda mais após 1981 devido a adoção de linguagens lógicas pelo projeto japonês de 5ª Geração [Cohen 85].

Informalmente, Prolog é uma linguagem na qual os programas são descritos como regras que são usadas para provar relações entre objetos; um programa consiste de um conjunto dessas regras. Cada relação em um programa Prolog é composta de um conjunto de cláusulas; o compilador ou interpretador Prolog possui um MI com raciocínio BCh que tenta provar a veracidade destas relações. O mecanismo usado para realizar essas provas está fundamentado no *princípio de resolução* [Robinson 65] que é um método de demonstração automática de teoremas.

No desenvolvimento de Sistemas Especialistas é possível identificar três formas diferentes de uso da linguagem Prolog [Rossi 86]:

- Escrever todo o Sistema Especialista diretamente em Prolog. Neste caso, o sistema serve exclusivamente para resolver o problema para o qual foi implementado e deve ser redefinido para resolver um outro problema diferente. Também, se for necessário adicionar novas facilidades para um problema já implementado, toda a Base de Conhecimento deve ser alterada. Este método retém a eficiência contida na Base de Conhecimento mas não permite realizar uma implementação modular e clara.

- Estender a linguagem Prolog a fim de incrementar a sua capacidade para implementar Sistemas Especialistas e, então, usar diretamente esta nova linguagem para implementar o sistema. Há várias propostas para estender a linguagem bem como várias implementações de novas linguagens que estendem Prolog; porém, estas linguagens não são ainda consideradas satisfatórias para implementar Sistemas Especialistas.

- Usar Prolog para implementar um meta-interpretador que manipula o conhecimento do Sistema Especialista. Neste caso, novas facilidades podem ser adicionadas ao sistema, introduzindo novos argumentos no meta-interpretador, sem que seja necessário alterar a Base de Conhecimento. Porém, deve ser lembrado que este

método pode apresentar problemas de eficiência se são usados muitos níveis de interpretação [Sterling 86a].

Este último método é o usado neste trabalho para implementar o Núcleo do Sistema Especialista no nível superior do Ambiente mostrado na figura 3.2.

3.4 Considerações sobre o Núcleo do Sistema Especialista

O Núcleo de um Sistema Especialista com raciocínio BCh deve, basicamente, pegar a representação interna de uma certa relação e provar que ela é verdadeira. Isto é o que faz o Motor de Inferência de Prolog, porém, há diferenças entre ambos.

- O NSE deve ser capaz de tirar conclusões sem ter certeza absoluta. Prolog somente trabalha com conclusões "seguramente verdadeiras" e "seguramente falsas".
- O NSE, através do Módulo Coletor de Dados, deve ser capaz de interagir com o usuário, equipamentos e outros programas para tirar conclusões que não podem ser tiradas por meio de inferências lógicas.
- O NSE deve ser capaz de explicar seus processos dedutivos fornecendo, no mínimo, explicações do tipo:

  porque faz certas perguntas

  e

  como chega a certas conclusões.

No sistema implementado, a Base de Conhecimento está armazenada internamente como relações Prolog, o que sugere, com as diferenças mostradas acima, uma divisão em cinco grandes grupos.

1. *Relações Primitivas*: são aquelas que devem ser executadas diretamente pelo Motor de Inferência do Prolog, ou seja, são programas Prolog. Relações primitivas não são capazes de explicar seu funcionamento ao usuário ou de tirar conclusões que sejam provavelmente corretas mas não seguramente corretas. Devem ser declaradas como relações primitivas aquelas regras da BC que efetuam cálculos e procedimentos algorítmicos.
2. *Relações de Sistema*: correspondem aos programas pré-definidos do Prolog e diretamente executados por ele. Relações de sistema são incapazes de explicar seu funcionamento ou de tirar conclusões que sejam provavelmente corretas mas não seguramente corretas.

3. *Relações de Decisão*: são interpretadas pelo NSE e, portanto, capazes de tirar conclusões provavelmente corretas mas não seguramente corretas, fazendo uso de fatores de certeza.

4. *Relações Perguntáveis*: são relações que exigem interação com um meio externo ao SE, que pode ser o usuário, equipamentos ou interação com outro sistema. O Módulo Coletor de Dados é responsavel pela interpretação destas relações.

5. *Relações Definidas*: são semelhantes às relações primitivas mas, por serem interpretadas pelo NSE e não pelo Prolog, elas são capazes de ativar tanto as relações de decisão quanto as perguntáveis.

3.4.1 Representação de Conhecimento

A representação de conhecimento -RC- é um dos problemas cruciais de IA e até agora não existe uma teoria geral de RC [Araribóia 89]. Entretanto, muitos métodos de RC têm sido estudados pelos pesquisadores de IA e, entre os que têm dado melhores resultados, encontram-se:

- a representação lógica [Kowalski 87];
- a representação procedimental [Kramer 87];
- os sistemas de produção [Buchaman 84];
- as redes semânticas [Brachman 83] e
- frames [Maida 87],[Minsky 75].

Neste trabalho, para representar o conhecimento do SE, é usada uma representação lógica, não necessariamente de primeira ordem, que permite deduzir novas informações a partir de fatos conhecidos [Hammond 80], [Clark 82], [Kowalski 86], [Walker 87].

O conhecimento manipulado pelo NSE implementado está internamente representado como relações Prolog e a divisão nos grupos acima mencionados é realizada usando tanto nomes de relações como argumentos específicos, que permitem reconhecer a que grupo pertence cada relação. A sintaxe do Prolog utilizado é a de Edinburgh; especificamente usamos o sistema Arity Prolog 5.1 [Arity 88] para a implementação. Nesta sintaxe, as cláusulas Prolog são escritas da seguinte forma

$$\texttt{atomo:- atomo}_1\texttt{, atomo}_2\texttt{, ..., atomo}_n\texttt{.}$$

terminada por '.' e pode ser lida como

$$\texttt{atomo se atomo}_1\texttt{ e atomo}_2\texttt{ ... e atomo}_n\texttt{.}$$

onde atomo denota um predicado aplicado a argumentos como na sintaxe do Cálculo de Predicados, atomo é a cabeça da cláusula e $\texttt{atomo}_1 \ldots \texttt{atomo}_n$ constituem o corpo da cláusula.

Informalmente, a sintaxe de Prolog de Edinburgh em Backus Normal Form -BNF- é dada por

```
<clausula>   ::=<atomo>.|
                <atomo>:-<listaatomo>.
<listaatomo>::=<atomo>|
                <atomo>,<listaatomo>|
                (<listaatomo>;<listaatomo>)
<atomo>      ::=<pred>|
                <pred>(<listaexp>)
<listaexp>   ::=<exp>|
                <exp>,<listaexp>
<exp>        ::=<const>|
                <var>|
                <fn>(<listaexp>)
```

onde

- ',' representa o operador e (and)
- ';' o operador ou (or)
- ':-' o operador 'se' (if)
- <pred> e <fn> são identificadores que começam com uma letra minúscula
- <var> são identificadores que começam com uma letra maiúscula
- o símbolo '_ ' que denota a variável anônima
- <const> é qualquer termo constante, incluindo as constantes numéricas.

Informalmente, a representação de conhecimento para os diversos grupos de relações da BC é a seguinte

1. *Relações Primitivas*: são reconhecidas por um predicado da forma

   ```
   primitiva(<atomo>).
   ```

   onde <atomo> é a cabeça de uma cláusula que será executada diretamente por Prolog. Por exemplo, se declaramos

   ```
   primitiva(media_aritimetica(Lista,Media)).
   ```

   o MI pode pedir para Prolog executar diretamente media_aritmetica/2. Portanto, este programa deve estar definido na BC. Por exemplo:

```
media_aritmetica(Lista,Media):-
   no_elementos(Lista,N),
   soma_elementos(Lista,S),
   Media is S/N.

no_elementos([],0).
no_elementos([Elem|Cauda],N):-
   no_elementos(Cauda,N1),
   N is N1 + 1.

soma_elementos([],0).
soma_elementos([X|Y],S):-
   soma(Y,S1),
   S is S1 + X.
```

2. *Relações de Sistema*: são as relações pré-definidas do Prolog, tais como `clause/2`, `is/3`, etc e não precisam ser especificamente declaradas na BC, o MI reconhece as mesmas e pede para Prolog executá-las diretamente.

3. *Relações de decisão*: são reconhecidas por cláusulas com a seguinte cabeça:

   `<pred>(fc,C,F,<expr>):- ...`

   onde `F` representa o fator de certeza dado pelo especialista a esta regra de conhecimento e `C` representa o fator de certeza que será calculado, se for necessário, durante uma consulta. O corpo destas relações admite qualquer tipo de relações 1 a 5.

4. *Relações Perguntáveis*: são reconhecidas por um predicado da forma

   `perguntavel(<atomo>).`

   e são as que interagem com o MCD. Este tipo de relação é explicada mais detalhadamente na próxima seção.

5. *Relações Definidas*: estas relações são reconhecidas pelo MI por não pertencer a nenhum dos tipos acima. Elas são definidas na seguinte forma:

   `<pred>(verifica(N),<listaexp>):-`

   `atomo or atomo`$_1$ `... or atomo`$_n$.

   com `N` inteiro ou, simplesmente, uma cláusula Prolog, onde o primeiro argumento da cabeça da cláusula não é o funtor `verifica(N)`.

No caso da regra de conhecimento pertencer ao grupo de relações definidas da forma:

`<pred>(verifica(N),<listaexp>):-`

`atomo or atomo`$_1$ `... or atomo`$_n$.

a regra de conhecimento especifica que ela é válida em um dado contexto, se for possível provar a validade de um conjunto que contém N das premissas que estão na cauda da regra. O corpo das relações definidas admite qualquer tipo de relações 1 a 5.

Todas estas relações da BC são reconhecidas e manipuladas pelo Núcleo do Sistema Especialista.

3.5 Considerações Finais

Foi apresentada neste capítulo uma introdução ao ambiente desenvolvido para auxiliar na construção de Núcleos de Sistemas Especialistas, evidenciando os passos a serem seguidos na construção de Núcleos Específicos adequados para manipular a Base de Conhecimento de interesse do projetista do Sistema.

Em seguida, foram descritas diferentes formas de implementar SE em Prolog, bem como a representação de conhecimento manipulada pelo Núcleo do Sistema Especialista implementado — NSE do Nível superior no Ambiente mostrado na figura 3.2.

Nos próximos capítulos serão mostradas várias das idéias usadas na implementação dos diversos subsistemas que constituem este Núcleo de Sistema Especialista.

Capítulo 4

O Módulo Coletor de Dados

4.1 Considerações Iniciais

Este capítulo tem o objetivo de apresentar o subsistema denominado Módulo Coletor de Dados, que é responsável pela extração de informações de um meio externo ao Sistema Especialista, tal como usuário, equipamentos ou um outro sistema. Várias das idéias incorporadas na implementação do MCD derivam de outros trabalhos [Hammond 83], [Niblett 84].

O MCD implementado está encarregado de formular perguntas ao usuário, cujas respostas são usadas pelo Motor de Inferência para chegar a certas conclusões. Quando o MI tenta provar uma meta que não pode ser deduzida da Base de Conhecimento ou de perguntas já respondidas pelo usuário, e se esta meta está declarada na BC como

```
perguntavel(Meta).
```

o Motor de Inferência ativa o Módulo Coletor de Dados fornecendo-lhe as seguintes informações:

- Meta a ser perguntada.
- Explicação a ser fornecida ao usuário caso ele desejar saber o motivo (por_que) desta pergunta.

Na implementação realizada do MI, esta explicação está baseada na cadeia de regras e metas que conectam esta informação com a interrogação original ao Sistema Especialista. Esta cadeia de regras e metas é usualmente chamada de "trace". O "trace" pode ser visualizado como uma cadeia de regras que conecta a meta que está sendo explorada no momento pelo Motor de Inferência e a meta original que interroga o SE, em uma árvore AND/OR.

Quando o Módulo Coletor de Dados faz uma pergunta e obtém do usuário a resposta por_que, ele ativa o Módulo de Explicação fornecendo-lhe a informação contida no "trace". Na primeira vez que é ativado, o ME mostra, em uma forma apropriada, a regra corrente, isto é, a regra de conhecimento que o Motor de Inferência está tentando provar e, logo após, volta o controle para o MCD que repete a pergunta. Se o usuário voltar a responder por_que, o ME é novamente ativado e a regra imediatamente acima no espaço de busca é mostrada e o controle volta para o MCD que faz a pergunta novamente.

Este processo pode, eventualmente, ser repetido até que a meta original seja alcançada. Neste caso, o ME informa ao usuário que não tem mais explicações a fornecer. Observe que este processo permite verificar a história da prova realizada até o momento.

É importante ressaltar que, se as explicações às perguntas do usuário forem de outro tipo, é muito simples reescrever o código correspondente a fim de acomodar outra estrutura de informação.

4.2 Base Algorítmica Manipulada pelo Módulo Coletor de Dados

Na estrutura do Sistema Especialista na qual o ambiente implementado está apoiado — figura 3.1 — a BC é considerada como um todo, ou seja, ela contém toda a informação necessária para a correta execução do sistema. Porém, algumas dessas informações, especificamente programas algorítmicos de escrita e crítica de dados, são manipulados somente pelo MCD. Portanto, é conveniente destacar, por motivos de clareza, esta distinção na Base de Conhecimento. Na figura 4.1, a área nomeada base Ba refere-se aos programas algorítmicos que são manipulados somente pelo MCD do Núcleo de um Sistema Especialista.

Para cada relação perguntável declarada na BC, deve existir informação específica na base Ba a fim de informar ao MCD o tipo de pergunta a ser realizada, qual a pergunta a ser feita, se a meta e a pergunta a serem feitas admitem variáveis instanciadas, responsabilidade de leitura e aceitação das respostas fornecidas pelo usuário, etc.

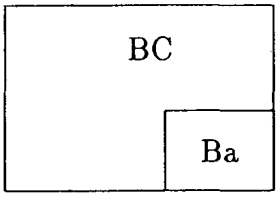


Figura 4.1: Divisão da Base de Conhecimento

Podemos dividir a informação requerida na base Ba em dois grandes grupos — figura

4.2 — tendo em conta se a responsabilidade da leitura e validação das respostas do usuário é do implementador da base Ba ou da rotina de leitura e funções de crítica do Módulo Coletor de Dados. Por sua vez, cada um destes grupos pode ser novamente subdividido dependendo da meta e da pergunta a serem realizadas admitirem ou não variáveis instanciadas. Cada um destes grupos correspondem a um predicado na base Ba como é mostrado a seguir.

No caso do MCD ser responsável pela leitura e crítica das respostas do usuário, deve existir na base Ba um predicado `pergunta/3` ou `pergunta/4` com os seguintes argumentos:

```
pergunta(Meta,Perguntas,Critica)
```

ou

```
pergunta(Meta,VarsInst,Perguntas,Critica)
```

onde

- `Meta` é o argumento da relação perguntável, que é a meta que está sendo provada pelo MI e que deverá ser preenchida com as respostas dadas pelo usuário;
- `Perguntas` corresponde a uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Estes argumentos são programas que devem estar definidos na base Ba e são ativados pelo MCD. Por exemplo, programas para criar janelas, escrever dentro de uma janela determinada, etc. No mínimo, um destes programas deve escrever o texto da pergunta a ser realizada;
- `Critica` determina o tipo de pergunta — afirmativa-negativa, única etc ... — e as funções de crítica das respostas que serão ativadas pelo MCD para validar as respostas fornecidas pelo usuário;
- `VarsInst` determina quais as variáveis de `Meta` que devem estar instanciadas para serem usadas nas perguntas realizadas.

Quando não é responsabilidade direta do Módulo Coletor de Dados realizar a leitura e criticar as respostas do usuário, então, deve existir na base Ba um predicado `pergunta_livre/3` ou `pergunta_livre/4`. Estes predicados devem ser declarados com os seguintes argumentos:

```
pergunta_livre(Meta,Perguntas,Tipo_Resps).
```

ou

```
pergunta_livre(Meta,VarInst,Perguntas,Tipo_Resps).
```

onde

- `Meta` e `VarsInst` são como explicadas anteriormente;

- `Perguntas` corresponde a uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Analogamente ao caso anterior, estes argumentos são programas que devem estar definidos na base Ba e que são ativados pelo MCD. A diferença com o caso anterior é que estes programas devem escrever o texto da pergunta a ser realizada, bem como ler e validar as respostas fornecidas pelo usuário. Além disso, as respostas do usuário devem ser unificadas com variáveis que sejam visíveis para o argumento `Tipo_Resps`;

- `Tipo_Resps` é uma lista onde o primeiro elemento determina o tipo de pergunta — afirmativa-negativa, única ou várias — como no caso anterior, mas os outros elementos são agora nome de variáveis Estas variáveis devem ser idênticas às usadas pelos programas de leitura definidos em `Perguntas`. É responsabilidade desses programas unificar estas variáveis com as respostas fornecidas pelo usuário.

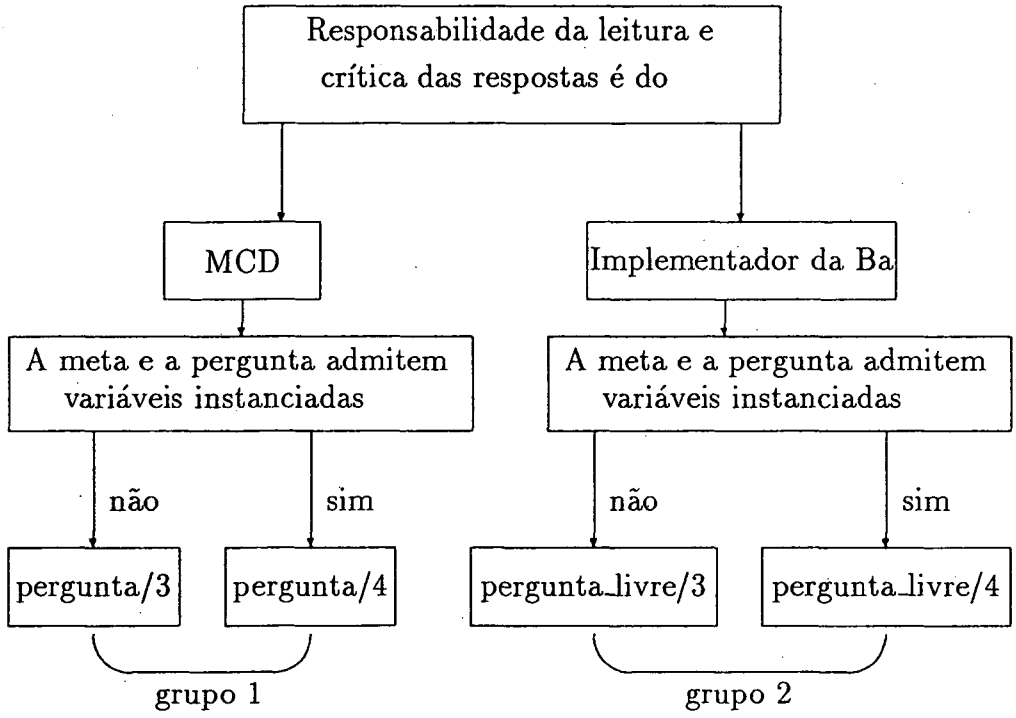


Figura 4.2: Informação na base Ba

Portanto, os predicados `pergunta/3` e `pergunta/4` correspondem, respectivamente, aos mesmos casos dos predicados `pergunta_livre/3` e `pergunta_livre/4`, com a diferença que nos casos em que `pergunta_livre` for usado, é responsabilidade do projetista da base Ba, definir a pergunta a ser feita bem como obter as respostas para esta pergunta.

O motivo de ter `pergunta` ou `pergunta_livre` com aridade 3 e 4 é o seguinte: suponha que a relação perguntável e o predicado pergunta/3 são

```
perguntavel(peso_altura_idade(Peso,Altura,Idade)).
```

```
pergunta(peso_altura_idade(Peso,Altura,Idade)),pg(perg),
        [u,critica_peso,critica_altura,critica_idade]).
```

e a pergunta a ser feita é

```
Qual o peso, a altura e a idade do paciente?
```

ou seja, a pergunta não carrega nenhuma informação adicional ligada a esta meta. Neste caso, pergunta/3 ou pergunta_livre/3 deve ser usada. Suponha, entretanto, que desejamos realizar uma pergunta personalizada, pois se está perguntando o peso a altura e a idade de um determinado paciente, por exemplo

```
Qual o peso, a altura e a idade de Jose Silva ?
```

neste caso, a relação perguntávele o predicado pergunta/4 seriam, por exemplo

```
perguntavel(peso_altura_idade(Paciente,Peso,Altura,Idade)).

pergunta(peso_altura_idade(Paciente,Peso,Altura,Idade)),
        [Paciente],pg(perg(Paciente)),
        [u,critica_peso,critica_altura,critica_idade]).
```

e pergunta/4 ou pergunta_livre/4 deve ser usada, onde o segundo argumento é uma lista de um elemento que contém a variável que deve estar instanciada na chamada, neste caso, a correspondente a Paciente.

A razão é que, ao ser ativado, o MCD faz com que todas as variáveis de Meta sejam variáveis livres ("desinstancia") exceto aquelas que estão declaradas no segundo argumento de pergunta/4 ou pergunta_livre/4, a fim de preencher estas variáveis livres com as respostas fornecidas pelo usuário.

A regra geral a ser usada é a seguinte: se a pergunta a ser realizada não carrega nenhuma informação, pergunta/3 ou pergunta_livre/3 deve ser definida, caso contrário pergunta/4 ou pergunta_livre/4 deve ser definida. Na próxima seção são descritos, informalmente, os argumentos das relações perguntavel/1, pergunta/3, pergunta_livre/3, pergunta/4 e pergunta_livre/4 bem como são mostrados alguns exemplos juntamente com os programas de crítica dos dados correspondentes.

Resumindo, o Motor de Inferência deve ser capaz de reconhecer as relações perguntáveis e, se a pergunta ainda não tiver sido realizada, o MI deve ativar o Módulo Coletor de Dados para realizar as perguntas correspondentes a esta meta. Portanto, as metas que são perguntáveis devem ser declaradas como tal na Base de Conhecimento, isto é

```
perguntavel(Meta).
```

Além disso, devem estar declarados na base Ba o modo como a pergunta correspondente a esta `Meta` deve ser formulada, o tipo da pergunta (explicado a seguir), bem como funções — explícitas no caso das perguntas do grupo 1 ou implícitas no caso das perguntas do grupo 2 — para criticar as respostas do usuário, ou seja, verificar se elas são ou não aceitáveis.

4.3 Tipos de Perguntas Implementadas

Os tipos de perguntas implementadas são:

- *afirmativas ou negativas*, para perguntas que necessitam de respostas sim ou não;
- *únicas*, para perguntas que devem ser feitas somente uma vez;
- *várias*, para perguntas que admitem várias respostas. Neste caso, a pergunta é feita várias vezes até que o usuário responda `basta` ou `pare`.

A seguir, são mostrados alguns exemplos descompromissados correspondentes a cada tipo de pergunta, que mostram a necessidade dos diversos tipos de informações na base Ba. Lembre que o Módulo Coletor de Dados permite ao projetista da base Ba decidir se a leitura das respostas fornecidas pelo usuário, para uma meta que foi declarada como perguntável na Base de Conhecimento, é realizada pelo Módulo Coletor de Dados ou é de exclusiva responsabilidade do implementador das perguntas. Esta diferença é feita usando as relações de nome `pergunta` e `pergunta_livre` declaradas na base Ba.

A fim de não confundir, são definidos programas muito simples para escrever e criticar dados. Porém, estes programas podem ser, eventualmente, bastante complexos. É mostrado também o resultado da execução do programa `faz_pergunta/4`, que ativa o Módulo Coletor de Dados onde as respostas fornecidas pelo usuário estão precedidas pelo símbolo '>'.

Deve ser salientado que a resposta `nao_sei` é aceita nos três tipos de perguntas bem como a resposta `por_que` que, como já foi visto, ativa o ME para explicar o motivo desta pergunta sempre que o MCD fizer parte de um NSE. No caso em que `pergunta_livre/3` ou `pergunta_livre/4` for definido, fica por conta do projetista da base Ba fornecer ao usuário a opção de responder `nao_sei` `por_que` e `basta` ou `pare` no caso em que o tipo de pergunta é *várias*.

4.3.1 Tipo de Pergunta que Admite Resposta Afirmativa ou Negativa

Suponha que seja necessário saber se um fiador de um cliente de uma imobiliária conhece suas obrigaçoẽs como fiador. Neste caso poder-se-ia ter:

- na BC

```
perguntavel(obrigacoes_fiador).
```

- na Ba

```
pergunta(obrigacoes_fiador,pg(perg1),[s-n]).

perg1:-
  write('Voce sabe das suas obrigacoes como fiador ?'),
  nl.
```

O argumento de perguntavel/1 é sempre a meta a ser perguntada e deve ser igual ao primeiro argumento da relação pergunta/3 correspondente. O segundo argumento é uma estrutura onde o funtor é um átomo qualquer — pg neste caso — e possui um número variável de argumentos. Estes argumentos correspondem aos programas de escrita que serão ativados pelo MCD. O terceiro argumento é uma lista onde o primeiro elemento — também o único elemento no caso de respostas afirmativas ou negativas, pois neste caso o MCD já possui a função de crítica destas respostas — indica o tipo de pergunta.

Segue-se um exemplo de execução correspondente a este caso.

```
?-faz_pergunta(obrigacoes_fiador,MSaida,Inf_PorQue,Classe).
Voce sabe das suas obrigacoes como fiador ?
>sim.
```

o programa é bem sucedido com

```
MSaida = obrigacoes_fiador
Classe = sim
```

No caso de ser necessário saber se um fiador específico conhece suas obrigações, a relação pergunta/4 deve ser usada, isto é:

- na BC

```
perguntavel(obrigacoes_fiador(Cliente)).
```

- na Ba

```
  pergunta(obrigacoes_fiador(Cliente),[Cliente],
           pg(perg2(Cliente)),[s-n]).

perg2(Cliente):-
  write(Cliente),
  write(' voce sabe das suas obrigacoes como fiador ?'),
  nl.
```

A seguir, são usados estes mesmos exemplos para ilustrar os casos onde a leitura das respostas é de exclusiva responsabilidade do projetista das perguntas.

Para o primeiro caso onde o predicado `pergunta/3` é substituído por `pergunta_livre/3` poder-se-ia ter:

- na BC

```
perguntavel(obrigacoes_fiador).
```

- na Ba

```
pergunta_livre(obrigacoes_fiador,pg(perg1(X)),[s-n,X]).

perg1(X):-
  faz_janela,
  write('Voce sabe das suas obrigacoes como fiador ?'),
  ativa(7,0,X),
  exit_popup.

faz_janela:-
  create_popup(' INTERACAO COM O USUARIO ',
               (14,10),(22,68),(79,-79)).

begin_menu(resposta,50,
           colors((113,31),(113,31),(123,27),(123,116))).
item($sim$,sim).
item($nao$,nao).
item($nao_sei$,nao_sei).
item(right($por_que$),por_que).
end_menu(resposta).

ativa(L,C,Resposta):-
    send_menu_msg(activate(resposta,(L,C)),Resposta).
```

Neste caso, o segundo argumento de `pergunta_livre/3` é uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Estes argumentos correspondem a programas de escrita e leitura que serão ativados pelo MCD. Neste exemplo a pergunta foi escrita em uma janela do tipo "popup" e a leitura, que é de exclusiva responsabilidade do projetista da base Ba, é realizada usando um menu que somente aceita respostas `sim`, `nao`, `nao_sei` e `por_que`. O terceiro argumento é uma lista onde o primeiro elemento indica o tipo de pergunta (s-n) e o outro é uma variável que vai ser instanciada com a resposta válida do usuário, para ser posteriormente avaliada, se for o caso, pelo MCD.

Segue-se um exemplo de execução correspondente a este caso.

```
File  Edit  Buffers  Info  Debug  Switch                                  Help
MAIN
yes
?- faz_pergunta(obrigacoes_fiador,MSaida,InfPorque,Classe).

        INTERACAO COM O USUARIO
        Voce sabe das suas obrigacoes como fiador ?

        sim  nao  nao_sei                          por_que
Arity/Prolog Interpreter Version 5.1  Copyright (C) 1989 Arity Corporation
```

após o usuário responder `sim` o programa é bem sucedido com

```
MSaida = obrigacoes_fiador
Classe = sim
```

No caso de saber se um fiador específico conhece suas obrigações, o predicado `pergunta_livre/4` deve ser usado, isto é:

- na BC

```
perguntavel(obrigacoes_fiador(Cliente)).
```

- na Ba

```
pergunta_livre(obrigacoes_fiador(Cliente),[Cliente],
                pg(perg2(Cliente,X)),[s-n,X]).

perg2(Cliente,X):-
  faz_janela,
  write(Cliente),
  write(' voce sabe das suas obrigacoes como fiador ?'),
  ativa(7,0,X),
  exit_popup.
```

4.3.2 Tipo de Pergunta a ser Realizada Somente uma Vez

Suponha que seja necessário saber de um fiador, que não é explicitamente identificado, o seu salário e se ele já é fiador de outra pessoa. Neste caso poder-se-ia ter:

- na BC

```
perguntavel(salario_fiador(Salario,Fiador)).
```

- na Ba

```
pergunta(salario_fiador(Salario,Fiador),pg(perg3),
        [u,critica_salario,critica_e_fiador]).

perg3:-
  write('Qual a sua renda mensal em salarios minimos)?'),
  nl,
  write('Voce ja e fiador de alguem ?'),
  nl.

critica_salario(X):-
  integer(X),
  X > 2,
  !.
critica_salario(_):-
  write('O valor aceitavel para renda deve ser maior que 2.'),
  nl,
  nova_resposta,
  fail.

critica_e_fiador(X):-
  pertence(X,[sim,nao]),
  !.
critica_e_fiador(_):-
  write('Os valores validos sao sim ou nao .'),
  nl,
  nova_resposta,
  fail.

nova_resposta:-
  write('Responda novamente, por favor.'),
  nl.
```

O primeiro e o segundo argumento de `pergunta/3` são como explicados anteriormente. O terceiro argumento é uma lista onde o primeiro elemento é um átomo, u, que identifica este tipo de pergunta. Os outros elementos da lista são os nomes das relações das funções de crítica na ordem de entrada dos dados. Todas as funções de crítica tem aridade 1.

No exemplo acima, o primeiro nome é `critica_salario`. Isto indica que o predicado `critica_salario/1` é aquele que verifica que a resposta do usuário para salário é um

elemento do domínio de valores definido para a variável `Salario` (um número inteiro e maior que 2). O segundo nome indica que o predicado `critica_e_fiador/1` realiza tarefa análoga para a variável `Fiador`. Se a pergunta a ser feita for:

```
Voce ja e fiador de alguem ?
Qual a sua renda mensal em salarios minimos ?
```

então, a ordem dos nomes das funções de crítica nesta lista também devem ser trocadas. Neste caso, o terceiro argumento de `pergunta/3` seria

```
[u,critica_e_fiador,critica_salario]
```

Observe que uma função de crítica do tipo

```
critica_nada(_).
```

permite processar respostas que admitem qualquer tipo de valor.

No caso de um fiador explicitamente identificado poder-se-ia ter:

- na BC

```
perguntavel(salario_fiador(Cliente,Salario,Fiador)).
```

- na Ba

```
pergunta(salario_fiador(Cliente,Salario,Fiador),[Cliente],
         pg(perg4(Cliente)),
         [u,critica_salario,critica_e_fiador]).

perg4(Cliente):-
  write(Cliente),
  write(' qual o sua renda mensal em salarios minimos ?'),
  nl,
  write('Voce ja e fiador de alguem ?'),
  nl.
```

com as mesmas funções de crítica definidas anteriormente. Segue-se um exemplo de execução deste último caso.

```
?-faz_pergunta(salario_fiador(pedro,Salario,Efiador),
               MSaida,Inf_PorQue,Classe).
pedro qual a sua renda mensal em salarios minimos ?
Voce ja e fiador de alguem ?
>-10.
>nao_sei.

O valor aceitavel para renda deve ser maior que 2.
Responda novamente, por favor.
>10.
```

o programa é bem sucedido com

```
MSaida = salario_fiador(pedro,10,nao_sei)
Classe = nao_sei
```

O Módulo Coletor de Dados classifica a resposta na classe `nao_sei` sempre que o usuário responder `nao_sei` a pelo menos uma das informações requeridas na pergunta em questão. Porém, pode ser observado no exemplo acima que as respostas válidas são processadas pelo MCD e somente as variáveis correspondentes à resposta `nao_sei` são instanciadas com este átomo. Assim, o Motor de Inferência que eventualmente estiver usando este MCD, tem liberdade para analisar individualmente a informação fornecida pelo usuário, bem como processar a classe `nao_sei` de respostas como um todo.

A seguir são considerados os mesmos exemplos anteriores para ilustrar o uso de `pergunta_livre`. No caso de `pergunta_livre/3` poder-se-ia ter:

- na BC

```
perguntavel(salario_fiador(Salario,Fiador)).
```

- na Ba

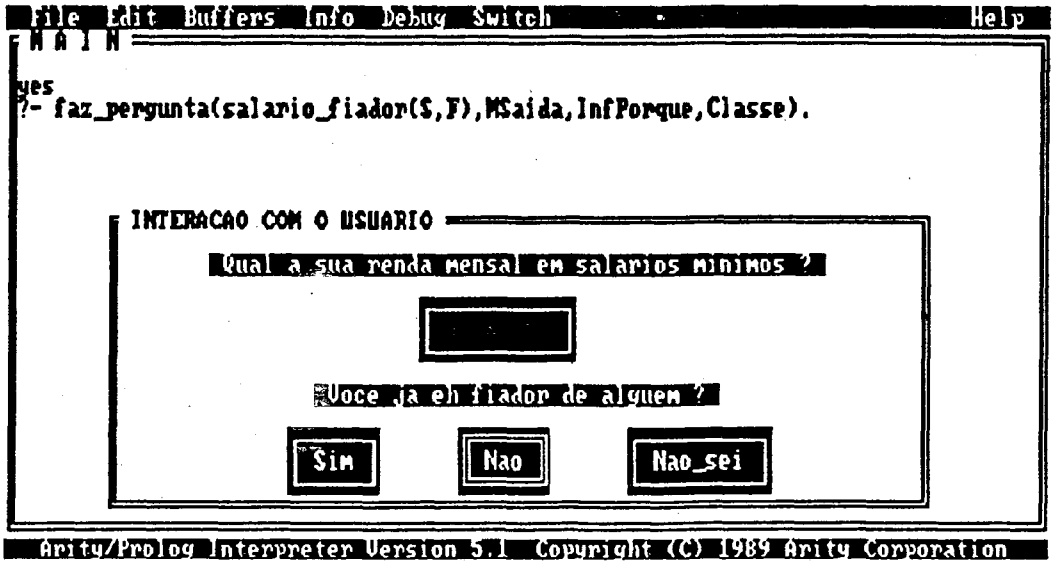
```
pergunta_livre(salario_fiador(Salario,Fiador),
               pg(perg3(Salario,Fiador)),
               [u,Salario,Fiador]).

perg3(Salario,Fiador):-
   dialog_run(j2,trataj2),
   busca_resposta(Salario,Fiador).

begin_dialog(j2,' INTERACAO COM O USUARIO ',(9,8),
             (22,70),(79,-79),112,popup).
ctrl(text,0,$ Qual a sua renda mensal em salarios minimos ? $,
```

```
      (1,7),112,47).
ctrl(efield,1,_,(3,23),(112,112),9,$$).
ctrl(text,0,$ Voce ja eh fiador de alguem ? $,(7,15),112,31).
ctrl(push,1,$Sim$,(9,13),(112,112),sim).
ctrl(push,1,$Nao$,(9,26),(112,112),nao).
ctrl(push,1,$Nao_sei$,(9,39),(112,112),ns).
end_dialog(j2).

trataj2(command(nobutton,ok),K):-
   send_control_msg(ef_set_text(Salario,Salario),2,K),
   verif_aceitavel(Salario,NSalario,K),
   assertz(salario(NSalario)),
   !,
   send_dialog_msg(def_dialog_fn,next_ctrl(2,4),K).
trataj2(command(_,sim),K):-
   !,
   assertz(fiador(sim)),
   exit_dbox(K).
trataj2(command(_,nao),K):-
   !,
   assertz(fiador(nao)),
   exit_dbox(K).
trataj2(command(_,ns),K):-
   !,
   assertz(fiador(nao_sei)),
   exit_dbox(K).
trataj2(Msg,K):-
   def_dialog_fn(Msg,K).

verif_aceitavel(Salario,Salario,K):-
   (pertence(Salario,[por_que,nao_sei]);
   integer(Salario)),!.
verif_aceitavel(Salario,NS,K):-
   send_control_msg(ef_set_text(Salario,$$),2,K),
   send_control_msg(focus(_,2),2,K),
   send_control_msg(ef_set_text(_,NS),2,K).

busca_resposta(Salario,Fiador):-
   salario(Salario),
   fiador(Fiador),
   abolish(salario/1),
   abolish(fiador/1).
```

O primeiro e o segundo argumento do predicado `pergunta_livre/3` são como explicados anteriormente. Neste exemplo, o segundo argumento contém somente o programa

perg3(Salario,Fiador), que deve escrever a pergunta e ler as respostas instanciando assim as variáveis da meta que está sendo perguntada — salario_fiador(Salario,Fiador) neste caso. Neste exemplo a pergunta é realizada com auxílio de uma caixa de diálogo. O terceiro argumento é uma lista onde o primeiro elemento indica o tipo de pergunta (u) e os outros são as mesmas variáveis que serão instanciadas com as respostas fornecidas pelo usuário.

Segue-se um exemplo correspondente a este caso.



após o usuário fornecer as respostas, o programa é bem sucedido com

```
MSaida = salario_fiador(8,nao)
Classe = sim
```

Para o caso correspondente a pergunta_livre/4 poder-se-ia ter:

- na BC

```
perguntavel(salario_fiador(Cliente,Salario,Fiador)).
```

- na Ba

```
pergunta_livre(salario_fiador(Cliente,Salario,Fiador),
               [Cliente],
               pg(perg4(Cliente,Salario,Fiador)),
               [u,Salario,Fiador]).
```

onde perg4(Cliente,Salario,Fiador) deve ser definido.

4.3.3 Tipo de Pergunta que Pode ser Feita mais de uma Vez

Suponha que seja necessário saber o tipo e a quantidade de imóveis que um fiador, não explicitamente identificado, possui. Neste caso poder-se-ia ter:

- na BC

```
perguntavel(imovel_quantidade(Tipo,Quantidade)).
```

- na Ba

```
pergunta(imovel_quantidade(Tipo,Quantidade),pg(perg5),
            [v,critica_tipo,critica_quantidade]).

perg5:-
  write('Que tipo de imovel voce possui ?'),nl,
  write('Qual a quantidade?'),
  nl.

critica_tipo(X):-
  pertence(X,[casa,apartamento,fazenda,lote,outros]),
  !.
critica_tipo(_):-
  write('Os valores validos para tipo de imovel sao:'),
  nl,
  write('casa, apartamento, lote, fazenda e outros.'),
  nl,
  nova_resposta,
  fail.

critica_quantidade(X):-
  integer(X),
  X > 0.
  !.
critica_quantidade(_):-
  write('A quantidade de imoveis deve ser maior que 0.'),
  nl,
  nova_resposta,
  fail.
```

Os argumentos de `pergunta/3` são como descritos no exemplo anterior exceto o do primeiro elemento da lista (terceiro argumento), que é agora o átomo `v` que identifica este tipo de pergunta. A pergunta será repetida até que o usuário responda `basta` ou `pare`.

Segue-se um exemplo de execução deste caso:

```
?-faz_pergunta(imovel_quantidade(Imovel,Quantidade),
                MSaida,Inf_PorQue,Classe).
Que tipo de imovel voce possui ?
Qual a quantidade ?
>casa.
>-3.
A quantidade de imoveis deve ser um numero maior que 0.
Responda novamente, por favor.
>3.
```

o programa é bem sucedido com

```
MSaida = imovel_quantidade(casa,3)
Classe = sim
```

no retrocesso, a pergunta é repetida

```
Que tipo de imovel voce possui ?
Qual a quantidade ?
>apartamento.
>5.
```

o programa é bem sucedido com

```
MSaida = imovel_quantidade(apartamento,5)
Classe = sim
```

novamente, no retrocesso, a pergunta é repetida

```
Que tipo de imovel voce possui e qual ?
Qual a quantidade ?
>pare.
```

e a execução termina.

No caso de um fiador explicitamente identificado poder-se-ia ter:

- na BC

```
perguntavel(imovel_quantidade(Cliente,Tipo,Quantidade)).
```

- na Ba

```
pergunta(imovel_quantidade(Cliente,Tipo,Quantidade),[Cliente],
         pg(perg6(Cliente)),
         [v,critica_tipo,critica_quantidade]).

perg6(Cliente):-
  write(Cliente),
  write(' que tipo de imovel voce possui ?'),nl,
  write('Qual a quantidade ?'),
  nl.
```

com as mesmas funções de crítica definidas anteriormente.

Os mesmos exemplos são considerados a seguir a fim de ilustrar o uso de `pergunta_livre`. Para o caso em que o predicado `pergunta_livre/3` é usado teria:

- na BC

```
perguntavel(imovel_quantidade(Tipo,Quantidade)).
```

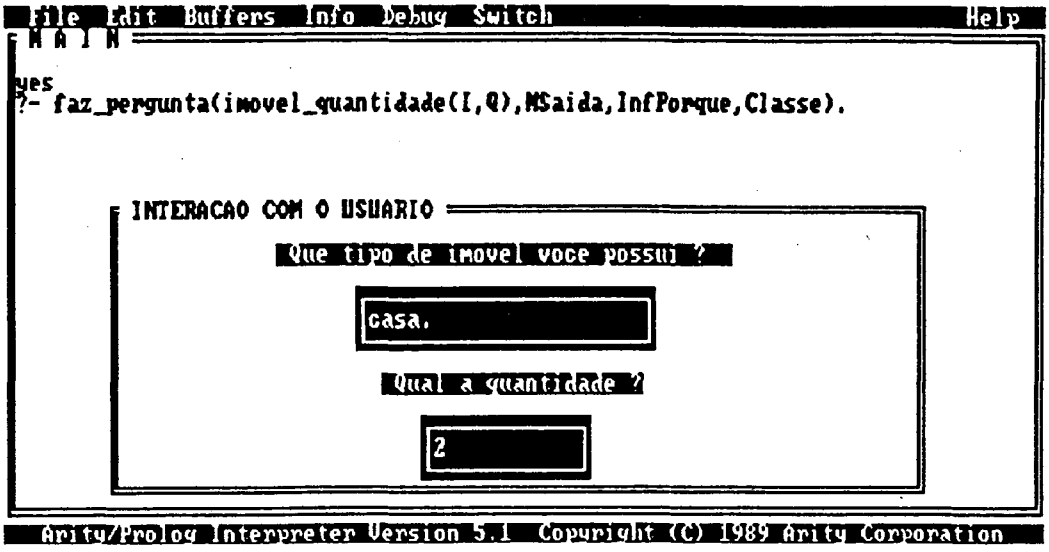
- na Ba

```
pergunta_livre(imovel_quantidade(Tipo,Quantidade),
               pg(perg5(Tipo,Quantidade)),
               [v,Tipo,Quantidade]).


perg5(Tipo,Quantidade):-
   dialog_run(j3,trataj3),
   buscaj3(Tipo,Quantidade).

begin_dialog(j3,' INTERACAO COM O USUARIO ',(9,8),(22,70),
             (79,-79),112,popup).

ctrl(text,0,$ Que tipo de imovel voce possui ? $,
     (1,12),112,35).
ctrl(efield,1,_,(3,18),(112,112),20,$$).
ctrl(text,0,$ Qual a quantidade ? $,(7,20),112,20).
ctrl(efield,1,_,(9,23),(112,112),10,$$).
end_dialog(j3).

trataj3(command(nobutton,ok),K):-
   send_control_msg(ef_set_text(Tipo,Tipo),2,K),
   verif_aceitavel_tipo(Tipo,NTipo,K),
```

```
    assertz(tipo(NTipo)),
    !,
    send_dialog_msg(def_dialog_fn,next_ctrl(2,4),K).

trataj3(command(nobutton,ok),K):-
    send_control_msg(ef_set_text(Quant,Quant),4,K),
    verif_aceitavel_quant(Quant,NQuant,K),
    assertz(quantidade(NQuant)),
    !,
    exit_dbox(K).

trataj3(Msg,K):-
    def_dialog_fn(Msg,K).

buscaj3(Tipo,Quantidade):-
    tipo(Tipo),
    quantidade(Quantidade),
    abolish(tipo/1),
    abolish(quantidade/1).
```

onde os programas
verif_aceitavel_tipo(Tipo,NTipo,K) e
verif_aceitavel_quant(Quant,NQuant,K)
devem ser definidos.

Os argumentos de pergunta_livre/3 são como descritos no exemplo de tipo de pergunta a ser realizada somente uma vez — subseção 4.3.2 — a única diferença é o primeiro elemento da lista (terceiro argumento) que agora é o átomo v que indica este tipo de pergunta. O resultado da execução de faz_pergunta/4 é semelhante ao mostrado na seção 4.3.3, somente que neste caso a pergunta é realizada com auxílio de uma caixa de diálogo, mostrada a seguir.



No caso de pergunta_livre/4 ser usado poder-se-ia ter:

- na BC

```
perguntavel(imovel_quantidade(Cliente,Tipo,Quantidade)).
```

- na Ba

```
pergunta_livre(imovel_quantidade(Cliente,Tipo,Quantidade),
               [Cliente],
               pg(perg6(Cliente,Tipo,Quantidade)),
               [v,Tipo,Quantidade]).

onde perg6(Cliente,Tipo,Quantidade) deve ser definido.
```

A implementação deste tipo de pergunta, que admite várias respostas, está baseada na solução apresentada em [Costa 85] e seu mérito é melhor apreciado quando o Módulo Coletor de Dados for parte integrante de um Núcleo de SE. Neste caso, quando o Motor de Inferência ativa o Módulo Coletor de Dados para fazer uma pergunta que admite várias respostas, o MCD retorna o controle para o MI após a primeira resposta do usuário. O MI é livre — dependendo de como está implementado — para continuar seu raciocínio com esta informação e até pode encontrar uma solução do problema que está analisando sem ter necessidade de outras respostas. A pergunta será repetida somente se o MI necessitar de mais informações para resolver o problema em questão.

Isto é realizado da seguinte forma: quando a pergunta é do tipo *várias*, o MCD fornece ao MI a resposta do usuário mas fica pendente uma outra ativação a si próprio. Portanto, se o MI não conseguir encontrar uma solução, no retrocesso ("backtracking"), a pergunta será repetida. A resposta basta ou pare termina a chamada recursiva dentro do MCD.

4.4 Descrição da Implementação

A seguir é apresentado o nível mais alto da implementação do Módulo Coletor de Dados. A implementação completa e detalhada deste módulo encontra-se em [Rodrigues 90a].

Quando for apropriado, os argumentos dos programas são descritos da seguinte forma.

[E]<arg_n>: n-ésimo argumento é de entrada

[S]<arg_n>: n-ésimo argumento é de saída

[A]<arg_n>: n-ésimo argumento é de entrada e de saída

O programa faz_pergunta, responsável pela ativação do Módulo Coletor de Dados, possui quatro argumentos

```
faz_pergunta(MEnt,MSaida,Regras,Classe)
```

onde

[E]<arg_1>: a meta a ser provada

[S]<arg_2>: quando o programa faz_pergunta é bem sucedido , este argumento contém a meta que foi provada.

[A]<arg_3>: uma lista que contém as metas anteriores a esta meta. Se o MCD está sendo usado como parte de um Núcleo de SE, a informação contida neste argumento deve ser passada ao Módulo de Explicação pelo MCD sempre que o usuário perguntar o por_que de uma pergunta

[S]<arg_4>: a classe da resposta dada pelo usuário. As respostas se classificam em

- sim se o usuário responder a todas as perguntas ou se a resposta for positiva
- nao para as respostas negativas
- nao_sei se o usuário responder nao_sei a algumas ou todas as perguntas

O programa faz_pergunta/4 está definido pelas seguintes quatro cláusulas, onde cada uma delas atende a um grupo de quatro perguntas definidas na seção 4.1.

A primeira cláusula do programa

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Perg,Tcritica),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].
```

verifica se está definido na Ba um predicado pergunta(MEnt,Perg,TCritica), para obter as informações necessárias para fazer a(s) pergunta(s) ao usuário e as funções de crítica para validar as respostas. Uma vez que existe uma cláusula pergunta/3, o programa faz_pergunta/4 ativa o programa desinstancie/2 da seguinte forma

```
desinstancie(MEnt,MInt)
```

cuja função é simplesmente tomar a meta de entrada (MEnt) e deixar todos os argumentos dessa meta livres. Por exemplo:

```
?-desinstancie(salario_fiador(S,nao),MSai).

S = _005D
MSai = salario_fiador(_01E9,_01DD) ->;

no
?-
```

Após a execução de desinstancie/2, o programa faz_pergunta/4 terá em MInt a meta de entrada com todos os seus argumentos livres. O programa pergunte/5 é ativado para fazer a interação com o usuário. Logo em seguida, usando a pré-definida "univ"(=..), o nome da relação de MEnt e a(s) resposta(s) do usuário, o programa constrói a meta de saída MSaida.

A segunda cláusula

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Lista,Perg,Tcritica),
   !,
   testa_instanciacao(MEnt,Lista,MSai,Tcritica),
   MEnt=..[Rel|LEnt],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].
```

é ativada somente se não existir um predicado pergunta/3 correspondente a esta MEnt. Ela verifica se existe alguma cláusula pergunta/4 na base Ba para obter as informações necessárias para fazer a(s) pergunta(s) ao usuário, as funções de crítica para as respostas e a lista das variáveis que devem permanecer instanciadas. O programa testa_instanciacao/4 verifica quais argumentos de entrada devem permanecer instanciados, que é o caso dos argumentos que estão presentes no segundo argumento de pergunta/4, e deixa os demais argumentos livres. Por exemplo:

```
?-testa_instanciacao(salario_fiador(pedro,S,nao),
                     [pedro],Msai,[u|_]).

MSai = salario_fiador(pedro,_01FE,_01D6) ->;

no
?-
```

Uma vez que testa_instanciacao/4 é bem sucedido, o programa faz_pergunta/4 ativa o programa pergunte/5 e em seguida constrói a meta de saída MSai.

A terceira claúsula

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Perg,Tresp),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].
```

é ativada somente se não estiverem definidos na base Ba os predicados `pergunta/3` ou `pergunta/4` correspondentes à meta de entrada. Ela verifica se existe o predicado `pergunta_livre/3` na base Ba, a fim de obter as informações necessárias para fazer a pergunta e ler as respostas fornecidas pelo usuário. Em seguida, o programa `desinstancie/2`, explicado anteriormente, é ativado. Finalmente, o programa `obter_resposta/6` é ativado para fazer a interação com o usuário.

A quarta claúsula do programa

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Lista,Perg,Tresp),
   testa_instanciacao(MEnt,Lista,MSai,Tresp),
   MEnt=..[Rel|_],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].
```

executada somente se nenhuma das anteriores tiver sido bem sucedida, procura na base Ba pelo predicado `pergunta_livre/4` e deixa livre todos os argumentos da meta de entrada, que não estiverem presentes no segundo argumento de `pergunta_livre/4`, usando o programa `testa_instanciação` explicado anteriormente. O programa `obter_resposta/6` é ativado para fazer a interação com o usuário e obter a classe da(s) resposta(s) — `sim`, `nao`, `nao_sei`.

4.5 Considerações Finais

Neste capítulo foram apresentados detalhes sobre a implementação do subsistema Módulo Coletor de Dados, que é parte integrante do ambiente desenvolvido para auxiliar a construção de Núcleos de Sistemas Especialistas com características definidas pelo projetista do Sistema.

A interação deste subsistema com o usuário admite várias categorias de perguntas que, em alguns casos, sobrepõem-se, como é o caso de respostas do tipo `s-n` que podem também ser definidas como de tipo `única`.

A fim de exemplificar, considere que é necessário confirmar se o voto de um cidadão é nulo. Isto pode ser realizado de duas formas. Na primeira, declarando

- na BC

```
perguntavel(voto_nulo).
```

- na Ba

```
pergunta(voto_nulo,pg(perg_voto_nulo),[s-n]).

perg_voto_nulo:-
  write('O voto e nulo ?'),nl.
```

Uma outra forma é declarando

- na BC

```
perguntavel(voto_nulo(X)).
```

- na Ba

```
pergunta(voto_nulo(X),pg(perg_voto_nulo),[u,critica_sn]).

perg_voto_nulo:-
  write('O voto e nulo ?'),nl.

critica_sn(X):-
  pertence(X,[sim,nao]),!.
critica_sn(X):-
  write('Os valores validos sao sim, nao.'),
  fail.
```

Ainda nestes casos, pode ser observado que não há conflito nos tipos de perguntas implementadas [Rodrigues 89].

Um outro aspecto que deve ser ressaltado na implementação deste subsistema é que não há restrição no número de variáveis que podem ser preenchidas com a(s) resposta(s) do usuário a uma pergunta específica.

Na implementação realizada, a entrada e a saída de dados pode ser programada usando um ambiente amigável, por exemplo, com janelas, menus, caixas de diálogo, etc. A idéia usada é a de deixar liberdade para o projetista das perguntas determinar o formato apropriado. Para isto, ele pode optar por definir na base Ba relações de nome `pergunta` ou `pergunta_livre` dependendo de como ele deseja implementar a interação com o usuário.

Na construção do Nível 1 — mostrada na figura 3.2 — o usuário decide quais os grupos de informações e tipos de perguntas que serão processadas pelo Núcleo Específico. Por

exemplo, se na Base de Conhecimento que será manipulada pelo Núcleo do Sistema Especialista específico as perguntas a serem realizadas não admitirem variáveis instanciadas — `pergunta/4` e `pergunta_livre/4` —, as cláusulas 2 e 4 do programa `faz_pergunta/4` não são necessárias ao MCD. Dessa forma, no Nível 1 tem-se um subconjunto do Módulo Coletor de Dados implementado.

Na construção do Nível 2, o projetista do NSE pode trocar a rotina de leitura do Módulo Coletor de Dados.

Capítulo 5

O Motor de Inferência

5.1 Considerações Iniciais

Como foi visto na seção 3.3, o método que será usado para construir o Motor de Inferência do Núcleo do SE é o que usa um meta-interpretador para manipular o conhecimento do Sistema Especialista. Neste caso, a Base de Conhecimento permanece intacta e novas cláusulas podem ser adicionadas ao meta-interpretador tornando-o mais abrangente.

Neste capítulo é apresentada uma descrição do subsistema denominado Motor de Inferência, que tem embutido o tipo de raciocínio do sistema.

5.2 O Meta-interpretador Implementado

O meta-interpretador implementado, `prove/3`, usa raciocínio Backward Chaining, isto é, o programa `prove` parte de uma conclusão e tenta encontrar evidências que suportam esta conclusão. Como a Base de Conhecimento está internamente representada por cláusulas Prolog, o programa `prove/3` tenta provar uma meta (conclusão) provando suas submetas.

Este programa armazena conclusões intermediárias a fim de não repetir avaliações de hipóteses já provadas. Este método é, computacionalmente, muito mais eficiente que aquele que não lembra provas bem sucedidas. Porém, deve ser ressaltado que no caso de reavaliar hipóteses, sem gravar as soluções na Base de Dados, há garantia de que a conclusão final é consistente, porque ela segue-se logicamente dos fatos e das regras existentes na Base de Conhecimento

O meta-interpretador `prove/3` é constituído de várias cláusulas — mostradas na figura 5.1 — sendo que uma única cláusula ou um grupo de cláusulas trata de algum tipo específico de conhecimento, como visto na seção 3.4.

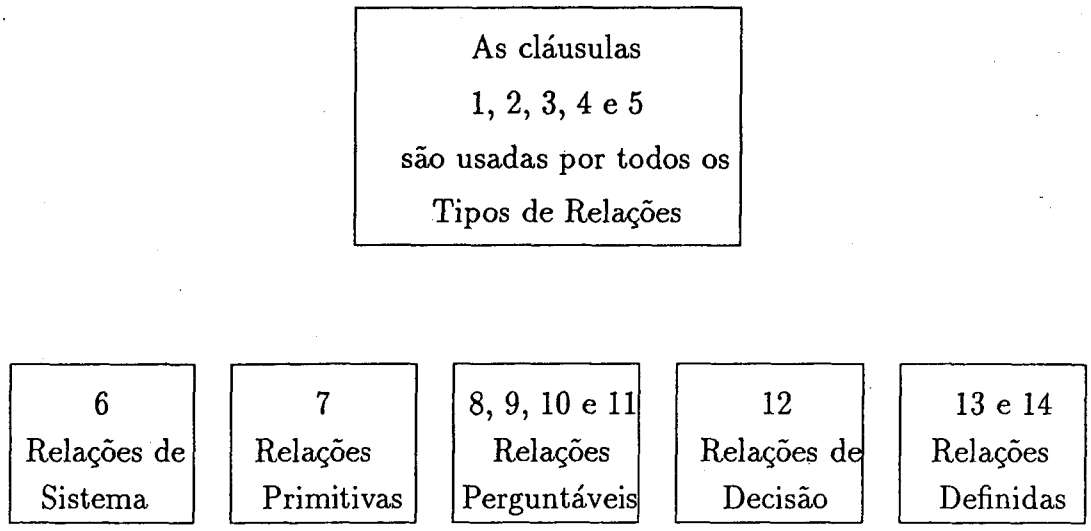


Figura 5.1: Diagrama que Representa as Cláusulas do Meta-interpretador prove/3

5.3 Descrição da Implementação

A seguir é apresentada em detalhe a implementação do meta-interpretador que constitui o Motor de Inferência.

O programa `prove` tem três argumentos

```
prove(Meta,Regras,Prova).
```

onde

`[E]`<arg_1>: é a meta a ser provada. Esta meta pode ser de qualquer um dos cinco tipos descritos anteriormente, ou seja, de decisão, perguntável, etc, bem como negação (por falha) ou uma conjunção ou disjunção de metas.

`[A]`<arg_2>: contém as regras que foram usadas até o momento, na ordem em que foram usadas. A informação contida neste argumento é passada ao Motor de Explicação pelo Módulo Coletor de Dados sempre que o usuário perguntar o porque de uma pergunta.

`[S]`<arg_3>: quando o programa `prove` é bem sucedido, e se este argumento estiver instanciado, ele contém a árvore de prova com todas as inferências realizadas durante o processo.

O programa `prove/3` completo está definido pelas seguintes 14 cláusulas

A primeira cláusula do programa

```
prove(true,Ef,Ef):-!.
```

é simplesmente o critério de parada.

A segunda cláusula

```
prove(Meta,Ef,Ef):-
   ja_provado(Meta).
```

verifica a existência de uma prova anterior bem sucedida de `Meta` que esteja armazenada na BD a fim de não repetir uma prova que já foi feita.

O programa `ja_provado/1` é responsável pelo acesso à Base de Dados e está intimamente ligado ao programa `lembre_provado/1` que é responsável pela gravação na BD das provas bem sucedidas das metas. Estes programas são explicados mais adiante e podem ser alterados sempre que verifiquem que a conjunção `lembre_provado(Meta), ja_provado(Meta)` é verdadeira.

Deve ser salientado que a eficiência da execução do programa `prove` pode ser melhorada consideravelmente se for usado o corte (!) nesta segunda cláusula. O problema é que, neste caso, dependendo da BC, o MI pode não encontrar todas as possíveis soluções da hipótese que está sendo verificada.

Se o corte for usado nesta cláusula, não é necessário verificar por

```
not(ja_provado(Meta))
```

nas cláusulas 5, 12, 13 e 14 do programa `prove`.

As cláusulas 3 e 4

```
prove((Meta1,Meta2),Ep,Ef):-
   prove(Meta1,Ep,Ef1),
   prove(Meta2,Ef1,Ef).

prove((Meta1;Meta2),Ep,Ef):-
   prove(Meta1,Ep,Ef);
   prove(Meta2,Ep,Ef).
```

são responsáveis, respectivamente, pelas provas de conjunção e disjunção de metas.

A negação de metas é provada pela cláusula 5

```
prove(not Meta,Ep,Ef):-
```

```
    not(ja_provado(not(Meta))),
    !,
    not prove(Meta,Ep,Ef),
    lembre_provado(not(Meta)).
```

5.3.1 Manipulação das Relações de Sistema

A cláusula 6

```
prove(Meta,Ef,Ef):-
    sistema(Meta),
    Meta\=not(X),!,
    Meta.
```

prova as relações pré-definidas do Prolog, tais como predicados aritméticos, comparação e classificação de termos, etc. O programa `sistema(Meta)` é responsável pelo reconhecimento deste tipo de relações. Além disso, a prova refere-se a uma prova positiva da meta. Provas negativas (negação por falha) são manipuladas pela cláusula 5.

5.3.2 Manipulação das Relações Primitivas

A cláusula 7

```
prove(Meta,Ef,Ef):-
    primitiva(Meta),
    !,
    Meta.
```

é semelhante à anterior, mas prova metas definidas como primitivas na BC, isto é, programas Prolog. Como a declaração primitiva refere-se à cabeça de um programa Prolog, a meta é positiva e não é necessário, como no caso anterior, verificar pela não negação da meta.

5.3.3 Manipulação das Relações Perguntáveis

No caso das relações perguntáveis, o procedimento geral é como mostrado na figura 5.2, sendo que o MI é capaz de reconhecer também respostas desconhecidas. Para o MI as respostas *não_sei* pertencem a uma classe diferente das respostas negativas.

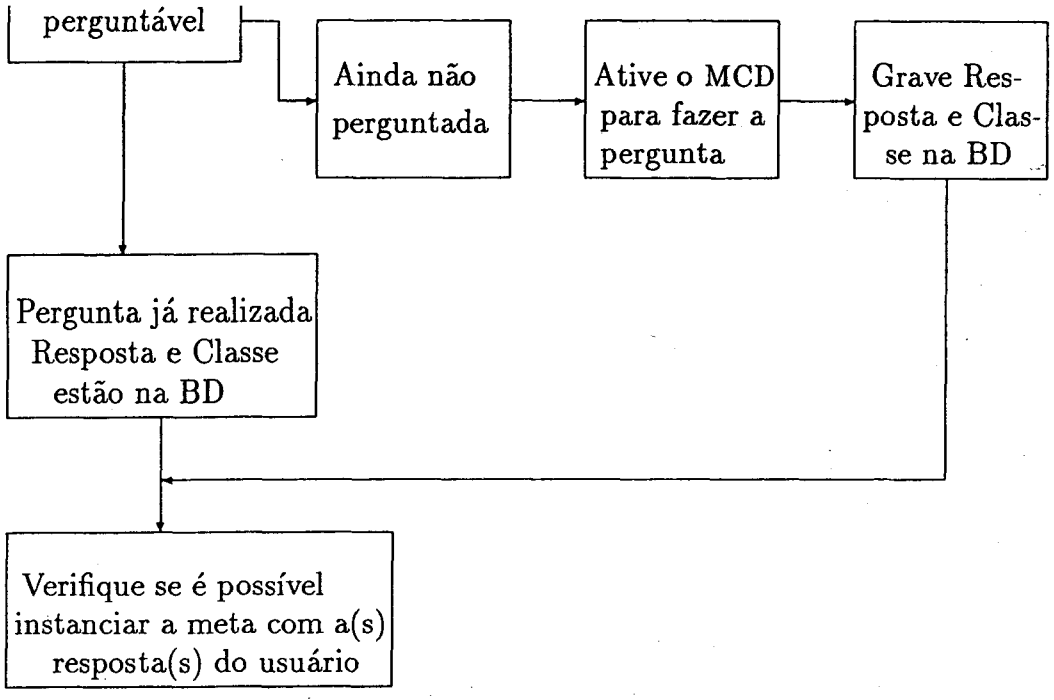


Figura 5.2: Processamento das Relações Perguntáveis

As relações perguntáveis da BC são manipuladas pelas cláusulas 8 a 11 do programa `prove/3`. Primeiramente é verificado se a pergunta já foi feita (cláusulas 8 e 9) e somente em caso negativo (cláusulas 10 e 11), o MCD é ativado, através do programa `faz_pegunta/4`, para interagir com o usuário.

A cláusula 8

```
prove(desconhecido(Meta),Ef,Ef):-
   perguntavel(Meta),
   resposta(nao_sei,Meta),
   !.
```

se encarrega de manipular as relações perguntáveis que aparecem em alguma regra da BC como `desconhecido(Meta)`. Isto é, `desconhecido(Meta)` é válida se a(s) resposta(s) do usuário à pergunta correspondente a esta Meta é(são) `nao_sei`.

Primeiramente é verificado que a relação é perguntável e logo após é verificado se a pergunta já foi realizada e a resposta dada foi nao_sei. Se este for o caso, a prova é bem sucedida e o corte não permite tentar outra prova.

A cláusula 9

```
prove(Meta,Ef,Ef):-
   perguntavel(Meta),
   resposta(sim,Meta),
   !.
```

é semelhante a anterior mas para perguntas que já foram realizadas e o usuário forneceu todas as respostas correspondentes a esta meta.

As cláusulas 10 e 11 do programa prove são as que ativam o MCD para fazer as perguntas ao usuário.

A cláusula 10

```
prove(desconhecido(Meta),Ef,Ef):-
   perguntavel(Meta),
   nao_foi_perguntada(Meta),
   !,
   faz_pergunta(Meta,MetaSaida,Ef,Tipo),
   lembre_resposta(MetaSaida,Tipo),
   Tipo=nao_sei.
```

processa as relações perguntáveis que aparecem em alguma regra da BC como `desconhecido(Meta)` e que ainda não foram perguntadas, ou seja, a relação `nao_foi_perguntada(Meta)` é válida. O corte é usado somente para incrementar a eficiência do programa, ou seja, é um corte "verde". Se a pergunta ainda não foi feita, o MCD é ativado através de `faz_pergunta/4`. Este programa é sempre bem sucedido.

Então, logo após fazer as perguntas necessárias, a resposta é lembrada na BD por

```
lembre_resposta(MetaSaida,Tipo)
```

e, finalmente, esta prova é bem sucedida se `Tipo` é igual a `nao_sei`, já que estamos tentando a prova de uma `Meta` perguntável do tipo `desconhecido(Meta)`, ou seja, é válida se o usuário responder `nao_sei`.

A cláusula 11

```
prove(Meta,Ef,Ef):-
   perguntavel(Meta),
   nao_foi_perguntada(Meta),
   !,
   faz_pergunta(Meta,MetaSaida,Ef,Tipo),
   lembre_resposta(MetaSaida,Tipo),
   Tipo=sim,
   Meta=MetaSaida.
```

é semelhante à 10 com a diferença que, após gravar na BD e verificar que `Tipo` é igual a `sim`, a prova é bem sucedida somente se a `Meta` original e a `MetaSaida` preenchida com as respostas do usuário unificam.

O motivo desta restrição é que a meta a ser provada pode ter algumas variáveis instanciadas, e esta é uma forma de verificar se as respostas dadas pelo usuário são as requeridas para a prova da meta em questão.

É importante observar que todas as respostas dadas são gravadas na BD e se a resposta for negativa estas duas cláusulas falham. Entretanto, devido a forma de como implementamos a negação (cláusula 5) a prova de uma negação com resposta negativa será bem sucedida.

Um outro ponto a ser ressaltado é referente ao programa `faz_pergunta/4` . Este programa possui somente uma solução se a meta perguntável admitir somente perguntas com tipo de respostas `sim-não` e `únicas`. No caso em que a meta perguntável admitir respostas do tipo `várias`, o programa `faz_pergunta` tem outra solução que será tentada no retrocesso até o usuário responder `basta`, como foi mostrado na seção 4.3.3.

Entretanto, deve ser ressaltado que esta outra solução fica pendente, no sentido de ser tentada somente quando outra solução do programa `prove/3` for necessária. Ou seja, o MCD faz as perguntas necessárias unicamente quando o MI precisar desta informação.

Infelizmente, há alguns sistemas que perguntam toda a informação no princípio da prova, sem levar em conta que dependendo das respostas dadas a algumas perguntas, outras perguntas passam a não ser relevantes, no sentido de que as regras de conhecimento que usam esta informação não serão necessariamente ativadas durante a prova.

5.3.4 Manipulação das Relações de Decisão

Há uma única cláusula que manipula relações de decisão, ou seja, as regras de conhecimento que possuem fatores de certeza (fc). Sua implementação foi trabalhosa do ponto de vista de eficiência do programa, ainda que esta versão não seja tão elegante como outras versões realizadas anteriormente.

Nesta última versão, o tempo de execução para este tipo de relação foi melhorado de forma considerável. Isto foi conseguido não deixando provas pendentes, bem como gravando todas as soluções na Base de Dados a fim de não repetir provas quando a Base de Conhecimento processada possui vários níveis de relações de decisão.

Antes de explicar o seu funcionamento, é preciso salientar a diferença que existe entre uma meta que inclui fator de certeza e uma que não o inclui.

Para provar uma meta que não inclui `fc`, o programa `prove` tenta encontrar um caminho bem sucedido para a prova e, se for o caso, fornece a solução para o usuário. Se o usuário desejar, novas soluções são processadas no retrocesso. No entanto, para provar uma meta que possui `fc` não basta somete encontrar um caminho para provar esta meta, mas todos

os possíveis caminhos da prova desta meta devem ser encontrados a fim de calcular e combinar os novos fatores de certeza. Após encontrar todas as soluções, estas devem ser ordenadas tal que a primeira solução usada é a que possui maior fator de certeza e, se for necessário, no retrocesso a segunda maior é usada e assim por diante até não haver mais evidências provadas.

A cláusula 12 do programa prove

```
prove(Meta,Ep,Ef):-
   Meta=..[RelMeta,fc,X,E,R],
   MetaLivre=..[RelMeta,fc,XX,EE,RR],
   not(ja_provado(MetaLivre)),
   not(ja_semsolucao(MetaLivre)),
   crie([RelMeta,fc,X,E,R],MetaAux),
   base_conhecimento(MetaAux,CorpoMeta),
   !,
   findall([fc,X1,E1,R1],
         (Meta1=..[RelMeta,fc,X2,E1,R1],
         base_conhecimento(Meta1,CorpoMeta1),
         prove(CorpoMeta1,[rg(Meta1,CorpoMeta1)|Ep],Ef),
         calcule_fc(X1,X2,E1,0.2)),
         Lista),
   sevazia_lembre_semsolucao_fail(Lista,MetaLivre),
   combine_evidencias(Lista,Lista1),
   ordene(Lista1,ListaOrd),
   lembre_provado_todos(RelMeta,ListaOrd),!,
   ja_provado(Meta).
```

verifica se `Meta` possui `fc`,bem como verifica se outra `Meta` com o mesmo nome (`MetaLivre`) não foi provada ou se já é sabido que não há solução para ela. Após verificar estas condições, e por problemas de instanciação de variáveis, uma `Meta` semelhante a ela (`MetaAux`) é criada a fim de verificar a existência de uma meta deste tipo na BC.

Logo após, através da pré-definida `findall`, todas as possíveis soluções para esta meta são encontradas, calculando o `fc` e coletando na lista `Lista` todas as soluções da forma `[fc,X1,E1,R1]` que satisfazem a conjunção do corpo do findall. O programa que calcula o `fc` é simplesmente

```
calcule_fc(X1,X2,E1,Min):-
   X1 is X2 * E1,
   X1 > Min.
```

ou seja, o novo `fc` é o produto do `fc` acumulado (`X2`) pelo `fc` da regra dado pelo especialista (`E1`). Além disso, `fc` menores ou iguais a `Min` (0.2 no programa), não são levados em conta pois eles contribuem muito pouco no cálculo do `fc` final.

Após a execução do `findall`, se não houver nenhuma solução, `Lista` é a lista vazia. Se for o caso, o programa

```
sevazia_lembre_semsolucao_fail(Lista,MetaLivre)
```

lembra o fato na BD e falha. Caso contrário, após a execução do `findall`, regras com os mesmos atributos devem ser combinadas para calcular o `fc` final de cada regra. Isto é realizado por

```
combine_evidencias(Lista,Lista1).
```

Há vários métodos para realizar este cálculo e este programa, que será explicado mais adiante, pode ser facilmente redefinido para atender a outros métodos. Após a execução deste programa, `Lista1` possui elementos diferentes, cada um relacionado com a prova da meta. A seguir

```
ordene(Lista1,ListaOrd)
```

ordena os elementos em ordem decrescente do fator de certeza calculado.

Finalmente, as provas são lembradas na BD também em ordem decrescente do `fc`, pelo programa

```
lembre_provado_todos(RelMeta,ListaOrd)
```

e elas são acessadas através de

```
ja_provado(meta)
```

Deve ser observado que se a meta a ser provada estiver instanciada, ela será bem sucedida se estiver gravada na BD. Caso contrário, se a meta estiver desinstanciada, primeiro será usada a meta gravada na BD com o maior `fc` para depois acessar, no retrocesso, a meta com o segundo maior `fc` e assim por diante até todas estas metas, se for o caso, serem usadas na prova.

O programa para combinar as evidências possui os seguintes argumentos:

[E] `<arg_1>`: uma lista cujos elementos são listas do tipo `[fc,C,E,Atrib]` onde o segundo elemento `C` representa o coeficiente de certeza calculado, o terceiro elemento `E` é o fator de certeza da regra, dado pelo especialista, e o quarto elemento `Atrib` representa o atributo que está sendo provado.

[S] `<arg_2>`: uma lista com elementos do mesmo tipo que a lista de entrada, onde os fatores de certeza dos elementos que possuem o mesmo valor de `Atrib` são combinados de acordo com o método implementado pelo programa `comb/6`

O programa completo é:

```
combine_evidencias([],[]):-!.

combine_evidencias([X|L],[Y|S]):-
   combine_evidencias1(X,L,Y,LM),
   combine_evidencias(LM,S).

combine_evidencias1(CertVal,CertVals,ComCertVal,
                    RestCertVals):-
   combine_aux(CertVals,CertVal,[],ComCertVal,RestCertVals).

combine_aux([],XM,LM,XM,LM):-!.

combine_aux(L,XP,LP,XM,LM):-
   comb(L,XP,LP,ProxL,ProxXP,ProxLP),
   combine_aux(ProxL,ProxXP,ProxLP,XM,LM).

comb([[fc,CFX,Z1,V]|L],[fc,CFY,Z2,V],
     LP,L,[fc,CF,Z1,V],LP):-
   !,
   CF is CFX + CFY - CFX * CFY.

comb([C|L],CP,LP,L,CP,[C|LP]).
```

Como já foi dito, o programa comb/6 é responsável pela implementação do método usado para combinar as evidências. Neste trabalho combinamos as evidências usando o método padrão e, para implementar outro método, somente o programa comb/6 deve ser redefinido.

Devido a importância deste tipo de relação, será mostrado um exemplo muito simples de execução desta cláusula na prova da relação de decisão a(X,Y,Z,W) definida pelas seguintes regras de conhecimento, que fazem parte da Base de Conhecimento usada na seção 8.2, com o objetivo de mostrar a execução do sistema completo.

```
a(fc,C,0.5,cd1):-
   c,
   b(fc,C,E,bb).
a(fc,C,0.5,cd2):-
   z(aa,C2),
   b(fc,C1,E,cc),
   min(C,[C1,C2]).

b(fc,C,1,bb):-
   e,
```

```
    c(fc,C,E,c4).
b(fc,C,1,bb):-
    d,
    c(fc,C,E,c5).
b(fc,C,1,cc):-
    c(fc,C,E,c6).

c(fc,C,0.5,c4):-
    c(c4,C).
c(fc,C,1,c4):-
    d(d4,C).
c(fc,C,1,c5):-
    e(e5,C).
c(fc,C,1,c6):-
    f(f6,C).
c(fc,C,0.9,c6):-
    g(g6,C).
```

onde

- c/0, e/0, d/0, c/2, d/2, z/2, f/2 e g/2 são relações perguntáveis;
- min/2 é uma relação primitiva

Considerando que o MI é ativado com

```
?- prove(a(X,Y,Z,W),[],Ef).
```

a cláusula 12 do programa prove será ativada, estabelecendo-se o seguinte diálogo com o usuário:

```
E verdade c ?
> sim.
E verdade e ?
> sim.
Qual o valor de c (c1, c2, c3, c4, c5 ou c6) ?
E qual o fator de certeza ?
> c4.
> 0.9.
Qual o valor de c (c1, c2, c3, c4, c5 ou c6) ?
E qual o fator de certeza ?
> pare.
Qual o valor de d (d1,d2,d3,d4,d5 ou d6) ?
```

```
E qual o fator de certeza ?
> d4.
> 0.7.
Qual o valor (e1, e2, e3, e4 ou e5) ?
E qual o fator de certeza ?
> e5.
> 0.8.
Qual o valor de f (f1,f2,f3,f4,f5 ou f6) ?
E qual o fator de certeza ?
> f6.
> 0.5.
Qual o valor de g (g1,g2,g3,g4,g5 ou g6) ?
E qual o fator de certeza ?
> g6.
> 0.8.
E verdade d ?
> sim.
Qual o valor de z (aa, bb, cc) ?
E qual o fator de certeza ?
> aa.
> 0.9.
```

o programa prove/3 é bem sucedido com

```
C = 0.86
E = _0080
X = cd1
Ef = _0070
```

e, no retrocesso, fornece a segunda e, neste caso, a última solução

```
C = 0.43
E = _0080
X = cd2
Ef = _0070
```

As regras de conhecimento são várias, com diferentes corpos, e como foi visto na prova das relações de decisão, todos os possíveis caminhos da prova são encontrados a fim de calcular e combinar os novos fatores de certeza — a figura 5.3 representa em forma de árvore as regras mostradas acima — com o objetivo de facilitar a visualização dos caminhos possíveis.

Após a execução da prova da relação de decisão **a/4**, a Base de Dados contém as seguintes informações:

```
provado(c(fc,0.86,A,c6)).
provado(c(fc,0.835,A,c4)).
provado(c(fc,0.8,A,c5)).
provado(b(fc,0.967,A,bb)).
provado(b(fc,0.86,A,cc)).
provado(a(fc,0.86,A,cd1)).
provado(a(fc,0.43,A,cd2)).

resp(c,sim).
resp(e,sim).
resp(c(c4,0.9),sim).
resp(d(d4,0.7),sim).
resp(e(e5,0.8),sim).
resp(f(f6,0.5),sim).
resp(g(g6,0.8),sim).
resp(d,sim).
resp(z(aa,0.9),sim).
```

Observe que como a relação `a(X,N,M,K)` é uma relação de decisão o programa `prove/3` percorre os 5 caminhos possíveis — figura 5.3 — e combina os fatores de certeza obtidos fornecendo como primeira solução `a(fc,0.86,A,cd1)` que possui maior fator de certeza e, no retrocesso, fornece a próxima solução `a(fc,0.43,A,cd2)` com fator de certeza menor.

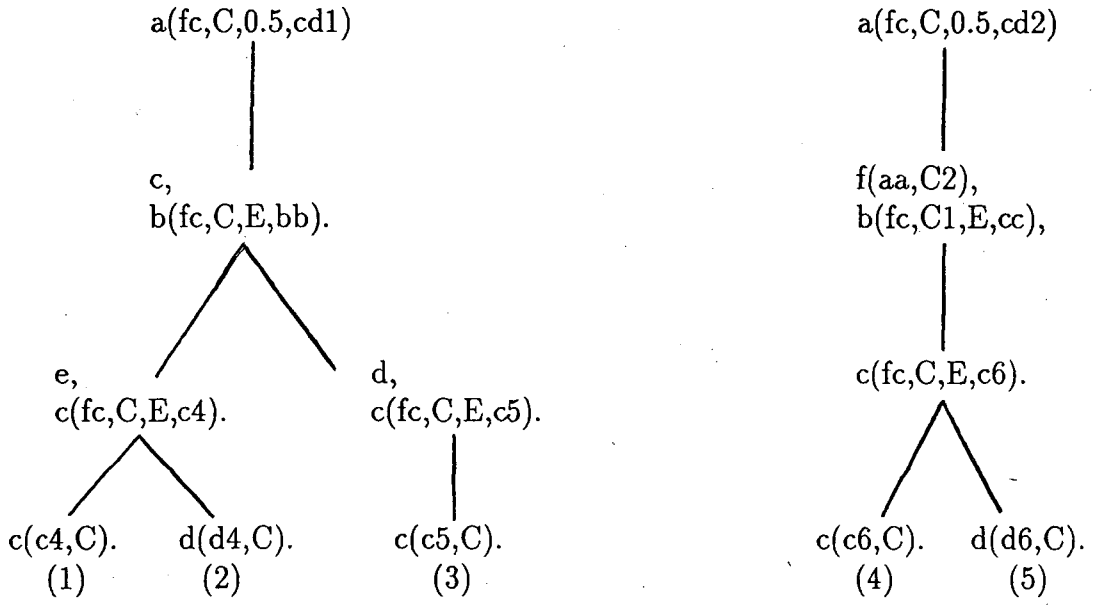


Figura 5.3: Diagrama que Representa um Exemplo das Relações de Decisão

5.3.5 Manipulação das Relações Definidas

Como visto na seção 3.4.1, estas relações são definidas na seguinte forma:

```
<pref>(verifica(N),<listaexp>):-
        atomo or atomo1 ...  or atomon.
```

com N inteiro ou elas são simplesmente uma cláusula Prolog.

A cláusula 13

```
prove(Meta,Ep,Ef):-
   base_conhecimento(Meta,CorpoMeta),
   not(ja_provado(Meta)),
   raciocinio_deterministico(Meta,N),
   prove(N,CorpoMeta,[rg(Meta,CorpoMeta)|Ep],Ef),
   lembre_provado(Meta).
```

prova as relações da BC que têm o funtor `verifique(N)` como primeiro argumento. `N` é um número inteiro que indica o número necessário de premissas verdadeiras na cauda desta regra de conhecimento para ela ser válida.

Após verificar que `Meta` não foi provada e que ela está na BC, o programa

```
raciocionio_deterministico(Meta,N)
```

é verdadeiro se `Meta` tiver como primeiro argumento o funtor `verifique(N)`. O programa `prove/4` é responsável por provar a validade de N premissas da cauda da regra, carregando a informação da regra que está sendo provada. O programa `prove/4`, definido em função do programa `prove/3`, é:

```
prove(0,_,Ef,Ef):-!.

prove(N,(Meta1 or Meta2),Ep,Ef):-
   prove(Meta1,Ep,Efi),
   N1 is N - 1,
   !,
   prove(N1,Meta2,Efi,Ef).

prove(N,(_ or Meta2),Ep,Ef):-
   !,
   prove(N,Meta2,Ep,Ef).

prove(1,Meta1,Ep,Ef):-
   prove(Meta1,Ep,Ef).
```

A cláusula 14

```
prove(Meta,Ep,Ef):-
    not(ja_provado(Meta)),
    base_conhecimento(Meta,Corpo),
    raciocinio_deterministico(Meta),
    prove(Corpo,[rg(Meta,Corpo)|Ep],Ef),
    lembre_provado(Meta).
```

prova as regras determinísticas de conhecimento, isto é, as regras que não possuem fator de certeza (`fc`) mas que carregam informação a ser lembrada na história da prova. Após verificar que a Meta não foi provada, a BC é acessada através da execução do programa `base_conhecimento(Meta,Corpo)`, que é bem sucedido se encontrar esta Meta a ser provada na BC, unificando o seu segundo argumento com o corpo da Meta. Todas as regras da BC que não são do sistema, primitivas, ou perguntáveis, são acessadas através deste programa, que pode ser facilmente redefinido.

As regras determinísticas são reconhecidas pelo programa `raciocinio_deterministico/1`, definido por:

```
raciocinio_deterministico(Meta):-
    functor(Meta,_,0),
    !.

raciocinio
_deterministico(Meta):-
    Meta=..[RelMeta,Arg1|Args],
    deterministico_arg1(Arg1),
    !.
```

Sabe-se que a Meta a ser provada pela cláusula 8 do programa prove/3, não foi provada pelas cláusulas anteriores. Isto ocorre devido aos cortes (!) usados nas cláusulas 5, 6 e 7, podendo-se concluir assim, que a Meta não é uma meta do sistema, primitiva, nem negativa, respectivamente. Portanto, a meta será uma regra de conhecimento determinístico se ela não possuir argumentos (aridade 0) ou se o primeiro argumento não for um dos átomos `fc` ou `verifica(_)`.

Então, após encontrar na BC a meta e o corpo da meta a ser provada e verificar que é uma regra de raciocínio determinístico, o programa `prove` é chamado recursivamente para provar o corpo da regra, lembrando no segundo argumento a regra que está sendo provada, com uma estrutura da forma `rg(Meta,Corpo)`. Quando uma pergunta é feita ao usuário e ele responder `por_que`, a informação contida neste segundo argumento é passada para o Módulo de Explicação a fim dele explicar ao usuário o motivo da pergunta. Finalmente,

se a prova é bem sucedida, a prova desta meta é lembrada na BD para não repetir provas já realizadas.

Observe que esta cláusula é não determinística, portanto, no retrocesso realizará todas as possíveis provas da meta que está sendo processada.

Maiores detalhes da implementação e uso do Motor de Inferência encontram-se em [Monard 89b].

5.4 Considerações Finais

A implementação do Motor de Inferência com raciocínio BCh mostrada, pode ser facilmente adaptada para resolver problemas específicos, com o objetivo de melhorar o tempo de execução do sistema. Por exemplo, se a Base de Conhecimento manipulada pelo Motor de Inferência não possuir relações de decisão, podemos suprimir a cláusula 12 do MI, bem como é possível melhorar o tempo de execução se estamos somente interessados em uma única prova da hipótese considerada, incluindo cortes (!) apropriados no meta-interpretador.

Porém, neste nível, é necessário um conhecimento não superficial da implementação para alterar o MI e montar um código dedicado à uma aplicação específica.

Observe que na construção do Nível 1 do núcleo específico — mostrado na figura 3.2 —, o usuário especifica os grupos de cláusulas que podem ser retiradas do Motor de Inferência. Por exemplo, se o usuário deseja construir um MI para manipular uma Base de Conhecimento que não possui relações primitivas e relações definidas, as cláusulas 7, 13 e 14 devem ser retiradas conforme mostra a figura 5.1.

Na construção do Nível 2 é possível realizar diversas alterações referentes a acesso e gravação na Base de Dados, manipulação de raciocínio incerto, etc. Por exemplo, o modelo que utilizamos para manipular relações de decisão é semelhante ao de MYCIN [Shortliffe 76] e este código pode ser alterado para implementar um outro método.

Capítulo 6

O Módulo de Explicação

6.1 Considerações Iniciais

Usualmente, os especialistas humanos podem explicar *porque* eles fazem uma pergunta, *como* eles chegam às suas conclusões, *o que aconteceria se* fosse tomada alguma atitude diferente, etc. De maneira análoga, os Sistemas Especialistas devem ser capazes de fornecer explicações plausíveis sobre o seu comportamento e suas decisões. Esta habilidade de explicação é necessária para que o usuário possa acompanhar o raciocínio do sistema, bem como para o projetista detectar possíveis erros nas regras da Base de Conhecimento, o que torna o sistema bem mais confiável.

Este capítulo descreve o Módulo de Explicação, que é o responsável pelas explicações solicitadas ao Sistema Especialista no Ambiente implementado, e os tipos de explicações nele implementadas.

6.2 Tipos de Explicações Implementadas

O Módulo de Explicação implementado — figura 6.1 — possui as seguintes facilidades de explicação:

- *por que* faz certas perguntas;
- *como* chegou a certas conclusões;
- *O que acontece se (what_if)* for trocada alguma informação já fornecida ao sistema.

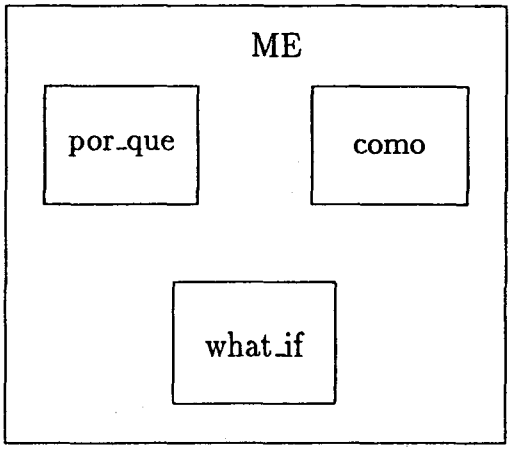


Figura 6.1: Tipos de Explicações Implementadas no Módulo de Explicação

6.2.1 A Explicação *por_que*

Para ser capaz de explicar o `por_que` de uma pergunta, o Módulo de Explicação deve ter acesso ao contexto em que foi feita tal pergunta. Um modo de fazer isto é passar a descrição do contexto como um argumento extra para cada regra que está sendo provada, como é feito pelo Motor de Inferência apresentado na seção 5.3 (veja $<\mathtt{arg}_2>$ das cláusulas 12, 13 e 14 do programa `prove/3`).

Este argumento extra, construído pelo Motor de Inferência, é passado para o Módulo Coletor de Dados cada vez que ele for solicitado para fazer alguma pergunta ao usuário. Se o usuário desejar saber o `porque` da pergunta, o MCD passa este argumento ao ME que se encarrega de interpretar e mostrar esta informação ao usuário de uma forma apropriada.

Além disso, a implementação realizada para o `por_que` é tal que, se logo após o ME explicar o `por_que` da pergunta e o MCD repetir a mesma pergunta, então o usuário pode responder `por_que` novamente à pergunta. Neste caso, o ME justifica com um motivo mais distante da pergunta inicial, até o momento em que não existem mais regras neste argumento extra para serem usadas na justificação do `por_que`. Observe que isto é equivalente a fornecer passo a passo, e em ordem inversa, a explicação de `como` está sendo realizada a prova.

O programa `explica_porque`, responsável pela ativação da explicação *por_que* no Módulo de explicação, possui um argumento:

```
explica_porque(Regra)
```

onde

`[E]`$<\mathtt{arg}_1>$: uma estrutura onde o funtor é o átomo `rg` que possui dois argumentos. O

primeiro argumento é a cabeça da regra e o segundo é a cauda da regra de conhecimento, que será usada para mostrar ao usuário por_que a pergunta foi realizada.

Ele consiste de duas cláusulas.

A primeira cláusula

```
explica_porque(rg(A,B)):-
  testa(A,N),
  !,
  mostra_tela(N,A,B).
```

verifica, através do programa `testa/2`, se a regra usada é uma relação definida que têm o funtor `verifica(N)` como primeiro argumento. Neste caso, o programa `mostra_tela/3` é ativado para interpretar e mostrar a informação contida na regra, ao usuário.

A segunda cláusula

```
explica_porque(rg(Meta,CorpoMeta)):-
  mostra_tela(Meta,CorpoMeta).
```

ativa o programa `mostra_tela/2`, instanciado com a cabeca e a cauda da regra, para mostrar esta informação ao usuário.

Para exemplificar a explicação *por_que* considere o exemplo da Base de Conhecimento mostrada na seção 5.3.4, onde é estabelecida a seguinte interação com o usuário.

```
E verdade c ?
> por_que.

SE
   c    E
   b(fc,_0FE8,_14A4,cc)

ENTAO
   a(fc,_0284,1,cd1)

E verdade c ?
> por_que.

Nao tenho mais informacoes para explicar "por_que".

E verdade c ?
```

6.2.2 A Explicação *como*

A explicação *como* descreve como o sistema chegou à conclusão que uma meta é verdadeira, usando os fatos e regras que foram gravadas na Base de Dados pelo Motor de Inferência.

A implementação realizada permite ao usuário ver uma descrição completa ou parcial do caminho percorrido na solução da meta; isto é realizado através de uma interação com o usuário. Em cada nível da regra, é perguntado ao usuário se ele quer ver a explicação da cauda da regra.

Quando a prova de uma meta é bem sucedida e a explicação *como* é ativada, o ME, com o auxílio de meta-interpretadores, constrói e interpreta uma árvore com a história da prova usando as informações contidas na Base de Dados [Amble 87]. A árvore é construída seguindo a classificação de relações mostradas na seção 5.3.

O programa `explica_como/1`, que é o responsável pela ativação da explicação *como* no Módulo de Exlicação, possui um argumento

```
explica_como(Meta)
```

onde

[E] <arg_1 >: uma meta que já foi provada e que o usuário deseja saber como foi provada.

Ela consiste de uma única cláusula

```
explica_como(Meta):-
   constroi_arvore(Meta,ArvoreMeta),
   interpreta_arvore(ArvoreMeta).
```

Com auxílio do meta-interpretador `constroi_arvore/2`, o programa `explica_como/1` constroi uma árvore contendo toda a história da prova da meta. A seguir, o meta-interpretador `interpreta_arvore/1` interpreta a árvore de prova explicando ao usuário como o sistema chegou à conclusão que a meta é verdadeira.

O programa `constroi_arvore` possui dois argumentos

```
constroi_arvore(Meta,ArvoreMeta)
```

onde

[E] <arg_1 >: a meta que o usuário deseja saber como foi provada;

[S] <arg_2 >: uma árvore, representada na forma `(Cabeca:-Cauda)`, que contém toda a história da prova da meta (<arg_1 >). O elemento `Cauda` contém as árvores de prova das premissas da `Cabeca`.

Este programa, que é constituído de 11 cláusulas, é semelhante ao meta-interpretador prove/3 do Motor de Inferência, com a diferença que este programa não exige interação com o usuário. No momento em que a explicação *como* é ativada, todas as informações já estão gravadas na Base de Dados.

As cláusulas 1, 2 e 3

```
constroi_arvore(true,true).

constroi_arvore((A,B),(ArvoreA,ArvoreB)):-
   constroi_arvore(A,ArvoreA),
   constroi_arvore(B,ArvoreB).

constroi_arvore((A;B),(ArvoreA;ArvoreB)):-
   constroi_arvore(A,ArvoreA);
   constroi_arvore(B,ArvoreB).
```

são responsáveis, respectivamente, pela condição de parada e pela construção das árvores de prova de conjunções e disjunções de metas.

A cláusula 4

```
constroi_arvore(not A,nao(A)):-
   not(constroi_arvore(A,ArvoreA)).
```

simplesmente afirma que a prova da meta não foi bem sucedida. Observe que esta cláusula não especifica o ponto em que a prova não foi bem sucedida. Ela somente armazena na árvore — usando o funtor nao — que a prova falhou.

A cláusula 5

```
constroi_arvore(A,p(A)):-
   primitiva(A),!,
   call(A).
```

é responsável pela construção da árvore das relações primitivas.

A cláusula 6

```
constroi_arvore(A,s(A)):-
   sistema(A),
   call(A).
```

usando o funtor s afirma que a meta é uma pré-definida do Prolog.

A cláusula 7

```
constroi_arvore(A,f(A)):-
   foi_perguntado(A,sim).
```

verifica se a meta é uma relação perguntável, analisando se ela já foi perguntada e a classe da resposta é `sim`. Neste caso, a meta é armazenada usando o funtor `f`.

A cláusula 8

```
constroi_arvore(A,desconhecido(A)):-
   foi_perguntado(A,nao_sei).
```

constrói a árvore de prova quando a meta é perguntável e a classe é `nao_sei`. Esta cláusula constrói a árvore das relações que aparecem na Base Conhecimento como `desconhecido(Meta)`.

A cláusula 9

```
constroi_arvore(A,decisao(ArvoreMeta)) :-
   A=..[Rel,fc,C,E,V],
   findall(d((Meta:-ArvoreCorpoA)),
           (clause(provado(Meta),true),
            Meta=..[Rel,fc,C1,E1,V],
            base_conhecimento(A,CorpoA),
            constroi_arvore(CorpoA,ArvoreCorpoA)),
           Lista),
   constroi(ArvoreMeta,Lista),!.
```

constrói a árvore das relações de decisão usando o funtor `decisao` para agrupar todos os caminhos percorridos na prova deste tipo de relação.

Com a execução do `findall` são construídas as árvores de prova de todos os caminhos percorrridos na prova da meta. O funtor `d` é usado para identificar as relações de decisão que são encontradas nestes caminhos percorridos.

Observe que esta árvore é construída buscando as metas que já foram provadas pelo Motor de Inferência e que estão gravadas na Base de Dados — esta busca é realizada usando `clause(provado(Meta),true)`.

A cláusula 10

```
constroi_arvore(A,ou(N,(A :- ArvoreA))) :-
```

```
    ja_provado(A),
    raciocinio_deterministico(A,N),!,
    base_conhecimento(A,CorpoA),
    constroi_arvore1(N,CorpoA,ArvoreA).
```

constrói a árvore de prova das relações definidas que têm `verifica(N)` como primeiro argumento. Esta cláusula é semelhante à cláusula do programa `prove/3` que manipula este tipo de relação.

A última cláusula

```
constroi_arvore(A,(A:-ArvoreA)):-
    ja_provado(A),!,
    base_conhecimento(A,CorpoA),
    constroi_arvore(CorpoA,ArvoreA).
```

constrói a árvore de prova das outras relações defindas. Ela também é semelhante à sua correspondente no Motor de Inferência; a diferença é que para construir a árvore esta cláusula verifica se a meta já foi provada.

Como o programa `constroi_arvore/2` constrói o caminho da prova agrupando as diferentes relações manipuladas pelo Motor de Inferência, a interpretação é mais simples, permitindo ao projetista do NSE específico escolher a forma mais apropriada para mostrar a saída de cada grupo de relações manipulado, bem como redefinir os códigos que mostram esta saída.

Veja um exemplo de execução do programa `constroi_arvore/2`, ativando o programa com a relação de decisão `c(fc,C,E,c4)` do exemplo que foi mostrado na seção 5.3.4.

```
?-constroi_arvore(c(fc,C,E,c4),Arvore).

Arvore = decisao((d((c(fc,0.835,_0350,c4) :- f(c(c4,0.9)))) ,
                  d((c(fc,0.835,_03A4,c4) :- f(d(d4,0.7)))))) ,

no
?-
```

O programa `interpreta_arvore/1`, que é o responsável por interpretar e mostrar a árvore de prova da explicação *como*, é constituído por 12 cláusulas.

As cláusulas 1 e 2

```
interpreta_arvore((A,B)):-
```

```
      interpreta_arvore(A),
      write('  E'),nl,
      interpreta_arvore(B).
   interpreta_arvore((A;B)):-
      interpreta_arvore(A);
      interpreta_arvore(B).
```

interpretam, respectivamente, a conjunção e disjunção de árvores.

As cláusulas 3 e 4

```
   interpreta_arvore((ArvoreA or ArvoreB)):-
      not var(ArvoreA),!,
      interpreta_arvore(ArvoreA),
      testa_var_ArvoreB(ArvoreB).
   interpreta_arvore((ArvoreA or ArvoreB)):-
      interpreta_arvore(ArvoreB).
```

interpreta a cauda de uma regra que tem o funtor `verifica(N)` como primeiro argumento. Na seção 3.4.1, foi visto que estas relações são definidas por

```
<pred>(verifica(N),<listaexp>):-
        atomo or atomo1 ...  or atomon.
```

com N inteiro, elas são verdadeiras se é possível provar a validade de N das premissas que estão na cauda da regra.

A cláusula 3 verifica se argumento `ArvoreA` não é uma variável, isto quer dizer que a cabeça da regra do argumento `ArvoreA` é uma das premissas verdadeiras do grupo de N premissas. Se `not var(ArvoreA)` for bem sucedido, `interpreta_arvore/1` é ativado para interpretar a árvore `ArvoreA`. O programa `testa_var_arvoreB/1` verifica se a árvore `ArvoreB` também não é uma variável e ativa o programa `interpreta_arvore/1` instanciado com `ArvoreB`.

A cláusula 4 é ativada somente se a cláusula 3 não for bem sucedida. Neste caso, o programa ativa `interpreta_arvore/1` é instanciado com `ArvoreB`, pois `ArvoreA` é uma das premissas que não foram provadas pelo MI.

As cláusulas 5 e 6

```
   interpreta_arvore(p(A)):-
      escreve_primitiva(A).
   interpreta_arvore(s(A)):-
      escreve_sistema(A).
```

interpretam, respectivamente, as relações primitivas e de sistema através dos programas `escreve_primitiva/1` e `escreve_sistema/1`.

A cláusula 7

```
interpreta_arvore(f(A)):-
   escreve_fato(A).
```

interpreta um fato, ou seja, as relações perguntáveis com `Classe = sim`.

A cláusula 8

```
interpreta_arvore(desconhecido(A,Classe)):-
   escreve_desconhecido(A).
```

interpreta as relações que aparecem na Base de Conhecimento como `desconhecido(Meta)`.

A cláusula 9

```
interpreta_arvore(nao(A)):-
   escreve_nao(A).
```

interpreta a relações que aparecem na BC como `not(Meta)`. O programa `escreve_nao/1` só afirma que a prova da meta A não foi bem sucedida uma vez que o programa `constroi_arvore/2`, neste caso, não gera a árvore de prova.

A cláusula 10

```
interpreta_arvore(ou(N,(A:-B))):-
   escreve_ou(N,(A:-B)).
```

interpreta, através do programa `escreve_ou/2` as regras que têm o funtor `verifica(N)` com primeiro argumento.

No primeiro nível de explicação desta regra, o programa `escreve_ou/2` mostra o número de premissas que são verdadeiras entre as premissas que compõem a cauda da regra, juntamente com a cabeça da regra. Neste nível não dá para identificar quais são as premissas verdadeiras entre as que compõem a cauda da regra. Por exemplo, se a Base de Conhecimento tem a seguinte regra:

```
a(verifica(2)):-
      b or
      c or
      d or
      e.
```

e foi provado que a regra é verdadeira com as premissas `c` e `d` verdadeiras, no primeiro nível tem-se a seguinte explicação

```
Considerando que 2 das clausulas abaixo e(sao) verdadeira(s)
     b OR
     c OR
     d OR
     e
Conclui-se a(verifica2)
```

Somente se o usuário quiser ver a árvore de prova dessas condições é que são mostrados quais são as premissas provadas.

A cláusula 11

```
interpreta_arvore(decisao(A)):-
   escreve_decisao_principal(decisao(A)).
```

interpreta a árvore de uma relação de decisão, através do programa `escreve_decisao_principal/1`. Este programa verifica se foi encontrado mais que um caminho na prova da meta (cabeça da regra). Se for bem sucedido, o programa mostra todas as regras usadas, ou seja, todos os caminhos encontrados. Isto é permitido pelo fato que o funtor `decisao` agrupa os caminhos percorridos na prova da meta; caso contrário, o programa `escreve_decisao_principal/1` simplesmente mostra o caminho encontrado.

A cláusula 12

```
interpreta_arvore((A:-B)):-
   escreve_regra((A:-B)).
```

mostra, através do programa `escreve_regra/1` , a regra que foi usada para chegar à conclusão que a meta é verdadeira.

É importante ressaltar que o meta-interpretador `interpreta_arvore/1` interpreta o caminho que levou à conclusão verdadeira através de níveis. O programa mostra primeiro a regra de mais alto nível, e somente se o usuário especificar que quer ver o corpo da regra, as árvores de prova das premissas das regras são mostradas.

A seguir é mostrado um exemplo de execução do programa `explica_como/1` instanciado com a relação de decisão que foi usada para exemplificar o programa `constroi_arvore/2`.

```
?-explica_como(c(fc,C,E,c4)).

Para mostrar  c(fc,0.835,_0314,c4)
Foram provadas as seguintes regras:

          REGRA
Considerando que
```

```
c(c4,0.9)
eh verdade.
Conclui-se  c(fc,0.835,_0314,c4)
Quer que explique os considerandos dessas condicoes ?
> sim.

Voce disse que c(c4,0.9) e verdade

        REGRA
Considerando que
d(d4,0.7)
eh verdade.
Conclui-se  c(fc,0.835,_02B0,c4)
Quer que explique os considerandos dessas condicoes ?
> nao.


Combinando os fatores de certeza obtidos com as regras que foram
mostradas, conclui-se   c(fc,0.835,_0314,c4)

C = _0074
E = _0078 ->;

no
?-
```

6.2.3 A explicação *what_if*

A explicação *what_if* verifica *o que acontece* com uma meta que foi provada *se* for trocada uma informação entre as que foram fornecidas pelo usuário do sistema, ou seja, as informações da Base de Dados que contém as respostas armazenadas no momento em que o Motor de Inferência foi ativado.

A capacidade de análise *what_if* pode ser implementada de duas maneiras. A primeira é quando são feitas suposições sob informações gerais, e o sistema é ativado para fazer deduções baseadas nestas informações. Neste caso, nem sempre a informação fornecida exerce influência sobre as deduções que serão realizadas, pois o sistema pode chegar a uma conclusão sem utilizar a informação fornecida.

A segunda opção é quando uma informação que já foi fornecida ao sistema é trocada, e o usuário deseja saber o que acontece com a meta que já foi provada, quando esta nova informação substitui a anterior. Neste caso, a nova informação influencia a meta que foi provada, pois esta informação pertence ao caminho percorrido para provar a meta. A explicação *what_if* implementada verifica este último caso.

O programa what_if/1 é constituído de uma única cláusula

```
what_if(Meta):-
   resp(Meta),
   busca_resp(Condicao),
   apaga_provado,
   acessa_Ba(Condicao,Tipo),
   apaga_condicao(Condicao,Tipo),
   ativa_MCD(Condicao,NovaCondicao,Classe),
   grava(NovaCondicao,Classe),
   ativa_MI(Meta).
```

O programa what_if/1 seleciona uma a meta que já foi provada usando o programa resp/1. O programa busca_resp/1 mostra todas as informações que estão armazenadas na BD sob a chave resp e interage com o usuário para saber qual informação ele deseja modificar.

O programa apaga_provado/0 apaga da Base de Dados todas as informações das regras que foram provadas na ativação do Motor de Inferência, garantindo, com isto, a consistência das informações. Consistência de informações é discutida em [Cuadrado 86].

A base Ba é acessada usando o programa acessa_ba/3 para buscar o tipo da pergunta — afirmativa-negativa, única ou várias — da informação que o usuário deseja modificar. Isto é necessário pois se a informação que está sendo manipulada admite várias respostas e elas já existem na Base de Dados, o Módulo de Explicação, através do programa apaga_condicao/2, mostra ao usuário as respostas existentes e pergunta qual informação deve ser substituída. Só após esta verificação ser realizada é que a informação é apagada da Base de Dados.

O programa ativa_MCD/3 ativa o Módulo Coletor de Dados com o objetivo de obter os novos dados da informação, estes novos dados são armazenados na BD através do programa grava/2. Em seguida, o Motor de Inferência é ativado para repetir a prova da meta e mostrar o que acontece usando esta nova informação.

Após o uso de *what_if* é possível que o sistema não consiga chegar a uma conclusão. Isto indica que a informação anterior, que foi substituida, é absolutamente relevante para a prova da meta.

6.3 Considerações Finais

Na construção do Nível 1 do Ambiente mostrado na figura 3.2, todas as restrições feitas no Motor de Inferência para construir o Núcleo Específico podem também ser realizadas na explicação *como*, tendo em vista que esta explicação está baseada na implementação de meta-interpretadores que manipulam todos os tipos de relações da Base de Conhecimento.

Na explicação *por_que* o usuário pode retirar a primeira cláusula do programa `explica_porque/1` se a Base de Conhecimento não possui relações definidas que têm o funtor `verifica(N)` como primeiro argumento.

Na explicação *what_if* o Nível 1 é sempre idêntico ao Nível superior.

Para todos os casos é possível redefinir, no Nível 2, o código que mostra ao usuário este tipo de informação. As implementações foram realizadas de forma bem estruturada, facilitando assim as redefinições desses códigos.

Maiores detalhes da implementação e uso do Módulo de Explicação encontram-se em [Rodrigues 90b].

Capítulo 7

Comunicação entre os Módulos

7.1 Considerações Iniciais

Este capítulo tem o objetivo de mostrar quais são os programas responsáveis pela comunicação entre os diversos subsistemas que constituem o Núcleo do Sistemas Especialistas implementado, mostrado no Nível superior do Ambiente — figura 3.2 —, bem como a comunicação destes subsistemas com a Base de Conhecimento e a Base de Dados.

7.2 Programas que fazem a Comunicação entre os Módulos que Constituem o Ambiente

A figura 7.1, mostra a comunicação entre os subsistemas que constituem o Ambiente implementando, considerando a divisão da Base de Conhecimento citada na seção 4.1. Isto é, a base Ba refere-se aos programas algorítmicos da BC, que são manipulados somente pelo MCD.

Estas comunicações são mostradas em detalhes com o objetivo de auxiliar o projetista do Núcleo específico na passagem do Nível 1 para o Nível 2 do Ambiente mostrado figura 3.2, onde são realizadas alterações no código do Núcleo do Sistema Especialista implementado.

É importante ressaltar que os programas estão bem estruturados, facilitando assim, possíveis redefinições na comunicação entre os módulos no processo de construção do Nível 2 do Ambiente. Caso o projetista do Núcleo específico não realize alterações do Nível 1 para o Nível 2, os programas responsáveis por estas comunicações também permanecem inalterados.

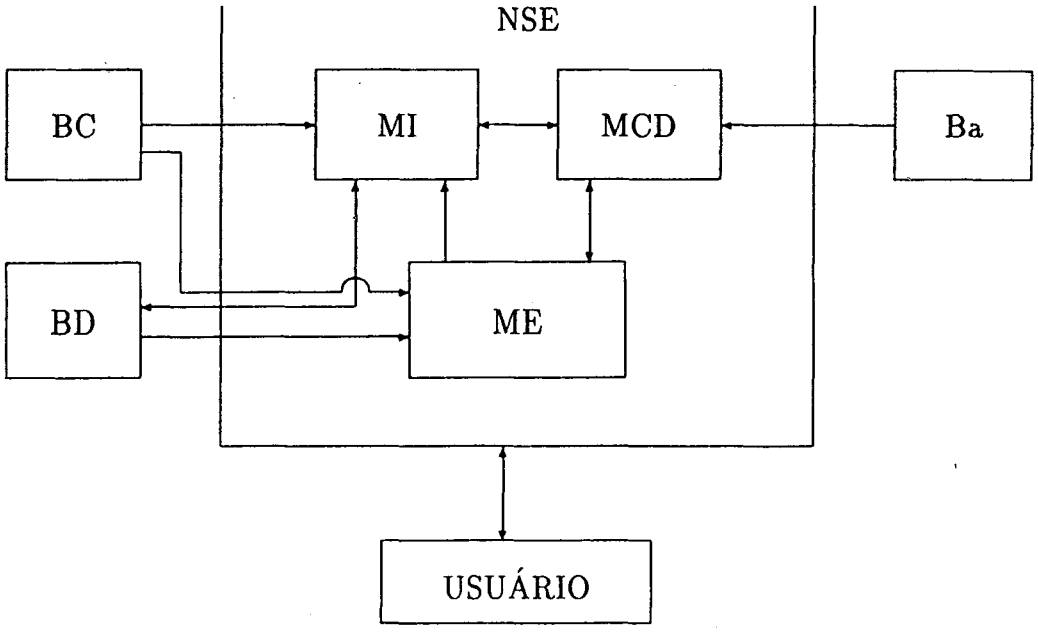
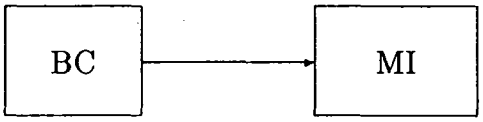


Figura 7.1: Comunicação entre os Subsistemas e as Bases

A seguir, são mostrados os programas responsáveis pela comunicação entre os módulos da figura 7.1.

7.2.1 Motor de Inferência e Base de Conhecimento



O MI acessa a BC através de

- `primitiva/1` para verificar se uma meta é primitiva;
- `perguntavel/1` para verificar se uma meta é perguntável;
- `base_conhecimento/2` para acessar uma regra de conhecimento

O Motor de Inferência também acessa a BC através do MI do Prolog.

Os dois primeiros predicados já foram explicados anteriormente. O predicado `base_conhecimento/2` é tal que, quando o MI necessita provar uma meta do tipo

```
Cabeca:- Cauda.
```

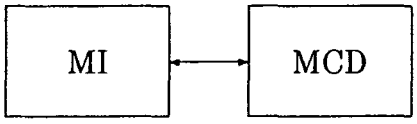
onde Cauda pode ser uma conjunção ou disjunção de átomos, bem como o corpo de uma regra de conhecimento pertencente a um dos grupo de relações da BC definidos na seção 3.4.1, então

```
base_conhecimento(Cabeca,Cauda)
```

é bem sucedida.

Se na construção do Nível 2 do Núcleo for trocada a representação de conhecimento armazenado na BC, o programa `base_conhecimento/2` deve ser redefinido para acessar corretamente a nova representação.

7.2.2 Motor de Inferência e Módulo Coletor de Dados

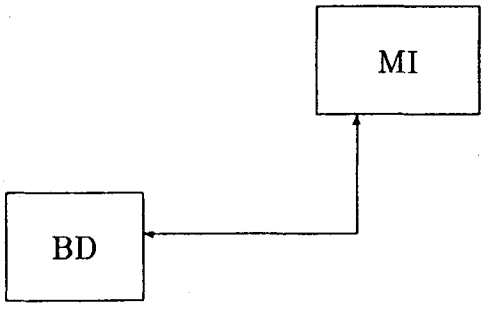


Esta comunicação é realizada através do programa

- `faz_pergunta/4`.

explicado na seção 4.4. O MI é responsável pela chamada e o MCD simplesmente fornece as respostas.

7.2.3 Motor de Inferência e Base de Dados



O MI grava na BD através de

- `lembre_provado/1`;
- `lembre_resposta/2`;
- `lembre_semsolucao/1`

Quando o MI consegue provar a validade ou a negação (por falha) de uma meta, então

```
lembre_provado(Meta)
```

é responsável por lembrar este fato na BD. Também, quando uma resposta a uma meta perguntável é obtida pelo MI através do MCD, esta resposta é gravada na BD por

```
lembre_resposta(Meta,Tipo)
```

onde `Tipo` é o tipo de resposta fornecida a esta `Meta` pelo usuário — `sim`, `nao`, `nao_sei`.

Por último, quando não é possível provar uma relação de decisão, isto é, relações que possuem fatores de certeza, então

```
lembre_semsolucao(Meta)
```

grava este fato na BD.

O MI acessa a BD através de

- `ja_provado/1`;
- `semsolucao/1`;
- `resposta/2`;
- `nao_foi_perguntada/1`

Antes de iniciar a prova de uma meta, o MI verifica se a meta não foi provada anteriormente. Se a meta já foi provada, então

```
ja_provado(Meta)
```

é bem sucedido. Observe que `ja_provado(Meta)` acessa a informação gravada na BD por `lembre_provado(Meta)`. O MI também verifica, no caso de relações que possuem fator de certeza e que não foram provadas anteriormente, se a prova dessa relação já foi tentada e se teve ou não sucesso

```
semsolucao(Meta)
```

é bem sucedido se este for o caso. Analogamente à situação acima, `semsolucao(Meta)` acessa a informação gravada por `lembre_semsolucao(Meta)`.

As metas perguntáveis são acessadas através de

```
resposta(Tipo,Meta)
```

Analogamente aos casos anteriores, `resposta(Tipo,Meta)` acessa a informação gravada por

```
lembre_resposta(Meta,Tipo).
```

Por último, `nao_foi_perguntada(Meta)` acessa a BC para verificar se uma relação perguntável foi ou não perguntada. Deve ser observado que por problemas de instanciação de variáveis, `nao_foi_perguntada(Meta)` não é equivalente a `not(resposta(_,Meta))`.

Observe que se, na construção do Nível 2 do Ambiente, o projetista do Núcleo específico redefinir os programas que o Motor de Inferência utiliza para gravar na Base de Dados — `lembre_provado/1`, `lembre_resposta/2` ou `lembre_semsolucao/1` —, os programas que o MI utiliza para acessar estas informações — `ja_provado/1`, `semsolucao/1`, `resposta/2`, ou `nao_foi_perguntada/1` — também devem ser alterados.

Em [Monard 89b] é mostrado em detalhes a implementação das comunicações entre os módulos das seções 7.2.1, 7.2.2 e 7.2.3.

7.2.4 Módulo Coletor de Dados e a base Ba

O MCD acessa a Ba através de

- `pergunta/3`;
- `pergunta/4`;
- `pergunta_livre/3`;
- `pergunta_livre/4`.

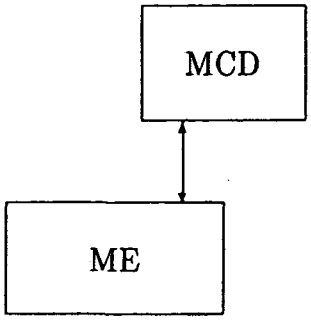
explicados na seção 4.3. No caso de ser usado `pergunta/3` ou `pergunta/4` o MCD também faz uso do programa

- `rotina_leitura/0`

se ele estiver definido na base Ba para informar ao usuário que está esperando pela sua resposta.

Em [Rodrigues 90a] é mostrada em detalhes a implementação das comunicações entre o MCD e a base Ba.

7.2.5 Módulo Coletor de Dados e Módulo de Explicação



O MCD ativa o ME, sempre que o usuário responder `por_que`, através de

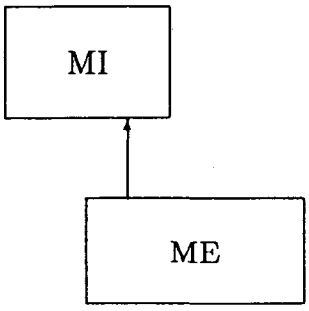
- `explica_porque/1`

passando as informações necessárias para que o ME explique ao usuário o motivo da pergunta realizada.

O ME ativa o MCD, para obter uma nova informação, através do programa

- `faz_pergunta/4` sempre que o usuário ativar a explicação *what_if*.

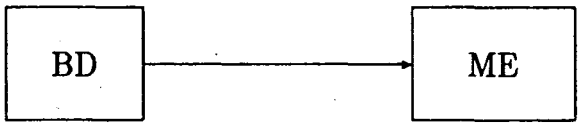
7.2.6 Módulo de Explicação e Motor de Inferência



O ME ativa o MI, sempre que o usuário solicitar a explicação *what_if*, através de

- `prove/3`.

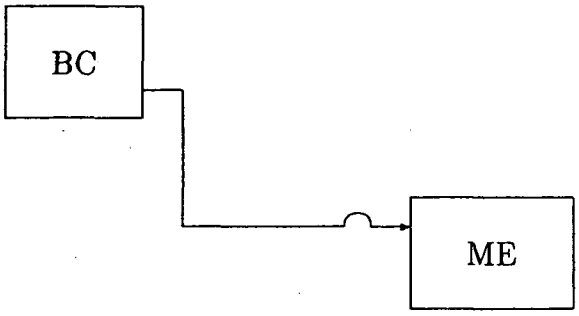
7.2.7 Módulo de Explicação e Base de Dados



O ME acessa a BD através de

- `ja_provado/1` para verificar se a meta já foi provada, quando a explicação *what_if* for solicitada;
- `resposta/2` para acessar as respostas armazenadas, quando as explicações *what_if* e *como* forem solicitadas.

7.2.8 Módulo de Explicação e Base de Conhecimento



O ME acessa a BC através de

- `primitiva/1` para verificar se uma meta é primitiva;
- `base_conhecimento/2` para acessar uma regra de conhecimento

quando a explicação *como* é solicitada pelo usuário.

Em [Rodrigues 90b] é mostrada em detalhes a implementação das comunicações entre os módulos das seções 7.2.5, 7.2.6, 7.2.7 e 7.2.8.

7.2.9 Usuário e Núcleo do Sistema Especialista

A comunicação entre o usuário e o Núcleo do SE é realizada através dos programas

- `prove/3`, ativando o Motor de Inferência;
- `explica_como/1`, ativando a explicação *como*;
- `what_if/1` ativando a explicação *what_if*.

Deve ser ressaltado que uma interface amigável com o usuário é de fundamental importância em qualquer sistema computacional. No caso de Sistemas Especialistas esta interface está, quase sempre, relacionada à área de conhecimento manipulado pelo sistema — comercial, científica, médica, etc ....

Portanto, o projetista do Núcleo é livre para definir esta interface da forma que ele achar mais conveniente.

7.3 Considerações Finais

As implementações das comunicações entre os módulos foram realizadas sem utilizar recursos específicos da versão da linguagem de implementação [Arity 88]. Por sua modularidade, elas podem ser facilmente reescritas a fim de aproveitar os recursos fornecidos pelo sistema Prolog em uso, que podem melhorar, em muitos casos, o tempo de execução do sistema.

Na construção do Nível 2 do Ambiente — figura 3.2 —, é possível redefinir o código dos programas que fazem estas comunicações.

Capítulo 8

Exemplo de Interação com o Sistema

8.1 Considerações Iniciais

Nos capítulos anteriores foram evidenciados os subsistemas que constituem os Núcleos de Sistemas Especialistas no Nível superior do Ambiente — mostrado na figura 3.2. Este capítulo tem o objetivo de mostrar a interação do usuário com aquele Nível superior.

A Base de Conhecimento que é usada não carrega nenhuma informação contextual nas suas regras. Após experiências realizadas para mostrar a interação com o usuário, decidimos usar este tipo de base pois ela permite seguir a execução do sistema do ponto de vista puramente computacional, sem interagir com o aspecto subjetivo que é introduzido quando se lida com Bases de Conhecimento do mundo real.

No restante do capítulo serão mostrados quais devem ser os procedimentos iniciais para o uso do NSE, que constitui o Nível superior do Ambiente desenvolvido, bem como uma interface simples, com o objetivo de ilustrar a comunicação entre o usuário e o Núcleo do Sistema Especialista.

8.2 Exemplo de uma Base de Conhecimento

A seguir, será mostrada a Base de Conhecimento, que, como já foi mencionado, não carrega nenhuma informação contextual nas suas regras.

```
/* Relações de Decisão */

a(fc,C,1,cd1):-
   c,
```

```
   b(fc,C,E,cc).
a(fc,C,0.5,cd2):-
   z(aa,C2),
   b(fc,C1,E,cc),
   min(C,[C1,C2]).

b(fc,C,1,bb):-
   e,
   c(fc,C,E,c4).
b(fc,C,1,bb):-
   d,
   c(fc,C,E,c5).
b(fc,C,1,cc):-
   c(fc,C,E,c6).

c(fc,C,0.5,c4):-
   c(c4,C).
c(fc,C,1,c4):-
   d(d4,C).
c(fc,C,1,c5):-
   e(e5,C1).
c(fc,C,1,c6):-
   f(f6,C).
c(fc,C,0.9,c6):-
   g(g6,C).
```

/* *Relações Definidas* */

```
s(b):-
   c(c2,C);
   c(c4,C).
m(z):-
   f.
t(c):-
   n(c),
   b.
n(c).
u.
v(a):-
   c(c5,E).
```

/* que tem o funtor `verifica(N)` como primeiro argumento */

```
r(verifica(2),regra1):-
```

```
    s(X) or
    a(fc,C,E,cd2) or
    m(Z).
r(verifica(3),regra2):-
    t(W) or
    c(fc,C,E,c5) or
    v(M) or
    u or
    b(fc,C,1,cc).
```

/* *Relações Primitivas* */

```
primitiva(min(X,Lista)).

min(X,[X]).
min(X,[X,Y]):-
   X =< Y,!.
min(Y,[X,Y]):-!.
min(Min,[X,Y|Z]):-
   X =< Y, !,
   min(Min,[X|Z]).
min(Min,[X,Y|Z]):-
   min(Min,[Y|Z]).
```

/* *Relações Perguntáveis* */

```
perguntavel(b).
perguntavel(c).
perguntavel(d).
perguntavel(e).
perguntavel(f).
perguntavel(z(X,FC)).
perguntavel(c(X,FC)).
perguntavel(d(X,FC)).
perguntavel(e(X,FC)).
perguntavel(f(X,FC)).
perguntavel(g(X,FC)).
```

/* Informações contidas na base Ba */

```
pergunta(b,pr(es('E verdade b ?')),[s-n]).
pergunta(c,pr(es('E verdade c ?')),[s-n]).
```

```
pergunta(d,pr(es('E verdade d ?')),[s-n]).
pergunta(e,pr(es('E verdade e ?')),[s-n]).
pergunta(f,pr(es('E verdade f ?')),[s-n]).
pergunta(z(X,FC),pr(es('Qual o valor (aa, bb, cc) ?'),
        es(' E qual o fator de certeza ?')),[u,critica_z,critica_fc]).
pergunta(c(X,FC),pr(es('Qual o valor de c (c1, c2, c3, c4, c5 ou c6) ?'),
        es('E qual o fator de certeza ?')),[v,critica_c,critica_fc]).
pergunta(d(X,FC),pr(es('Qual o valor de d (d1,d2,d3,d4,d5 ou d6) ?'),
        es('E qual o fator de certeza ?')),[u,critica_d,critica_fc]).
pergunta(f(X,FC),pr(es('Qual o valor de f (f1,f2,f3,f4,f5 ou f6) ?'),
        es('E qual o fator de certeza ?')),[u,critica_f,critica_fc]).
pergunta(g(X,FC),pr(es('Qual o valor de g (g1,g2,g3,g4,g5 ou g6) ?'),
        es('E qual o fator de certeza ?')),[u,critica_g,critica_fc]).
pergunta(e(X,FC),pr(es('Qual o valor (e1, e2, e3, e4 ou e5) ?'),
        es('E qual o fator de certeza ?')),[u,critica_e,critica_fc]).

critica_z(X):-pertence(X,[aa,bb,cc]),!.
critica_z(X):-write('Responda novamente. '),nl,fail.

critica_c(X):-pertence(X,[c1,c2,c3,c4,c5,c6,c7,c8]),!.
critica_c(X):-write('Responda novamente o valor de c. '),nl,fail.

critica_d(X):-pertence(X,[d1,d2,d3,d4,d5,d6]),!.
critica_d(X):-write('Responda novamente o valor de d. '),nl,fail.

critica_e(X):-pertence(X,[e1,e2,e3,e4,e5]),!.
critica_e(X):-write('Responda novamente o valor de e. '),nl,fail.

critica_f(X):-pertence(X,[f1,f2,f3,f4,f5,f6]),!.
critica_f(X):-write('Responda novamente o valor de f. '),nl,fail.

critica_g(X):-pertence(X,[g1,g2,g3,g4,g5,g6]),!.
critica_g(X):-write('Responda novamente o valor de g. '),nl,fail.

critica_fc(X):- X>0, F=<1,!.
critica_fc(X):-write('Responda novamente o valor do fc.'),nl,fail.

es(X):-write(X),nl.
pertence(X,[X|_]).
pertence(X,[R|Z]):-pertence(X,Z).
```

8.3 Uso do Sistema

A seguir será ilustrada a interação do usuário com o Ambiente, utilizando a Base de Conhecimento acima descrita. Para isto foi desenvolvida uma interface muito simples. Esta interface utiliza recursos oferecidos pelo Arity Prolog versão 5.1, como menu de opções e caixas de diálogos entre outros, os quais permitem uma fácil interação com o sistema implementado. Esta interface foi construída só para demonstração do trabalho até aqui descrito. O projetista que for utilizar o Ambiente em questão deverá implementar uma interface orientada para a aplicação específica.

Na interface implementada, o Ambiente desenvolvido é ativado usando a palavra *ambiente*. Após ativar o sistema aparece a janela mostrada na figura 8.1.

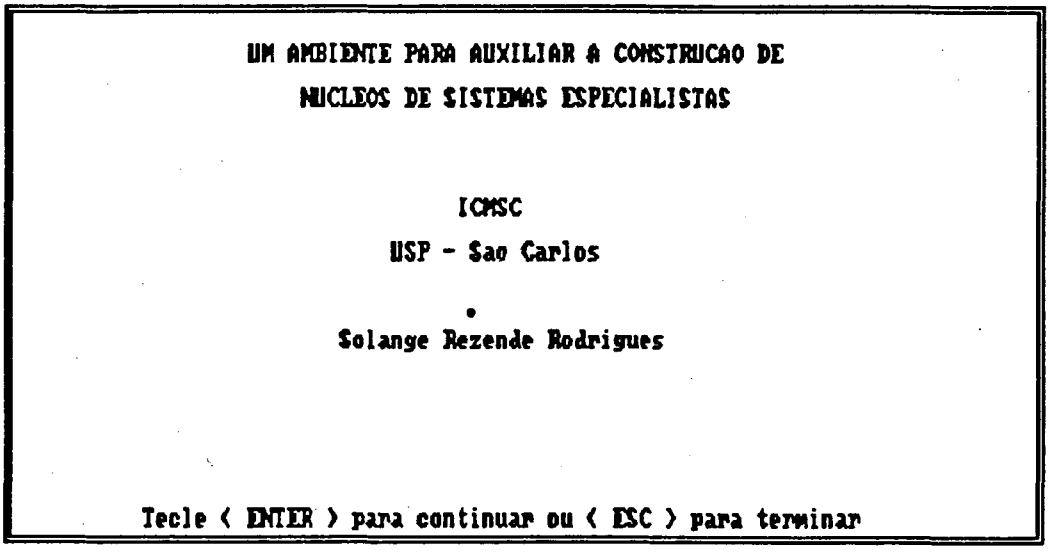


Figura 8.1: Abertura do Ambiente Implementado

Posteriormente aparece o menu principal do sistema, o qual pode ativar somente os subsistemas descritos na seção 7.2.9. Nesta interface, as opções do menu principal são: `Carrega BC` , `Ativa Prova`, `Ativa Explicacao como`, `Ativa Explicacao what_if`, `Limpa Base de Dados` e `Terminar`. A figura 8.2 mostra como este menu aparece na tela.

Na interface utilizada, sempre que aparecerem diversas opções, só poderão ser ativadas aquelas onde o símbolo ▒ estiver ausente. Assim, a primeira vez que o menu principal é ativado, o usuário só pode escolher as opções `Carrega BC` ou `Terminar`.

A escolha da opção `Carrega BC` ativa uma caixa de diálogo — figura 8.3 — que mostra as Bases de Conhecimento disponíveis. Uma restrição desta interface é que as BC devem ter extensão `bc`.

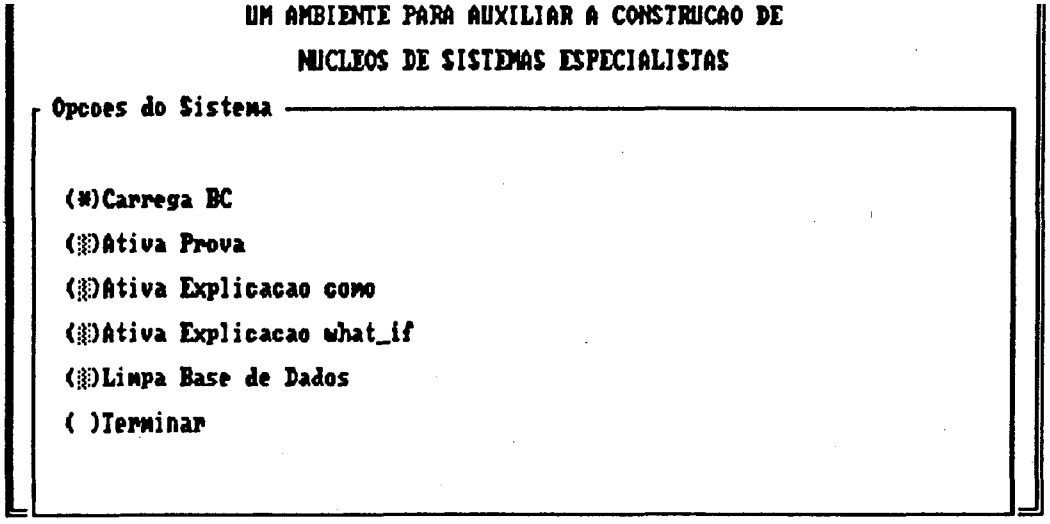


Figura 8.2: Menu Principal

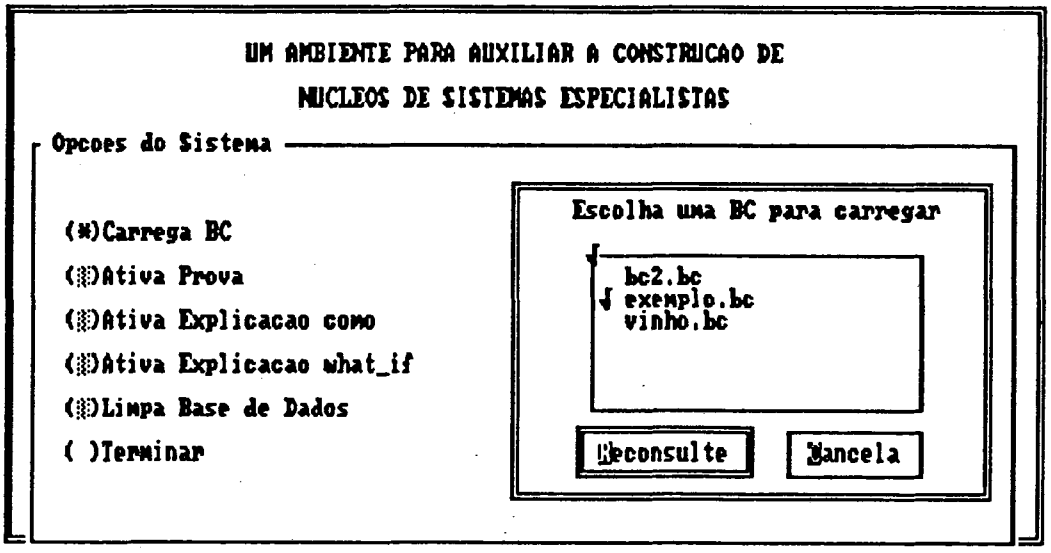


Figura 8.3: Carrega Base de Conhecimento

Após a confirmação da escolha de uma Base de Conhecimento o sistema volta ao menu principal. A Base de Conhecimento apresentada na seção anterior encontra-se no arquivo `exemplo.bc`. Depois de ter carregado a base, o usuário já pode ativar a opção `Ativa Prova`. Esta opção escolhe uma meta para ser provada e ativa o Motor de Inferência, descrito no capítulo 5.

Deve ser ressaltado que o Engenheiro de Conhecimento, juntamente com o projetista do Núcleo do Sistema Especialista pode definir quais as metas da Base de Conhecimento que

podem ser selecionadas pelo usuário para ativar a prova. A figura 8.4 mostra a janela que é ativada no momento em que a opção `Ativa Prova` é selecionada. A figura 8.5 mostra uma parte da sessão na qual o sistema está provando a meta previamente selecionada (a primeira meta na figura 8.4).

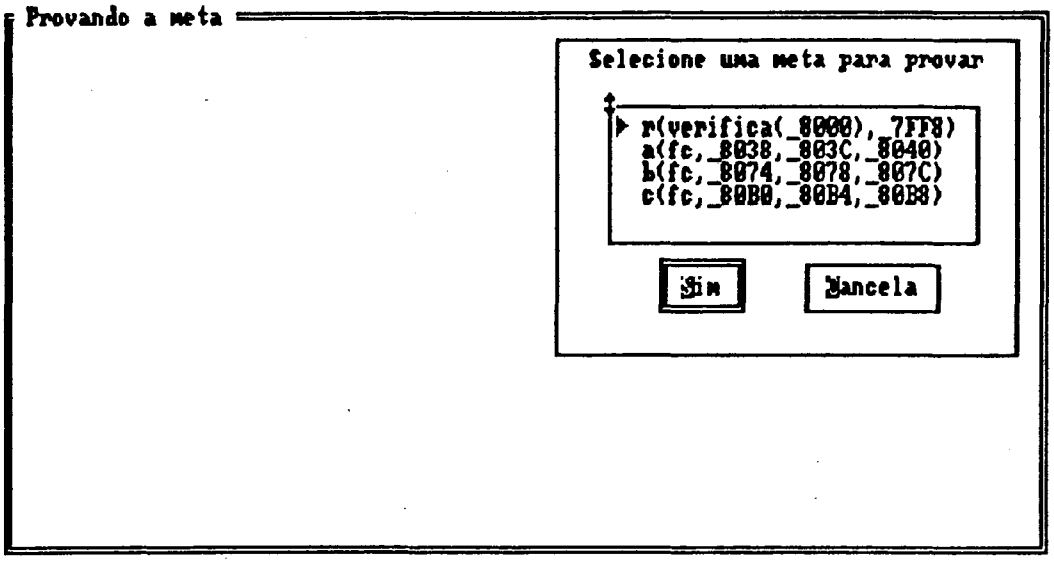


Figura 8.4: Escolha da Meta a ser provada

Após conseguir provar esta meta — r(verifica(2),regra1) — a BD, inicialmente vazia, contém toda a história da prova na seguinte forma:

```
provado(s(b)).
provado(c(fc,0.886,A,c6)).
provado(c(fc,0.7,A,c4)).
provado(b(fc,0.886,A,cc)).
provado(b(fc,0.7,A,bb)).
provado(a(fc,0.886,A,cd1)).
provado(a(fc,0.3,A,cd2)).
provado(r(verifica(2),regra1)).

resp(c(c2,0.8),sim).
resp(c,sim).
resp(e,sim).
resp(d(d4,0.7),sim).
resp(e(e5,0.6),sim).
resp(f(f6,0.4),sim).
resp(g(g6,0.9),sim).
resp(d,sim).
resp(z(aa,0.6),sim).
```

podem ser selecionadas pelo usuário para ativar a prova. A figura 8.4 mostra a janela que é ativada no momento em que a opção `Ativa Prova` é selecionada. A figura 8.5 mostra uma parte da sessão na qual o sistema está provando a meta previamente selecionada (a primeira meta na figura 8.4).

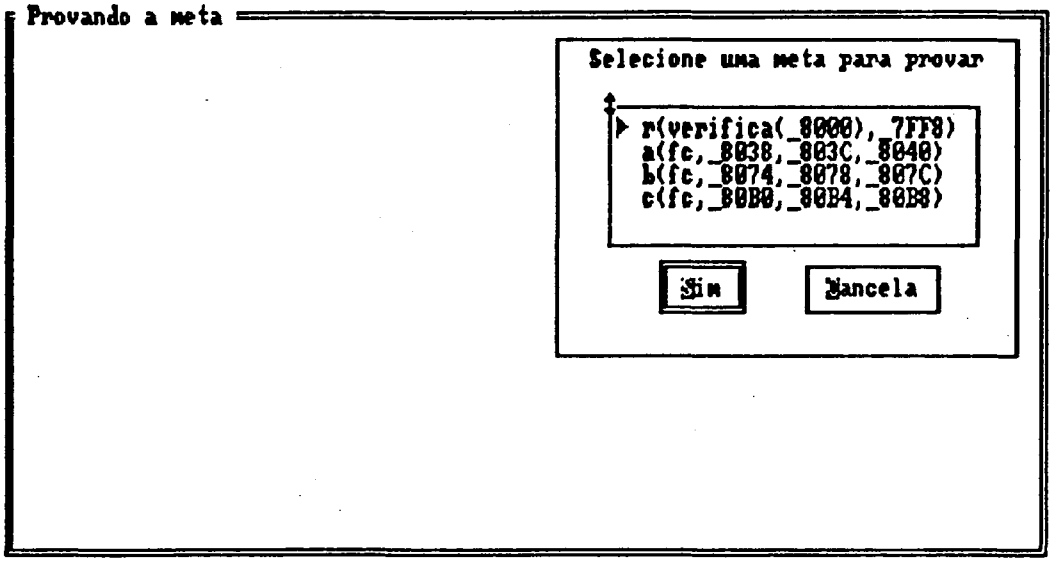


Figura 8.4: Escolha da Meta a ser provada

Após conseguir provar esta meta — r(verifica(2),regra1) — a BD, inicialmente vazia, contém toda a história da prova na seguinte forma:

```
provado(s(b)).
provado(c(fc,0.886,A,c6)).
provado(c(fc,0.7,A,c4)).
provado(b(fc,0.886,A,cc)).
provado(b(fc,0.7,A,bb)).
provado(a(fc,0.886,A,cd1)).
provado(a(fc,0.3,A,cd2)).
provado(r(verifica(2),regra1)).

resp(c(c2,0.8),sim).
resp(c,sim).
resp(e,sim).
resp(d(d4,0.7),sim).
resp(e(e5,0.6),sim).
resp(f(f6,0.4),sim).
resp(g(g6,0.9),sim).
resp(d,sim).
resp(z(aa,0.6),sim).
```

```
Provando a meta
Qual o valor de c (c1, c2, c3, c4, c5 ou c6) ?
E qual o fator de certeza ?
>c2.
>0.8.
E verdade c ?
>sim.
E verdade e ?
>sim.
Qual o valor de d (d1,d2,d3,d4,d5 ou d6) ?
E qual o fator de certeza ?
>d4.
>0.7.
Qual o valor (e1, e2, e3, e4 ou e5) ?
E qual o fator de certeza ?
>e5.
>0.6.
Qual o valor de f (f1,f2,f3,f4,f5 ou f6) ?
E qual o fator de certeza ?
>f6.
>0.4.
Qual o valor de g (g1,g2,g3,g4,g5 ou g6) ?
E qual o fator de certeza ?
>
```

Figura 8.5: Ativação do Motor de Inferência

No momento em que a ativação do Motor de Inferência é bem sucedida e o sistema retorna ao menu principal e, as opções `Ativa Explicacao como` e `Ativa Explicacao what_if` são liberadas para serem ativadas. Isto se deve ao fato que o Módulo de Explicação precisa das informações armazenadas na Base de Dados pelo Motor de Inferência, para fornecer as explicações solicitadas pelo usuário.

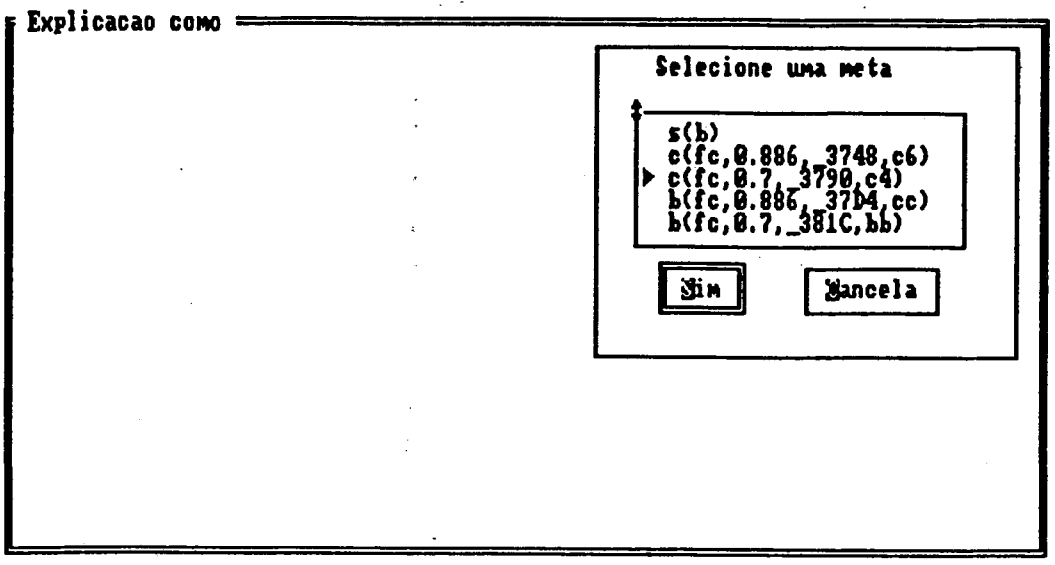


Figura 8.6: Escolha da Meta a ser explicada

Quando a opção `Ativa Explicacao como` é ativada, o sistema mostra ao usuário as metas que foram provadas e solicita a escolha de uma delas — figura 8.6. Em seguida, ativa a Explicação *como* do Módulo de Explicação — seção 6.2.2 — para que o usuário acompanhe o caminho que foi percorrido na prova da meta selecionada. A figura 8.7 mostra a explicação *como* quando a meta `c/4` é selecionada.

```
Explicacao como
Considerando que
d(d4,0.7)
eh verdade.
Conclui-se  c(fc,0.7,_8600,c4)
Quer que explique os considerandos dessas condicoes ?
sim.

Voce disse que d(d4,0.7) e verdade

Tecle <ESC> para retornar ao menu principal
```

Figura 8.7: Ativação da Explicação *como*

Se o usuário seleciona a opção `Ativa explicacao what_if` o sistema mostra ao usuário as metas que foram provadas — figura 8.8 —, solicita a escolha de uma delas e, em seguida, ativa a explicação *what_if* do Módulo de Explicação. A figura 8.9 mostra a ativação da explicação *what_if*.

```
Explicacao what_if

Selecione uma meta

  b(fc,0.778,_8600,cc)
  b(fc,0.7,_8650,bb)
  a(fc,0.778,_8870,cd1)
  a(fc,0.389,_8870,cd2)
▶ r(verifica(Z),regra1)

Sim        Cancela
```

Figura 8.8: Escolha da Meta a ser explicada com *what_if*

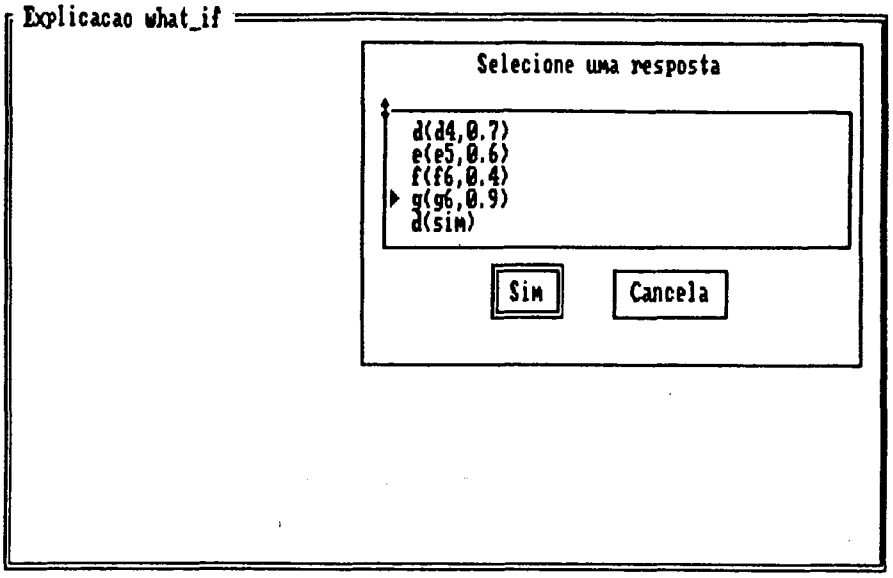


Figura 8.9: Ativação da Explicação *what_if*

Para iniciar uma nova sessão deve-se limpar a BD. A opção `Limpa BD` é responsável por limpar a Base de Dados.

8.4 Considerações Finais

Neste capítulo foi apresentada uma pequena Base de Conhecimento que não carrega nenhuma informação contextual, para exemplificar os passos que devem ser seguidos em uma sessão de trabalho com o Núcleo do Sistema Especialista. Esta demonstração utiliza uma interface simples que mostra uma possível interação entre o usuário e o NSE.

Como já foi mencionado, a responsabilidade da implementação desta interface é do projetista do sistema.

Capítulo 9

Conclusões Finais

9.1 Avaliação do Ambiente Proposto

Este trabalho, teve como objetivo principal, mostrar a viabilidade de um Ambiente com características específicas, para auxiliar projetistas de SE na construção de Núcleos de Sistemas Especialistas.

Para atingir este objetivo, foi implementado o Núcleo de um Sistema Especialista, levando em consideração a maioria dos problemas encontrados na construção deste tipo de sistemas.

O Núcleo implementado é bastante abrangente e possui as facilidades que são consideradas mais importantes neste tipo de software. Ele foi implementado na linguagem de programação lógica Prolog. Há implementações comercialmente disponíveis desta linguagem, que usam convenções sintáticas diferentes. Neste trabalho foi usada a chamada sintaxe de Edinburgh, que tem tido ampla aceitação na comunidade científica. Especificamente, foi usado o Arity-Prolog versão 5.1 para microcomputadores IBM PC - compatíveis [Arity 88].

Entretanto, levando em conta os possíveis problemas de transportabilidade para computadores de maior porte, não foi utilizado nenhum dos comandos "não padrão" oferecidos no Arity-Prolog. Assim, o código implementado pode ser diretamente transportado para outros sistemas computacionais que possuam, por exemplo, Quintus Prolog [Quintus 85], C-Prolog [Pereiral 82], ou semelhantes.

A linguagem Prolog, que está solidamente construída sobre bases científicas [Maier 88], possui características apropriadas para desenvolver Sistemas baseados em conhecimento.

Programas Prolog, quando bem definidos, podem ser facilmente estendidos ou reduzidos através da adição ou eliminação de cláusulas, respectivamente. Esta facilidade foi de extrema importância na construção do Ambiente proposto.

No entanto, para desenvolver sistemas de médio ou grande porte em Prolog, é imprescindível conhecer dois aspectos desta linguagem:

- *aspecto declarativo*, que descreve a estrutura lógica do problema;
- *aspecto procedimental*, que descreve como o computador resolve o problema.

Conseguir conciliar estes dois aspectos não é sempre uma tarefa fácil. Isto se reflete na dificuldade de documentar detalhadamente programas Prolog, no caso em que ambos os aspectos citados devem ser evidenciados. Muito esforço foi consumido no desenvolvimento deste trabalho, a fim de documentar tanto o aspecto declarativo quanto o procedimental do Núcleo implementado.

Um outro ponto que é interessante mencionar é referente ao uso prático de um subconjunto do Núcleo apresentado neste trabalho. Em 1989 ministramos dois seminários intensivos (40hs/seminário) do curso CAVIAR I (Curso AVançado de Inteligência ARtificial), junto ao ILTC (Instituto de Lógica Filosofia e Teoria da Ciência) no Rio de Janeiro, para engenheiros da Petrobrás.

O primeiro seminário foi dedicado à linguagem Prolog, e o segundo a Sistemas Especialistas. No segundo, foi explicado em detalhes uma versão muito reduzida dos diversos subsistemas do Núcleo de SE, onde foram expostas as idéias usadas na implementação dos mesmos, enfatizando meta programação.

Como trabalho final, os participantes dos cursos implementaram em pouco tempo, pequenas Bases de Conhecimento de diversas áreas, tais como seleção de pessoal, recomendação de compra de automóveis, entre outras, usando esta versão restrita do Núcleo.

Foi interessante observar que, nos melhores trabalhos, um grande esforço de programação foi realizado, com o objetivo de implementar mais facilidades no Módulo Coletor de Dados, que naquela época não tinha todas as opções que apresentamos aqui. Portanto, este subsistema foi reestruturado, a fim de incluir várias opções de interação com o usuário. A versão apresentada neste trabalho, inclui as opções e facilidades mais relevantes.

Os testes realizados com o Núcleo do Sistema Especialista completo, usando Bases de Conhecimento de médio porte, mostram que o tempo de resposta do sistema é satisfatório, considerando os recursos computacionais usados. Uma das propostas é reduzir ainda mais o tempo de resposta, construindo um Núcleo de SE específico, contendo somente o código necessário para manipular a Base de Conhecimento usada.

Um outro trabalho de pesquisa que está em fase inicial é a modelagem de uma Base de Conhecimento de maior porte [Monard 89c], que permitirá uma melhor avaliação do tempo de resposta do Sistema.

A memória utilizada pelo Núcleo completo é de aproximadamente 250 Kbytes, e é menor para Núcleos que não fazem uso de todas as facilidades implementadas.

9.2 Sugestões de Trabalhos Futuros

As linhas de pesquisa que dão continuidade e complementam aspectos apresentados neste trabalho, são várias. Delineamos algumas a seguir:

Um ponto, que merece consideração especial, é o desenvolvimento de um software que automatize o processo de construção de Núcleos específicos, já que, até o momento, esta tarefa é realizada manualmente. Na passagem do Nível superior para o Nível 1, este software deve ser controlado pelo projetista através de um conjunto de opções a ele apresentadas, para cada um dos subsistemas. Desta forma, o projetista pode retirar as facilidades que não são de seu interesse, reduzindo assim o código do sistema. Com isto, a construção deste primeiro nível torna-se segura e confiável.

A automação da passagem para o Nível 2, onde é permitido ao projetista inserir e/ou alterar código, é bem mais complicada e merece um estudo cuidadoso, pois é computacionalmente impossível validar estes programas. Ainda assim, é possível realizar algumas verificações para auxiliar o projetista nesta tarefa.

Com relação ao Nível superior, que possui um Motor de Inferência com raciocínio *Backward Chaining*, é possível estendê-lo incluindo também um Motor de Inferência com raciocínio *Forward Chaining*.

Também é interessante incluir uma opção que permita ao usuário se utilizar de uma Base de Conhecimento Auxiliar, onde serão armazenadas as regras com os quais ele discorda, sem que com isso a BC original seja alterada. Esta BC Auxiliar será consultada, quando for conveniente, pelo Engenheiro de Conhecimento, a fim de se verificar se as discordâncias feitas pelos usuários são consistentes.

Uma outra pesquisa relacionada que está em andamento, é o desenvolvimento de ferramentas para auxiliar a construção de Bases de Conhecimento usando Aprendizado de Máquina por Exemplos [Castiñeira 89], na tentativa de atenuar o gargalo existente no processo de aquisição de conhecimento. Estas ferramentas servirão de auxílio na construção de regras de conhecimento, na forma manipulada pelo Ambiente proposto.

Bibliografia

[Alty 84] Alty, J.C.; Coombs, M.J. *Expert Systems: Concepts and Examples.* NCC Publications, 1984.

[Amble 87] Amble, T. *Logic Programming and Knowledge Engineering.* Addison-Wesley, 1987.

[Araribóia 89] Araribóia G. *Inteligência Artificial: Um Curso Prático.* Livros Técnicos e Científicos Editora Ltda, 1989.

[Arity 88] Arity Corporation. *The Arity/Prolog Programming Language.* 1988.

[Barros 86] Barros, L.N. *Um Compilador de Regras para Geração de um S.E. com Encadeamento Regressivo.* ICMSC-USP. Dissertação de Mestrado, 1986.

[Brachman 83] Brachman, R.J. *What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks*, 1986.

[Buchaman 84] Buchaman, B.G. ; Shortliffe, E.H. (Ed.) *Rule-Based Expert Systems.* Addison-Wesley Series in Artificial Intelligence, 1984.

[Carnota 88] Carnota, R.J.; Tezkiewiz, A.D. *Sistemas Expertos y Representación del Conocimiento.* Editora Ebai, 1988.

[Castiñeira 89] Castiñeira, M.I. *Aprendizado de Máquina por Exemplos: Família TDIDT.* Mini-dissertação, ICMSC - USP, Novembro, 1989.

[Clark 82] Clark, K.L.; McCabe, F.G.; Hammond, P. *Prolog: a Language for Implementing Expert Systems.* Machine Intelligence 10, Ellis Horwood, 1982.

[Cohen 85] Cohen, J. *Describing Prolog by its Implementation and Compilation.* CACM, Vol 18, No 2, pp 1311-1324, December 1985.

[Costa 85] Costa Pereira, A.E. *Comunicação pessoal*, 1985.

[Cuadrado 86] Cuadrado, C.Y.; Cuadrado, J.L. *Handling Conflicts in Data.* Byte, pp 193-202. November 1986.

[Duda 79] Duda, R.O.; Hart, P.E.; Konolige, K.; Rebo, R.A. *Computer_Based Consultation for Mineral Exploration*. Technical Report SRI International, Sept. 1979.

[Hammond 80] Hammond, P. *Logic Programming for Expert Systems*. MSc Thesis, Dept. of Computing, Imperial College, Inglaterra, 1980.

[Hammond 83] Hammond, P. *APES: A Prolog Expert System Shell*. Dept. of Computing, Imperial College, 1983.

[Hayes Roth 83] Hayes-Roth, F.; Waterman, D.A.; Lenat, D.B. (Eds). *Building Expert Systems*. Addison-Wesley, 1983.

[Hebihara 88] Hebihara, S.M. *Implementação Lógica do Núcleo de um S.E.* ICMSC-USP. Dissertação de Mestrado, 1988.

[Jackson 86] Jackson, P. *Introduction to Expert Systems*. Addison-Wesley, 1986.

[Kowalski 79] Kowalski, R.A. *Logic for Problem Solving*. Artificial Intelligence Series, Vol 7, Elsevier-North Holland, 1979.

[Kowalski 86] Kowalski, B. *Logic for Expert Systems*. AI Masters-Logic Programming, pp 92-104, 1986.

[Kowalski 87] Kowalski, R.A ; Hogger, C.J. *Logic Programming*. Encyclopedia of Artificial Intelligence. Shapiro, S.C. (Ed.), John Wiley & Sons, pp 544-558, 1987.

[Kramer 87] Kramer, B.M. ; Mylopoulos, J. *Knowledge Representation*. Encyclopedia of Artificial Intelligence. Shapiro, S.C. (Ed.), John Wiley & Sons, pp 882-890, 1987.

[Maida 87] Maida, A.S. *Frame Theory*. Encyclopedia of Artificial Intelligence. Shapiro, S.C. (Ed.), John Wiley & Sons, pp 302-312, 1987.

[Maier 88] Maier, D. ; Warren, D.S. *Computing with Logic: Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company Inc., 1988.

[Michie 82] Michie, D. (Ed). *Introductory Readings in Expert Systems*. Gordon and Brach, 1982.

[Minsky 75] Minsky, M. *A Framework for Representing Knowledge*. The Psycology of Computer Vision. McGraw-Hill, N.Y., 1975.

[Monard 86] Monard, M.C. *Um Sistema de Representação de Conhecimento e Raciocínio por "default"*. Tese de Livre-Docência. ICMSC-USP, 1986.

[Monard 87] Monard, M.C.; Hebihara S.M. *Um Sistema que Utiliza Raciocínio "default"*. IV Simpósio Brasileiro de I.A., pp 143-156, 1987.

[Monard 89a] Monard, M.C.; Prado, A.H.A. *Uso de Incerteza em Sistemas Baseados em Conhecimento*. ILTC, 36 pp, Maio 1989.

[Monard 89b] Monard, M.C.; Rodrigues, S.R. *Implementação Lógica de um Motor de Inferência com Raciocínio Backward Chaining para a Construção de Sistemas Especialistas*. Notas do ICMSC-USP, nº 51, 58 pp, 1989.

[Monard 89c] Monard, M.C.; Scarpelli, H.A ; Rodrigues, S.R. *Implementação de uma Base de Conhecimento para Apoio a Usuários da Biblioteca NAG*. Anais do XII CNMAC, pp 349-352, Setembro 1989.

[Niblett 84] Niblett,T. *YAPES - Yet Another Prolog Expert System*. Technical Report TIRM-84-008, Turing Institute, 38 pp, 1984.

[Nunes 86] Nunes, M.G.V.; Costa Pereira, A.E.; Monard, M.C. *O Núcleo de um S.E. com Raciocínio "Forward Chaining" usando Inferência Probabilística Bayesiana*. Anais do VI Congresso da SBC, 1986.

[Pearl 86] Pearl, J. *Bayes Decision Methods*. Technical Report CSD-850023. University of California at Los Angeles, January 1986.

[Pereiral 82] Pereira, F. *C-Prolog User's Manual*. University of Edinburgh: Department of Computer-Aided Architectural Desing, 1982.

[Quintus 85] *Quintus Prolog User's Guide and Reference Manual*. Palo Alto: Quintus Computer System Inc., 1985.

[Robinson 65] Robinson, J.A. *A Machine-Oriented Logic Based on the Resolution Principle*. JACM, Vol 12, No 1, pp 23-41, January 1965.

[Rodrigues 89] Rodrigues, S.R.; Monard, M.C. *Um subsistema de um Núcleo de Sistema Especialista para Interrogar o Usuário Admitindo Diversas Categorias de Perguntas*. Anais VI SBIA, pp 78-92, Novembro 1989.

[Rodrigues 90a] Rodrigues, S.R.; Monard, M.C. *Implementação Lógica de um Módulo Coletor de Dados para a Construção de Sistemas Especialistas*. Notas do ICMSC-USP, nº 63 41pp, 1990.

[Rodrigues 90b] Rodrigues, S.R.; Monard, M.C. *Implementação Lógica de um Módulo de Explicação para a Construção de Sistemas Especialistas*. (Em preparação).

[Rossi 86] Rossi, G. *Uses of PROLOG in Implementation of Expert Systems*. New Generation Computing, 4, pp. 321-329, 1986.

[Shell 85] Shell, P.S. *Expert Systems: A Practical Introduction*. McMillan Publishers Ltd., 1985.

[Shortliffe 76] Shortliffe, E.H. *Computer_Based Medical Consultation Mycin*. American Elsevier, New York, 1976.

[Sterling 86a] Sterling, L.; Ranall, D.B. *Incremental Flavors-Mixing of Meta-interpreters for Expert System Construction.* Proceedings Symposium on Logic Programming, pp 20-27, 1986.

[Sterling 86b] Sterling, L.; Shapiro, E. *The Art of Prolog.* The MIT Press, 1986.

[Walker 87] Walker, A. (Ed); McCord, M.; Sowa, J.E.; Wilson, W.G. *Knowledge Systems and Prolog. A Logical Appoach for Expert systems and Natural Language Processing.* Addison-Wesley, 1987.

[Warren 77] Warren, D.H.D.; Pereira, L.M.; Pereira, F. *PROLOG-The Language and its Implementation Compared with Lisp.* SIGART Newsletter, No 64, pp 109-115, August 1977.

[Waterman 86] Waterman, D.A. *A Guide to Expert Systems.* Addison - Wesley, 1986.

[White 85] White, A.P. *Inference Deficiencies in Rule-based Expert Systems.* Research and Development in Expert Systems, Cambridge University Press, pp 39-50, 1985.

[Zadeh 65] Zadeh, L.A. *Fuzzy Sets.* Information and Control, Vol 8, pp 338-353, 1965.