UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

CURSO DE MESTRADO EM INFORMÁTICA

CONTRIBUIÇÃO PARA A CONSTRUÇÃO DE SISTEMAS BASEADOS EM CONHECIMENTO VOLTADOS PARA O CONHECIMENTO PROFUNDO

MARIA LÍGIA BARBOSA PERKUSICH

Campina Grande, PB

Dezembro - 1990

CONTRIBUIÇÃO PARA A CONSTRUÇÃO DE SISTEMAS BASEADOS EM CONHECIMENTO VOLTADOS PARA O CONHECIMENTO PROFUNDO

MARIA LÍGIA BARBOSA PERKUSICH

CONTRIBUIÇÃO PARA A CONSTRUÇÃO DE SISTEMAS BASEADOS EM CONHECIMENTO VOLTADOS PARA O CONHECIMENTO PROFUNDO

Dissertação apresentada ao curso de **MESTRADO EM INFORMÁTICA** da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do grau de mestre.

ÁREA DE CONCENTRAÇÃO: CIÊNCIA DA COMPUTAÇÃO

MISAEL ELIAS DE MORAIS

Orientador

Campina Grande, PB

Dezembro - 1990

P447c Perkusich, Maria Ligia Barbosa
Contribuicao para a construcao de sistemas baseados em conhecimento voltados para o conhecimento profundo / Maria Ligia Barbosa Perkusich. - Campina Grande, 1990.
102 f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Ferramentas de Programacao 2. Redes de Petri 3. Dissertacao I. Perkusich, Maria Ligia Barbosa, Dra. II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 004.4'23(043)

CONTRIBUIÇAO PARA A CONSTRUÇAO DE SISTEMAS BASEADOS EM CONHECIMENTO VOLTADOS PARA O CONHECIMENTO PROFUNDO

MARIA LIGIA BARBOSA PERKUSICH

DISSERTAÇAO APROVADA EM 26.12.1990

MISAEL ELIAS DE MORAIS, Dr. Ing.
Orientador

DAVID SIMONETTI BARBALHO, Dr. Ing.
Componente da Banca

JOAO MARQUES DE CARVALHO, Ph.D
Componente da Banca

HELIO DE MENEZES SILVA, M.Sc
Componente da Banca

Campina Grande, 26 de dezembro de 1990

AGRADECIMENTOS

Agradeço àquelas pessoas que direta ou indiretamente contribuíram para que este trabalho pudesse ser concluído.

Agradeço em especial:

Ao prof. MISAEL ELIAS DE MORAIS, pela orientação e ensinamentos.

Aos amigos JORGE CÉSAR ABRANTES DE FIGUEIREDO e ADRIANA GUERRA, pelo incentivo e apoio.

Ao prof. TOMÁZ DE C. BARROS pelo incentivo.

Ao prof. ANGELO PERKUSICH pelos ensinamentos, apoio, compreensão, e orientação.

A minhas irmãs LIGINEY BARBOSA E LAUDILENE BARBOSA, pelo apoio e contribuição.

A minha mãe Zefinha,
ao meu esposo Angelo e
aos meus filhos Mirko
e Andrei.

RESUMO

Este trabalho apresenta a contribuição à implementação de ferramentas para construção de sistemas baseados em conhecimento voltados para o conhecimento profundo. São discutidos ainda aspectos relacionados com a natureza do conhecimento.

As redes de Petri são empregadas como ferramenta formal para construir bases de conhecimento para sistemas baseados em conhecimento voltados para o conhecimento profundo. O conhecimento profundo é caracterizado por uma base de conhecimento construída a partir do modelo do sistema. Apresenta-se ainda as similaridades existentes entre as redes de Petri e os sistemas baseados em conhecimento, bem como uma ferramenta destinada a gerar a base de conhecimento a partir da definição matemática da rede de Petri e um motor de inferência destinado a executá-la.

Duas aplicações para as ferramentas desenvolvidades são também apresentadas.

ABSTRACT

This work presents a contribution to the implementation of tools to construct knowledge based systems oriented to deep knowledge. Aspects related to the knowledge nature are discussed.

Petri nets are used as a formal tool to construct knowledge bases for deep knowledge oriented knowledge based systems. The deep knowledge is characterized by a knowledge base constructed oriented for the model of the system. The similarities between Petri nets and knowledge based systems are also presented. A tool for knowledge base generation from the Petri net model of the system and a inference engine to execute it is also presented.

Two application of the implemented tools are also presented.

CONTEÚDO

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

1.1. Introdução

A importância dos sistemas especialistas vem crescendo consideravelmente. Diversas pesquisas e experiências aplicadas tem sido desenvolvidas, criando assim um horizonte cada vez maior para as aplicações.

O desenvolvimento da tecnologia vem provendo técnicas mais eficientes e poderosas, as quais propiciam um espectro de aplicação dos sistemas especialistas cada vez mais amplo, quer como diagnosticador, controlador, planejador, etc. Estas tecnologias proveem meios de raciocinar sob modelos profundos, algumas vezes obtidos através de metodologias formais, envolvendo uma grande quantidade de informações. Além disto, é cada vez mais importante a utilização dos sistemas especialistas integrado a ferramentas de auxílio mais eficientes, tais como aquelas de processamento de sinais e, instrumentação e controle, desta forma possibilitando extrair informações sobre o comportamento de sistemas ou dispositivos.

Uma tendência corrente é o desenvolvimento de sistemas voltados para o modelo do sistema ou dos dispositivos associados a ele utilizando o conhecimento profundo ou funcional [MILN 87, FINK 87, DVOR 87]. A aplicação do conhecimento empírico ou

superficial tem sido predominante no desenvolvimento das aplicações de sistemas especialistas. Entretanto, é crescente a a necessidade de desenvolver-se abordagens e métodos que possibilitem o desenvolvimento de sistemas especialistas ou baseados em conhecimento voltados para o conhecimento funcional ou profundo, baseado no modelo do sistema.

1.2. Objetivos

O objetivo principal deste trabalho é criar um conjunto básico de ferramentas que possibilitem implementar sistemas especialistas ou sistemas baseados em conhecimento com as características descritas no item anterior. Destacam-se então os seguintes pontos:

- implementação de um Editor de Regras e de um Motor de Inferência;
- criação de opção no Editor de modo a gerar automáticamente a Base de Conhecimento a partir de um modelo formal.

Com o objetivo de ilustar e comprovar a validade da ferramenta desenvolvida apresenta-se a aplicação da ferramenta desenvolvida a um pequeno sistema integrado para auxílio a análise de eletrocardiograma [PERK 89, DEEP 89, PERK 90].

De modo a construir um sistema onde a Base de Conhecimento seja voltada para o modelo do sistema optou-se por utilizar as redes de Petri para gerar a Base de Conhecimento, [PETE 81, REIS 82, BRAN 83, MURA 89]. A aplicação das redes de Petri é possível pelas similaridades existentes entre as redes de Petri e os

sistemas baseados em regras de produção. A aplicação das redes de Petri como base para a construção da Base de Conhecimento é desejável pois seus formalismos permitem tanto uma precisa construção do modelo quanto sua validação, sendo portanto desnecessário validar a Base de Conhecimento através de testes exaustivos. De forma a utilizar-se das redes de Petri para a construção da Base de Conhecimento implementou-se um Editor, no caso, um interpretador da rede que gera a partir da definição matricial formal da rede a Base de Conhecimento. Implementaram-se ainda alterações no Núcleo proposto de modo a possibilitar a execução das redes de Petri. Apresenta-se ainda o desenvolvimento de uma aplicação da ferramenta construída para executar redes de Petri descrevendo sistemas de transporte do tipo metrô [BARB 89, BARB 90, CARV 90, PERK 90].

1.3. Apresentação

A apresentação do trabalho está organizada de modo a mostrar o desenvolvimento dos sistemas e das aplicações, sendo assim dividiu-se a apresentação deste trabalho em 8 capítulos.

No capítulo 2 apresentam-se conceitos básicos sobre Inteligencia Artificial e Sistemas Especialistas e, enfatiza-se os conceitos de conhecimento compilado, conhecimento funcional, conhecimento estrutural e conhecimento comportamental.

No capítulo 3 apresentam-se as características desejadas para o sistema proposto e detalha-se cada módulo do sistema.

No capítulo 4 apresenta-se e descreve-se a implementação do

sistema proposto no capítulo 3.

No capítulo 5 apresenta-se uma aplicação para o sistema proposto: o desenvolvimento de um sistema para auxílio a análise de eletrocardiograma (ECG). Apresenta-se também uma seção com o sistema implementado, mostrando-se a sucessão de telas e os passos seguidos pelo sistema.

No capítulo 6 introduzem-se conceitos sobre redes de Petri e apresentam-se as similaridades existentes entre as redes de Petri e os sistemas baseados em regras de produção.

No capítulo 7 apresenta-se e descreve-se a implementação de um sistema baseado em regras de produção com o objetivo de avaliar e implementar modelos descritos por redes de Petri [PETE 81] podendo assim operar como controlador em operação normal ou como diagnosticar em caso de falha. Apresenta-se também o desenvolvimento de um Editor de Regras para construção de regras de produção a partir de uma matriz de incidência e de um Motor de Inferência para executar uma rede de Petri, onde a rede é descrita por um conjunto de regras de produção e a validade do emprego desse sistema na simulação do comando de um pequeno trecho de um sistema de transporte.

Finalmente, no Capítulo 8 apresentam-se conclusões e perspectivas.

CAPÍTULO 2

CONCEITOS BÁSICOS

2.1. Introdução

Neste capítulo apresentam-se os conceitos de Inteligência Artificial e Sistemas Especialistas e enfatiza-se os conceitos de conhecimento compilado, conhecimento funcional, conhecimento comportamental e conhecimento estrutural.

2.2. Conceitos Básicos

A Inteligência Artificial (IA) pode ser definida como a área da computação que desenvolve conceitos e métodos com o objetivo de fazer uma máquina se comportar inteligentemente, capaz de adquirir, transformar e aplicar conhecimento [BARR 81].

Sistema Especialista (SE) é um programa que contém informações sobre um certo campo de conhecimento e, quando interrogado, responde como se fosse um técnico especialista [SCHI 89].

Os sistemas que utilizam a filosofia dos sistemas especialistas, ou seja, são construídos baseados no mesmo paradigma, mas não no sentido clássico (MYCIN, PROSPECTOR, DENDRAL, etc [WATE 86]) não podem ser considerados como tais, são chamados de Sistemas Baseados em Conhecimento. Neste trabalho os dois conceitos são utilizados indistintamente com o sentido do

segundo.

Os Sistemas Especialistas são constituídos por duas partes: a Base de Conhecimento e a Máquina de Inferência.

A Base de Conhecimento é um banco de dados que armazena o conhecimento. Existem diversas metodologias para a representação do conhecimento [RICH 88, WATE 86]. A seguir destacam-se as três metodologias mais aplicadas.

A representação por regras de produção é a metodologia mais popular de representação do conhecimento. Uma regra representa uma porção do conhecimento através da lógica de predicados. As regras são estruturadas da seguinte forma:

SE < premissa >
ENTAO < consequente >.

Em um sistema especialista baseado em regras, o conhecimento é representado por um conjunto de regras que são verificadas contra uma coleção de fatos sobre a situação corrente. Quando um conjunto de fatos satisfaz as premissas de uma regra, as ações especificadas pelo consequente serão executadas. Essa ação ou conjunto de ações especificadas pelo consequente podem modificar o conjunto de dados do sistema, por exemplo adicionando novos fatos aos já existentes.

Uma outra metodologia de representação do conhecimento são as redes semânticas. As redes semânticas são grafos, isto é, o conhecimento é representado por um conjunto de nós ligados um ao outro através de um conjunto de arcos rotulados, onde os nós

representam os objetos, conceitos ou eventos e os arcos representam as relações entre os nós.

Uma das metodologias de representação do conhecimento de grande utilização são os frames. Um frame é basicamente uma descrição estruturada de um objeto ou uma classe de objetos, podendo inclui-se procedimentos, descrições dos membros de uma classe, etc. Um frame é dividido em "slots" que são os lugares aonde o conhecimento é armazenado dentro de um largo contexto criado pelo frame. Os frames, ao estilo das redes semânticas, se vinculam entre si por relações de pertinência e inclusão que permitem que os atributos e características de uma classe sejam "herdadas" por subclasses ou indivíduos membros, permitindo inferências do tipo: "se A está incluído em B e B está incluído em C, então A está incluído em C".

A Máquina de Inferência tem o objetivo de buscar o conhecimento, ordená-lo de uma maneira lógica e, a partir daí, ir direcionando o processo de inferência. Existem três grandes categorias de Máquina de Inferência: determinística, probabilistica e possibilística [STEP 87, SCHI 89].

Além das três grandes categorias citadas, existem três métodos básicos para construir uma Máquina de Inferência: encadeamento para a frente, encadeamento para trás e valor da regra. As diferenças entre esses métodos relacionam-se à maneira como a máquina busca o objetivo procurado [SCHI 89].

O Método do encadeamento para a frente (Forward-Chaining) parte de fatos para alcançar os objetivos, ou seja, ele usa os dados fornecidos pelo usuário para se movimentar na Base de Conhecimento até encontrar um ponto terminal, que é o objetivo.

No método do encadeamento para trás (Backward-Chaining) segue-se o procedimento inverso do encadeamento para a frente, ou seja, parte de uma hipótese (um objetivo) e pede informação para confirmar ou refutar.

O método do valor da regra é superior tanto ao encadeamento para a frente quanto ao encadeamento para trás, pois ele pede informações que têm grande importância de acordo com o estado atual do sistema. A teoria geral de operação é de que os pedidos do sistema, assim como sua próxima informação, removerão a maior incerteza do sistema.

2.3. Aspectos Relacionados com a Natureza da Base de Conhecimento

Na construção da Base de Conhecimento deve observar-se que o conhecimento pode ser representado em diferentes níveis [MILN 87]:

- conhecimento compilado;
- conhecimento funcional;
- conhecimento comportamental e;
- conhecimento estrutural.

De modo a simplificar a discussão sobre os aspectos relacionados a cada um dos níveis de conhecimento considerar-se-á um exemplo simples, um chuveiro elétrico.

Os sistemas que utilizam o conhecimento compilado são aqueles que fazem uso de informações superficiais ou empíricas. A maioria dos sistemas desenvolvidos atualmente baseiam-se neste tipo de conhecimento. A Base de Conhecimento é construída a partir da experiência de especialistas ou através de casos conhecidos. Considerando o exemplo do chuveiro pode-se observar que se a água não esquenta adequadamente e a temperatura ambiente da água é satisfatória então deve-se fechar ligeiramente o registro do chuveiro de modo a diminuir a quantidade de água.

O conhecimento funcional é voltado para o desempenho dos componentes associados a um dispositivo ou sistema, bem como seus relacionamentos, ou seja, enfatizam-se as intenções da aplicação de um determinado dispositivo ou sistema. No caso do chuveiro, sabe-se que a água é aquecida convenientemente se sua temperatura ambiente for compatível com a capacidade do aquecedor utilizado e se a pressão da água estiver a níveis tais que proporcione uma vazão compatível com a capacidade de aquecimento do aquecedor.

O conhecimento comportamental é aquele que detalha o comportamento físico de um sistema ou um dispositivo de forma qualitativa. No caso do chuveiro, sabe-se que o aquecedor funciona corretamente se estiver convenientemente alimentado por energia elétrica.

O conhecimento estrutural baseia-se em informações estruturais ou conectivas. Em muito sistemas, é difícil entender ou descrever o comportamento ou a função do sistema. Neste caso podem ser estabelecidas relações entre as partes constituintes. No

caso do chuveiro, sabe-se que para que ele funcione corretamente devem ser providos meios pelos quais a água possa chegar ao aquecedor sob uma pressão adequada e que o aquecedor esteja apto a aquecer a água.

De fato os sistemas especialistas ou baseados em conhecimento que utilizam os conhecimentos funcional, comportamental e estrutural são também chamados de sistemas baseados no modelo. Neste tipo de sistemas destacam-se os sistemas funcionais onde o modelo dos dispositivos ou de sistemas são usados de modo a gerar todos os possíveis cenários de falha. Ou seja é possível, a partir do modelo, introduzir uma falha em algum elemento do modelo e então simular seu funcionamento obtendo assim o resultando da falha. Voltando ao exemplo do chuveiro, a partir do modelo, pode-se causar uma falha no aquecedor ou no limitador de pressão e observar que estas duas situações levam a um aquecimento inadequado da água.

2.4. Sumário

Neste capítulo dilenearam-se os conceitos sobre Inteligência Artificial e Sistemas Especialistas. Estabeleceu-se o conceito de conhecimento compilado, conhecimento funcional, conhecimento comportamental e conhecimento estrutural voltado para o modelo, ou profundo.

CAPÍTULO 3

SISTEMA PROPOSTO

3.1. Características Desejadas para o Sistema

Geralmente os sistemas especialistas são constituídos por duas partes: a **Base de Conhecimento** e o **Motor de Inferência**. A Base de Conhecimento contém o conhecimento sobre uma determinada área e pode ser armazenado segundo uma das formas existentes para a representação do conhecimento [RICH 88, WATE 86]. O sistema proposto armazenará o conhecimento em forma de **regras de produção**. O **Motor de Inferência** é o responsável pelo contrôle e avaliação da Base de Conhecimento e pelo processo de inferência.

Considerando-se o apresentado acima, define-se a estrutura do sistema com quatro módulos básicos: **Interface, Editor, Núcleo e Base de Conhecimento**. A seguir são detalhados cada uma destes módulos.

3.2. Detalhamento do Sistema

3.2.1. Interface

A Interface tem por objetivo facilitar a comunicação entre o sistema e o usuário. Ela será feita basicamente através de menus e janelas.

3.2.2. Editor

O Editor foi definido com o objetivo de tornar o sistema dinâmico, isto é, estar sujeito a modificações que por ventura tornem-se indispensáveis à boa execução do sistema. O Editor é dividido em três partes: **Editor de Diagnósticos, Editor de Qualificadores e Editor de Regras.**

- **Editor de Diagnósticos**

O Editor de Diagnósticos tem o objetivo de incluir todos os possíveis **diagnósticos** para um determinado problema. Pode também, excluí-los, alterá-los e listá-los.

- **Editor de Qualificadores**

O Editor de Qualificadores tem o objetivo de incluir todos os **qualificadores**, ou seja, os dados necessários para a identificação de um determinado problema. Pode também, excluí-los, alterá-los e listá-los.

- **Editor de Regras**

O Editor de Regras tem o objetivo de construir regras de produção do tipo:

REGRA(NUM_REG, SE <PREMISSA> ENTÃO <CONSEQUENTE>).

Estas regras representarão todo o conhecimento a respeito de um problema. Pode também, excluí-las, alterá-las e listá-las. Simultâneamente a construção do Arquivo de Regras, o Editor de Regras também constrói 3 arquivos que serão utilizados pelo Núcleo

para otimizar o processo de inferência: **Arquivo de Influência, Arquivo de Contagem e Arquivo de Prioridades.**

3.2.3. Núcleo

O Núcleo do sistema tem por objetivo controlar e avaliar o conteúdo da Base de Conhecimento e da Base de Trabalho e a partir daí direcionar o processo de inferência. O Núcleo é constituído basicamente por 4 sub_módulos principais: **Motor de Inferência, Diagnósticador, Justificador e Base de Trabalho.**

- **Motor de Inferência**

O Motor de Inferência será o responsável pela marcação e execução das regras de produção. Ele será dividido em dois sub-módulos: Escalonador e Acionador.

- **Escalonador**

O Escalonador tem o objetivo de pesquisar e selecionar todos os dados da Base de Conhecimento e da Base de Trabalho que se relacionam e, a partir daí fazer o escalonamento, ou seja, deverá marcar todas as regras que devem ser executadas pelo Acionador.

O mecanismo utilizado para marcar as regras será a Contagem Regressiva de Regras [ARAR 89] com algumas modificações. Estas modificações tornaram-se necessárias para que o Motor de Inferência também possa tratar com regras que contenham disjunções.

A contagem regressiva de uma regra é o número de dados que ainda devem ser verificados para que uma regra seja marcada.

As regras que contenham apenas **conjunções** (e) seraõ marcadas quando **todas** as suas premissas forem verificadas e, as regras que contenham **disjunções** (ou) serão marcadas quando **pelo menos uma** de suas premissas for verificada.

Quando todas as regras possíveis forem marcadas, o Núcleo deverá ter acesso a alguma informação que o possibilite decedir qual a primeira regra que deverá ser executada. Este Núcleo utilizará um mecanismo de definição de precedencia de disparo que constituí-se na associação de prioridades a cada regra.

• **Acionador**

O Acionador executará todas as regras que foram marcadas pelo Escalonador.

O mecanismo de inferência utilizado para executar as regras será o encadeamento progressivo, ou seja, parte de um conjunto de fatos, os quais são identificados pelo Escalonador, verifica em relação a Base de Conhecimentos e, tenta chegar a alguma conclusão.

• **Diagnósticador**

O Diagnósticador será o responsável pela elaboração do diagnóstico final.

• **Justificador**

O Justificador poderá responder a seguinte pergunta: **como**.

como é uma opção que será oferecida ao usuário sempre que ele deseja saber em que o sistema baseou-se para obter um diagnóstico.

• **Base de Trabalho**

Na Base de Trabalho serão armazenados, em forma de predicados Prolog (Claúsulas de Horn), tanto os dados fornecidos pelo arquivo Fatos, quanto os dados gerados pelo próprio Núcleo. Estes predicados também serão utilizados pelo Escalonador durante o processo de escalonamento das regras.

3.2.4. Base de Conhecimento

A Base de Conhecimento do sistema será construída através do Editor de Regras e armazenará todo o conhecimento a respeito de um problema na forma de regras de produção do tipo:

REGRA(NUM_REG, SE <PREMISSA> ENTÃO <CONSEQUENTE>).

O **NUM_REG** (número da regra), servirá para rotular a regra visando facilitar o seu uso no processo de escalonamento, execução, explanação, etc. A **PREMISSA** será composta por um conjunto de fatos e/ou diagnósticos. O **CONSEQUENTE** será constituído por um conjunto de ações que poderão ser diagnósticos ou novos fatos.

Estas regras serão verificadas em relação a uma coleção de dados (Base de Conhecimento e Base de Trabalho) e quando todas as

premissas forem satisfeitas a regra será marcada para posteriormente ser executada pelo Motor de Inferência.

Além do Arquivo de Regras a Base de Conhecimento contém mais 3 arquivos que serão utilizados pelo Núcleo. Estes arquivos são:

- **Arquivo Contagem**, indica a quantidade de dados na premissa de uma regra;

- **Arquivo Influência**, indica quais os fatos e/ou diagnósticos que influênciam uma regra e

- **Arquivo Prioridade**, indica a prioridade de uma regra.

Estes arquivos serão acessados pelo Núcleo e transformados em predicados Prolog.

3.3. Sumário

Neste capítulo apresentou-se as características desejadas para o sistema proposto e detalhou-se cada módulo do sistema.

CAPÍTULO 4

SISTEMA IMPLEMENTADO

4.1. Introdução

Neste capítulo apresenta-se detalhes da implementação do sistema proposto. Mostra cada módulo implementado e apresenta a estrutura de dados utilizada.

4.2. Descrição do Sistema

Na atual implementação, o sistema apresenta 3 etapas de operação. A primeira etapa é a de aquisição de dados básicos correspondendo a fatos voluntariados por um módulo externo como será mostrado no capítulo 5. A segunda fase é a de edição de regras de produção através do módulo **Editor** e a última é a de execução das regras através do módulo **Núcleo.**

Como foi mostrado no capítulo anterior o sistema é constituído basicamente por 4 módulos: Interface, Editor, Núcleo e Base de Conhecimento. A seguir detalhar-se-á cada módulo.

4.2.1. Interface

A Interface foi implementada em linguagem C para microcomputadores de 16 bits. Ela é constituída basicamente de menus e janelas. As escolhas devem ser feitas através da inicial da opção ou através das teclas de movimento do cursor para cima, para baixo, próxima página e página anterior seguido de <ENTER>.

O sistema é formado por 26 telas.

4.2.2. Editor

O Editor foi implementado em linguagem C. Ele é constituído por três sub-módulos: Editor de Diagnósticos, Editor de Qualificadores e Editor de Regras. A seguir detalhar-se-á cada sub-módulo.

- **Editor de Diagnósticos**

O Editor de Diagnósticos utilizará a estrutura **diagnóstico** e oferece quatro opções: Incluir, Excluir, Atualizar e Listar Diagnósticos. A estrutura diagnóstico possuí o seguinte formato:

```
struct diagnosticos {
                     char nome[25];
                     int numero;
struct diagnosticos  *prox;
struct diagnosticos  *ant;};
struct diagnosticos  *dstart;
struct diagnosticos  *dlast;
```

onde, **nome** : nome do diagnóstico;
número : número do diagnóstico;
***prox** : apontador para o próximo diagnóstico;
***ant** : apontador para o diagnóstico anterior;
***dstart** : apontador para o primeiro diagnóstico;
***dlast** : apontador para o último diagnóstico.

• **Incluir Diagnósticos**

O usuário indica os nomes dos diagnósticos e eles são armazenados na Base de Conhecimento.

• **Excluir Diagnósticos**

O usuário indica o nome de um diagnóstico e ele é retirado da Base de Conhecimento. Todas as regras que utilizam esse diagnóstico também serão retiradas da Base de Conhecimento.

• **Atualizar Diagnósticos**

O usuário indica o nome de um diagnóstico e ele torna-se disponível para atualizações, ou seja, o nome do diagnóstico poderá ser modificado. Todas as regras que utilizam esse diagnóstico também serão atualizadas.

• **Listar Diagnósticos**

O usuário indica se deve listar na impressora ou na tela e todos os diagnósticos serão listados.

• **Editor de Qualificadores**

O Editor de Qualificadores utilizará a estrutura **qualificadores** e oferece quatro opções: Incluir, Excluir, Atualizar e Listar Qualificadores. A estrutura **qualificadores** possuí o seguinte formato:

```
struct atributo {
                    char nome_a[25];
                    int numero_a;
```

```
        struct atributo  *ptr;};
        struct qualificador {
                              char nome_q[25];
                              int numero_q;
                              struct atributo *a;
        struct qualificador  *prox;
        struct qualificador  *ant;};
        struct qualificador  *qstart;
        struct qualificador  *qlast;
```

onde, **nome_a** : nome de um atributo para o qualificador;

numero_a : número de um atributo para o qualificador;

***ptr** : apontador para o próximo atributo;

nome_q : nome do qualificador;

numero_q : número do qualificador;

***a** : apontador para a estrutura atributo;

***prox** : apontador para o próximo qualificador;

***ant** : apontador para o qualificador anterior;

***qstart** : apontador para o primeiro qualificador;

***qlast** : apontador para o último qualificador.

- **Incluir Qualificadores**

O usuário indica os nomes dos qualificadores e seus atributos e eles são armazenados na Base de Conhecimento.

- **Excluir Qualificadores**

O usuário indica o nome de um qualificador e ele é retirado da Base de Conhecimento. Todas as regras que utilizam esse qualificador também serão retiradas da Base de Conhecimento.

• **Atualizar Qualificadores**

O usuário indica o nome de um qualificador e ele poderá ser atualizado, ou seja, o usuário pode mudar o nome do qualificador, pode mudar o(s) nome(s) do(s) atributo(s) do qualificador ou adicionar novo(s) atributo(s) ao qualificador. Se o usuário mudar o nome do qualificador ou o nome de um atributo do qualificador todas as regras que utilizam esse qualificador também serão atualizadas.

• **Listar Qualificadores**

O usuário indica se deve listar na impressora ou na tela e todos os qualificadores serão listados.

• **Editor de Regras**

O Editor de Regras é o responsável pela construção de regras de produção do tipo:

regra(<NUM_REG>,se <PREMISSA> entao <CONSEQUENTE>).

as quais são armazenadas em estruturas regras como definidas a seguir:

```
struct regra_premissa {
                        char premissa[50];
                        char conectivo[2];
struct regra_premissa  *ptr;};
struct regra_conseq {
                      char consequente[50];
struct regra_conseq  *ptr;};
```

```
struct regras {
                char *regra;
                char numero[3];
                char *se;
                struct regra_premissa *rp;
                char *entao;
                struct regra_conseq *rc;
     struct regras  *prox;
     struct regras  *ant;};
     struct regras  *rstart;
     struct regras  *rlast;
```

onde, **premissa** : uma premissa da regra;

conectivo : um conectivo da regra (conjunções[e], disjunções[ou], fecha[)];

***ptr** : apontador para a próxima premissa;

consequente : um consequente da regra;

***ptr** : apontador para o próximo consequente;

***regra** : apontador para a constante "regra (";

numero : número da regra;

***se** : apontador para a constante "se";

***rp** : apontador para a estrutura regra_premissa;

***entao** : apontador para a constante "entao";

***rc** : apontador para a estrutura regra_consequente;

***prox** : apontador para a próxima regra;

***ant** : apontador para a regra anterior;

***rstart** : apontador para a primeira regra;

***rlast** : apontador para a última regra.

O <NUM_REG> é um número **inteiro** que serve para rotular uma regra.

A <PREMISSA> pode conter conjunções(,) e/ou disjunções(;) da **estrutura regra_premissa**, que é parte da **estrutura regras**, a qual, pode armazenar dois tipos de estruturas, a saber: **qualificador** e **diagnóstico.**

O <CONSEQUENTE> pode conter conjunções da **estrutura regra_conseq**, que é parte da **estrutura regras**, a qual pode armazenar dois tipos de estruturas: **qualificador** e **diagnóstico.**

Simultâneamente a construção do Arquivo de Regras, o Editor de Regras constrói, automaticamente, três arquivos que são utilizados pelo **Núcleo** para otimizar o processo de inferência: **Arquivo de Influência, Arquivo de Contagem e Arquivo de Prioridades.**

O Arquivo de Influência armazena informações que indicam **quais** os dados que influênciam uma regra, ou seja, todos os dados que fazem parte da premissa de uma regra. Essas informações são armazenadas em estruturas do tipo **influência**, como definida a seguir:

```
struct influencia {
                    char influencia[12];
                    char nome[42];
                    int num_regra;
                    char fecha[3];
                   };
```

onde, **influencia** : armazena a constante "influencia";

nome : armazena o qualificador ou o diagnóstico que influencia a regra;

num_regra : número da regra influenciada;

fecha : fecha a estrutura.

O Arquivo de Contagem armazena informações que indicam a quantidade de fatos e/ou diagnósticos que influênciam cada regra e mostra quando a regra tem apenas conjunções (1) ou quando tem pelo menos uma disjunçaõ (0). Essas informações são armazenadas em estruturas do tipo **contagem**, como definida abaixo:

```
struct contagem {
                 char contagem[10];
                 int  contador;
                 char virgula1[2];
                 int num_regra;
                 char virgula2[2];
                 int sinal;
                 char fecha[3];
                };
```

onde, **contagem** : armazena a constante "contagem";

contador : quantidade de dados que influenciam a regra;

virgula1 : uma virgula;

num_regra: número da regra;

virgula2 : uma virgula;

sinal : sinal que indica se a regra tem ou não disjunção;

fecha : fecha a estrutura.

O Arquivo de Prioridades armazena informações que indicam a prioridade de cada regra. Essas informações são armazenadas em estruturas do tipo **prioridade**, como definida abaixo:

```
struct prioridade {
                    char prioridade[7];
                    int prior;
                    int num_regra;
                    char fecha[3];
                  };
```

onde, **prioridade** : armazena a constante "prioridade";
prior : é a prioridade da regra;
num_regra : é o número da regra;
fecha : fecha a estrutura.

O Editor de Regras oferece quatro opções: Incluir, Excluir, Atualizar e Listar Regras.

• **Incluir Regras**

O usuário indica os qualificadores, os diagnósticos e os conectivos que devem fazer parte da premissa e, os qualificadores e os diagnósticos que devem fazer parte do consequente e, a partir daí, as regras serão formadas e posteriormente incluídas no Arquivo de Regras. Para todo qualificador ou diagnóstico que for selecionado, serão criadas, automaticamente, as influências (Arquivo Influência) e os contadores (Arquivo Contagem). Para cada regra incluída, será solicitado ao usuário que entre com a prioridade para a regra (Arquivo Prioridade).

• **Excluir Regras**

O usuário indica o número de uma regra e ela será excluída da Base de Conhecimento.

• **Atualizar Regras**

O usuário indica o número de um regra e ela torna-se disponível para atualizações, ou seja, o usuário poderá adicionar ou retirar premissas, adicionar ou retirar consequentes, mudar a prioridade e/ou mudar os conectivos. As influências e os contadores serão atualizados automaticamente.

• **Listar Regras**

O usuário indica se deve listar na impressora ou na tela e todas as regras serão listadas.

• **Funcionamento do Editor**

Neste item apresenta-se basicamente os passos seguidos na execução do Editor durante uma seção.

PASSO 1: Inicialmente pergunta-se se o usuário deseja criar um arquivo novo ou se deseja utilizar um já existente. Caso ele deseje abrir um arquivo já existente, o usuário deve entrar com o nome do arquivo e, se ele existir, o mesmo será aberto e seus dados serão recuperados. Caso deseje um novo arquivo, o usuário entra com o nome do novo arquivo e, se ele ainda não existir, o mesmo será criado.

PASSO 2: Caso o usuário deseje trabalhar com um arquivo já existente apresenta-se um menu no qual o usuário escolhe uma das seguintes opções: **Incluir, Excluir, Atualizar ou Listar** e em seguida executa-se o **passo 3**. Caso seja arquivo novo, o **passo 3** será apresentado. Para sair teclar <ESC> e a seção será encerrada.

PASSO 3: Apresenta-se um menu no qual o usuário deverá escolher uma das seguintes opções: **Diagnósticos, Qualificadores ou Regras.** O usuário poderá escolher qualquer opçao caso não esteja utilizando um arquivo novo. Caso seja arquivo novo, terá que escolher inicialmente uma das duas primeiras opções, pois ele deverá incluir diagnósticos e qualificadores antes de incluir regras. Para sair teclar <ESC> e o **passo 2** será executado.

Considerando-se que o usuário deseje criar um arquivo novo.

Se a escolha foi **Diagnósticos** o **passo 4** será executado.
Se a escolha foi **Qualificadores** o **passo 5** será executado.
Se a escolha foi **Regras**, o **passo 6** será executado.

PASSO 4: O usuário deverá entrar com os nomes dos diagnósticos e <ENTER> para sair. Novos diagnósticos poderão ser incluidos posteriormente. O **passo 3** será repetido.

PASSO 5: O usuário deverá entrar com os nomes dos qualificadores e seus atributos e <ENTER> para sair. Novos qualificadores poderão ser incluídos posteriormente. O **passo 3** será repetido.

PASSO 6: Será apresentado um menu com as seguintes opções: **qualificadores ou diagnósticos.** O usuário deverá selecionar os qualificadores ou diagnósticos que devem fazer parte das premissas de uma regra. Em seguida o **passo 7** será executado.

PASSO 7: Será apresentado um menu com as seguintes opções: **(e), (ou) e (fecha).** O usuário deverá selecionar um conectivo para a premissa da regra ou deverá fechar as premissas. Se as premissas não foram fechadas, ou seja, as premissas não foram completadas, o **passo 6** será repetido, senão o **passo 8** será executado, ou seja, o consequente da regra será construido .

PASSO 8: Será apresentado um menu com as seguintes opções: **diagnósticos ou qualificadores.** O usuário deverar selecionar um diagnóstico ou um qualificador que deva fazer parte do consequente de uma regra. Será perguntado ao usuário se ele deseja adicionar mais consequentes ou não. Se a resposta for positiva este passo será repetido, senão a prioridade da regra será solicitada. A seguir será perguntado se ele deseja incluir novas regras. Se a resposta for positiva o **passo 6** será repetido, senão o **passo 3** será repetido.

4.2.3. Núcleo

O Núcleo foi implementado utilizando-se a linguagem Arity Prolog. Ele é constituído basicamente por 4 sub_módulos: Base de Trabalho, Motor de Inferência, Diagnósticador e Justificador. A seguir, descrever_se cada sub-módulo.

• **Base de Trabalho**

Este módulo é constituído durante a execução do Núcleo. O Núcleo acessa o Arquivo de Fatos e o armazena em forma de **predicados fato(dado)** na Base de Trabalho. Durante a execução das regras, poderão ser criados novos **predicados fato(dado)** e **predicados diagnóstico(dado)**, os quais também serão armazenados na Base de Trabalho, para posteriormente serem utilizados pelo Motor de Inferência. A Base de Trabalho é dividida em 2 conjuntos de predicados: fato(dado) e diagnóstico(dado).

• **fato(dado)** armazena os fatos fornecidos pelo Arquivo de Fatos gerado por um módulo externo (ver Capítulo 5), bem como os gerados pelo próprio Núcleo. Estes predicados serão utilizados pelo Escalonador durante o processo de escalonamento das regras. Exemplo: fato(ritmo(regular)), ou seja, o ritmo é regular.

• **Diagnóstico(dado)** armazena todas as conclusões que o Núcleo consegue alcançar. Estes predicados serão utilizados pelo Escalonador durante o processo de escalonamento das regras e também será requisitado pelo Diagnósticador para elaboração do diagnóstico final. Exemplo: diagnóstico(taquicardia_sinusal).

• **Motor de Inferência**

O Motor de Inferência é o responsável pela execução das regras de produção e foi dividido em dois sub-módulos: Escalonador e Acionador.

• **Escalonador**

O Escalonador é o responsável pelo escalonamento de todas as regras que devam ser executadas, ou seja, ele pesquisará e analizará a Base de Trabalho e a Base de Conhecimento e marcará todas as regras que devem ser executadas pelo Acionador.

Sempre que um predicado **fato** for instanciado, ele será excluído e os predicados **contagem** das regras influenciadas por tal fato são atualizadas. No exemplo abaixo, quando **fato(frequencia(maior_100)** for instanciado, os predicados **influencia** correspondentes serão excluídos e fornecerão os números das regras influenciadas por ele, no caso, a regra R1, e o **contador** dos predicados **contagem** de todas as regras influenciadas serão diminuídas de um, ou seja, o **contador** de R1 passará a ser 1. Sempre que o **contador** tornar-se nulo, a regra será marcada, isto é, a seguinte sentença é colocada na base de trabalho: **Executável(Num_reg)**, onde Num_reg é o número da regra cujo contador é 0.

Terminada a fase de leitura dos dados (predicados fato(dado) e diagnostico(dado)), todas as sentenças **Executável(Num_reg)** são verificadas em relação aos predicados **prioridade** para formarem **listas de regras executáveis** que conterão o número da regra e sua prioridade. Em seguida, estas listas serão ordenadas em ordem decrescente da prioridade e formarão uma **lista de regras executáveis por prioridade**. Finalmente, a primeira regra da lista, ou seja, a de maior prioridade, será acionada pelo **Acionador**.

As regras são fornecidas pelo Arquivo de Regras, as informações sobre contagem de regras são fornecidas pelo Arquivo de Contagem, as informações sobre as regras cujas contagens são influênciadas pela inserção de um fato no contexto são fornecidas pelo Arquivo de Influência e as informações sobre as prioridades das regras são fornecidas pelo Arquivo de Prioridades, todos gerados pelo Editor de Regras. Exemplos destas informações:

```
regra(R1, se (ritmo(regular), frequencia(maior_100))
então ( guarde_d(taquicardia_sinusal))).
regra(R2, se (ritmo(regular), frequencia(menor_60))
então ( guarde_f(onda_p(normal)))).

contagem(2,R1,0).
contagem(2,R2,0).

influencia(ritmo(regular),R1).
influencia(ritmo(regular),R2).
influencia(frequencia(maior_100),R1).
influencia(frequencia(menor_60),R2).

prioridade(100,R1).
prioridade(200,R2).
```

Na regra R1 o predicado guarde_d no consequente coloca na Base de Trabalho um diagnóstico se o mesmo ainda não existir e, na regra R2 o predicado guarde_f coloca na Base de Trabalho um novo fato, se o mesmo ainda não existir.

• Acionador

O Acionador é o responsável pela execução de todas as regras que foram marcadas pelo Escalonador.

O mecanismo de inferência utilizado para provar as regras foi o **encadeamento progressivo**, ou seja, parte de um conjunto de dados, os quais são identificados pelo Escalonador, e tenta chegar a uma conclusão.

O Acionador acessa a **lista de regras executáveis por prioridade** criada pelo Escalonador e executa a primeira regra da lista através do seguinte engenho de inferência:

```
aciona(X) :- regra(X, se Conds então Ações),
             call(Conds), call(Ações).
```

Se esta regra modificar a base de trabalho, os predicados contagem das regras influenciadas por estes dados são atualizados e caso alguma regra seja marcada, as listas das regras executáveis e a lista das regras executáveis por prioridade serão refeitas e a próxima de maior prioridade será acionada.

Se esta regra não modificar a base de trabalho, a próxima regra da lista das regras executáveis por prioridade será executada.

Após todas as regras executáveis terem sido executadas o diagnóstico final será elaborado.

• **Diagnosticador**

Após a execução de todas as regras executáveis este módulo será acionado. Ele buscará tudo o que estiver nos predicados **diagnóstico(dado)** e elaborará a partir daí a impressão do laudo final.

• **Justificador**

O Justificador poderá responder a seguinte pergunta: como.

como é uma opção que será oferecida ao usuário sempre que ele deseje saber em que o sistema baseou-se para obter um diagnóstico.

• **Funcionamento do Núcleo**

Neste item apresenta-se basicamente todos os passos seguidos pelo Núcleo durante uma seção.

PASSO 1: Inicialmente o Núcleo acessa os arquivos Fatos, Influência, Contagem, Prioridade e Regras e criará os seguintes predicados: fatos, influ_cont, contagem, prioridade e regras, respectivamente. O passo 2 será executado.

PASSO 2: O Escalonador acessa um dado da base de trabalho.

PASSO 3: O Escalonador identifica as regras que são influenciadas pelo dado do passo 2 através dos predicados **influencia(dado,Num_reg).**

PASSO 4: O Escalonador decrementa o **contador** dos predicados **contagem(contador,regra, sinal)** das regras influenciadas e verifica através do **sinal** se a regra tem disjunções ou não. Se tiver disjunções a regra é marcada, ou seja, a cláusula **executável(regra)** será incluida. Senão, apenas quando o **contador** for zero a regra será marcada.

PASSO 5: Repete os passos 2), 3) e 4) até que todos os dados tenham sidos pesquisados.

PASSO 6: Todas as cláusulas **executável(regra)** serão verificadas em relação aos predicados **prioridade(prior, regra)** correspondentes para formarem listas de regras executáveis que conterão o número da regra e a sua prioridade. O **passo 7** será executado.

PASSO 7: As listas de regras executáveis formarão uma **lista de regras executáveis por prioridade** que será ordenada em ordem decrescente de sua prioridade. O **passo 8** será executado.

PASSO 8: A primeira regra da **lista de regras executáveis por prioridade** será executada pelo Acionador. Se esta regra adicionar algum fato ou diagnóstico o **passo 2** será novamente executado, senão o **passo 9** será executado.

PASSO 9: O Diagnosticador elaborará o diagnóstico final e a seção será encerrada.

4.2.4. Base de Conhecimento

A Base de Conhecimento foi construída através do Editor de Regras e contém regras de produção que apresentam-se da seguinte forma:

regra(<NUMERO_DA_REGRA> se (PREMISSA) entao (CONSEQUENTE).

Estas regras são verificadas em relação a uma coleção de predicados fato(dado) e/ou diagnóstico(dado) e em relação aos predicados **influência**(dado, Num_reg) e **contagem(contador, Num_reg, sinal)**. Se um conjunto de fatos e/ou diagnósticos satisfaz as premissas de uma regra, ela será marcada e, quando não existir mais predicados fato(dado) e diagnóstico(dado) a serem verificados todas as regras ativadas serão acionadas, ou seja, as ações especificadas pelo consequente serão executadas. Se um diagnóstico for premissa de uma regra ele será excluído do conjunto de predicados **diagnóstico(objeto)** para facilitar a obtenção do diagnóstico final.

A Base de Conhecimento conta com o auxílio de três arquivos: **Arquivo de Contagem, Arquivo de Influência** e o **Arquivo de Prioridade.**

Arquivo de Contagem - indica a quantidade de premissas que influencia a regra e se ela contém apenas disjunções ou não. Exemplo: contagem(3,R2,1), ou seja, a regra R2 contém 3 premissas e é uma regra que contém apenas conjunções.

Arquivo de Influência - indica quais os dados que influênciam uma regra. Exemplo: influência(ritmo(regular), R2), ou seja, o fato **ritmo(regular)** influencia a regra R2).

Arquivo de Prioridade - indica a prioridade de uma regra. Exemplo: prioridade(100, R1), ou seja a prioridade da regra R1 é 100.

Estes arquivos são acessados pelo Núcleo e transformados nos seguintes predicados: **contagem(contador, Num_reg, sinal), influência(dado, Num_reg), prioridade(prior, Num_reg).**

4.3. Dados sobre a implementação

O sistema consta de 2 arquivos, num total de 208 Kbytes, distribuídos do seguinte modo:

- Editor (incluindo Interface): fonte 4500 linhas (C)
  código 68 Kbytes
- Núcleo: fonte 150 linhas (Prolog)
  código 140 Kbytes

4.4. Sumário

Neste capítulo descreveu-se as partes implementadas do sistema proposto, quais sejam, a Interface, o Editor, o Núcleo e a Base de Conhecimento.

CAPÍTULO 5

APLICAÇÃO EM ELETROCARDIOGRAFIA

5.1. Introdução

Neste capítulo descreve-se o desenvolvimento de uma aplicação para a ferramenta apresentada no capítulo 4: Sistema para auxílio a análise de eletrocardiograma (ECG).

O desenvolvimento da tecnologia de instrumentação eletrônica tem levado ao aumento da complexidade e precisão dos sistemas de medição [SZIT 88]. Como consequência direta do aumento da complexidade surge a necessidade dos usuários disporem de ferramentas mais eficientes para auxiliar a análise e interpretação dos dados experimentais. Uma solução natural para contornar os problemas acima citados é integrar as técnicas de processamento digital de sinais e das ferramentas de inteligência artificial, notadamente os sistemas especialistas ou sistemas baseados em conhecimento [WATE 86], aos sistemas de medição com o objetivo de auxiliar na tarefa de analizar e interpretar os resultados de um experimento de medição.

Na área de diagnóstico médico, onde o desenvolvimento dos sistemas de instrumentação biomédica é notório, dois aspectos devem ser considerados: O primeiro relaciona-se com o desenvolvimento dos sistemas médicos computadorizados com o objetivo de aumentar o desempenho da análise dos dados

experimentais através da agregação de facilidades para processamento de dados e capacidade de reações inteligentes [PAPP 88]. O segundo aspecto a ser considerado é a necessidade de integrar-se aos sistemas de consulta médica utilizando inteligência artificial existentes [KULI 88] alguma capacidade para processamento dos sinais medidos. A análise destes dois aspectos leva à necessidade de sistemas de instrumentação onde uma abordagem utilizando processamento digital de sinais e sistemas especialistas possam ser integrados.

5.2. Descrição do Sistema

A seguir descrevem-se os módulos de software do sistema como apresentados em [PERK 89]. Todas os módulos forão implementados utilizando-se linguagem C e Prolog.

5.2.1. Gerenciamento de Sinal

• Aquisição do Sinal de ECG

O módulo para aquisição do sinal de ECG foi implementado em linguagem C. A temporização da aquisição dos dados foi feita a partir do relógio em tempo real do microcomputador. A rotina de aquisição a partir das informações sobre a derivação a ser amostrada e discretizada e do número de amostras (tempo de amostragem) executa o contrôle de um amplificador de ECG com chaveamento eletrônico e um conversor A/D. As amostras do sinal são armazenadas em uma estrutura de dados contendo informações sobre: derivação, número de amostras e amostras. A frequência de amostragem pode ser programada pelo usuário, neste trabalho a taxa

de amostragem está definida em 200 Hz.

• **Gerênciamento de Banco de Dados**

O gerenciamento do banco de dados (baseado no modelo relacional) foi implementado em linguagem C. As relações existentes no banco de dados dizem respeito a dados cadastrais como: número, nome, idade, sexo e peso do paciente; dados de histórico: indicação clínica de exames, dor précordial, arritmia de pulso e origem do paciente (ambulatório, CTI, etc); e, dados de histórico de ECG, contendo: número do paciente, data, hora, número de amostras por derivação e as amostras das derivações, bem como um Arquivo de Fatos, descrito a seguir.

• **Processamento do Sinal de ECG**

Neste módulo são implementadas as funções de processamento do sinal de ECG, extração de parâmetros, e criação de um Arquivo de Fatos para o Sistema Especialista, em linguagem C. Os detalhes de implementação deste módulo fogem do escopo deste trabalho e podem ser encontrados em [PERK 90].

5.2.2. Interface Homem-Máquina

A interface homem-máquina é baseada em menus e janelas, de modo a prover flexibilidade de operação. A interface provê ainda acesso a recursos gráficos para visualização e impressão do ECG. Na Figura 5.1. apresenta-se a impressão do sinal após o processamento descrito no item anterior.

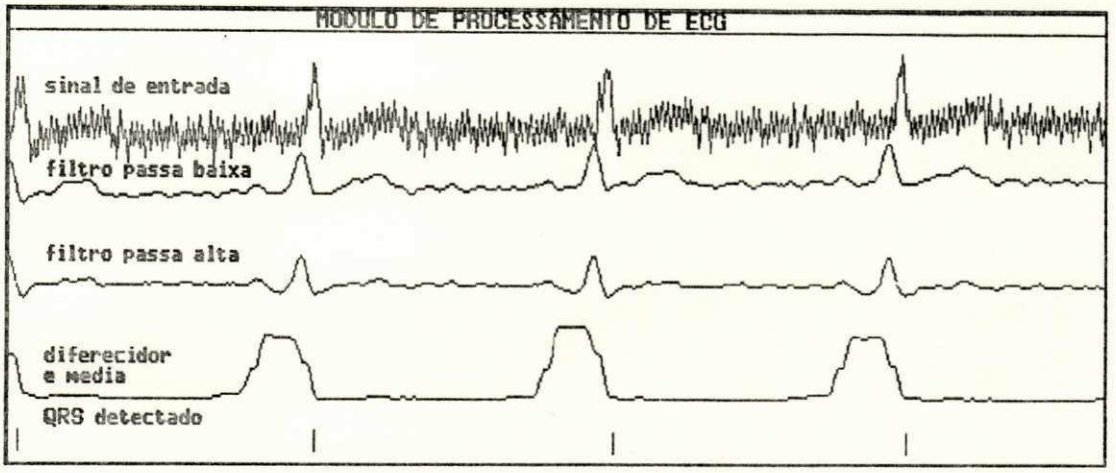
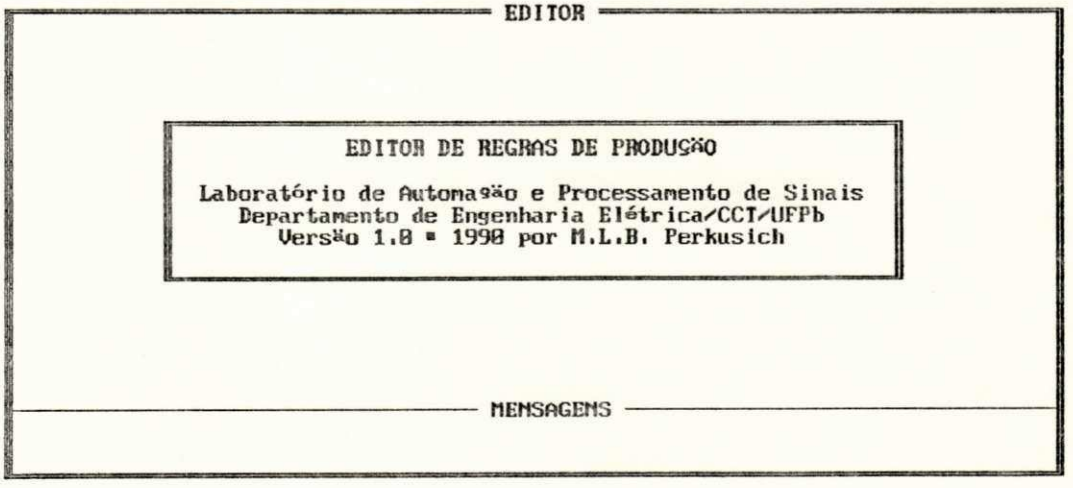


Figura 5.1.: Tela do Módulo de Processamento

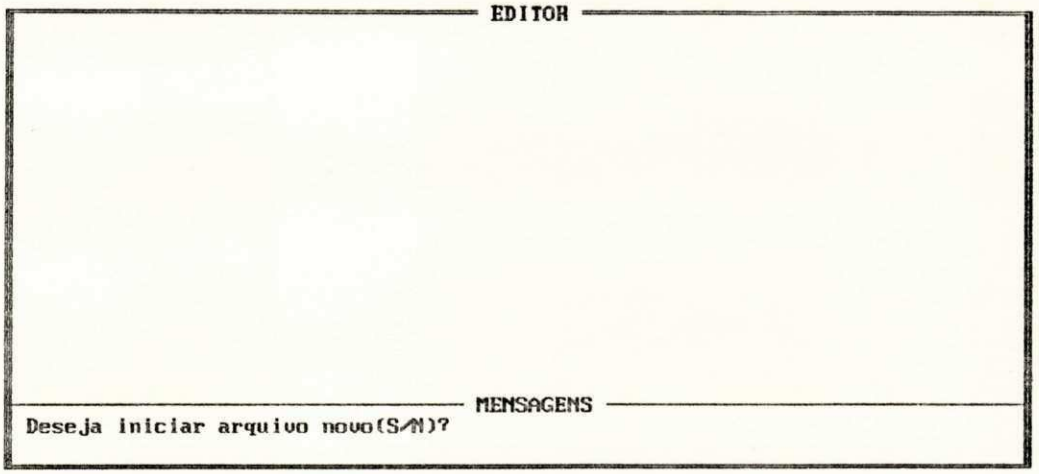
5.3 Seção com o Editor

Neste ítem apresenta-se uma seção com o Editor descrito no capítulo 2, mostrando-se a sucessão de telas apresentadas.

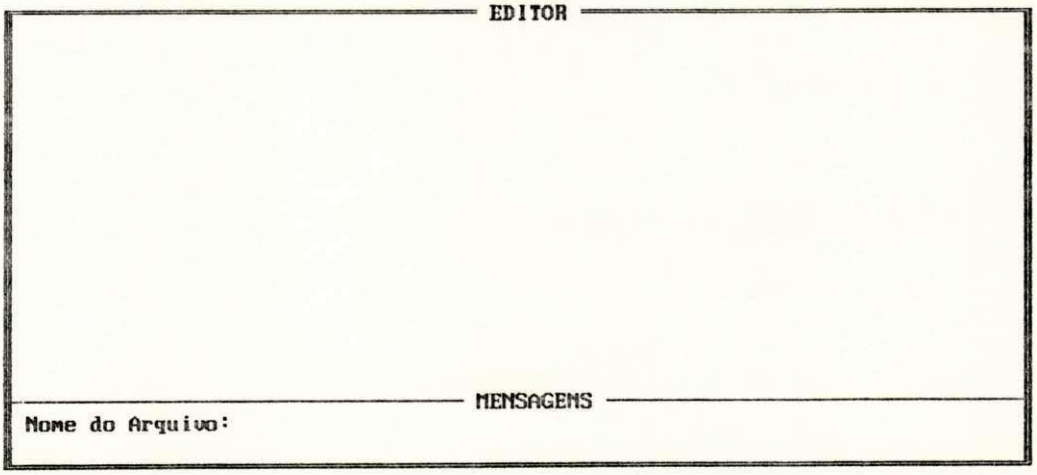
Inicialmente o Editor mostrará sua tela de apresentação;



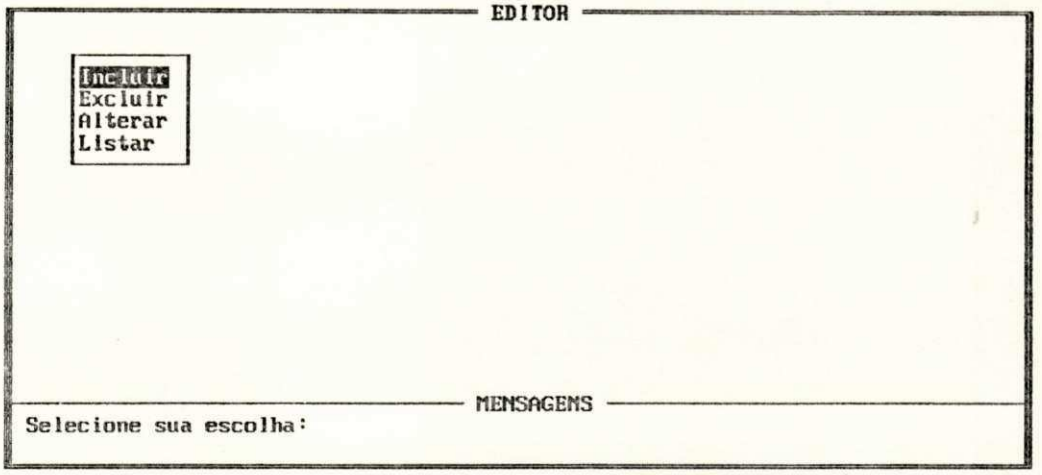
A seguir perguntará se o usuário deseja criar um novo arquivo.



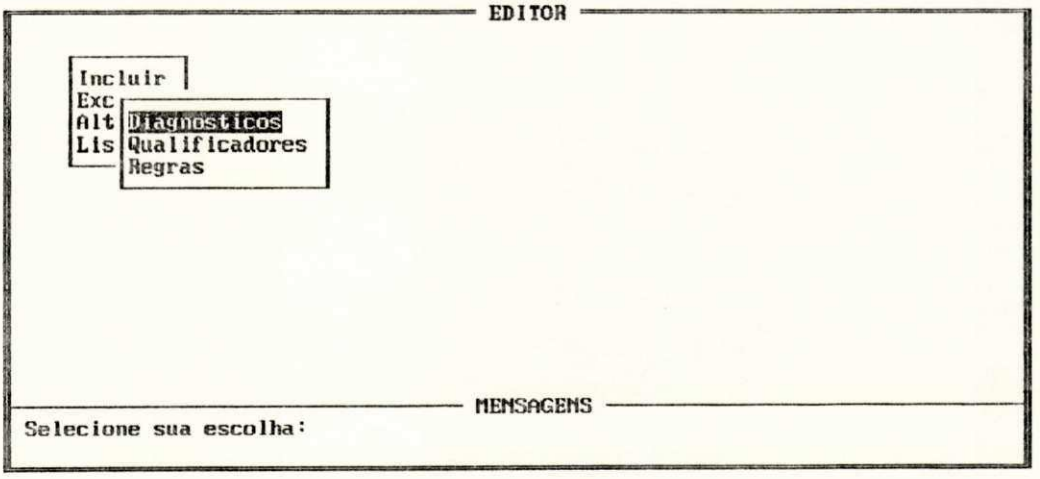
O usuário deverá entrar com o nome do arquivo.



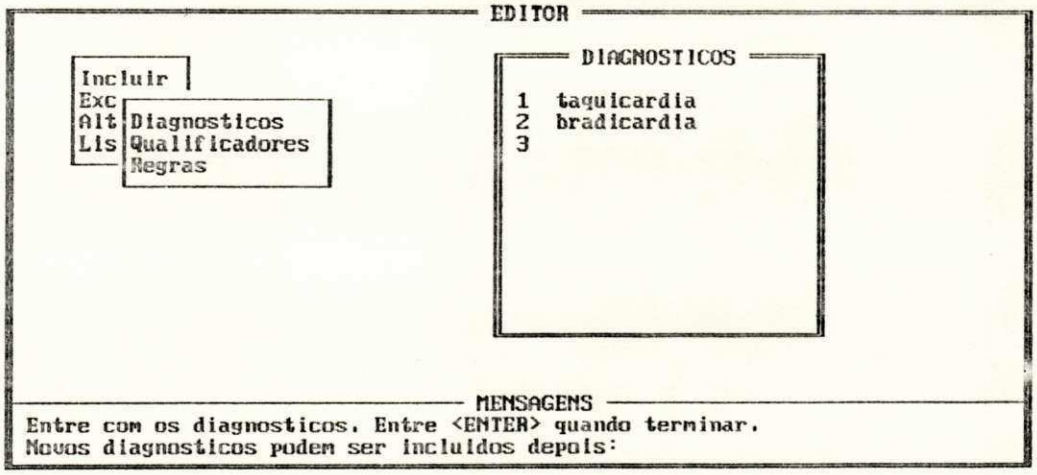
Será solicitado ao usuário que ele faça uma escolha. Na tela abaixo a primeira opção (Incluir) foi selecionada.



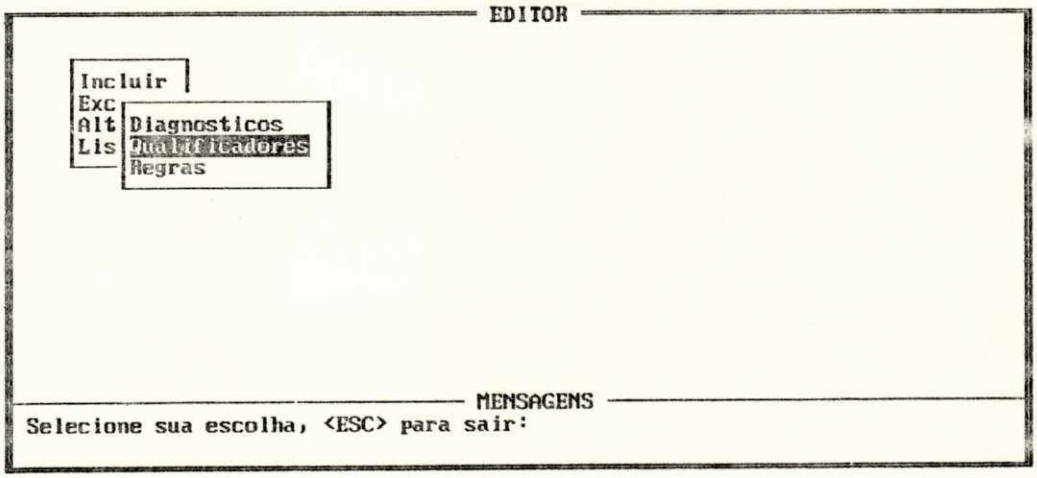
Será solicitado que o usuário faça uma escolha. Na tela abaixo a primeira opção (Diagnóstico) foi selecionada.



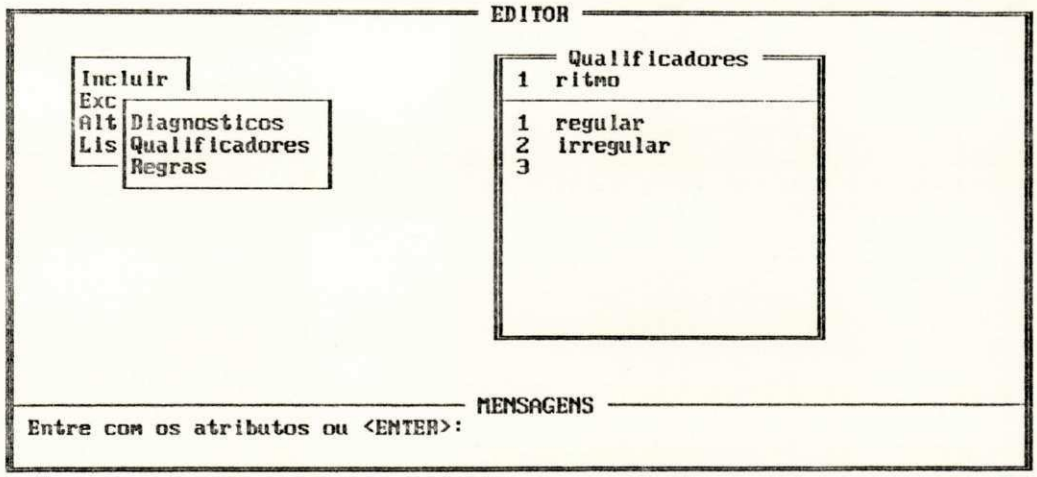
O Editor solicitará ao usuário que entre com os nomes dos diagnósticos.



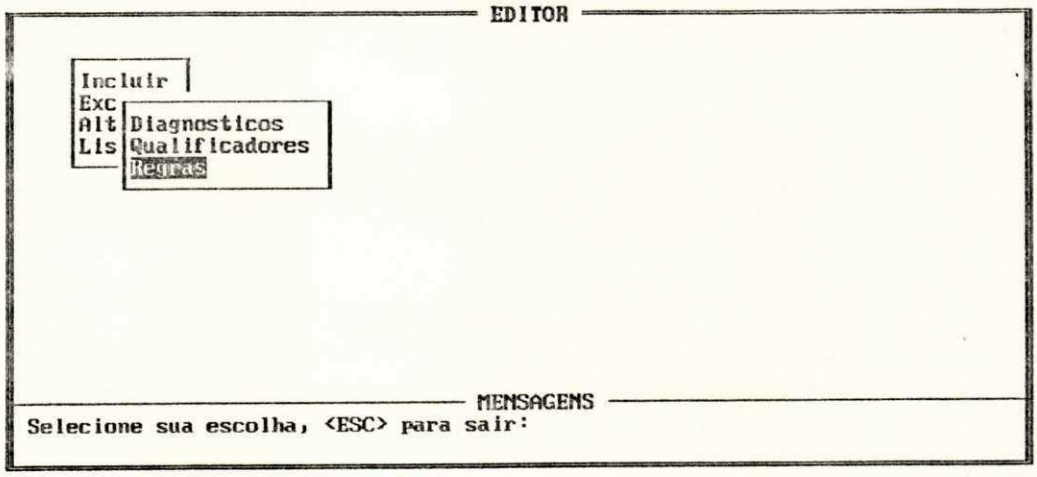
Será solicitado que o usuário faça uma escolha. Na tela abaixo a segunda opção (Qualificadores) foi selecionada.



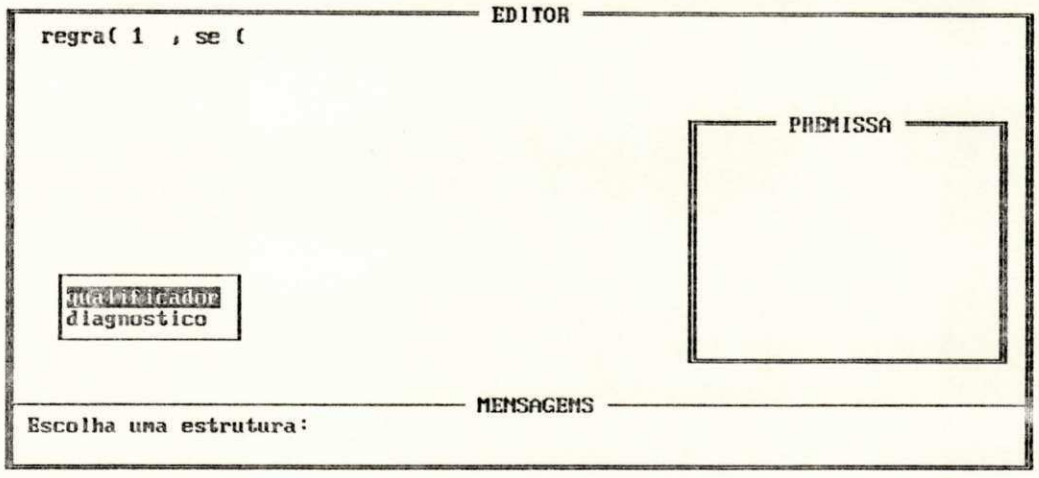
O usuário deverá entrar com os nomes dos qualificadores e seus atributos.



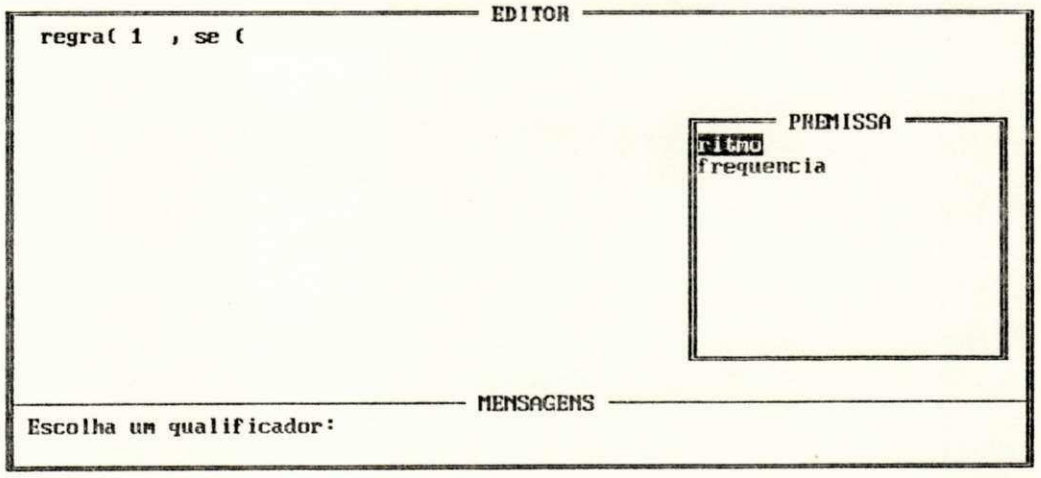
Será solicitado que o usuário faça uma escolha. Na tela abaixo a última opção (Regras) foi selecionada.



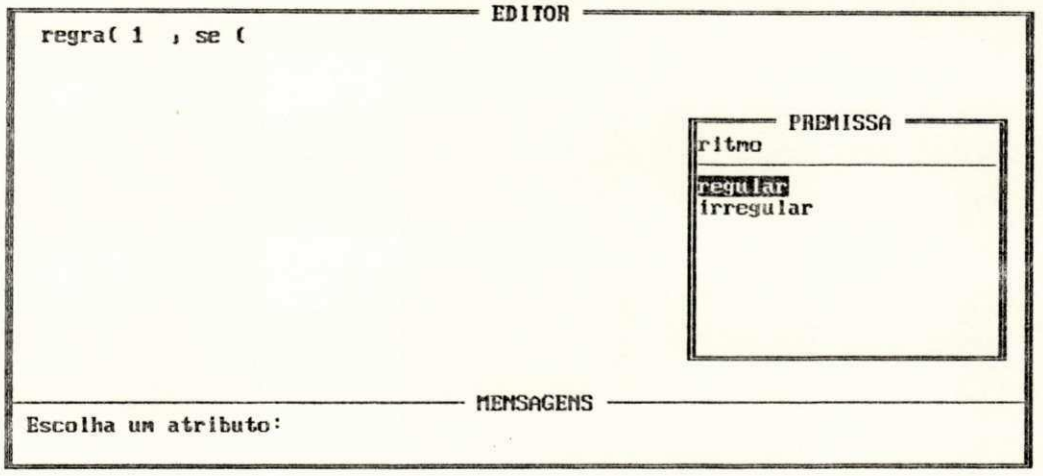
O usuário deverá escolher uma estrutura para a premissa da regra. Na tela abaixo a escolhida foi a primeira (Qualificadores).



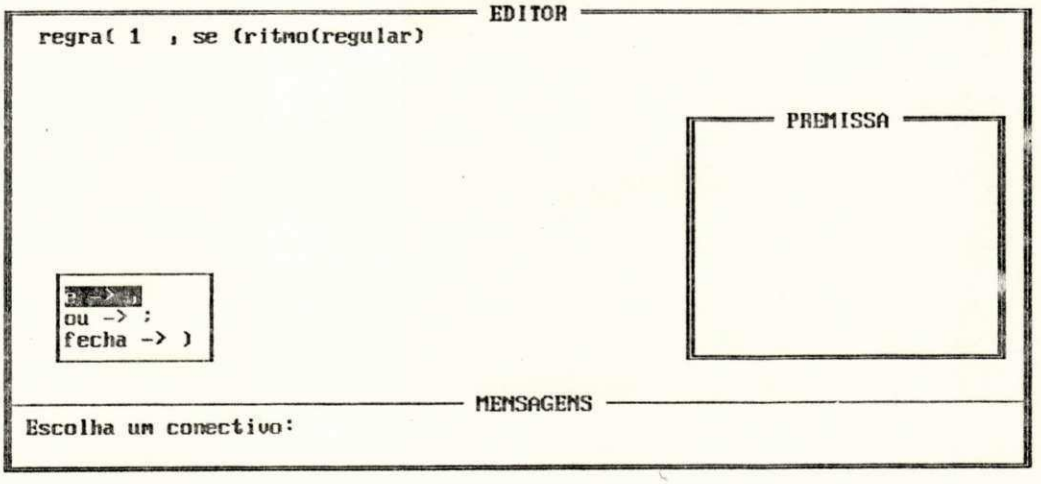
O usuário deverá escolher o qualificador que deverá fazer parte da premissa da regra.



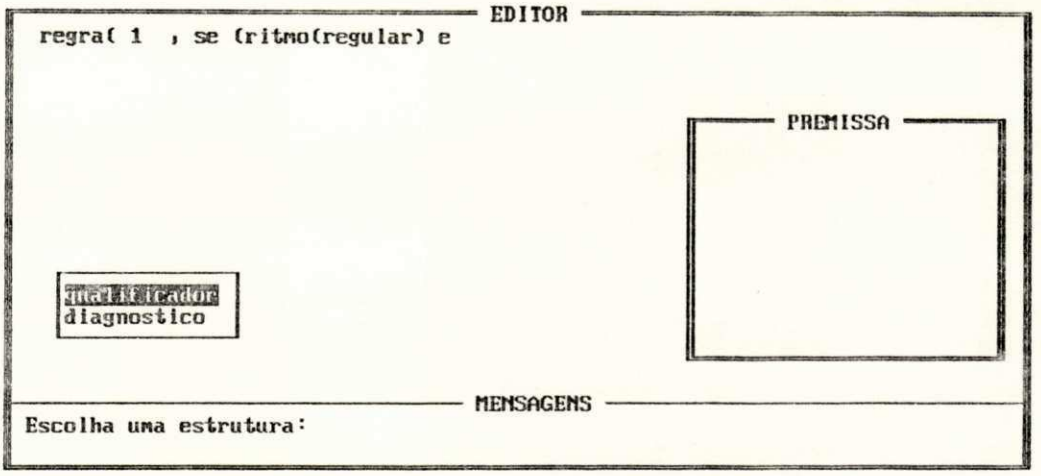
O usuário deverá escolher o atributo do qualificador que deverá fazer parte da premissa da regra.



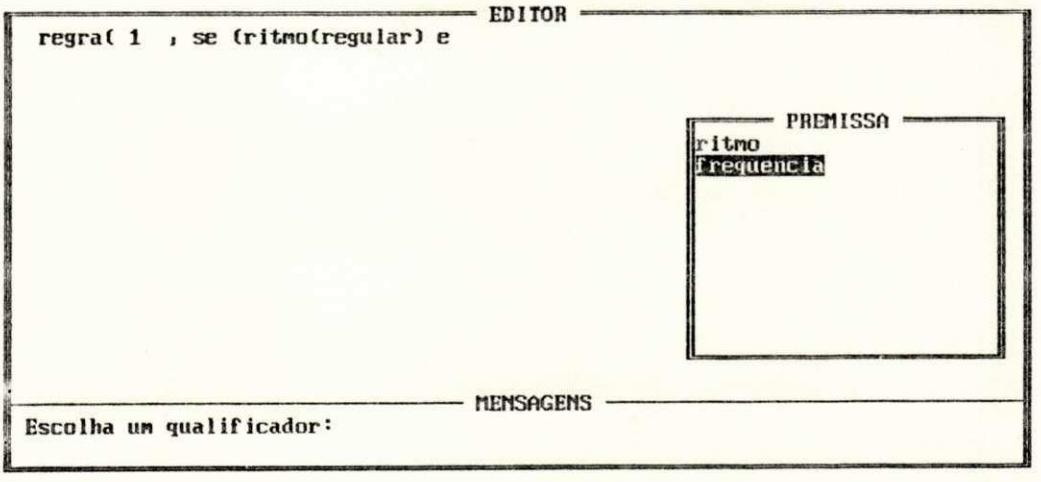
O usuário deverá escolher um conectivo para a regra. Na tela abaixo o escolhido foi o primeiro (e).



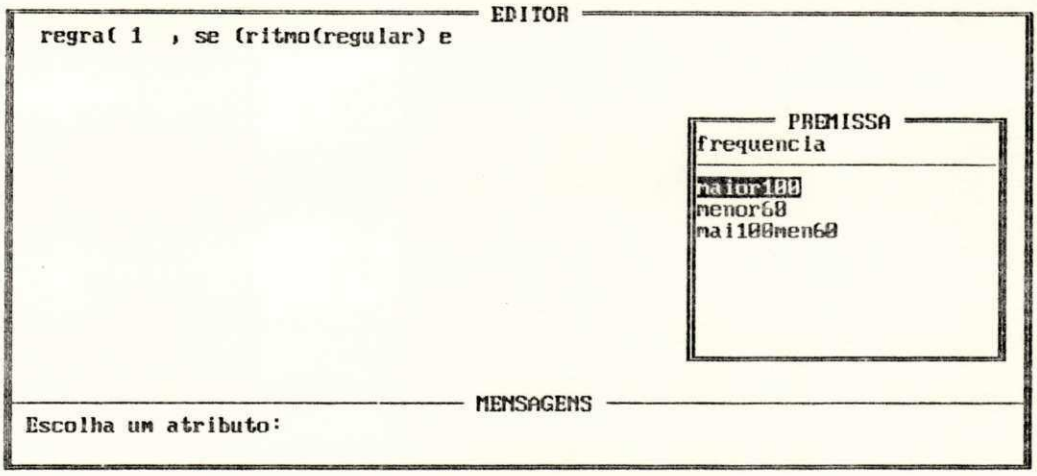
O usuário deverá escolher uma estrutura para a premissa da regra. Na tela abaixo a escolhida foi a primeira (Qualificadores).



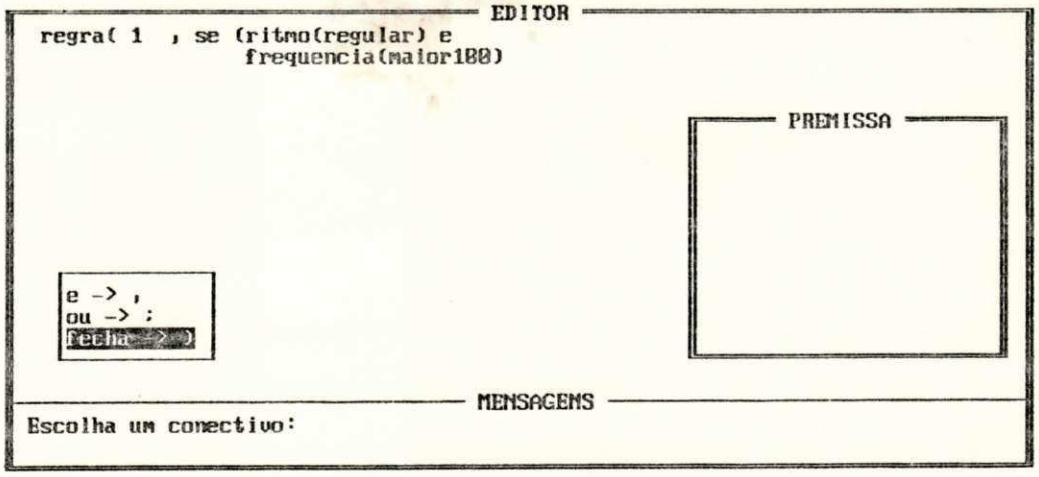
O usuário deverá escolher o qualificador que deverá fazer parte da premissa da regra.



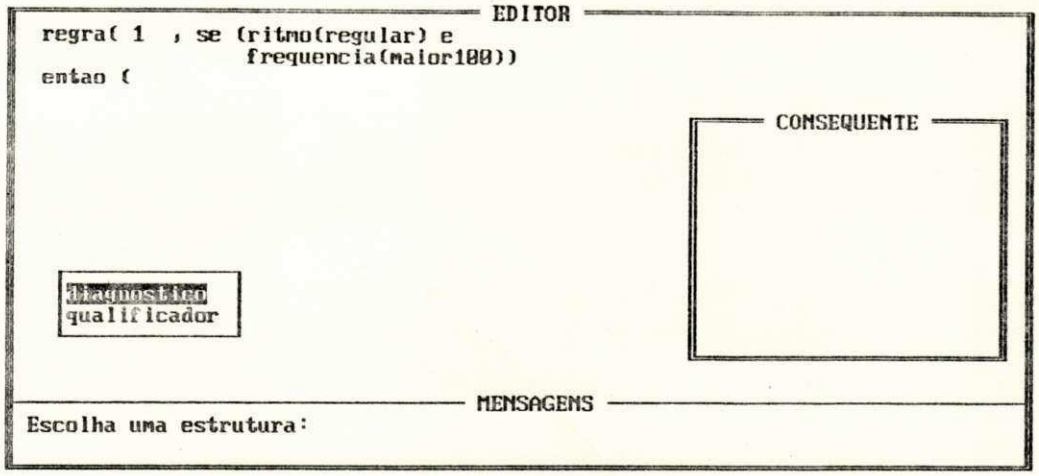
O usuário deverá escolher o atributo do qualificador que deverá fazer parte da premissa da regra.



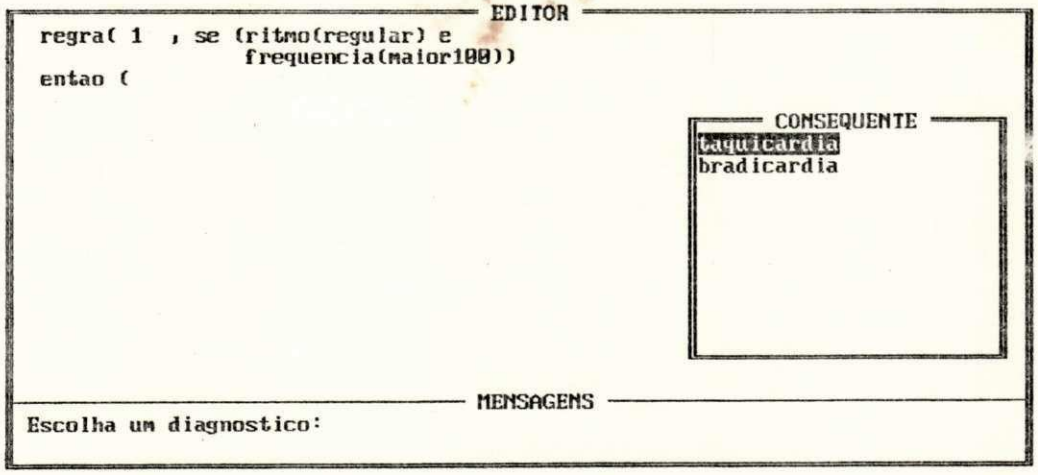
O usuário deverá escolher um conectivo para a regra. Na tela abaixo o escolhido foi último (fecha).



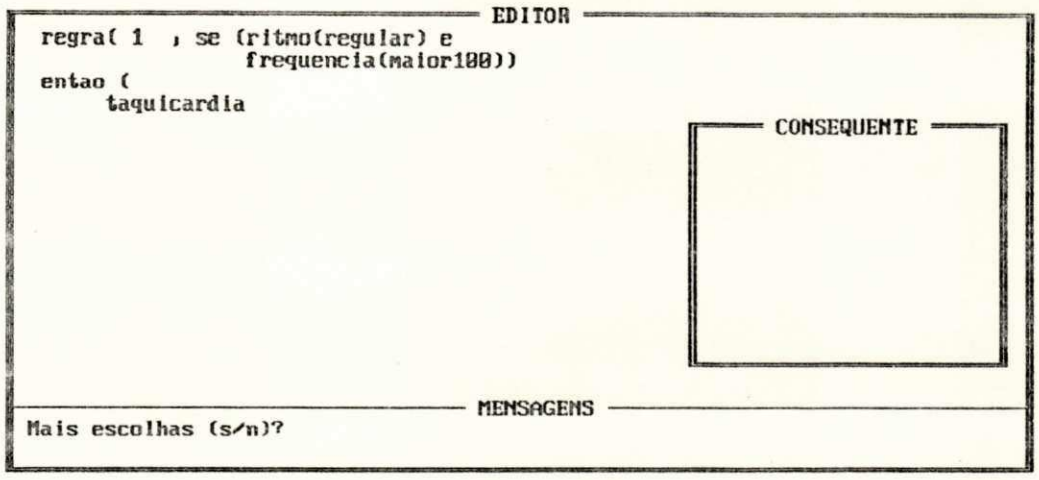
O usuário deverá escolher uma estrutura para o consequente da regra. Na tela abaixo a escolhida foi a primeira (Diagnóstico).



O usuário deverá escolher o diagnóstico que deverá fazer parte do consequente da regra.

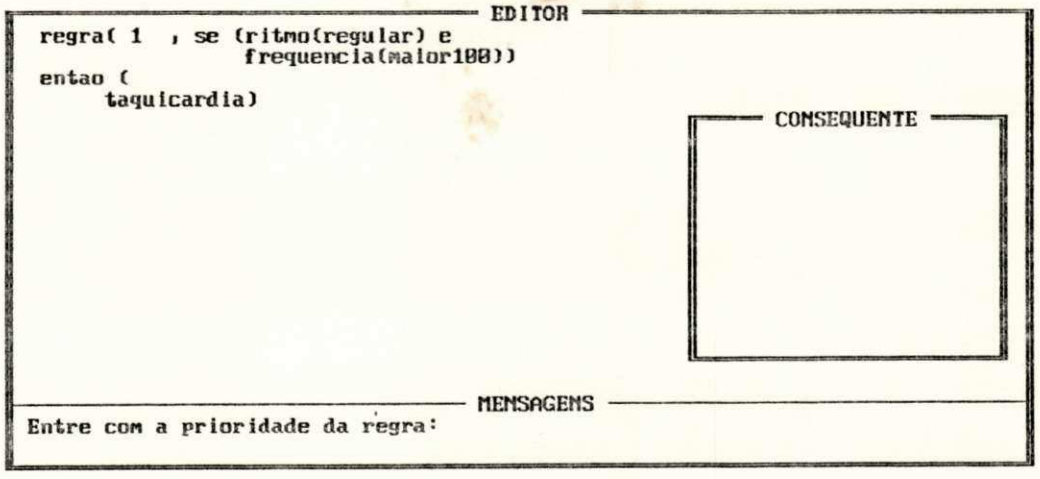


O Editor perguntará se o usuário deseja adicionar mais algum ítem ao consequente.

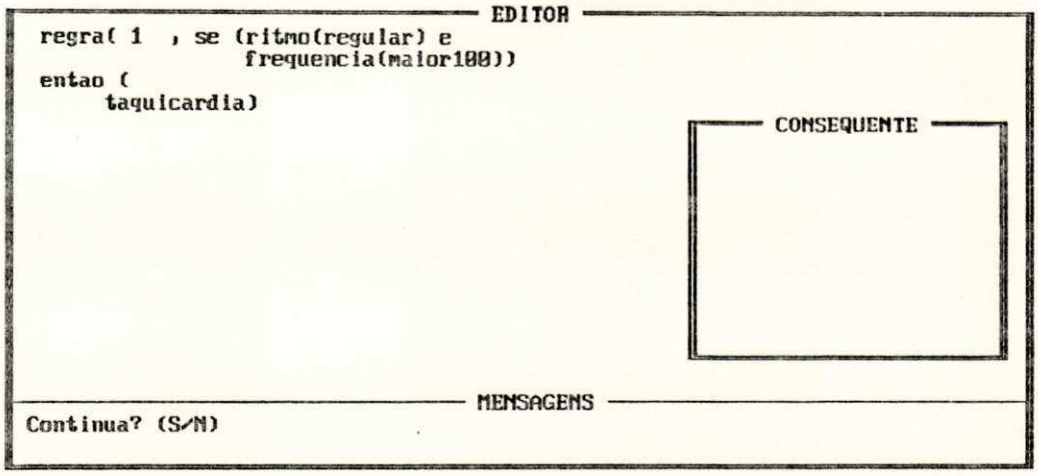


Considerando-se que o usuário teclou **N** (Não).

O usuário deverá entrar com a prioridade da regra.



Será perguntado se o usuário deseja adicionar mais regras.



5.4. Sumário

Neste capítulo apresentou-se uma aplicação para o sistema implementado e mostrou-se uma seção com o sistema.

CAPÍTULO 6

UMA INTRODUÇÃO AS REDES DE PETRI

6.1. Introdução

Uma vez que um dos principais objetivos do trabalho é prover meios para construir a Base de Conhecimento com base no modelo do sistema desta forma provendo um conhecimento profundo ou funcional do sistema optou-se pelo uso das redes de Petri como base para geração da Base de Conhecimento. Neste capítulo apresenta-se o conceito básico de redes de Petri e mostra-se as similaridades existentes entre a execução das redes de Petri e os sistemas baseados em regras de produção.

6.2. Conceitos de Rede de Petri

Redes de Petri são uma ferramenta gráfica e matemática para a modelagem formal de diversos tipos de sistemas. Elas são uma poderosa ferramenta para a descrição e análise de vários sistemas de processamento de informação que podem ser caracterizados como concorrentes, assíncronos, distribuídos, paralelos, não determinísticos, estocáticos e/ou nebulosos [MURA 89]. Como uma ferramenta gráfica, as redes de Petri podem ser usadas como um mecanismo de visualização similar aos diagramas de blocos, fluxogramas e redes. Como uma ferramenta matemática, é possível definir-se equações de estado , equações algébricas e outros modelos matemáticos que governem o comportamento do sistema.

O comportamento de muitos sistemas pode ser descrito pelos seus estados e suas trocas de estado. Um estado pode ser visto como um conjunto de condições. Uma troca de estado significa o fim de algumas condições e o início de outras. Uma troca de estados elementar, atômica, é chamada de evento. Como ferramenta de modelagem gráfica as redes de Petri descrevem o comportamento de sistemas utilizando as noções de condições e de eventos.

Uma rede de Petri é definida formalmente por uma 6-tupla **N** = **(P, T, E, M_0, K, W)**. Os três primeiros conjuntos **(P, T, E)** são utilizados para descrever a estrutura gráfica estática de uma rede de Petri, onde **P** e **T** são dois conjuntos disjuntos chamados **lugares** (ou P elementos) e **transições** (ou T elementos), e **E** é um conjunto de arcos orientados de um lugar para uma transição ou de uma transição para um lugar. Na modelagem, utilizando o conceito de condições e eventos, lugares representam condições e transições eventos. Uma transição (evento) tem **lugares de entrada e saída** representando as pré-condições e pós-condições de um evento, respectivamente. Um exemplo bastante simples é mostrado na figura 6.1. [MURA 85], neste exemplo a rede de Petri possui três lugares de entrada **(p_1, p_2, p_3)** representando as pré-condições e dois lugares de saída **(p_4, p_5)** representando os lugares de saída (pós-condições) da transição (o evento de uma cerimônia de casamento). Como mostrado na Figura 6.1., lugares são representados notacionalmente por círculos e transições por barras (quando as transições representam eventos atômicos indivisíveis). De modo a simular o comportamento dinâmico do sistema ou o fluxo de dados/contrôle são utilizadas **fichas**. A presença de um certo

número de fichas em um determinado lugar pode ser interpretada como o número de ítens de dados ou condições associadas ao lugar. Uma distribuição de fichas pelos lugares pode ser considerada como o estado do sistema. M_0 na tupla **N** denota a distribuição inicial das fichas pelos lugares e é chamada de **marcação inicial**, e sua função, M_0: **P** -> **N**, de **P** (conjunto de lugares) para **N** (o conjunto de inteiros não negativos). $M_0(p)$ denota número de fichas no lugar p. **K** é uma função de **P** para **N** U {∞} e **K(p)** denota a capacidade do lugar p, o máximo número de fichas que um lugar p pode conter. **W** é uma função de **E** para **N** e **W(e)** denota o peso ou multiplicidade associada ao arco e. Um arco com peso **W(e)** representa um conjunto de **W(e)** arcos com peso unitário paralelos como mostrado na figura 6.2., onde **W(a) = W(c) = 2** e **W(b) = 1**. O peso de um arco unitário é usualmente omitido no desenho da rede.

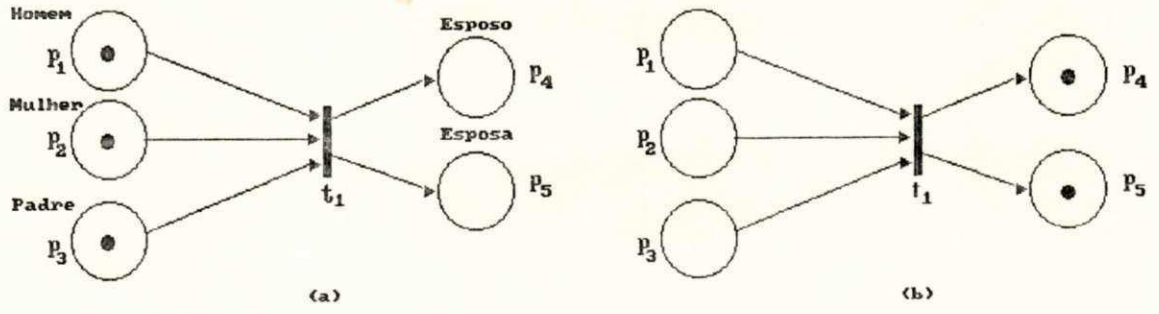


Figura 6.1.: Uma representação por rede de Petri do evento da cerimônia de casamento(a) e a ilustração da regra de transição(b).

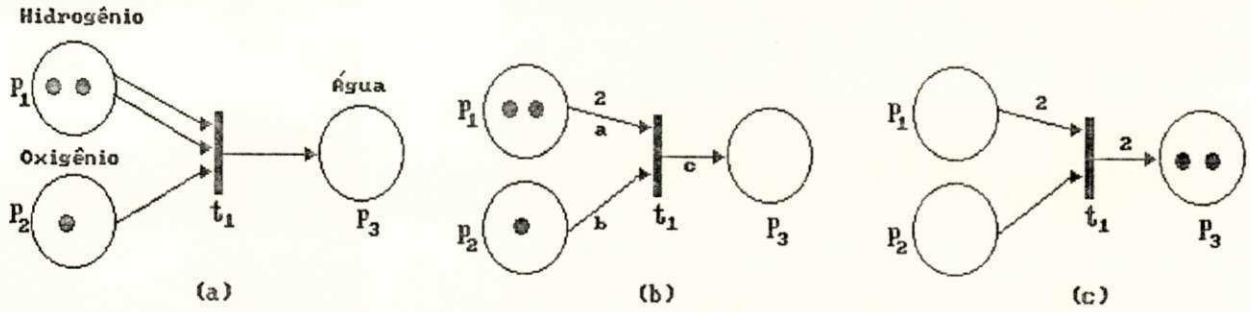


Figura 6.2.: Uma representação por rede de Petri da reação química $2H_2 + O_2 \rightarrow 2H_2O$ (a e b) e a ilustração da regra de transição(c).

De modo a simular o comportamento dinâmico do sistema, uma distribuição de fichas ou **marcação** é modificada quando ocorre um evento de acordo com as seguintes regras de operação, ditas **regras de transição:**

1) Uma transição é dita **habilitada** ou **disparável** se cada um de seus lugares de entrada tem no mínimo tantas fichas quanto são os pesos dos arcos chegando na transição. Uma transição sem nenhum lugar de entrada é sempre habilitada, por definição.

2) Uma transição habilitada pode ou não disparar (dependendo se o evento correspondente a transição pode ou não ter lugar).

3) Quando uma transição habilitada dispara, o número de fichas dos seus lugares de entrada decresce correspondentemente ao peso do respectivo arco associando o lugar à transição e, o número de fichas nos lugares de saída aumentam correspondentemente ao peso do arco associando a transição ao lugar.

No caso das Figuras 6.1 e 6.2 as transições nas Figuras 6.1a e 6.2b estão habilitadas, e quando elas disparam, a nova marcação será a mostrada nas Figura 6.1b e 6.2c, respectivamente.

As regras de transição descritas acima são chamadas **regras fracas de transição** e são aplicáveis a redes de Petri com capacidade infinita, ou seja, ($K(p) = \infty$ para todo p). Para redes de Petri com capacidade finita ($K(p) < \infty$) existe uma condição adicional para habilitação da transição de modo que o número de fichas em cada lugar de saída não excederá a capacidade após o disparo. Esta regra é referenciada como **regra estrita de transição.** Entretanto, deve-se ressaltar que uma rede de Petri **N** satisfazendo a regra estrita de transição pode ser transformada em uma rede de Petri equivalente **N'** satisfazendo a regra fraca de transição, porém sua prova esta fora do escopo deste trabalho.

Para uma rede de Petri contendo **n** transições e **m** lugares, define-se a matriz de incidência como sendo matriz **C** = $[c_{ij}]$, onde $c_{ij} = c^{+}_{ij} - c^{-}_{ij}$ com c^{+}_{ij}= número de arcos diretos da transição **i** para o lugar **j** e c^{-}_{ij}= número de arcos diretos da transição **j** para o lugar **i**. A Figura 6.3. mostra a matriz de incidência para a rede da Figura 6.1.

	t1
homem	-1
mulher	-1
padre	-1
esposo	1
esposa	1

Figura 6.3.: Matriz de incidência para a rede da Figura 6.1.

6.3. Redes de Petri e os Sistemas de Produção

Neste ítem apresentam-se, suscintamente, as similaridades existentes entre a execução de redes de Petri e de sistemas baseados em regras de produção [SAHR 87, BARB 87, NILS 82].

6.3.1. Similaridades entre Transições e Regras de Produção

Cada transição em uma rede de Petri pode ser considerada como uma regra de produção. As premissas da regra proveem as condições de marcação dos lugares de entrada, enquanto que as ações ou consequentes da regra proveem as modificações na marcação da rede decorrentes do disparo da transição.

6.3.2. Similaridades entre o Mecanismo de Evolução das redes de Petri e o Motor de Inferência

Fica claro que é possível executar uma rede de Petri diretamente através de um programa interpretador conhecido como **jogador de redes de Petri**. Neste caso a rede de Petri e sua interpretação são traduzidos para uma estrutura de dados que é então manipulada pelo jogador, seguindo-se as regras de disparo das transições. A função do jogador de rede de Petri é bastante parecida com o mecanismo de inferência através de encadeamento progressivo nos sistemas especialistas baseados em regras de produção. Este mecanismo basicamente constitui-se na tentativa de provar as premissas de uma regra de modo a verificar a aplicabilidade do consequente.

Uma ficha em um lugar representa um lugar marcado. Uma ficha no lugar p_i é representada por um ponto ù. No contexto deste trabalho

um lugar pode estar marcado ou não, associando um rótulo **verdadeiro** ou **falso** d_j ao lugar p_i. Deste modo o disparo de uma transição cujos lugares de entrada p_i estão marcados causará a marcação dos lugares de saída p_k.

Uma regra de produção é expressa então por:

R_i: SE d_j ENTÃO d_k

e pode ser modelada pela rede de Petri **N** da Figura 6.4.

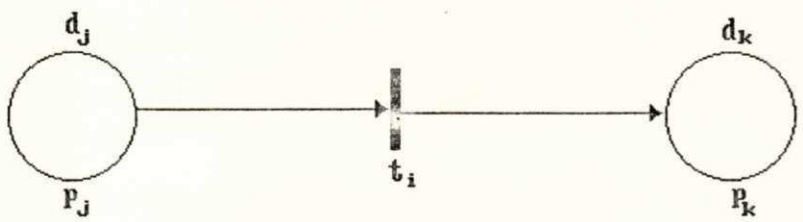


Figura 6.4.: Uma rede de Petri representando uma regra

Se o antecedente ou o consequente da regra contém conectivos "e" ou "ou" esta regra é chamada de regra de produção composta.

Neste trabalho serão considerados três tipos de regras de produção compostas:

- **Tipo 1:** SE d_{j1} e d_{j2} e $\cdots$ e d_{jn} ENTÃO d_k. Este tipo de regra pode ser modelada pela rede de Petri da Figura 6.5.
- **Tipo 2:** SE d_{j1} ENTÃO d_{k1} e d_{k2} e $\cdots$ e d_{kn}. Este tipo de regra pode ser modelada pela rede de Petri da Figura 6.6.

- **Tipo 3:** SE d_{j1} ou d_{j2} ou $\cdots$ ou d_{jn} ENTÃO d_k. Este tipo de regra pode ser modelada pela rede de Petri da Figura 6.7.

Regras contendo somente disjunções na conclusão não são consideradas pois não levam a nenhuma conclusão específica.

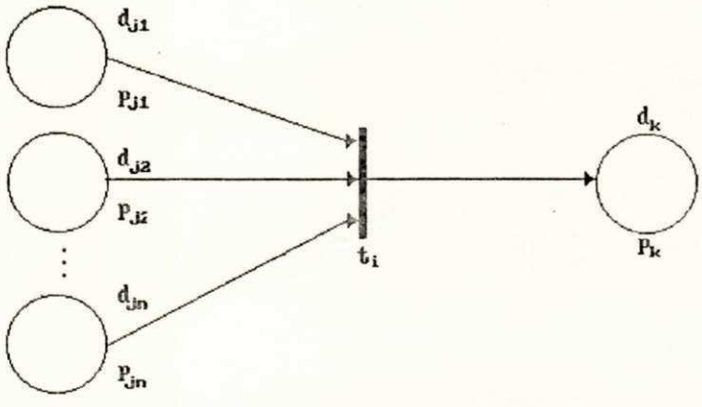


Figura 6.5.: Rede de Petri para regras do tipo 1

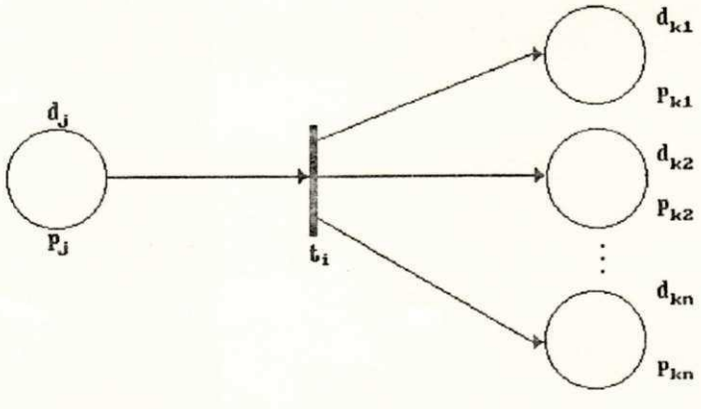


Figura 6.6.: Rede de Petri para regras do tipo 2

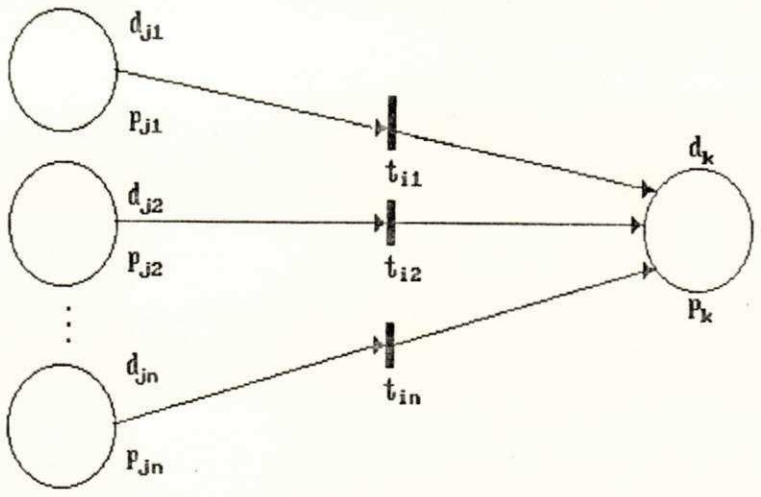


Figura 6.7.: Rede de Petri para regras do tipo 3

Observando-se a matriz de incidência da Figura 6.3. fica evidente que é possível montar-se a seguinte regra de produção:

SE (homem(marcado) e mulher(marcado) e padre(marcado))
ENTÃO (esposo(marcado) e esposa(marcado))

Neste exemplo o **então** corresponde a ocorrência do evento "cerimônia de casamento". Desta observação fica evidente a possibilidade de construírem-se regras de produção diretamente a partir da matriz de incidência.

No caso de modelos por rede de Petri é evidente que pode ser construída uma Base de Conhecimento consistente tanto para controlar um sistema, como para diagnosticar falhas. No caso de diagnótico de falha, tanto pode-se partir de uma marcação de falha conhecida, ou seja, um cenário claramente estabelecido, diagnosticar as causas da falha encadeando-se para tráz, como pode-se gerar hipóteses de falha e encadear a rede para frente de

modo a obter um cenário de falha, e observar se este é compatível com o cenário.

6.4. Sumário

Neste capítulo introduziram-se os conceitos básicos relacionados com redes de Petri e apresentou-se as similaridades existentes entre redes de Petri e os sistemas de produção.

CAPÍTULO 7

ABORDAGEM POR REDES DE PETRI

7.1. Introdução

Neste capítulo apresenta-se e descreve-se a implementação de um sistema baseado em regras de produção com o objetivo de avaliar e implementar modelos descritos por redes de Petri. Apresenta-se o desenvolvimento de um Editor para construção de regras de produção a partir de uma matriz de incidência e, de um Motor de Inferência para executar uma rede de Petri, onde a rede é descrita por um conjunto de regras de produção e, a viabilidade do emprego do Motor de Inferência na simulação do comando de um pequeno trecho de um sistema de transporte.

7.2. Editor de Regras e Motor de Inferência para Executar Redes de Petri

Neste ítem apresentam-se um Editor de Regras e um Motor de Inferência implementados para tratar com Redes de Petri.

7.2.1. Editor de Regras

O Editor de Regras aqui apresentado é uma adaptação do Editor de Regras descrito nos Capítulos 2 e 3 e também foi implementado em linguagem C. A principal diferença é que o Editor original solicita ao operador que ele selecione qualificadores e diagnósticos dos arquivos Diagnósticos e Qualificadores,

respectivamente, para comporem as regras, enquanto que este novo Editor gera as regras automaticamente a partir de uma matriz de incidência. A regras geradas são do tipo:

regra (<num_reg>, se (<premissa>) então <consequente>).

onde, **num_reg** : número inteiro que serve para rotular a regra;

premissa : contém um conjunto de fatos (<fato_1>, <fato_2>,....,<fato_3>). A **vírgula** denota o conectivo lógico "e".

consequente : contém um conjunto de fatos (<fato_1>, <fato_2>,....,<fato_3>). A **vírgula** denota o conectivo lógico "e".

As premissas nas regras são as pré-condições na rede de Petri e o consequente são as pós-condições. A execução da regra corresponde ao disparo da transição sensibilizada.

Essas regras serão armazenadas em um Arquivo de Regras. De modo a otimizar o desempenho do Motor de Inferência o Editor de Regras constroí além do Arquivo de Regras, mais dois arquivos auxiliares: Arquivo de Influência e Arquivo de Contagem. Estes arquivos formarão a Base de Conhecimento que será utilizada pelo Motor de Inferência.

O Arquivo de Influência armazena informações que indicam quais são os dados que influenciam cada regra. Exemplo: **influência (<fato_1>, R1)**. Esta declaração indica que a regra R1 é influenciada pelo fato_1.

O Arquivo de Contagem armazena informações que indicam a quantidade de fatos na premissa das regras. Exemplo: **contador(<numero_fatos>,<num_reg>).** Este contador indica quantos lugares de entrada sensibilizam uma dada transição da rede.

O Motor de Inferência deverá ter acesso a alguma informação que o possibilite tomar uma decisão quando existirem mais de uma regra pronta para ser executada. Sendo assim, criou-se um mecanismo de definição de precedencia de disparo que constituí-se da associação de prioridades a cada uma das regras que representam as transições. Estas informações serão geradas por um módulo externo e armazenado em um Arquivo de Prioridades. Exemplo: **prioridade(<grau_de_prioridade>, <num_reg>).** No caso de duas ou mais regras terem a mesma prioridade elege-se arbitrariamente a primeira a ser observada.

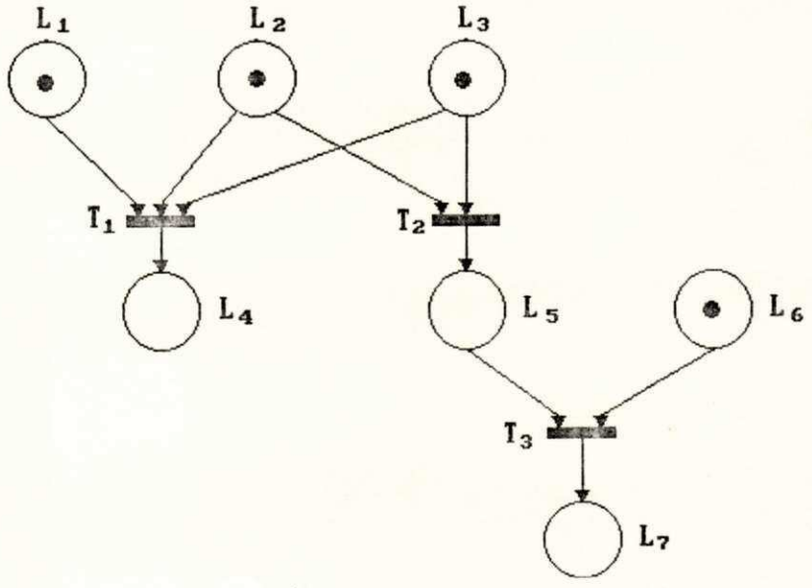
Como introduzido no capítulo anterior as regras são geradas diretamente a partir da matriz de incidência. Desta forma as informações contidas na matriz de incidência, são interpretadas pelo Editor. Considere a matriz de incidência da Figura 7.1.a, referente a rede da Figura 7.1.b. Deve ficar claro que esta rede tem somente o objetivo de ilustar a estratégia de geração da Base de Conhecimento e a evolução da rede de Petri.

Da matriz de incidência podem ser diretamente geradas todas as informações da Base de Conhecimento, com exceção da precedência utilizada somente para simulação (Arquivo de Prioridades). Observa-se que para a transição **T1** temos três lugares de entrada (**L1, L2 e L3**), os negativos na coluna, ou seja três premissas,

definindo assim o contador, as influencias e as premissas para a regra **R1** que descreve o comportamento da transição **T1**. Observando agora os positivos tem-se um lugar na coluna (**L4**), ou seja um valor no consequente da regra.

lugares	t1	t2	t3
l1	-1	0	0
l2	-1	-1	0
l3	-1	-1	0
l4	1	0	0
l5	0	1	-1
l6	0	0	-1
l7	0	0	1

(a) Matriz de incidência



(b) Representação Gráfica

Figura 7.1.: Rede de Petri - Exemplo

Para a rede de Petri da Figura 7.1. gera-se a Base de Conhecimentos detalhada a seguir.

• **Arquivo de Regras:**

```
Transição T1: regra (R1,  se(l1(marcado),l2(marcado),l3(marcado))
              entao  guarde_fato (l4(marcado)).
Transição T2: regra (R2, se (l2(marcado), l3(marcado))
              entao  guarde_fato (l5(marcado)).
Transição T3: regra (R3, se (l5(marcado), l6(marcado))
              entao  guarde_fato (l7(marcado)).
```

• **Arquivo de Influência:**

```
influencia (l1 (marcado), R1).
influencia (l2 (marcado), R1).
influencia (l3 (marcado), R1).
influencia (l2 (marcado), R2).
influencia (l3 (marcado), R2).
influencia (l5 (marcado), R3).
influencia (l6 (marcado), R3).
```

• **Arquivo de Contagem:**

```
contagem (3, R1).
contagem (2, R2).
contagem (2, R3).
```

• **Arquivo de Prioridades:**

```
prioridade (100, R1).
prioridade (200, R2).
prioridade (100, R3).
```

onde, o maior grau de prioridade associa-se ao maior valor.

Neste caso associou-se prioridade maior à transição T2 descrita pela regra R2. Ressalta-se mais uma vez que estas prioridades simulam ordens externas.

7.2.2. Motor de Inferência

Para se executar uma rede de Petri, utilizou-se um Motor de Inferência em linguagem Prolog, capaz de satisfazer as regras de disparo das transições. De fato, o Motor de Inferência aqui apresentado é uma adaptação do Motor de Inferência descrito no Capítulos 3. O motor original foi alterado, pois no caso de um sistema especialista convencional, todas as regras que puderem ser executadas serão executadas. No caso da execução de redes de Petri, as regras que descrevem o comportamento das transições que puderem ser executadas não serão todas executadas, isto é, somente prova-se uma delas, escolhida arbitrariamente. Após executada a regra, criam-se novos fatos determinados pelo seu consequente, isto é, executa-se a marcação dos lugares de saída da transição correspondente. Para as regras que não foram executadas, restauram-se os fatos que faziam parte dos seus antecedentes, ou seja, a marcação inicial dos lugares de entrada das transições correspondentes. Esse Motor de Inferência também foi dividido em dois módulos: Escalonador e Acionador.

- **Escalonador**

O Escalonador é o responsável pelo escalonamento de todas as regras que devam ser executadas pelo Acionador.

Inicialmente o Motor de Inferência lê o Arquivo de Regras, o Arquivo de Contagem, o Arquivo de Influência, o Arquivo de Prioridades e o Arquivo de Fatos e os armazena em forma de predicados Prolog. Sempre que um fato for instanciado, ele é retirado do predicado fato(dado) e as contagens das regras influenciadas por tal fato são atualizadas. No exemplo anterior, quando **fato(l1(marcado))** for instanciado os predicados **influência(dado,Num_reg)** correspondentes serão excluídos e fornecerão os números das regras influenciadas por ele, no caso a regra R1, e a contagem de todas as regras influenciadas serão diminuídas de um, no caso a contagem de R1 passará a ser 2. Sempre que a contagem tornar-se nula, a regra será ativada, ou seja, a seguinte sentença é colocada na base de trabalho: **Executável(Num_reg)**, onde N é o número da regra cuja contagem é 0.

Terminada a fase de leitura dos fatos, todas as sentenças **Executável(Num_reg)** serão verificadas em relação aos predicados **prioridade(prior,Num_reg)** para formarem listas de regras executáveis que conterão o número da regra e sua prioridade, depois estas listas formarão uma lista de regras executáveis por prioridade que será ordenada em ordem decrescente de sua prioridade. Em seguida a primeira regra da lista, ou seja, a de maior prioridade, será acionada pelo Acionador.

- **Acionador**

O Acionador é o responsável pela execução de todas as regras que foram ativadas pelo Escalonador.

O Acionador acessa a **lista de regras executáveis por prioridade** criada pelo Escalonador e executa a primeira regra da lista através do seguinte engenho de inferência:

```
aciona(X) :- regra(X, se Conds então Ações),
             call(Conds), call(Ações).
```

Sempre que uma regra é executada, a base de trabalho anterior é restaurada, ou seja, os predicados **contagem(contador,Num_reg)** passarão a ter seus valores originais, os predicados **influência(dado,Num_reg)** serão recuperados, os predicados **fato(dado)** que fazem parte das regras que não forão executadas também serão recuperados e a lista das regras executáveis por prioridade será excluída. Finalmente, todos os fatos (novos e anteriores) serão novamente instanciados na tentativa de se preparar outras regras para execução.

7.2.3. Funcionamento do Motor de Inferência

Neste ítem apresenta-se basicamente todos os passos seguidos pelo Motor de Inferência durante uma seção.

PASSO 1: Inicialmente ele acessa os arquivos Fatos, Influência, Contagem, Prioridade e Regras e criará os seguintes predicados: fato(dado), influência(dado, Num_reg), contagem(contador, Num_reg), prioridade(prior,Num_reg) e regras respectivamente. O **passo 2** será executado.

PASSO 2: O Escalonador acessa um predicado **fato(dado)** da memória de trabalho.

PASSO 3: O Escalonador identifica as regras que são influenciadas pelo predicado **fato(dado)** do passo 2 através dos predicados **influência(dado,Num_reg)** e os predicados **influência(dado,Num_reg)** e **fato(dado)** correspondentes serão excluídos.

PASSO 4: O Escalonador decrementa o **contador** dos predicados **contagem(contador,Num_reg)** das regras influenciadas. Quando o **contador** tornar-se igual a zero a regra é marcada, ou seja, a cláusula **executável(Num_reg)** será incluida.

PASSO 5: Repete os passos 2), 3) e 4) até que todos os predicados fato(dado) tenham sidos pesquisados.

PASSO 6: Restaura os predicados **influência(dado, Num_reg)** das regras executáveis e os predicados **influência(dado, Num_reg)** e **fato(dado)** das regras não_executáveis.

PASSO 7: Todas as cláusulas **executável(Num_reg)** serão verificadas em relação aos predicados **prioridade(prior, regra)** correspondentes para formarem listas de regras executáveis que conterão o número da regra e a sua prioridade e o **passo 8** será executado. Caso não exista nenhuma cláusula **executável(Num_reg)**encerrar a seção.

PASSO 8: As listas de regras executáveis formarão uma **lista de regras executáveis por prioridade** que será ordenada em ordem decrescente de sua prioridade. O **passo 9** será executado.

PASSO 9: A primeira regra da **lista de regras executáveis por prioridade** será executada pelo Acionador e a **lista de regras executáveis por prioridade** será excluída.

PASSO 10: Os predicados **fato(dado)** das regras executáveis que não foram executadas serão restaurados.

PASSO 11: Restaura os predicados **contagem (contador, Num_reg)** de todas as regras e retorna ao **passo 2.**

Com o objetivo de exemplificar a operação do Motor de Inferência considere a Base de Conhecimento gerada para a rede da Figura 7.1. O Arquivo de Fatos, representando a marcação inicial da rede é mostrado a seguir:

• **Arquivo de Fatos:**

```
fato (l1(marcado)).
fato (l2(marcado)).
fato (l3(marcado)).
fato (l6(marcado)).
```

Executando-se os passos 1), 2), 3), 4) e 5) para a rede da Figura 7.1, tem-se os seguintes valores na memória de trabalho, com relação aos valores que foram modificados:

```
influência (l5 (marcado), R3).
contagem (0, R1).
contagem (0, R2).
contagem (1, R3).
```

Executando-se o passo 6) tem-se na memória de trabalho os seguintes predicados:

```
influencia (l5 (marcado), R3).
influencia (l6 (marcado), R3).
influencia (l2 (marcado), R2).
```

influencia (l3 (marcado), R2).

influencia (l1 (marcado), R1).

influencia (l2 (marcado), R1).

influencia (l3 (marcado), R1).

fato (l6(marcado)).

Executando-se o passo 7) cria-se um lista contendo as regras R1 e R2.

Dado que a prioridade da regra R2 é maior do que a da regra R1, após a execução do passo 8) cria-se uma lista com o seguinte formato: [R2, R1].

Executando-se agora o passo 9) a regra R2 é executada e a memória de trabalho apresenta o seguinte formato:

influencia (l5 (marcado), R3).

influencia (l6 (marcado), R3).

influencia (l2 (marcado), R2).

influencia (l3 (marcado), R2).

influencia (l1 (marcado), R1).

influencia (l2 (marcado), R1).

influencia (l3 (marcado), R1).

fato (l6(marcado)).

fato (l5(marcado)).

O passo 10) restaura os fatos das regras executáveis que não foram executadas, sendo assim a memória de trabalho será adicionada com os seguintes predicados:

fato (l1(marcado)).

fato (l2(marcado)).

Finalmente, no passo 11) restauram-se os contadores das regras para seus valores iniciais e executa-se o passo 2). Quando o programa encerrar a execução, ou seja quando não houver mais transições habilitadas para o disparo, tem-se os seguintes fatos na memória de trabalho:

fato (l1(marcado)).

fato (l7(marcado)).

o que corresponde a marcação final atingida pela evolução da rede da Figura 7.1.

7.3. Aplicação a um Modelo por Rede de Petri

A aplicação de redes de Petri na modelagem de sistemas de transporte do tipo trem de metrô visa garantir o melhor fluxo de trens com o mais alto grau de segurança, ou seja, sem o risco de ocorrerem colisões.

O sistema é caracterizado por **regras operacionais** [BARB 90, BARB 89, CARV 90] que devem ser observadas, rigorosamente, por cada parte ou percurso. As regras operacionais são obedecidas pelos trens para evitar colisões, independentemente dos itinerários escolhidos. No caso de percursos unidirecionais decompõe-se o sistema de transporte em seções chamadas células [VALE 85]. No caso de percursos bidirecionais o modelo é construído a pa de partes elementares, unidirecionais comuns a todo sistema de transport são chamadas de seções.

Uma seção então, é um percurso unidirecional que pode ser do tipo simples, com duas entradas e com duas saídas, e deve obedecer a seguinte regra operacional:

<R1>: Dentro de uma seção, somente é permitido, no máximo um trem parado ou em movimento.

Considera-se como via um percurso bidirecional que pode ser de três tipos: via simples, via bifurcada e via em "Y". Esta vias devem obedecer a seguinte regra operacional:

<R2>: Dentro de uma via, somente é permitido, no máximo, um trem em movimento.

Considera-se uma conexão um percurso bidirecional que permite ao trem a passagem de uma via para outra. Uma conexão deve obedecer a seguinte regra operacional:

<R3>: Dentro de uma conexão, somente é permitido, no máximo, um trem em movimento durante a troca de vias.

O procedimento de modelagem de cada parte de um sistema de transporte segue os seguintes passos:

1) estabelecimento das redes de Petri para cada um dos tipos de seção, garantindo a regra operacional <R1>;

2) estabelecimento das redes de Petri para cada uma das vias, garantindo a regra operacional <R2> sem que a regra <R1> seja violada;

3) estabelecimento das redes de Petri para as conexões garantindo a regra operacional <R3> sem que as regras <R1> e <R2> sejam violadas;

4) construção da rede global pela composição de vias e conexões sem violar as regras <R1>, <R2>, e <R3>.

Uma vez que os detalhes do processo de modelagem por redes de Petri dos sistemas de transporte foge do escopo deste trabalho será apresentado aqui somente o desenvolvimento da implementação de um sistema de via única bidirecional.

7.4. A Aplicação

Desenvolveu-se a aplicação das ferramentas ao trecho de um sistema de transporte mostrado na Figura 7.2. A rede de Petri para este exemplo é mostrada na Figura 7.3. Deve-se observar que na Figura 7.3 os índices **d** indicam o movimento da direita para esquerda e os índices **e** o movimento da esquerda para a direita e os números dos índices indicam o número da seção, por exemplo Pd1 significa trem parado à direita na primeira seção do trecho. A Base de Conhecimento foi gerada utilizando-se o Editor de Regras descrito no ítem 7.2.1. .

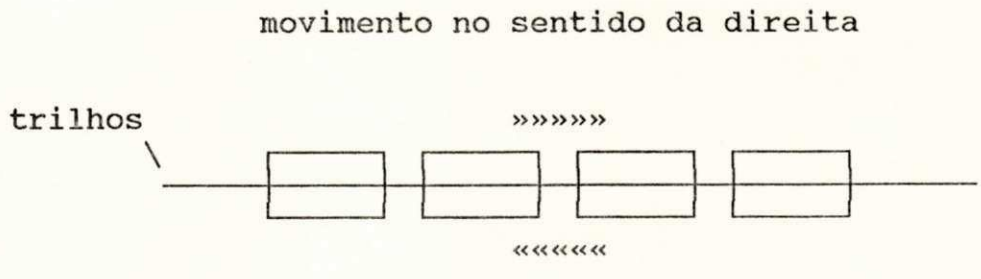


Figura 7.2.: Trecho do sistema considerado

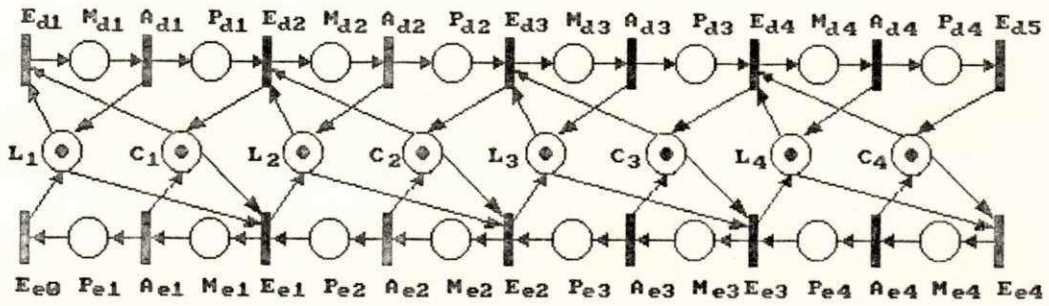


Figura 7.3.: Rede de Petri para o trecho da Figura 7.2 com a marcação inical.

Abaixo mostram-se as regras descrevendo as transições da rede de Petri da Figura 7.3

• **Arquivo de Regras**

Regras descrevendo o movimento da esquerda para a direita:

```
Transição Ed1: regra(1, se (l1(verd),c1(verd))
               entao guarde_f(md1(verd))).
Transição Ad1: regra(2, se (md1(verd))
               entao  (guarde_f(pd1(verd)), guarde_f(c1(verd)))).
Transição Ed2: regra(3, se (pd1(verd),c2(verd),l2(verd))
               entao  (guarde_f(md2(verd)), guarde_f(l1(verd)))).
Transição Ad2: regra(4, se (md2(verd))
               entao  (guarde_f(pd2(verd)), guarde_f(c2(verd)))).
Transição Ed3: regra(5, se (pd2(verd),c3(verd),l3(verd))
               entao  (guarde_f(md3(verd)), guarde_f(l2(verd)))).
Transição Ad3: regra(6, se (md3(verd))
               entao  (guarde_f(pd3(verd)), guarde_f(c3(verd)))).
```

```
Transição Ed4: regra(7, se (pd3(verd),c4(verd),l4(verd))
               entao  (guarde_f(md4(verd)), guarde_f(l3(verd)))).
Transição Ad4: regra(8, se (md4(verd))
               entao  (guarde_f(pd4(verd)), guarde_f(c4(verd)))).
Transição Ed5: regra(9, se (pd4(verd))
               entao  guarde_f(l4(verd))).
```

Regras descrevendo o movimento da direita para a esquerda:

```
Transição Ee4: regra(10, se (c4(verd),l4(verd))
               entao  guarde_f(me4(verd))).
Transição Ae4: regra(11, se (me4(verd))
               entao  (guarde_f(pe4(verd)), guarde_f(l4(verd)))).
Transição Ee3: regra(12, se (pe4(verd),l3(verd),c3(verd))
               entao  (guarde_f(me3(verd)), guarde_f(c4(verd)))).
Transição Ae3: regra(13, se (me3(verd))
               entao  (guarde_f(pe3(verd)), guarde_f(l3(verd)))).
Transição Ee2: regra(14, se (pe3(verd),c2(verd),l2(verd))
               entao  (guarde_f(me2(verd)), guarde_f(c3(verd)))).
Transição Ae2: regra(15, se (me2(verd))
               entao  (guarde_f(pe2(verd)), guarde_f(l2(verd)))).
Transição Ee1: regra(16, se (pe2(verd),c1(verd),l1(verd))
               entao  (guarde_f(me1(verd)), guarde_f(c2(verd)))).
Transição Ae1: regra(17, se (me1(verd))
               entao  (guarde_f(pe1(verd)), guarde_f(l1(verd)))).
Transição Ee0: regra(18, se (pe1(verd))
               entao  guarde_f(c1(verd))).
```

• **Arquivo de Influência**

Influência dos fatos nas premissas das regras traduzindo os lugares de entrada das transições, definido no Arquivo de Influência:

```
influ_cont(l1(verd), 1).        influ_cont(c4(verd), 10).
influ_cont(c1(verd), 1).        influ_cont(l4(verd), 10).
influ_cont(md1(verd), 2).       influ_cont(me4(verd), 11).
influ_cont(pd1(verd), 3).       influ_cont(pe4(verd), 12).
influ_cont(c2(verd), 3).        influ_cont(c3(verd), 12).
influ_cont(l2(verd), 3).        influ_cont(l3(verd), 12).
influ_cont(md2(verd), 4).       influ_cont(me3(verd), 13).
influ_cont(pd2(verd), 5).       influ_cont(pe3(verd), 14).
influ_cont(c3(verd), 5).        influ_cont(c2(verd), 14).
influ_cont(l3(verd), 5).        influ_cont(l2(verd), 14).
influ_cont(md3(verd), 6).       influ_cont(me2(verd), 15).
influ_cont(pd3(verd), 7).       influ_cont(pe2(verd), 16).
influ_cont(c4(verd), 7).        influ_cont(c1(verd), 16).
influ_cont(l4(verd), 7).        influ_cont(l1(verd), 16).
influ_cont(md4(verd), 8).       influ_cont(me1(verd), 17).
influ_cont(pd4(verd), 9).       influ_cont(pe1(verd), 18).
```

• **Arquivo de Prioridades:**

Prioridade de disparo das regras, correspondendo às transições:

movimento da esquerda para a direita	movimento da direita para a esquerda
prior(1,200).	prior(10,100).
prior(2,190).	prior(11,90).
prior(3,180).	prior(12,80).
prior(4,170).	prior(13,50).
prior(5,10).	prior(14,5).
prior(6,10).	prior(15,5).
prior(7,10).	prior(16,5).
prior(8,10).	prior(17,5).
prior(9,10).	prior(18,5).

• **Arquivo de Contagem:**

Informação sobre o número de premissas das regras, correspondendo ao número de lugares de entrada na rêde:

movimento da esquerda para a direita	movimento da direita para a esquerda
contagem(2,1).	contagem(2,10).
contagem(1,2).	contagem(1,11).
contagem(3,3).	contagem(3,12).
contagem(1,4).	contagem(1,13).
contagem(3,5).	contagem(3,14).
contagem(1,6).	contagem(1,15).
contagem(3,7).	contagem(3,16).
contagem(1,8).	contagem(1,17).
contagem(1,9).	contagem(1,18).

Após a execução do programa obteve-se uma sequencia de disparos de transições correspondendo a seguinte marcação final da rede da Figura 7.3, dadas abaixo:

(a) Sequência de transições:

regra 01:Ed1
regra 02:Ad1
regra 03:Ed2
regra 01:Ed1
regra 02:Ad1
regra 04:Ad2
regra 10:Ee4
regra 11:Ae4
regra 12:Ee3
regra 10:Ee4
regra 11:Ae4
regra 13:Ae3

(b) Marcação final:

c1(verd).
c2(verd).
pd1(verd).
l4(verd).
pe3(verd).
pe4(verd).
l3(verd).
pd2(verd).

Observe que, executado-se esta sequência de disparos de transições na rede da Figura 7.3 obtem-se a mesma marcação final obtida pela execução da rede utilizando-se o Motor de Inferência. Nota-se então que o Motor de Inferência realmente executou as funções de coordenação do comando do sistema, segundo as ordem de prioridades estabelecido para o disparo de cada transição. Na verdade, a prioridade das regras simula as decisões dos níveis superiores de comando, responsáveis por definir os possiveis itinerários que devam ser seguidos pelos trens. Concluí-se desta forma, que realmente é possível a implementação do comando deste

sistema através das técnicas de Inteligência Artificial.

No caso do modelo mostrado na Figura 7.3. pode ser observado que o sistema pode ser utilizado para diagnosticar falhas de duas formas. A primeira, consiste em a partir de um cenário de falha determinar-se qual a possível falha. Por exemplo, supondo que um trem entrou em movimento da esquerda para a direita e não há trem nem parado nem em movimento em nenhuma seção da direita para esquerda, ou seja os lugares C1, L1, C2, L2, C3, L3, C4 e L4 estão marcados, no trecho mostrado, se o trem não sair do trecho significa que ele ficou parado em alguma das seções que constituem o trecho. Então é possível, através de um encadeamento para tráz, determinar a falha, ou seja que transição, ou transições que deveriam disparar não dispararam, ou qual lugar ou lugares deveriam estar marcados e não estão. Entretanto, este tipo de raciocínio necessita que se implemente o raciocínio por encadeamento para tráz no Núcleo. A segunda forma de diagnosticar a falha, seria provocar a falha, ou seja retirar uma ficha, por exemplo do lugar C2, e executar a rede. Isto fará com que o trem fique parado na primeira seção da esquerda para a direita, ou seja haverá uma ficha em Pd1 indicando trem parado. Por outro lado, se não houver um trem parado nem em movimento nas outras seções da esquerda para direita, no caso específico não há ficha nem em Pe2 nem Me2, concluí-se que o trem parou por um defeito no sistema. Supondo que uma ficha no lugar L1 representa um sinal verde indicando para o trem avançar e sua ausência indica um sinal vermelho indicando para o trem parar, pode-se concluir que a falha é do sinal que deveria estar verde e não está. Desta observação

conclui-se que executando-se a rede a partir de uma marcação inicial representando um estado inicial com algum tipo de falha é possível verificar qual será o cenário decorrente da falha.

7.5. Sumário

Neste capítulo apresentou-se e descreveu-se a implementação de um sistema baseado em regras de produção para avaliar e implementar modelos descritos por redes de Petri. Apresentou-se um Editor para construir regras de produção a partir de uma matriz de incidência e de um Motor de Inferência para executar uma rede de Petri, onde a rede é descrita por um conjunto de regras de produção e a viabilidade do emprego desse sistema na simulação do comando de um pequeno trecho de um sistema de transporte.

CAPÍTULO 8

CONCLUSÕES E PERSPECTIVAS

Os objetivos iniciais do trabalho foram alcançados. Desenvolveu-se um Editor de Regras e um Motor de Inferência para o desenvolvimento de sistemas especialistas ou baseados em conhecimento. Introduziram-se meios pelos quais sistemas voltados para modelos pudessem também ser desenvolvidos.

Este trabalho apresentou ainda a aplicação das ferramentas para a implementação de dois sistemas. O primeiro voltado para a área de instrumentação, especificamente eletrocardiografia. Neste trabalho investiga-se a possibilidade de aquisição sobre o comportamento do sinal de eletrocardiografia utilizando-se de informações qualitativas, como ritmo normal, obtidas através de algoritmos para processamento do sinal de ECG. O objetivo da aplicação foi o desenvolvimento de um sistema de aquisição e auxílio a análise e diagnóstico a partir do ECG. A implementação no estágio atual somente tem capacidade de tratar com alguns tipos de arritmias. Os testes efetuados utilizaram um conjunto de sinais de ECG gerados a partir de um simulador, junto ao NETEB/UFPb.

A abordagem adotada na aplicação de ECG criou a oportunidade de investigar e verificar a excelência da solução proposta. O estágio atual de implementação do sistema proposto, qual seja, criar-se um sistema integrado para processamento digital de sinais e inteligência artificial com o objetivo de automatizar os

procedimentos de análise e interpretação de dados experimentais dentro da área de instrumentação eletrônica, no caso específico da instrumentação biomédica.

Evidentemente que a nível de diagnóstico e deteção de características importantes do sinal de ECG necessita-se ampliar a base de conhecimentos e introduzir outras técnicas automáticas para deteção de padrões. Deve ficar claro que diversas moléstias cardíacas podem ser diagnosticadas a partir do ECG. Isto é notoriamente importante quando o ECG é utilizado concomitantemente com outras metodologias, como ecocardiagrafia. Obviamente que um sistema que considera as diversas possibilidades da cardiologia do ponto de vista clínico resultaria em um sistema de grande escala.

A segunda aplicação desenvolvida foi a execução de uma rede de Petri modelando um sistema de transporte a nível de coordenação. De fato, a execução das redes de Petri, como mostrado possuem similaridades bastante grandes com os sistema de produção. Neste trabalho empregou-se os sistemas especialistas utilizando regras de produção para representar o conhecimento e executar redes de Petri. Entretanto, deve ser ressaltado que diversos esforços devem ser conduzidos no sentido de aperfeiçoar esta ferramenta. Um primeiro ponto a ser considerado é na execução em tempo real das redes, isto leva a necessidade de implementar-se um núcleo de sistema especialista com característica em tempo real.

Deve-se observar ainda que as aplicações de redes de Petri na área de programação em lógica são bastante promissoras, tendo tido alguns avanços consideráveis no sentido de aplicá-la como

ferramenta para representação de programas baseados em programação em lógica [MURA 88, PETE 88, MURA 89]. Esta aplicação pode ser facilmente entendida aplicando-se uma transformação de implicação para um conjunto de cláusulas e prová-las utilizando os conceitos de lógica proposicional [NILS 82, CASA 86, MEIE 88]. A abordagem mostrada neste trabalho é extendível à lógica proposicional desde que o consequente das regras tenham apenas um termo, desta forma reduzindo a transformação da implicação à uma cláusula de Horn, ou seja as regras dos tipos 1 e 3 apresentadas no capítulo 2.

Outra perspectiva que parece bastante promissora e a aplicação de redes de Petri como ferramenta para validação de bases de conhecimentos utilizando regras de produção [WILL 90]. Evidentemente que as similaridades entre redes de Petri e sistemas de regras de produção aplicadas neste trabalho podem seguir um mecanismo inverso. De fato, com o desenvolvimento desta perspectiva cria-se uma poderosa ferramenta para validação e análise de bases de conhecimentos constituídas por regras de produção. Esta possibilidade permite ainda uma ampla aplicação dos sistemas especialistas em controle, pois com a utilização de uma ferramenta formal para validação das regras, as resistências a aplicação das abordagens por inteligência artificial em controle seriam naturalmente quebradas. No caso de modelos tratando com incerteza esta solução ainda é aplicável através da utilização de redes de Petri nebulosas [LOON 88].

Entretanto, restringindo-se ao contexto deste trabalho, fica evidente que a proposta colocada atingiu plenamente seus

objetivos. O motor de inferência implementado possui cerca 100 linhas em Prolog o que caracteriza a simplicidade e eficiência desta abordagem. Dentro ainda desta perspectiva aplicou-se esta mesma ferramenta para validar e executar uma rede de Petri para um sistema de trasporte com todas as possibildades, ou seja, contendo conexões, bifurcações, cruzamentos, trechos bidirecionais e unidirecionais, etc. Evidentemente que pela extensão da base de conhecimento esta aplicação não é apresentada neste trabalho.

Deve ser ressaltado que o desenvolvimento desta aplicação serviu como validação da metodologia proposta. Deste modo diversas possibilidades de aplicação podem ser desenvolvidas. Desde sistemas bastante simples como o controle de uma pequena máquina industrial até sistemas sofisticados para automação da manufatura.

Neste trabalho, o núcleo utilizado executa a rede de Petri. Entretanto, deve-se ressaltar que com o auxílio de outras regras para dignóstico é naturalmente possivel introduzir conhecimento para deteção de falhas. Para tanto basta partir-se da marcação da rede no instante da falha e então identificar quais as transições que não dispararam desta forma identificando a partir da marcação de seus lugares de entrada porque elas não dispararam. Ou ainda, como dito no capítulo 2 utilizar o sistema como um simulador de falhas. Uma perspectiva futura é implementar o encadeamento pra tráz no núcleo, de modo a possibilitar diagnóstico de falhas como descrito no capítulo 7.

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

[BARB 87] D.S. Barbalho: Conception et Mise en Oeuvre de la Fonction Coordination pour une Commande Distribuée d'Atelier, Thèse de Doctorat de l'Université Paul Sabatier, Toulouse, France. 1987.

[BARB 89] D.S. Barbalho, M.R. Santos e T.C. de Barros: "Uma Metodologia Estruturada e Sistemática de Suporte à Automação da Manufatura: Comando de Sistemas de Transporte", Anais do 1º ERAI, Vitória, ES, Brasil, Vol. II: AM(07-18).

[BARB 90] D.S. Barbalho, M.R. Santos e T.C. de Barros: "Uma Técnica Estruturada de Modelagem por Rede de Petri: Função Coordenação de Sistema de Transporte", Anais do 8º Congresso Brasileiro de Automática, Vol. 2, Belém, PA, Setembro, 1990, pp. 1158-1164.

[BRAM 83] G.W. Brams: Réseaux de Petri: Thérie et Pratique, Editions Masson, Tome 1 e Tome 2, 1983

[CARV 90] T. de B. de Caravalho: Uma Técnica de Modelagem por Redes de Petri Voltada a Automação da Manufatura, Dissertação de Mestrado, Universidade Federal de Pernambuco, Recife, PE, Brasil, Outubro, 1990.

[CASA 86] M.A. Casanova, F.A. de C. Giorno e A.L. Furtado: Progração em Lógica, V Escola de Computação, Belo Horizonte, 1986.

[DEEP 89] G.S. Deep, A. Perkusich, M.L.B. Perkusich e M.L. Varani: "Amplificador de ECG com Chaveamento Eletrônico: Aplicação a um Sistema Automático de Aquisição e Análise", Revista Brasileira de Engenharia, Caderno de Engenharia Biomédica, Vol. 6, Setembro, 1989, Nº 2, pp. 298-305.

[DOUG 88] E.R. Dougherty e C.R. Giardina: Mathematical Methods for Artificial Inteligence and Autonomous Systems, Prentice-Hall International Editions, Englewood Clifs, NJ, 1988.

[DVOR 87] D.L. Dvorak: "Expert Systems for Monitoring and Control", Report AI87-55, University of Texas at Austin, 1987, 23 pags.

[DUBI 76] D. Dubin: Interpretação Rápida do ECG: um Curso Rápido, Rio de Janeiro: EPUC, 1976.

[FAIR 85] R. Fairley: Software Engineering Concepts, MacGraw-Hill International Student Editions, Computer Science Series, New York, 1985.

[FINK 87] P.K. Fink e J.C. Lusth: "Expert Systems and Diagnostic Expertise in the Mechanical and Electrical Domains", IEEE Trans. on Systems Man an Cybernetics, Vol. 17, Nº 3, May/June, 1987, pp. 340-349.

[HALH 80] M.J. Halhuber, R. Gunther e M. Ciresa: Manual de Eletrocardiografia, São Paulo: E.P.U./Springer, 1980.

[KULI 88] K. Kulikowski: "Artificial Intelligence in Medical Consultation Systems: A Review", IEEE Engineering in Medicine and Biology Magazine, Junho, 1988, pp. 34-39.

[LOON 88] C.G. Looney: "Fuzzy Petri Nets for Rule-Based Decision-Making", IEEE Trans. on Systems Man an Cybernetics, Vol. 18, Nº 1, Feb./Mar., 1988, pp. 178-183.

[MAIE 88] D. Maier e D.S. Warren: Computing with Logic: Logic Programming with Prolog, Benjamin/Cummings, Menlo Park, California, 1988.

[NELS 83] R.A. Nelson, L. M. Haibt e P.B. Sheridan: "Casting Petri Nets into Programs", IEEE Trans. on Software Engineering, Vol. SE-9, Setembro, 1983, pp. 590-602.

[NILS 82] N.J. Nilsson: Principles of Artificial Intelligence, Springer-Verlag, Berlin, 1982.

[MILN 87] R. Milne: "Strategies for Dignosis", IEEE Trans. on Systems Man an Cybernetics, Vol. 17, Nº 3, May/June, 1987, pp. 333-339.

[MURA 88] T. Murata e D. Zhang: "A Predicate-Transition Net Model for Parallel Interpretation of Logic Programs", IEEE Trans. on Software Engineering, Vol. SE-14, Abril, 1988, pp. 481-497.

[MURA 89] T. Murata: "Petri Nets: Properties, Analysis and Applications", Proceedinds of the IEEE, Vol. 77, Nº 4, April, 1989, pp.541-580.

[MURA 84] T. Murata: Modelling and Analysis of Concurrent Systems, Handbook of Software Engineering, Eds. C.R. Vick e C.V. Ramamoorthy, Van Nostrand Reinhold Company Inc, NY, 1984, pp. 39-63.

[PAPP 88] Z. Papp et alli: "Intelligent Medical Instruments", IEEE Engineering in Medicine and Biology Magazine, junho, pp. 18-23.

[PERK 87] A. Perkusich, G.S. Deep e J.H.F. Cavalcanti, "Data and Control Flow Diagram and Process Automation", Proceedings of IEEE Industrial Electronics Society Conference IECON'87, Cambridge, Massachusetts, 1-6 de Novembro de 1987, pp. 48-55.

[PERK 89] A. Perkusich, G.S. Deep, M.L.B. Perkusich e M.L. Varani: "An Expert ECG Acquisition and Analysis System", Anais do IMTC/89, IEEE Instrumentation and Measurement Technology Conference, Washington, D.C., 25-27 Abril, 1989 pp. 184-189.

[PERK 90] A. Perkusich, M.L.B. Perkusich , G.S. Deep, M.E. de Morais, A.M.N. Lima: "Sistema de Auxílio a Análise e Dagnóstico a Partir de Eletrocardiograma", Revista Brasileira de Engenharia, Caderno de Engenharia Biomédica, Vol. 8, Outubro, 1990, Nº 2, pp. 298-305.

[PETE 81] J.L. Peterson: Petri Net Theory and the Modelling of Systems, Prentice-Hall Inc., New Jersey, 1981.

[PETE 89] G. Peterka e Tadao Murata: "Proof Procedure and Answer Extraction in Petri Nets Model of Logic Programs", IEEE Trans. on Software Engineering, Vol. SE-15, Fevereiro, 1989, pp. 209-217.

[PRES 87] R.S. Pressman: Software Engineering: A Practioner's Aproach, MacGraw-Hill International Student Editions, Computer Science Series, Second Edition, New York, 1987.

[REIS 82] W. Reizig: Petri Net Theory An Introduction, Springer-Verlag, Berlin, 1982.

[RICH 88] E. Rich: Inteligência Artificial, McGraw-Hill, São Paulo, 1988.

[SAHR 87] A. Sahraoui, H. Atabakhche, M. Corvousier e R. Valette: "Joining Petri Nets and Knowledge Based Systems for Monitoring Purposes", Raport LAAS, Toulouse, France, 1987, 22 pags.

[SCHI 89] H. Schildt: Inteligência Artificial e Linguagem C, MacGraw-Hill, Rio de Janeiro, 1989.

[STEP 87] H.F. Sthephanou e A.P. Sage: "Perspectives on Imperfect Information Processing", IEEE Trans. on Systems Man an Cybernetics, Vol. 17, Nº 5, September/October, 1987, pp. 780-798.

[SZTI 88] J. Sztipanovits e G. Karsai: "Knowledge-Based Techniques in Instrumentation", IEEE Engineering in Medicine and Biology Magazine, Junho, 1988, pp. 13-17.

[VALE 85] R. Valette, M. Courvoisier e C. Desclaux: "Putting Petri Nets to Work for Controlling Flexible Manufacturing Systems", Proceedinds of IEEE International Symposium on Circuits and Systems, ISCAS 85, Kyoto, Japan.

[WATE 86] D.A. Watermann AGuide to Expert Systems, 1986, Addison-Wesley Publishing Company.

[WILS 90] R.G. Wilson e B.H. Krogh: "Petri Nets Tools for the Specification and Analysis of Discrete Controllers",IEEE Trans. on Software Engineering, Vol. 16, Nº 1, January, 1990, pp. 39-50.

APÊNDICE A

NÚCLEO DO SISTEMA

```
:- op(900,fx,se).
:- op(600,xfy,':').
:- op(800,xfx,entao).
:- op(750,xfy,e).
:- op(400,fx,i).
:- op(300,fx,[guarde_f,guarde_d]).


/* consulta  arquivos e os armazena em forma de predicados  Prolog
   */

consulte(Arq) :- see(Arq),repeat,
                 read(X),armazene(X),!,seen.

armazene(end_of_file) :- !.
armazene(A) :- assertz(A),fail.

                 /* E S C A L O N A D O R */

/* Instancia um  predicado FATO ou DIAGNOSTICO, depois o instancia
   com  os predicados INFLU_CONT para encontrar o numero da  regra
   influenciada. Finalmente atualiza o predicado CONTAGEM da regra
   correspondente  e  verifica o valor do conectivo C. Se C  =  1,
   cria uma clausula EXECUTAVEL, ou seja, prepara a regra para ser
   executada, senao chamara a rotina de disjuncao disj */

escalonador :- retract(fato(D)),assertz(D),assertz(dado(D)),
               influ_cont(D,N), retract(influ_cont(D,N)),
               contagem(C1,C2,N,C), ifthenelse(C == 1,
               (assertz(executavel(N)), exec1(D)),
               disj(D,N)).

escalonador :- call(diagnostico(D)),influ_cont(D,N),
               assertz(dado(D)),
               retract(diagnostico(D)), retract(influ_cont(D,N)),
               contagem(C1,C2,N,C),
               ifthenelse(C == 1, (assertz(executavel(N)),
               exec1(D)), disj(D,N)).

escalonador.

/* Procura o numero de todas as regras influenciadas por um dado e
   o instancia com o predicado CONTAGEM correspondente e  verifica
   o valor do conectivo C. Se C = 1, cria uma clausula EXECUTAVEL,
   senao chamara o predicado disj */
```

```
exec1(D) :- influ_cont(D,N), retract(influ_cont(D,N)),
            contagem(C1,C2,N,C),
            ifthenelse(C == 1,
            (assertz(executavel(N)), exec1(D)), disj(D,N)).

exec1(X) :- escalonador.

/* Atualiza  o contador C2 do predicado CONTAGEM da  regra
   correspondente.  Se C2 = 0, cria uma clausula EXECUTAVEL, ou
   seja, prepara a regra para ser executada, senao chamara o
   predicado exec1. */

disj(D,N) :- retract(contagem(C1,C2,N,C)),C3 is C2 - 1,
             assertz(contagem(C1,C3,N,C)),
             ifthenelse(C3 == 0, (assertz(executavel(N)), exec1(D)),
             exec1(D)),!.

                  /* M O T O R  D E  I N F E R E N C I A */


/* Ler  uma  lista (com numero e prioridade da regra) e  aciona  a
   primeira da lista */

acionador([[A,B]|Ys]) :- aciona(B), not(modificou), acionador(Ys),!.
acionador([]).

/* Aciona  uma regra, ou seja, procura os fatos na base e  executa
   as acoes. */

aciona(Y) :-  not(executou(Y)),
              regra(Y,se Conds entao Acoes),
              asserta(regra_corrente(Y)),
              call(Conds), call(Acoes),
              assertz(executou(Y)), retract(prior_exec(Y,X)).

aciona(Y) :- assertz(para).

                  /* D I A G N O S T I C A D O R */

/* Lista  todos  os  diagnosticos  que  estao  nos  predicados
   DIAGNOSTICO */

diagnosticador :- diagnostico(X), call(X), write(X), nl, fail.
diagnosticador.

/* Procura um fato na base. Se nao existir, o coloca na base. */

guarde_f(X) :- call(X).
guarde_f(X) :- assertz(fato(X)), modificou.
guarde_f(_) :- assertz(modificou).

retira_f(X) :- call(X).
retira_f(X) :- retract(fato(X)).
```

```
/* Procura  um  diagnostico na base. Se nao existir, o  coloca  na
   base. */

guarde_d(X) :- call(X).
guarde_d(X) :- assertz(diagnostico(X)), assertz(X), modificou.
guarde_d(_) :- assertz(modificou).

/* executa  o  nucleo no modo CONTINUO ou  no  modo  PASSO_A_PASSO
   dependendo da opçao do usuario */

execute :- consulte($exemplo.reg$), consulte($exemplo.inf$),
           consulte($exemplo.pri$), consulte($exemplo.con$),
           assertz(modificou), passo.

/* Escalona as regras para serem executadas, cria listas de regras
   executaveis,  classifica em ordem decrescente de  prioridade  a
   lista  de regras executaveis e aciona todas as regras da  lista
   */

continuo :- modificou, escalonador, exec_prior,
            findall([Y,X], prior_exec(X,Y), L), sort(L,L2),
            invertel(L2,[],L1), retract(modificou), acionador(L1),
            continuo.

continuo :- diagnosticador, !.

passo :-  modificou, escalonador, exec_prior, passo2,!.

passo :- diagnosticador,!.

passo2 :- findall([Y,X], prior_exec(X,Y), L), sort(L,L2),
          invertel(L2,[],L1), retract(modificou), socabeca(L1),
          passo,!.

passo2 :- diagnosticador,!.

/* Recupera o numero da primeira regra e a aciona */

socabeca([[A,B]|Ys]) :- aciona(B), socabeca2(Ys),!.
socabeca([]) :- !.

socabeca2([[A,B]|Ys]) :- not(modificou), aciona(B), socabeca2(Ys),!.
socabeca2([[A,B]|Ys]) :- !.
socabeca2([]) :- !.

/* Recupera  o numero das regras executaveis, acha a prioridade  e
   cria  uma  lista  com  o  numero  da  regra  executavel  e  sua
   prioridade */

exec_prior :- executavel(X), prior(X,Y), retract(executavel(X)),
              assertz(prior_exec(X,Y)), exec_prior.

exec_prior.
```

```
/* inverte a lista das regras executaveis */

invertel([X|Xs], Acc,Ys) :- invertel(Xs,[X|Acc],Ys).
invertel([],Ys,Ys).

/* Lista todas as regras executaveis que nao foram executadas. */

lista_executaveis :- prior_exec(X,Y), write(X),
                     retract(prior_exec(X,Y)),
                     nl, fail.

como :- executou(X), write('Pela(s) regra(s) '), mostra_regras, nl,
        write(' baseado nos dados: '), mostra_dados, nl,
        write(' concluiu-se que: '), diagnosticador.

como :- write('Atualmente nao diponho de dados suficientes'),
        write(' para responder.').

mostra_regras :- call(executou(X)), write('R'), write(X),
                 write(', '), fail.
mostra_regras.

mostra_dados :- call(dado(X)), write(X), write(', '), fail.
mostra_dados.
```

APÊNDICE B

NÚCLEO PARA REDES DE PETRI

```
:- op(900,fx,se).
:- op(600,xfy,':').
:- op(800,xfx,entao).
:- op(750,xfy,e).
:- op(400,fx,i).
:- op(300,fx,[guarde_f, guarde_d]).


/* consulta arquivos e os armazena em forma de predicados Prolog */

consulte(Arq) :- see(Arq),repeat,
                 read(X),armazene(X),!,seen.

armazene(end_of_file) :- !.
armazene(A) :- assertz(A),fail.

/* instancia um  predicado FATO, depois instancia o predicado FATO
com  os  predicados INFLU_CONT para encontrar o  numero  da  regra
influenciada.  Finalmente atualiza o predicado CONTAGEM  da  regra
correspondente.  Se  for zero, cria uma  clausula  EXECUTAVEL,  ou
seja, prepara a regra para ser executada */

escalonador :- retract(fato(X)), vesex(X), influ_cont(X,Y),
               assertz(influ(X,Y)), retract(influ_cont(X,Y)),
               cont(Z,Z1,Y),
               retract(cont(Z,Z1,Y)),Z2 is Z1 - 1,
               assertz(cont(Z,Z2,Y)),
               ifthenelse(Z2 == 0,
               /* then */
               (assertz(executavel(Y)),
                 exec1(X)),
              /* else */
                exec1(X)),!.
escalonador :- restaura.
escalonador.

vesex(X) :- call(X), !.
vesex(X) :- assertz(X).

exec1(X) :- influ_cont(X,Y), assertz(influ(X,Y)),
            retract(influ_cont(X,Y)),
            cont(Z,Z1,Y), retract(cont(Z,Z1,Y)),
            Z2 is Z1 - 1, assertz(cont(Z,Z2,Y)),
            ifthenelse(Z2 == 0, (assertz(executavel(Y)), exec1(X)),
            exec1(X)),!.
```

```
exec1(X) :- escalonador, !.

restaura :- cont(Z,Z1,Y),
            ifthenelse(Z1 == 0,
            /* then */
            (veinflu(Y), retract(cont(Z,Z1,Y))),
            /* else */
            (retract(cont(Z,Z1,Y)), veinfli(Y))),
            restaura.
restaura.

veinflu(Y) :- influ(X,Y), retract(influ(X,Y)), guarde_influ(X,Y), fail.
veinflu(Y) :- gc(full).

veinfli(Y) :- influ(X,Y), retract(influ(X,Y)), guarde_influ(X,Y),
              guarde_fato(X), fail.
veinfli(Y) :- gc(full).

guarde_influ(X,Y) :- call(influ_cont(X,Y)).
guarde_influ(X,Y) :- assertz(influ_cont(X,Y)).


acionador([[A,B]|Ys]) :- aciona(B), acionador(Ys),!.
acionador([]).

aciona(Y) :-  regra(Y,se Conds entao Acoes),
              call(Conds), call(Acoes),
              gc(full), retract(prior_exec(Y,X)), retira_conds(Y), !.
aciona(Y).

retira_conds(Y) :- influ_cont(X,Y), retract(X), retira_f(X), fail.
retira_conds(Y).

diagnosticador :- diagnostico(X), call(X), write(X), nl, fail.
diagnosticador.

liste :- fato(X), write(X),nl,fail.
liste.


liste(F) :- call(F),write(F),nl,fail.
liste(_).

guarde_f(X) :- call(X), retract(X), guarde_f(X).
guarde_f(X) :- assertz(fato(X)).

guarde_fato(X) :- call(fato(X)), !.
guarde_fato(X) :- assertz(fato(X)).

retira_f(X) :- not(call(fato(X))), !.
retira_f(X) :- retract(fato(X)).

guarde_c(Z,Z1,Y) :- Z2 is Z1 + 1, call(cont(Z,Z2,Y)),
                    retract(cont(Z,Z2,Y)),
                    assertz(cont(Z,Z1,Y)).
```

```
guarde_c(Z,Z1,Y) :- assertz(cont(Z,Z1,Y)).

guarde_d(X) :- call(X), !.
guarde_d(X) :- assertz(diagnostico(X)).

execute :- consulte($petri.reg$),consulte($petri.inf$),
           consulte($petri.pri$),consulte($petri.con$),
           consulte($fatos.ini$), fazcont, passo.

fazcont :- contagem(X,Y), assertz(cont(X,X,Y)), fail.
fazcont.

passo :-  not(para), escalonador, vesep, fazcont, exec_prior,
          findall([Y,X], prior_exec(X,Y), L), sort(L,L2),
          invertel(L2,[],L1), socabeca(L1), passo.

passo :- nl, liste, halt.

vesep :- executavel(X).
vesep :- assertz(para), passo.

socabeca([[A,B]|Ys]) :- aciona(B), resto(Ys).
socabeca([]).

lp :- prior_exec(X,Y), retract(prior_exec(X,Y)), fail.
lp.

resto([[A,B]|Ys]) :- procura_influ(B), resto(Ys).
resto([]) :- lp.

procura_influ(Y) :-  influ_cont(X,Y), retract(influ_cont(X,Y)),
                     assertz(ic(X,Y)), retract(X),
                     guarde_fato(X), fail.
procura_influ(Y) :- icif.
procura_influ(Y).

icif :- ic(X,Y), retract(ic(X,Y)), assertz(influ_cont(X,Y)), fail.
icif.

exec_prior :- executavel(X), prior(X,Y), retract(executavel(X)),
              assertz(prior_exec(X,Y)), exec_prior.

exec_prior.

invertel([X|Xs], Acc,Ys) :- invertel(Xs,[X|Acc],Ys).
invertel([],Ys,Ys).
```

APÊNDICE C

VISÃO GERAL DO SISTEMA DE AUXÍLIO A ANÁLISE DE ECG

A aplicação descrita no capítulo 5 faz parte de um sistema de auxílio a análise e interpretação de eletrocardiogramas (ECG) voltado para a deteção de arritmias, como descrito em [PERK 89, DEEP 89, PERK 90]. O objetivo principal do sistema é construir um sistema experimental de modo a investigar a possibilidade de integrar técnicas de processamento digital de sinais, reconhecimento de padrões e sistemas especialistas, com ênfase no conhecimento funcional, para auxiliar a análise e interpretação do ECG. Neste sentido a base de conhecimento implementada não tem finalidade clínica pois para o diagnóstico de enfermidades cardíacas outros dados são extremamente importantes, como dor precordial, tipos de drogas ministradas, e informações de outros exames como ecocardigrafia e angiografia.

Na figura A.1 apresenta-se o DFDC (Diagrama de Fluxo de Dados e Controle) [PERK 87] de primeiro nível do sistema completo. O sistema baseado em conhecimento apresentado no capítulo 5 corresponde à transformação 6 no DFDC. Detalhes sobre a implementação de hardware podem ser encontradas em [PERK 89 e DEEP 89].

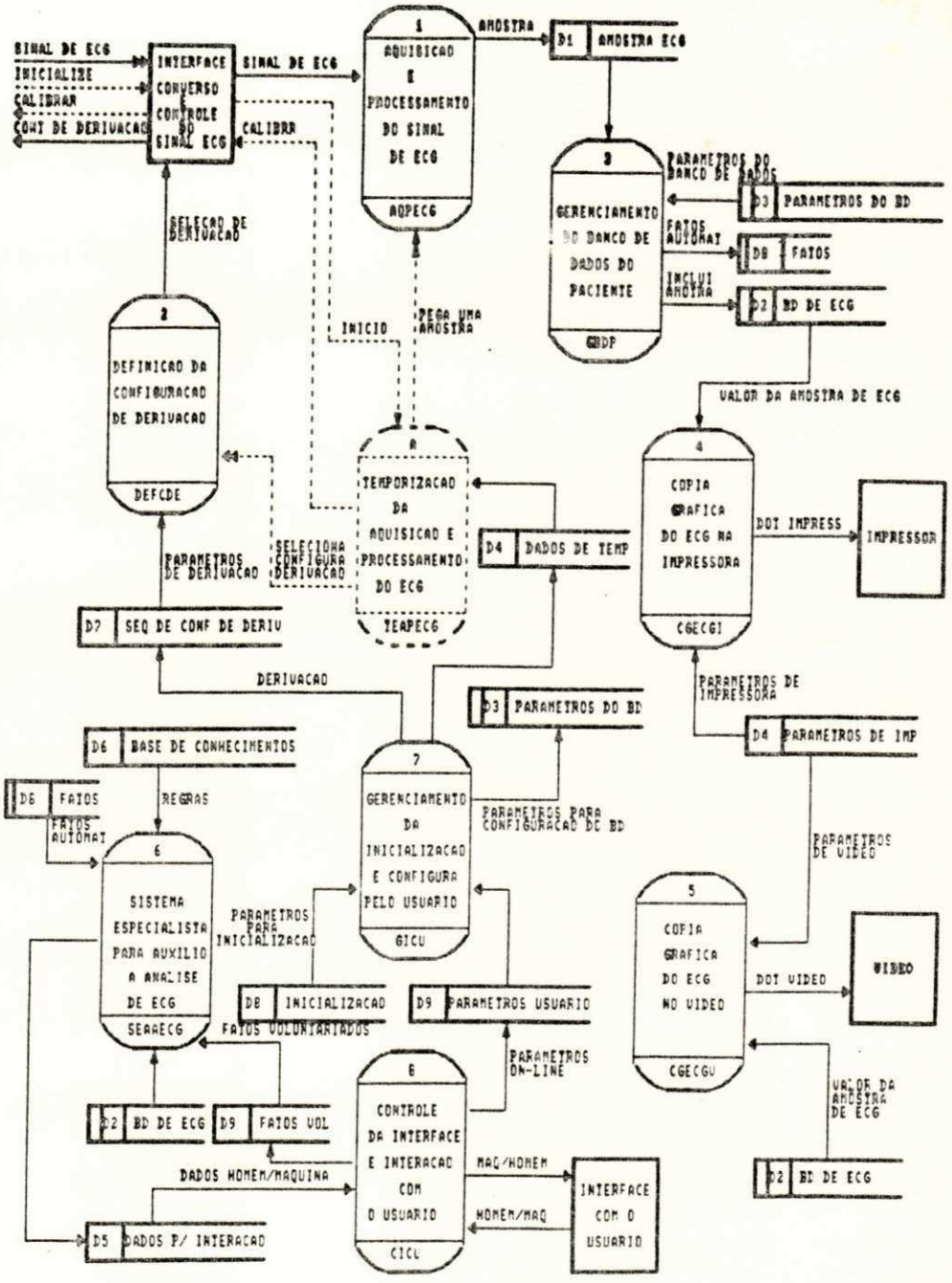


Figura A.1.: DFDC de primeiro nível para o sistema.