UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CURSO DE POS-GRADUAÇAO EM CIENCIA DA COMPUTAÇAO

FERRAMENTA PARA APOIO A MODELAGEM DE SISTEMAS COM REDES DE PETRI

por

Alvaro Guarda

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

Orientador:

Prof. Carlos Alberto Heuser

Co-orientador:

Prof. Antônio Carlos da Rocha Costa

Porto Alegre, novembro de 1989.

Guarda, Alvaro

Ferramenta para apoio à modelagem de sistemas com redes de Petri.

Porto Alegre, CPGCC da UFRGS, 1989.

1v.

Diss. (mestr. ci. comp.) UFRGS-CPGCC, Porto Alegre, BR-RS, 1989.

Dissertação: Modelagem de Sistemas: Redes de Petri: Verificação Automática de Propriedades: Simulação.

AGRADECIMENTOS

A minha mãe Remy e ao meu pai Angelo, pela educação e incentivo aos meus estudos. A Antônio e Elena, José Luiz, Nelcy e Miguel, Maria, e em especial à Vera Lúcia Miranda pelo apoio e paciência nos momentos difíceis.

Aos Profs. Carlos Alberto Heuser e Antônio Carlos da Rocha Costa pela orientação, excelentes sugestões e grande paciência durante a elaboração desta dissertação.

Aos Profs. Claudio Walter e José Mauro Volkmer de Castilho pelas valiosas sugestões e críticas por ocasião do seminário de andamento da dissertação.

Aos colegas e amigos Aliomar e Adja Mariano Rego, Ana Teresa C. Martins, Ana Murr, Beatriz Regina Tavares Franciosi, Carlos A. Prolo, Deoni e Marlete Segalin, Edeval Ari Vieira, Eduardo Todt, Eloi Favero, Griselda E. Jara, Javan e Rosa de Castro Machado, Javier Lopez, João Paulo Kitajima, José Carlos Bins Filho, José Dirceu G. Ramos, José M. Rodrigues Junior, José Roque Voltolini da Silva, Jorge Sampaio Farias (Baiano), Karin Christine Kipper, Mara Abel, Marco A. de Oliveira, Marco A. Visintin, Paulo Henrique Lemelle Fernandes, Remis e Alba Balaniuk, Renata Vieira, Ricardo Vieira, e Walcélio e Keila Lousada de Melo, pelo apoio, companheirismo e amizade sempre presentes.

Aos funcionários da biblioteca do CPGCC-UFRGS pela presteza na obtenção de obras e periódicos solicitados.

Aos funcionários da administração pelo apoio e serviços prestados.

Aos demais professores, colegas e funcionários do CPGCC-UFRGS que embora não citados aqui, contribuiram para o desenvolvimento deste trabalho.

Aos colegas e amigos da PUC/RS pelo constante incentivo e companheirismo.

Ao CNPq e CAPES pelo apoio financeiro, sem o qual, apesar dos atrasos, não seria possível a realização deste trabalho.

A minha mãe

Ao meu pai (in memoriam)

A futura geração:
Diego,
Denis,
Marynes,
Francisco,
Lisi,
Levi,
Andressa, e
Vinicius.

"A luta contra o erro tipográfico tem algo de homèrico. Durante a revisão os erros se escondem, fazem-se positivamente invisíveis. Mas assim que o livro sai, tornam-se visibilíssimos, verdadeiros sacis vermelhos a nos botar a língua em todas as páginas. Trata-se de um mistério que a ciência não consegue decifrar..."

Monteiro Lobato

SUMARIO

GLOSSARIO

ALCANÇABILIDADE é a propriedade de ser alcançável. Tradução para a palavra inglesa "reachability". Alguns autores chamam de atingibilidade.

CARDINALIDADE é a propriedade que denota o número de elementos de um objeto ou de um conjunto.

DEFAULT é uma palavra inglesa que significa assunção em caso de falha ou ausência.

INICIALIZAR é um verbo, do jargão da computação, que significa efetuar ações iniciais, como por exemplo, atribuir um valor inicial a uma variável.

LOOPING é uma palavra inglesa que significa, em computação, execução infinita, normalmente causada pela existência de ciclos ou laços.

MANIPULAÇÃO DIRETA é um estilo de interação com o usuário cuja característica principal é a manipulação de objetos e ações de interesse, os quais são apresentados graficamente na tela. Isto evita a necessidade da utilização de uma linguagem de comandos.

MENU é um conjunto de opções, no jargão da computação, que são apresentadas na tela para que o usuário faça alguma escolha. Palavra francesa que significa cardápio.

PONTO DE BLOQUEIO é uma marcação que não possui alterações habilitadas. Neste ponto uma rede de Petri fica bloqueada.

TOP-DOWN é, em computação, um método ou maneira de definição ou de desenvolvimento de programas, em que primeiro define-se um conceito ou solução com outros conceitos ou problemas menos complexos, ainda que não suficientemente detalhados. Palavra inglesa que significa de cima para baixo.

UNIVERSO DE DISCURSO é o conjunto de todas entidades (objetos) sobre as quais se está tratando.

VIVACIDADE é a propriedade de uma rede de Petri marcada estar viva, isto é, não existem marcações alcançáveis que sejam mortas. Tradução para a palavra inglesa "liveness".

LISTA DE ABREVIATURAS

BD: Banco de Dados.

BNF: Backus-Naur Form.

BR: Banco de Redes.

CAPES: Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior.

CNPq: Conselho Nacional de Desenvolvimento Científico e Tecnológico.

CPGCC: Curso de Pós-Graduação em Ciência da Computação.

EC: Editor de Consultas.

ER: Editor de Redes.

IA: Inteligência Artificial.

IBM: International Business Machine.

LARP: Linguagem de Anotação das Redes de Petri.

LC: Linguagem de Consulta.

LRC: Linguagem de Representação do Conhecimento.

PC: "Personal Computer".

PUC/RS: Pontifícia Universidade Católica do Rio Grande do Sul.

UD: Universo de Discurso.

UFRGS: Universidade Federal do Rio Grande do Sul.

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O trabalho propõe uma ferramenta para apoio à modelagem de sistemas utilizando como linguagem de modelagem as Redes de Petri.

São discutidos que tipos de auxílio são necessários no processo de modelagem de sistemas e as classes de Redes de Petri que podem ser utilizadas na ferramenta proposta.

A dissertação mostra a estrutura e a arquitetura da ferramenta, descreve a implementação de um protótipo e apresenta um exemplo de uso deste. Na definição da ferramenta é dada ênfase na verificação automática de propriedades das redes.

ABSTRACT

A tool to support system modeling with Petri Nets is proposed.

The kinds of assistance needed in the modeling system process, and the Petri Net classes that can be used in the proposed tool are discussed.

The dissertation shows the structure and the architecture of the tool, describing the prototype implementation and presenting an example of its use. In the definition of the tool, emphasis is given in the automatic verification of the net properties.

1 INTRODUÇAO

1.1 Motivação

A constante diminuição do custo do hardware nos anos anteriores, teve por consequência um crescente interesse das mais diversas àreas na utilização do computador para a solução de problemas. Para facilitar o desenvolvimento de soluções para estes problemas, foram desenvolvidas vàrias linguagens de modelagem de sistemas, entre elas estão as Redes de Petri.

O fato de Redes de Petri terem uma expressão gràfica facilita a visualização e compreensão de soluções, permitindo ainda modelar tanto propriedades estàticas quanto propriedades dinâmicas e descrever situações onde ocorre paralelismo e concorrência. Associado à estas vantagens tambèm hà o fato de Redes de Petri terem uma base formal que permite uma interpretação exata e precisa, e possibilitou o desenvolvimento e consolidação de grande parte de sua teoria. Estas e outras qualidades que Redes de Petri possuem como linguagem de modelagem, o que mostra todo o seu potencial de utilização, serão abordadas com maior profundidade na seção 2.3 Redes de Petri como Linguagem de Modelagem.

Apesar da evidente vantagem do uso da modelagem que è a possibilidade de estudar um sistema modelado sem o custo, perigo, inconveniência ou atè mesmo impossibilidade de observar o comportamento de um sistema real, o processo de modelagem de sistemas è extremamente caro e demorado, como serà visto no segundo capítulo. Procurando minimizar este problema começaram a aparecer diversas ferramentas que facilitassem o processo de modelagem de sistemas. Porèm são poucas as ferramentas que auxiliam o usuàrio em todas as

fases no processo de modelagem, principalmente no que se refere à análise ou validação de modelos.

Todos estes fatos motivaram o desenvolvimento deste trabalho que tem por objetivo definir uma ferramenta computacional que auxilie o usuário na criação de modelos de sistemas utilizando Redes de Petri, permitindo verificar se o modelo é consistente ou se as propriedades desejadas estão presentes no modelo. Por exemplo, confirmar se uma determinada marcação é alcançável ou a existência de bloqueios.

Uma ferramenta deste tipo, para ser completa, deve auxiliar o modelador em todas as fases que envolvem a modelagem de sistemas. Porèm, pela questão da impossibilidade de tratar adequadamente todos os aspectos que envolvem uma ferramenta completa, foi dada uma ênfase menor no que se refere à interface com o usuário, visando concentrar os esforços na verificação automática de propriedades.

1.2 Estudos Iniciais

A idéia inicial desta dissertação era fazer um mapeamento das redes de Petri em uma linguagem cuja teoria estivesse bastante consolidada, no caso, lógica de primeira ordem. Com isto tencionava-se utilizar provadores automáticos de teoremas para deduzir propriedades de modelos. Ou seja, propriedades de um modelo, cujas verificações fossem interessantes, seriam colocadas como teoremas e algum provador automático de teoremas faria a verificação destes teoremas (propriedades) contra uma teoria que seria o mapeamento deste modelo na lógica.

Porèm, a lógica é uma linguagem bastante genèrica, isto é, com um poder de expressão muito grande. Este fato a

torna uma linguagem não decidível e faz com que o desenvolvimento de provadores automàticos de teoremas seja bastante complexo, alèm de existirem provadores apenas para algumas partes da lògica como Clàusulas de Horn.

Associado a estes problemas, hà outro inconveniente que è a monotonicidade da lògica que torna obrigatòria a utilização de artifícios, como o uso de estados nos predicados e o predicado especial **depois(e1, e2)**, para descrever situações não-monotònicas, típicas das redes de Petri. Isto torna a descrição de Redes de Petri através da lògica não muito natural.

Procurou-se outras alternativas como a utilização de lògica temporal. Porèm, o desenvolvimento de provadores automàticos de teoremas para este tipo de lògica parece estar muito pouco consolidado jà que não se encontrou muito material sobre o assunto.

Assim achou-se conveniente utilizar simulação em um esquema de representação mais limitado e orientado para Redes de Petri, e portanto mais simples de desenvolver-se. Consegue-se tambèm, desta maneira, fazer um tratamento computacional mais eficiente e efetivo.

1.3 Trabalhos Relacionados

Nesta seção serà vista a contribuição de algumas ferramentas às diversas fases que envolvem o processo de modelagem utilizando Redes de Petri.

No que se refere à fase de modelagem ou criação de modelos existem diversas ferramentas que auxiliam o usuàrio fornecendo uma interface gràfica que facilita a criação e edição de modelos. Alguns colocam atè mesmo regras de integridade visando evitar erros por parte do usuàrio. Ferramentas deste tipo são vistas em [MEL 89], [MEN 89] e

[OLI 86].

Na fase de anàlise ou validação, que è onde està a contribuição maior desta dissertação, encontrou-se apenas dois trabalhos que são fortemente relacionados. Estes trabalhos são comentados, a seguir, com maior profundidade.

Em [NIE 86] è apresentada uma ferramenta que faz verificação de propriedades em redes de Petri do tipo predicado/transição. Para isto è feita simulação utilizando conceitos de IA e a linguagem Prolog. O trabalho è excelente porèm apresenta alguns problemas que esta dissertação procura contornar. Para verificar a alcançabilidade de marcações, por exemplo, è feita pesquisa em espaço de estados utilizando o mètodo de pesquisa em profundidade. Este è um mètodo dito cego porque faz uma pesquisa exaustiva, sem questionar o caminho tomado. Alèm disto, não leva em conta estados duplicados podendo, desta forma, entrar em ciclos infinitos, o que torna a verificação da propriedade indecidìvel. Outro problema è o fato do usuàrio não ter influência no processo de pesquisa (simulação), ou seja, não pode tentar melhorar a performance da verificação. A interface com o usuàrio è ponto o fraco desta ferramenta. As idèias de como representar redes de Petri e de utilizar pesquisa em espaço de estados tiveram uma forte influência nesta dissertação.

O outro trabalho, [OBE 87], apresenta uma ferramenta de gerenciamento e anàlise (verificação de propriedades) para redes de Petri do tipo predicado/transição. E utilizado um banco de dados para armazenar redes e marcações, e para efetuar a interface com o usuàrio. Tem facilidades para a criação de modelos, porèm a interface não è gràfica apesar de permitir a observação de redes na sua forma gràfica. O mòdulo da ferramenta que permite

analisar redes foi baseado em [NIE 86] e está implementado em Prolog. A comunicação deste módulo com o restante da ferramenta é através de arquivos e a sua execução é feita de forma isolada.

1.4 Estrutura Geral do Texto

O texto deste trabalho tem a seguinte estrutura geral:

O segundo capítulo aborda e discute alguns aspectos relacionados com a modelagem de sistemas, procurando mostrar o papel das redes de Petri como linguagem de modelagem.

O terceiro capítulo define as Redes Marcadas, linguagem de modelagem suportada pela ferramenta, e expõe as restrições impostas à sua utilização na ferramenta proposta neste trabalho.

No quarto capítulo são levantadas as necessidades básicas que uma ferramenta de apoio à modelagem de sistemas deve satisfazer, e, baseado nisto, apontados os tipos de auxílio que devem ser oferecidos. Neste capítulo, são também apresentados a estrutura e os tipos de auxílios oferecidos ao usuário, bem como o modo pelo qual o usuário deve solicitar alguns destes auxílios. É dada ênfase ao auxílio relativo à verificação automática de propriedades de redes.

No capítulo cinco é apresentada e estudada a arquitetura do módulo da ferramenta que faz a verificação de propriedades. Isto é feito discutindo os métodos e técnicas utilizados para este propósito, e apresentando detalhadamente os elementos que compõem esta arquitetura.

O capítulo seis mostra o estado atual da implementação do protótipo, em Prolog, descrevendo a sua

estrutura e como são verificadas, a nível de instrução, as propriedades que podem ser consultadas.

Os capítulos quatro, cinco e seis vão progressivamente delimitando o escopo do assunto estudado e detalhando-o até o nível de implementação de alguns tópicos.

O sétimo capítulo apresenta um exemplo de utilização do protótipo implementado procurando deixar claro o seu potencial como auxílio para a compreensão e validação de modelos.

Finalmente, no capítulo oito são apresentadas as conclusões e sugestões para futuras pesquisas.

Inclui-se ainda neste trabalho, um apêndice onde é apresentada a listagem do programa completo que implementa o protótipo explicado no sexto capítulo.

2 MODELAGEM DE SISTEMAS

2.1 Introdução

Um modelo é uma representação, frequentemente em termos matemáticos, das características, consideradas importantes para um certo fim, de um sistema. Esta representação é obtida através de um processo de abstração, chamado modelagem, utilizando uma linguagem de modelagem.

A principal motivação do uso da modelagem é a possibilidade de estudar um sistema modelado sem o custo, perigo, inconveniência ou até mesmo impossibilidade de observar o comportamento do sistema real. Um exemplo disto é a criação de um modelo para uma linha de produção existente. Se houver segurança de que este modelo representa satisfatoriamente o sistema real, então é possível estudar-se este sistema através do estudo do comportamento do seu modelo. Pode-se, também, alterar o modelo visando estudar sistemas propostos alternativos e desta forma identificar as alterações necessárias para otimizar a performance do sistema real. Isto tudo, sem o custo de criar-se sistemas alternativos reais. Além do exemplo visto, há uma grande variedade de possíveis aplicações do processo de modelagem de sistemas, como por exemplo sistemas financeiros para auxílio na tomada de decisões, sistemas de informação, sistemas astronômicos para estudo de fenômenos muito demorados, sistemas biológicos ou sociológicos, entre outros. Para maiores informações sobre modelagem e simulação veja [EMS 70] e [FRA 77].

Resumindo, é possível efetuar-se análises do modelo visando derivar propriedades ainda não identificadas. Isto pode ser utilizado para avaliar e sugerir mudanças ou melhorias no sistema modelado. Dependendo da linguagem de modelagem, estas análises podem ser feitas através de

anàlise matemàtica ou de simulação do modelo, o que, por sua vez, pode ser efetuado de maneira automàtica.

O processo de modelagem, alèm de poder ser aplicado em sistemas jà existentes, pode ainda ser utilizado em sistemas inexistentes. Neste caso, o resultado da modelagem pode ser visto como uma descrição ou uma especificação. Se a linguagem de modelagem for suficientemente formal, o modelo poderia ser utilizado tambèm para efetuar algum tipo de implementação automàtica.

Apesar de todas estas vantagens è importante salientar que a modelagem de um sistema complexo pode ser um processo extremamente caro e demorado. Este problema tem origem em trés principais possiveis causas, alèm da pròpria complexidade inerente ao sistema modelado. A primeira causa està relacionada com as caracteristicas das linguagens de modelagens, como serà visto na pròxima seção.

A segunda està relacionada com o fato de que o modelo deve comportar-se da maneira desejada, representando adequadamente a realidade modelada, ou seja, o modelo deve ser consistente. Não existe alguma maneira formal, e portanto segura, de garantir a consistència de um modelo. No estàgio atual da tecnologia de modelagem de sistemas, qualquer afirmação a respeito da consistència de modelos è feita pelas pessoas que desenvolvem estes modelos ou atè mesmo por especialistas na àrea de conhecimento do sistema modelado [TUR 87]. Entretanto, dependendo da linguagem de modelagem, podem existir tècnicas de anàlise ou ferramentas que dão subsidios ou auxiliam nesta questão.

E, finalmente, a ultima causa està relacionada com a falta de uma metodologia de modelagem suficientemente madura para uso pràtico [HEU 88].

Apesar da inexistência de uma metodologia madura, o processo completo de modelagem de um sistema envolve uma série de atividades que podem ser organizadas visando auxiliar este processo. Estas atividades basicamente dividem-se em três fases que formam um ciclo que é repetido até o modelo não apresentar problemas inaceitáveis. Como pode ser observado na figura 2.1, estas fases são criação do modelo (modelagem), validação e/ou análise, e revisão. Podem, ainda, ser observados na figura, os objetos que tomam parte no processo.

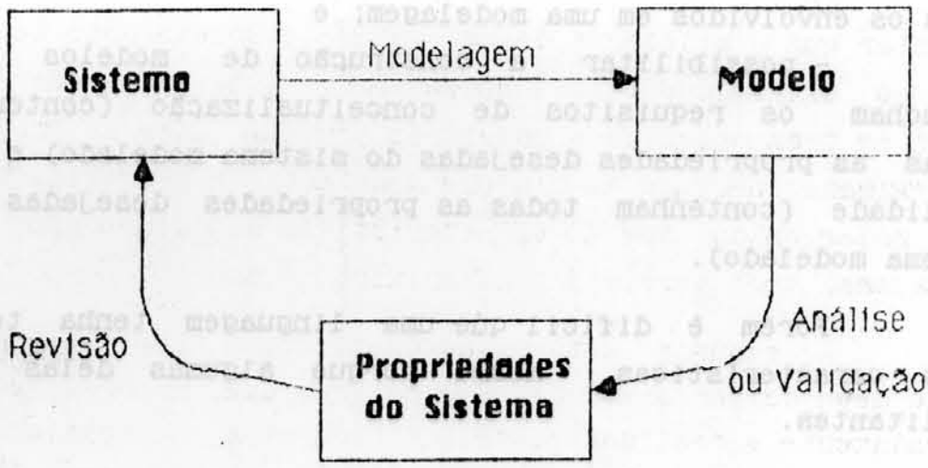


Figura 2.1 - Fases de atividades no processo de modelagem de sistemas.

Neste processo, um sistema é modelado utilizando uma linguagem de modelagem adequada para modelar as características relevantes do objeto. Após obter-se um modelo, não necessariamente completo, ele é validado e/ou analisado para verificar se o mesmo se comporta de acordo com o sistema real e/ou possue todas características desejadas. Caso o modelo apresentar problemas ou propriedades inaceitáveis, então ele passa por uma revisão onde são feitas alterações visando torná-lo consistente.

2.2 Linguagens de Modelagem de Sistemas

Uma linguagem de modelagem é o meio pelo qual se expressam modelos. Como o principal objetivo de uma linguagem de modelagem é a descrição de sistemas (construção de modelos), ela deve possuir uma série de características orientadas a esta atividade. Em [HEU 88] são apresentadas como necessárias as seguintes características:

- possuir uma base formal, visando obter-se uma interpretação exata e precisa;
- clareza, visando facilitar a comunicação entre todos os envolvidos em uma modelagem; e
- possibilitar a construção de modelos que preencham os requisitos de conceitualização (contenham apenas as propriedades desejadas do sistema modelado) e de totalidade (contenham todas as propriedades desejadas do sistema modelado).

Porém é difícil que uma linguagem tenha todas estas características, mesmo porque algumas delas são conflitantes.

Além disto, existem outras qualidades não mencionadas, mas que seria interessante que uma linguagem de modelagem as possuisse. Algumas estão abaixo relacionadas:

- facilidade de aprendizado e utilização;
- existência de ferramentas e de técnicas formais de análise;
- existência de ferramentas de auxílio à modelagem;
- fartura de documentação, como livros; e
- existência de um bom número de aplicações (divulgação).

É interessante salientar que as linguagens de modelagem, normalmente, são projetadas visando o

desenvolvimento de classes especificas de sistemas, ou seja, sistemas que apresentam algum conjunto de caracteristicas em comum. Portanto, nem sempre elas são adequadas para o desenvolvimento de modelos de qualquer tipo de sistema.

2.3 Redes de Petri como Linguagem de Modelagem

Os sistemas, em geral, possuem uma série de caracteristicas comuns. Qualquer sistema è composto por vàrios componentes individuais que interagem muitas vezes de maneiras complexas. Por sua vez cada componente pode ser visto como um sistema que pode ser descrito independentemente de outros componentes, exceto pela interação destes componentes. Como sistemas normalmente estão relacionados com a idèia de tempo, uma caracteristica importante è a noção de estado dos elementos de um sistema. Alèm disto è muito comum que componentes de sistemas apresentem atividades concorrentes ou paralelas.

Redes de Petri são uma linguagem de modelagem que foi desenvolvida especificamente para utilização em sistemas discretos que possuem componentes que interagem concorrentemente [PET 81]. Deste modo ela tem caracteristicas que não a tornam a linguagem mais adequada para modelagem de alguns tipos de sistemas, como por exemplo programas sequenciais, porèm são ideais para sistemas que apresentam concorrência.

As pricipais caracteristicas de redes de Petri são:

- permitem a modelagem de sistemas discretos com alto grau de paralelismo;
- possibilitam utilizar a mesma tècnica de representação em diferentes niveis de abstração;
- permitem modelar propriedades estàticas e

dinâmicas;

- permitem a representação gráfica de modelos o que facilita a visualização e compreensão de soluções;

- possuem uma base formal;

- são inadequadas para modelar certas classes de sistemas como, por exemplo, sistemas estáticos (sem atividades) e sistemas sequenciais (atividades não concorrentes);

- possuem regras pré-definidas de "execução", assim como existem regras de inferência na lógica; e

- o comportamento de modelos é não determinístico.

Como já foi dito, as redes de Petri têm uma base formal, matemática. Isto permitiu o desenvolvimento e consolidação de sua teoria, como por exemplo técnicas de análise, ferramentas básicas e conceitos necessários para a sua aplicação. Com isto a pesquisa e a aplicação de redes de Petri estão cada vez mais sendo divulgadas e utilizadas.

3 REDES DE PETRI

3.1 Introdução

Redes de Petri são uma linguagem de modelagem de sistemas bastante formal, porèm não hà consenso na forma de defini-la, como pode ser observado em [PET 81], [REI 86] e [HEU 89]. Apesar disto, todas definições têm componentes em comum. A grosso modo, uma rede de Petri è composta por uma estrutura, uma marcação inicial, e um conjunto de regras de transformações de estados.

A estrutura de uma rede de Petri pode ser representada de duas maneiras: representação gràfica e representação formal. A representação formal è a mais utilizada para desenvolvimento de trabalhos teòricos [PET 81]. Jà a representação gràfica de redes de Petri è mais usual em modelagem, devido a maior facilidade de compreensão e visulização dos modelos.

A definição da estrutura de redes de Petri utilizada aqui, a mesma de [HEU 89], tem três tipos de elementos:

- **lugares**, cuja representação gràfica è um circulo;
- **conexões**, cuja representação gràfica è um retângulo mais um conjunto de setas, chamadas **portas**, ligando lugares; e
- **anotações**, opcionais, associadas à rede (lugares, conexões ou portas).

As **portas** podem ser de quatro tipos: alteradora de entrada, restauradora de entrada, alteradora de saìda, e restauradora de saìda.

Existem diversas classes de redes de Petri, como pode ser constatado na literatura, e as definições dos

outros dois componentes, marcação inicial e conjunto de regras de transformações de estados, dependem da classe de rede de Petri.

3.2 Redes Marcadas

Como foi dito, existem várias classe de redes de Petri, porèm a linguagem objeto deste trabalho são as Redes Marcadas.

Segue abaixo a definição de Redes Marcadas [HEU 89]. Desta definição já fazem parte as definições da marcação inicial e do conjunto de regras de transformações de estados.

1. Elementos de uma Rede Marcada:

a) Universo de discurso (UD), sendo este formado por um conjunto de entidades e um conjunto de relações e funções.

b) LARP (Linguagem de Anotação das Redes de Petri) para este UD, sendo esta definida através da indicação dos simbolos de relação e dos simbolos de função, junto com seu significado dentro do UD considerado.

c) Uma rede de Petri, anotada com a LARP, como segue:

- A cada conexão é associada uma fòrmula da LARP. Esta fòrmula, **fòrmula de conexão**, aparece, na representação gràfica da rede, dentro do retângulo representativo da conexão.

- A cada lugar são associados dois termos da LARP, e cada um deve designar um conjunto do UD. O termo designador do **tipo de marca** aparece, entre parènteses, após o nome do lugar. O termo designador da **marcação inicial** aparece dentro da elipse representativa do lugar.

- A cada porta é associado um termo, **termo de porta**, que aparece junto à seta representativa da porta.

2. Definições para uma Rede Marcada:

a) Um lugar define uma **marca** para cada entidade do conjunto designado pelo termo de tipo de marca do lugar.

b) A **marcação** de um lugar (da rede) é o conjunto de marcas presentes neste lugar (em todos lugares da rede).

c) As variáveis de uma conexão são as variáveis livres dos termos das portas da conexão.

d) Uma conexão da rede define um conjunto de **alterações**, uma para cada valorização das variáveis da conexão, que satisfaça as seguintes condições:

- o conjunto designado pelo termo de cada porta, sob a valorização considerada, deve fazer parte do tipo de marca do lugar; e

- a fórmula da conexão, quando presente, deve resultar em verdadeiro, sob a valorização considerada.

e) Para uma alteração define-se:

- As **marcas de entrada** da alteração como sendo o conjunto de marcas indicadas pela valorização dos termos das portas de entrada da conexão.

- As **marcas de saída** da alteração como sendo o conjunto de marcas indicadas pela valorização dos termos das portas de saída da conexão.

- As **marcas alteradas** pela alteração como sendo o conjunto de marcas indicadas pela valorização dos termos das portas alteradoras da conexão.

- As **marcas restauradas** pela alteração como sendo o conjunto de marcas indicadas pela valorização dos termos

das portas restauradoras da conexão.

f) **Regra de habilitação**: uma alteração está habilitada frente a uma marcação, quando:

- as marcas de entrada da alteração estiverem presentes dentro da marcação considerada; e
- as marcas de saída da alteração estiverem ausentes na marcação considerada.

g) Duas alterações são conflitantes quando ambas possuem marcas de entrada ou marcas de saída comuns.

h) Um conjunto de alterações é um **passo** frente a uma marcação (marcação precursora) quando:

- todas alterações do passo estão habilitadas dentro da marcação precursora; e
- as alterações do passo não são conflitantes entre si.

i) O **efeito** da ocorrência das alterações de um passo é a transição da marcação precursora para uma marcação sucessora de tal forma que:

- todas marcas alteradas de entrada das alterações do passo desapareçam na marcação sucessora;
- todas marcas alteradas de saída das alterações apareçam na marcação sucessora; e
- nenhuma outra diferença exista entre as duas marcações, precursora e sucessora.

j) Frente a uma marcação inicial da rede define-se marcações alcançáveis da rede como sendo qualquer marcação obtida pelo efeito da ocorrência de sucessivos passos sobre a marcação inicial.

Um UD específico deve ser definido para cada modelo particular, já que as entidades, relações e funções do UD são os objetos que se está tratando em um modelo.

A LARP representa uma classe de linguagens com alguns elementos prè-definidos, e com outros elementos que devem ser definidos conforme as necessidades dos modelos. A semântica dos elementos prè-definidos da linguagem podem ser vistos em [HEU 89].

A parte da LARP que deve ser definida tem uma semântica particular para cada modelo e os seus símbolos de relações e de funções são específicos para cada modelo. Porèm è importante que a sintaxe da LARP seja universal, ou seja, sirva para qualquer modelo. Segue abaixo a sintaxe da LARP [HEU 89].

Para a descrição da sintaxe das sentenças na LARP è empregado o mètodo BNF. São adotadas as seguintes convenções para a BNF:

::	è definido como;
¦	alternativa;
.	fim da definição;
grifo	meta-símbolo;
negrito	símbolo terminal;
...	quantidade variável de símbolos (zero ou mais).

Definição da LARP:

fòrmula :: **verdadeiro** ¦
falso ¦
relação(termo, ...) ¦
(termo relação binária termo) ¦
(fòrmula **e** fòrmula) ¦
(fòrmula **ou** fòrmula) ¦
não fòrmula ¦
(fòrmula **impl** fòrmula) ¦
paratodo var (fòrmula) ¦

existe var (fórmula) ¦
(fórmula) .

relação binária :: Elem ¦
Sub ¦
= ¦
outros definidos de acordo com o modelo.

termo :: constante ¦
var ¦
função(termo) ¦
(termo função binária termo) ¦
{termo, ... } ¦
{var ¦ fórmula} ¦
<termo, ... > ¦
(termo x termo) ¦
(termo) .

função binária :: x ¦
outros definidos de acordo com o modelo.

Observações:

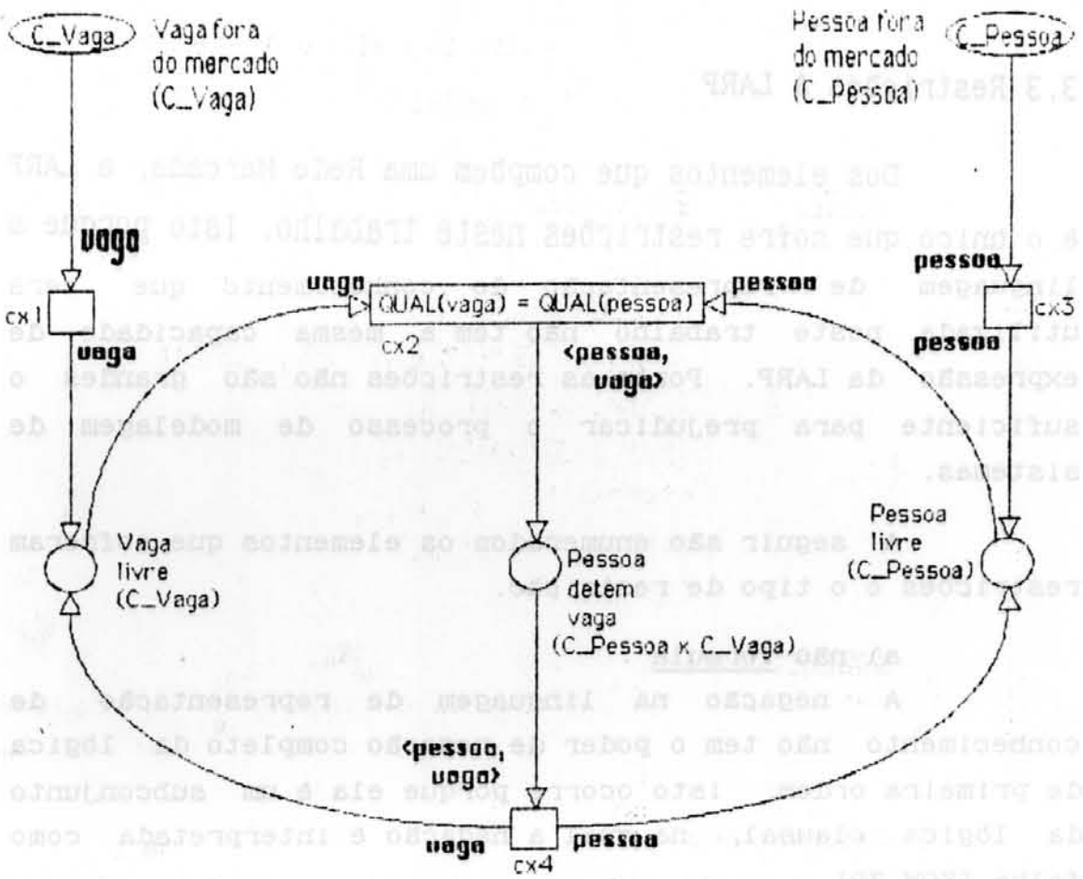
a) relação ou relação binária é um símbolo relacional representado por uma palavra iniciada por letra maiúscula e podendo conter o símbolo _. Excepcionalmente o símbolo de relação binária pode ser um símbolo especial, como os símbolos =, >, <.

b) função ou função binária é um símbolo funcional representado por uma palavra composta exclusivamente por letras maiúsculas e podendo conter o símbolo _.

c) var é uma palavra composta exclusivamente de letras minúsculas, podendo conter o símbolo _.

d) constante é uma cadeia de caracteres,

delimitada por apòstrofes. No caso de constantes numèricas, os apòstrofes devem ser suprimidos. No caso de uma constante que designa um conjunto de entidades, constante è uma palavra iniciada por C_.



Universo de Discurso (UD)

C_Vaga = { v1, v2 }

C_Pessoa = { p1, p2, p3 }

C_Qual = { q1, q2, q3 }

função QUAL

Pessoa ou Vaga	Qualificação
p1	q1
p2	q2
p3	q1
v1	q1
v2	q3

Figura 3.1 - Modelo do mercado de trabalho e seu Universo de Discurso.

A figura 3.1 mostra um exemplo de rede marcada com o respectivo UD. Este modelo apresenta três conjuntos de entidades (pessoas, vagas e qualificações) e apenas uma função (QUAL) que associa às pessoas e às vagas suas respectivas qualificações.

3.3 Restrições à LARP

Dos elementos que compõem uma Rede Marcada, a LARP é o único que sofre restrições neste trabalho. Isto porque a linguagem de representação do conhecimento que será utilizada neste trabalho não tem a mesma capacidade de expressão da LARP. Porèm as restrições não são grandes o suficiente para prejudicar o processo de modelagem de sistemas.

A seguir são enumerados os elementos que sofreram restrições e o tipo de restrição.

a) **não** fòrmula :

A negação na linguagem de representação de conhecimento não tem o poder de negação completo da lògica de primeira ordem, isto ocorre porque ela è um subconjunto da lògica clausal, na qual a negação è interpretada como falha [KOW 79].

b) (fòrmula **impl** fòrmula) :

O impl è definido atravès da negação:

f1 impl f2 = não f1 ou f2

portanto, o que foi dito para o operador anterior è vàlido para este operador lògico.

c) **paratodo** var (fòrmula) :

Não è colocado porque as fòrmulas da linguagem de representação de conhecimento são, na verdade, sentenças, ou seja, todas as variàveis são quantificadas com o

quantificador universal, e por simplicidade são omitidos.

d) **existe** var (fórmula) :

Não é definido.

e) Funções :

Não são definidas.

f) {termo, ... } :

É permitida a existência de duplicidade de elementos dentro do conjunto.

g) {var ¦ fórmula} :

Não é definido.

h) (termo **x** termo) :

Não é definido.

3.4 Utilização de Outros Tipos de Redes

Na literatura existem diversos tipos de redes de Petri, cada uma com suas características particulares. Esta seção procura mostrar, rapidamente, que procedimentos adotar para permitir o uso da ferramenta, definida neste trabalho, na modelagem de sistemas utilizando alguns destes tipos de redes.

As redes Condição/Evento (C/E) têm as seguintes características:

- cada lugar pode ter apenas uma marca (cardinalidade unitária);
- não interessa o tipo de marca dos lugares;
- a única anotação deste tipo de rede é a marcação inicial; e
- possui apenas portas alteradoras.

Então para criar-se modelos com redes C/E basta que se defina capacidade igual a 1 (um) para todos os

lugares da rede, não se coloque anotações na rede alèm da marcação inicial, e utilize-se apenas portas alteradoras.

As redes Predicado/Transição (Pr/T) estritas têm as seguintes caracteristicas:

- as marcas têm identidade;
- os lugares podem ter diversas marcas, porèm não è permitido mais que uma còpia de uma mesma entidade em um lugar;
- podem existir portas com mais de um termo (lista de termos), desde que todos os termos sejam diferentes e a cardinalidade desta lista de termos seja constante;
- possui apenas portas alteradoras; e
- não existem anotações nas conexões.

Então, para criar-se modelos com redes Pr/T restritas, basta que se utilize apenas portas alteradoras sem conjuntos como termos de porta, e não se coloque anotações nas conexões da rede. Como, na linguagem suportada pela ferramenta deste trabalho, não è permitido multiplicidade de marcas, então não è necessàrio preocupar-se com isto.

As redes com multiplicidade não podem ser utilizadas porque não são permitidas mais que uma còpia de uma mesma entidade em um lugar.

4 FERRAMENTA PARA MODELAGEM

O principal objetivo deste trabalho è definir uma ferramenta computacional que efetivamente auxilie o usuàrio no desenvolvimento de modelos de sistemas utilizando, como linguagem de modelagem, Redes de Petri. Para atingir este objetivo deve-se fazer um levantamento de que tipos de auxìlio são necessàrios no processo de modelagem.

4.1 Necessidades de uma Ferramenta

Como foi visto no capitulo 2, o processo completo de modelagem de um sistema envolve uma sèrie de atividades que, basicamente, se dividem em très fases, independente da linguagem de modelagem escolhida. Estas fases são modelagem, validação e revisão.

Para uma ferramenta ser completa, do ponto de vista da utilidade, è necessàrio que a mesma auxilie o usuàrio em todos estas fases. Cada uma destas fases tem um conjunto de atividades que são caracteristicas, pròprias de cada fase, e estão diretamente relacionadas com a linguagem de modelagem. Assim, o tipo de auxìlio e a maneira como este auxìlio è oferecido e/ou implementado depende, basicamente, de duas coisas: do tipo de atividade e das caracteristicas especificas da linguagem de modelagem.

Alèm disto, o auxìlio a ser oferecido em cada fase, deve possuir uma interface com o usuàrio que tenha levado em conta, no seu desenvolvimento, principios e orientações de projeto de interfaces amigàveis. Isto visa facilitar a interação com o usàrio e afeta diretamente o tempo de aprendizado, a produtividade, o nivel das taxas de erros, e a satisfação do usuàrio. Em [SHN 87] o tema è abordado com bastante propriedade.

4.1.1 Fase de Modelagem

Durante a fase de modelagem, propriamente dita, todas as atividades estão relacionadas com a criação de modelos, portanto torna-se necessário que haja um editor de redes de Petri. Tendo em vista que redes de Petri têm uma expressão gràfica, è interessante para o usuàrio que este editor seja um editor gràfico.

Para tonar mais efetiva a interação deste editor com o usuàrio, o estilo de interação mais adequado è o de manipulação direta. E è conveniente, tambèm, que este editor possua um conjunto de caracteristicas, orientadas à linguagem de modelagem, que facilitem a criação de modelos de sistemas nesta linguagem.

4.1.2 Fase de Validação

Na fase de validação, o usuàrio normalmente deseja saber se o modelo è consistente, ou seja, se o mesmo comporta-se da maneira desejada, representando adequadamente a realidade modelada. Para isto, è necessàrio que a ferramenta permita que o usuàrio faça consultas ao sistema, com o intuito de verificar se propriedades desejadas estão presentes no modelo. Por exemplo, verificar a inexistência de marcações no conjunto de marcações alcançàveis, que habilitem alguma alteração definida por uma asserção estàtica; ou verificar a inexistência de pontos de bloqueio na rede de Petri.

Neste sentido, è interessante que a ferramenta possua um editor de consultas, visando facilitar a efetuação de consultas por parte do usuàrio. E, alèm disto, tambèm è interessante que a obtenção das respostas, às consultas do usuàrio, seja feita o mais automaticamente possivel. Portanto, a ferramenta deve ter, na sua estrutura interna,

algo que faça a anàlise de modelos.

E importante ressaltar que a validação de modelos, permitida pela ferramenta, não è completa; isto por dois motivos. O primeiro è que a validação de um modelo envolve um processo informal. Este processo è o julgamento do usuàrio, informal e arbitràrio, sobre quais propriedades o modelo deve possuir para garantir a correção do mapeamento com a realidade, ou seja, para garantir que o modelo seja vàlido.

O segundo motivo è a possibilidade de fazer-se necessària a verificação de certos tipos de propriedades, para garantir que um modelo seja vàlido, e estes tipos de propriedades não são verificàveis pela ferramenta. Isto ocorre porque o nùmero de propriedades verificàveis, neste trabalho, è limitado. Uma discussão mais aprofundada sobre validação pode ser vista em [TUR 87] e [BER 82].

Alèm de validar modelos, o usuàrio pode utilizar este editor de consultas para fazer anàlises de modelos jà existentes e validados visando apenas observar o comportamento destes.

4.1.3 Fase de Revisão

Esta fase è necessària quando o modelo criado possui propriedades indesejàveis. Neste ponto o usuàrio provavelmente desejarà fazer alterações no modelo, visando eliminar as propriedades indesejàveis. Para auxilià-lo nesta tarefa, a ferramenta deve permitir que um modelo jà criado possa ser editado e facilmente alterado.

Isto reforça a necessidade de um editor de redes de Petri amigàvel, jà que a criação e alterações de modelos são atividades que exigem um esforço consideràvel por parte do usuàrio.

4.2 Estrutura e Descrição Geral da Ferramenta

Nesta seção, é proposta uma estrutura para a ferramenta de apoio à modelagem de sistemas que é objeto deste trabalho. Esta estrutura foi elaborada tendo em vista a satisfação das necessidades que foram levantadas na seção anterior. A descrição geral desta ferramenta é feita detalhando-se todos os módulos que compõem a sua estrutura.

A parte da ferramenta que se dedica a auxiliar o usuário na fase de validação de modelos é o objetivo principal desta dissertação. Portanto, as demais partes, relacionadas com as outras fases, modelagem e revisão, não estão detalhadas no nível de profundidade necessário para efetuar-se a sua implementação sem maiores estudos.

A figura 4.1 mostra a estrutura proposta e procura indicar o interrelacionamento dos módulos que a compõem. Como pode ser observado, a ferramenta é composta basicamente por três módulos: a interface com o usuário, a base de redes e o analisador. Cada um deste módulos é descrito nos itens subsequentes.

Na figura também pode ser observada a influência da interação com o usuário, sobre os submódulos que compõem o módulo de interface, em cada fase do processo de modelagem. No submódulo editor de redes o usuário pode interagir para criar redes (modelagem) e/ou para efetuar alterações de redes (revisão). No submódulo editor de consultas o usuário pode interagir para verificar propriedades do modelo (validação) e/ou para efetuar análise de modelos (estudar o comportamento de modelos).

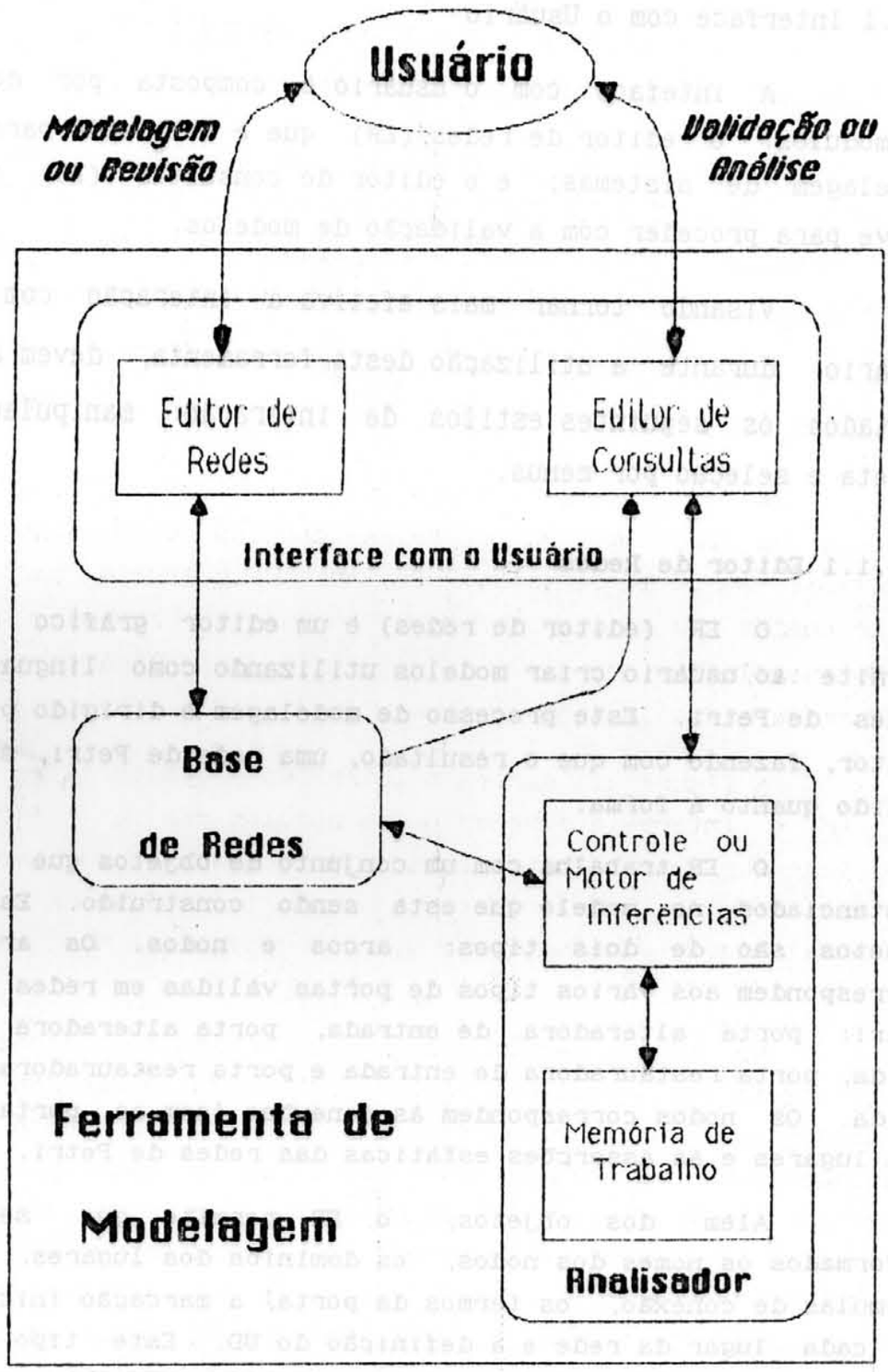


Figura 4.1 - Estrutura da ferramenta.

4.2.1 Interface com o Usuàrio

A inteface com o usuàrio è composta por dois submòdulos: o editor de redes (ER) que è utilizado para a modelagem de sistemas; e o editor de consultas (EC) que serve para proceder com a validação de modelos.

Visando tornar mais efetiva a interação com o usuàrio, durante a utilização desta ferramenta, devem ser adotados os seguintes estilos de interação: manipulação direta e seleção por menus.

4.2.1.1 Editor de Redes

O ER (editor de redes) è um editor gràfico que permite ao usuàrio criar modelos utilizando como linguagem redes de Petri. Este processo de modelagem è dirigido pelo editor, fazendo com que o resultado, uma rede de Petri, seja vàlido quanto à forma.

O ER trabalha com um conjunto de objetos que são instanciados no modelo que està sendo construido. Estes objetos são de dois tipos: arcos e nodos. Os arcos correspondem aos vàrios tipos de portas vàlidas em redes de Petri: porta alteradora de entrada, porta alteradora de saida, porta restauradora de entrada e porta restauradora de saida. Os nodos correspondem às conexões (sem as portas), aos lugares e às asserções estàticas das redes de Petri.

Alèm dos objetos, o ER permite que sejam informados os nomes dos nodos, os dominios dos lugares, as fòrmulas de conexão, os termos da porta, a marcação inicial de cada lugar da rede e a definição do UD. Este tipo de informação deve respeitar a sintaxe da LARP [HEU 89], e as restrições impostas no terceiro capitulo.

Durante a modelagem, alèm das restrições quanto ao

conjunto de objetos e das restrições de sintaxe das anotações, o ER impõem uma sèrie de restrições de integridade que servem para impedir que o usuàrio crie uma rede de Petri invàlida quanto à forma. Isto não permite, por exemplo, que seja possivel construir uma rede de Petri que tenha portas ligando nodos do mesmo tipo.

Os modelos criados são armazenados na base de redes, conforme serà visto na respectiva seção.

Existem duas camadas de comandos no ER. A primeira camada são os comandos de manipulação de modelos: criar modelo, editar modelo, salvar modelo e imprimir modelo. A segunda são os comandos de edição de modelos: deslocar janela de edição, inserir nodo ou arco, deletar nodo ou arco, e alterar nodo ou arco.

4.2.1.2 Editor de Consultas

Uma consulta è gerada atravès da interação do usuàrio com o EC (editor de consultas). O editor vai guiando o usuàrio interativamente e o resultado deste processo è a geração de uma consulta vàlida na linguagem de consulta.

Uma consulta è, na verdade, uma solicitação de verificação de alguma propriedade em um certo modelo. O processo de verificação de propriedades è um processo formal. Portanto, para que seja possivel verificar uma propriedade, è necessàrio que esta propriedade esteja formalizada. Ou seja, não deve haver ambiguidades a respeito do nome e sintaxe da propriedade e da maneira como ela è verificada. Por causa disto existe um número limitado de propriedades que esta ferramenta permite que sejam verificadas automaticamente.

Após a geração de uma consulta na LC, a propriedade correspondente è verificada automaticamente

através do analisador, como será visto com maiores detalhes no quinto capítulo.

4.2.1.2.1 Linguagem de Consulta

A principal vantagem da utilização de uma LC (Linguagem de Consulta) é a independência do procedimento de geração de consultas. Isto é, não interessa como uma consulta é gerada, desde que seja válida na LC.

Além das consultas propriamente ditas, a LC permite, ainda, mais um tipo de construção que são as informações de execução (meta-comandos de controle). Apesar deste tipo de construção não ser uma consulta, ele é incluído na LC. Isto porque a LC é, de certo modo, uma interface entre as solicitações do usuário e o analisador.

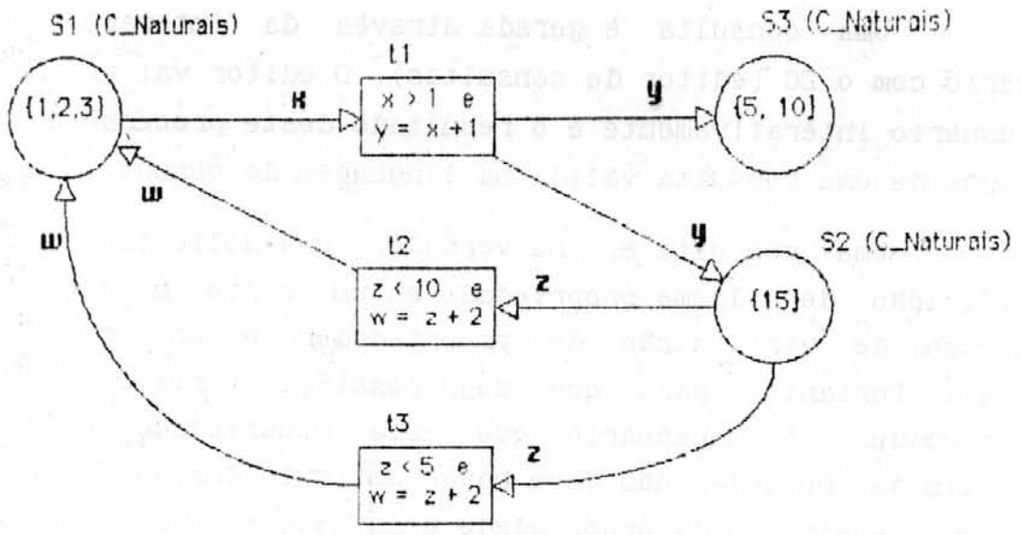


Figura 4.2 - Exemplo de Rede Marcada, com a marcação inicial.

No próximo item será definida a sintaxe dos vários tipos de construções da LC e alguns exemplos de sentenças válidas. Os exemplos serão construídos em cima da rede de

Petri da figura 4.2, a qual apresenta uma marcação inicial.

Para a descrição da sintaxe das sentenças na LC é empregada a BNF descrita no terceiro capitulo. São omitidos detalhes que complicam o entendimento da linguagem e não acrescentam vantagens relevantes para fins de uso ou anàlise da linguagem.

As convenções da BNF, utilizadas para a descrição das sentenças, são repetidas abaixo:

::	è definido como;
¦	alternativa;
.	fim da definição;
grifo	meta-simbolo;
negrito	simbolo terminal;
...	quantidade variàvel de simbolos (zero ou mais).

Como a marcação è o elemento, da sintaxe das sentenças, mais frequente e mais complicado, a seguir serà dada uma explicação do mesmo e apresentada a sua sintaxe.

A marcação de uma rede de Petri è o conjunto de marcas que estão presentes em todos os lugares da rede. Isto è representado atravès de uma lista de marcações de todos os lugares da rede. Cada lugar da rede è representado atravès de uma lista contendo dois elementos: o nome do lugar e a marcação deste lugar. A marcação de um lugar è representada atravès de uma lista das marcas que estão presentes neste lugar.

Sintaxe:

marcação :: nome de variàvel em Prolog ¦
lista de marcações de lugares.

lista de marcações de lugares ::
[marcação de lugar, ... ,marcação de lugar] ¦
[marcação de lugar¦Resto].

marcação de lugar :: [nome de lugar, lista marcas].

lista marcas :: nome de variável em Prolog ¦
[marca, ... ,marca] ¦
[marca¦Resto].

nome de lugar :: nome de um dos lugares da rede de Petri.

marca :: elemento que faz parte da marcação de um dos lugares da rede de Petri; deve pertencer ao domínio deste lugar.

Observações:

a) Resto é nome de variável e, como está sendo utilizada, significa que o resto da lista em questão pode ser qualquer lista de marcas válidas, até mesmo lista vazia.

b) A utilização de nome de variável em Prolog, na construção de uma sentença, significa que esta variável pode ser qualquer coisa válida para o item em questão.

Como exemplo de uma marcação, é mostrada a marcação inicial da figura 4.2 nesta sintaxe:

[[s1, [1, 2, 3]], [s2, [15]], [s3, [5, 10]]]

onde,

[s1, [1, 2, 3]] e'a marcação do lugar s1 que tem 1, 2 e 3 como marcas;

[s2, [15]] e'a marcação do lugar s2 que tem 15 como marca; e

[s3, [5, 10]] e' a marcação do lugar s3 que tem 5, 10 como marcas.

Outro exemplo:

[[s1, [1, 2, 3]]¦Resto]

significa que o lugar **s1** tem as marcas **1**, **2** e **3**; e os demais lugares (resto da marcação) podem ter quaisquer marcações válidas.

4.2.1.2.1.1 Tipos de Consultas

A LC, dependendo do tipo de consulta, tem uma sintaxe bem flexível, permitindo uma grande variedade de consultas válidas. Abaixo segue a descrição dos tipos de consultas permitidos e a correspondente sintaxe na LC.

a) Verificação da existência de conflito entre alterações definidas por duas conexões para uma dada marcação.

Duas alterações, de diferentes conexões, são conflitantes quando as mesmas estão habilitadas, e têm marcas de entrada e/ou saída comuns. Isto significa que a ocorrência de uma desabilita a outra, portanto elas nunca podem ocorrer em um mesmo passo.

Quando é feita uma solicitação desta consulta, o resultado obtido é a relação de pares de alterações, uma de cada conexão, que estão em conflito. Ou, caso não houver conflito, uma mensagem informando que não existem alterações conflitantes para estas conexões.

É importante salientar que as alterações definidas para uma única conexão podem ser conflitantes. Isto também vale para concorrência, o próximo tipo de consulta. Assim, é permitido informar, nos dois tipos de consulta, a mesma conexão duas vezes.

Sintaxe:

conflito(conexão**,** conexão**,** marcação**).**

onde,

conexão :: nome de variável em Prolog

nome de conexão.

Exemplos de sentenças válidas:

```
conflito(t2, t3, [[s1, [1, 2, 3]], [s2, [4, 6], [s3, []]]).
conflito(t1, t1, [[s1, [1]], [s2, [2, 3], [s3, []]]).
conflito(t1, Conexao, [[s1, [1]], [s2, [2, 3], [s3, []]]).
conflito(Conx1, Conx2, [[s1, [1]], [s2, [2, 3], [s3, []]]).
```

b) Verificação da existência de concorrência (paralelismo) entre alterações definidas por duas conexões para uma dada marcação.

Duas alterações são concorrentes quando elas estão habilitadas e, além disto, não possuem marcas de entrada e/ou saída comuns. Isto significa que estas alterações podem ocorrer independentemente uma da outra.

O resultado obtido é a relação de pares de alterações, uma de cada conexão, que são concorrentes. Se não houver nenhum caso de concorrência, então aparece uma mensagem informando que não existem alterações concorrentes para estas conexões.

Sintaxe:

concorrencia(conexão, conexão, marcação**).**

Exemplos de sentenças válidas:

```
concorrencia(t2, t3, [[s1, [2, 3]], [s2, [6], [s3, []]]).
concorrencia(t1, t1, [[s1, [1]], [s2, [2, 3], [s3, []]]).
concorrencia(t1, Cnx, [[s1, [1]], [s2, [2, 3], [s3, []]]).
concorrencia(Cx1, Cx2, [[s1, [1]], [s2, [2, 3], [s3, []]]).
```

c) Verificação de bloqueio na rede de Petri com relação a uma marcação específica.

Esta consulta verifica se a marcação informada causa um bloqueio na rede de Petri. Quando isto acontece, a rede fica "parada", não sendo mais possível ocorrerem

alterações. Isto é, não existem alterações habilitadas para esta marcação e a mesma é chamada, neste trabalho, de ponto de bloqueio.

O resultado obtido é uma mensagem informando que a marcação bloqueia a rede de Petri; ou, caso não seja um ponto de bloqueio, uma alteração que esteja habilitada para a marcação.

Sintaxe:

bloqueio(marcação).

Exemplos de sentenças válidas:

```
bloqueio([[s1, [1, 2, 3]], [s2, [4, 5, 6], [s3, []]]).
bloqueio([[s1, [1, 3]]|Resto]).
bloqueio([[s1, [1]], [s2, [10, 12]],
                    [s3, [3, 4, 6, 7, 9, 10, 12]]]).
```

d) Verificação da vivacidade de marcações ou da inexistência de pontos de bloqueio na rede de Petri.

Há várias noções de vivacidade. E uma rede de Petri pode ser dita viva se, com respeito a alguma destas noções de vivacidade para marcações, existe uma marcação inicial que seja viva [REI 86].

A noção de vivacidade adotada aqui, é de que uma marcação é viva se o seu conjunto de marcações alcançáveis não contem marcações que sejam pontos de bloqueio. Desta maneira, quando esta propriedade for verificada, isto significa também que a rede de Petri é viva.

E permitido informar também, além da marcação, a profundidade em que é feita a pesquisa para efetuar-se a verificação. A profundidade é a quantidade de ocorrência de alterações, em sequência, efetuadas em extensão na rede de Petri. Serve para limitar a pesquisa e, portanto pode não chegar a um resultado válido.

O resultado obtido é uma mensagem informando que a marcação é viva; ou, em caso contrário, a relação de pontos de bloqueio.

Sintaxe:

viva(marcação**).**

viva(marcação**,** profundidade**).**

onde,

profundidade :: número inteiro que limita a profundidade da pesquisa na verificação da propriedade.

Exemplos de sentenças válidas:

```
viva([[s1,[1,2,5],[s2,[]],[s3,[3]]]).
viva([[s1,[1,2,5],[s2,[]],[s3,[3]]],5).
```

e) Verificação da validade de estado futuro ou estado passado a partir de um estado inicial.

É verificado se uma marcação (estado) especifica é alcançável (válido) para uma marcação inicial. Ou seja, a existência de uma sequência válida de ocorrências de alterações na rede de Petri que altere a marcação de tal modo que fique igual à marcação final.

Se não for desejado fazer-se esta verificação com a marcação inicial da rede de Petri, então basta informá-la. Se a marcação inicial não for informada então é assumida a marcação inicial da rede de Petri.

A alcançabilidade, além da verificação para frente que é a normal, pode ser verificada para trás. Isto é feito através das informações de execução como será visto no próximo item.

A resposta obtida é a sequência de alterações que levam à marcação informada, ou uma mensagem informando que

esta marcação não é alcançável. Além disto, também são informados os pontos de bloqueio que foram encontrados durante a pesquisa.

Sintaxe:

alcancabilidade(marcação inicial**,** marcação final**).**

alcancabilidade(marcação final**).**

onde,

marcação inicial :: marcação.

marcação final :: marcação.

Exemplos de sentenças válidas:

```
alcancabilidade([[s1,[5,6]],[s2,[]],[s3,[3,4]]],
                [[s1,[6]],[s2,[6]],[s3,[3,4,6]]]).

alcancabilidade([[s1,[1,2]],[s2,[4]],[s3,[4]]],M).

alcancabilidade([[s1,[1,3]]|Resto],
                [[s1,[1],[s2,[8]],[s3,[3,4,8]]]).

alcancabilidade([[s1,[1]],[s2,[8]],[s3,[3,4,8]]]).

alcancabilidade([[s1,[2]]|Resto]).
```

4.2.1.2.1.2 Informações de Execução

Na LC também estão definidas as possíveis opções de execução do processo de simulação na rede de Petri. Estas informações de execução são, na verdade, informações específicas do domínio da aplicação, as quais o usuário pode fornecer. No quinto capítulo estas informações são chamadas de meta-comandos de controle, e têm este nome porque os mesmos alteram a maneira como é feita a verificação de propriedades. Para entender-se bem este tipo de informação é necessário saber como é feita a verificação de propriedades, o que é explicado no quinto capítulo. No item 5.3.3.2.2

Entrada de Conhecimento Antes da Simulação, é explicado qual é o efeito de cada um destes meta-comandos de controle. Segue abaixo a relação de meta-comandos de controle permitidos com um breve comentário e a sua sintaxe.

a) Validação de alterações.

É possível definir-se quais alterações serão válidas, em uma verificação de alcançabilidade específica. Isto serve para que o processo de verificação seja feito de maneira mais rápida já que apenas as alterações informadas como válidas serão utilizadas na verificação. Se nenhuma alteração for informada, então todas serão consideradas válidas.

Para entrar-se com este tipo de informação deve-se fornecer, para cada alteração que deve ser válida, o nome da conexão e o conjunto de portas alteradoras de entrada desta conexão com os termos de porta de entrada já valorizados, ou seja, com as marcas que tomarão parte na alteração já especificadas. Isto é, em outras palavras, uma alteração definida pela conexão.

Sintaxe:

alteracao_valida(nome conexão, conj entrada**).**

onde,

conj entrada :: nome de variável em Prolog ¦
lista portas entrada.

lista portas entrada :: [porta e, ... , porta e] ¦
[porta e¦Resto].

porta e :: [nome de lugar, marca].

Exemplos de sentenças válidas:

```
alteracao_valida(t1,[[s1,3]]).
alteracao_valida(t1,[[s2,6]]).
alteracao_valida(t2,[[s2,4]]).
```

alteracao_valida(t2,[[s2,7]]).

b) Validação de conexões.

Este tipo de informação, da mesma forma que o anterior, também serve para tornar mais rápido o processo de verificação de propriedades, pois apenas as conexões validadas pelo usuário serão utilizadas. Se nenhuma for informada, então todas conexões serão consideradas válidas.

Para definir-se quais conexões serão válidas, em uma verificação de alcançabilidade específica, deve-se enumerá-las conforme a sintaxe definida abaixo. Cada conexão válida equivale a uma classe de alterações válidas, isto é, a todas alterações definidas por esta conexão.

Sintaxe:

conexao_valida(conexão**).**

Exemplo de sentença válida:

conexao_valida(t1).

c) Declaração de procedimento para validar conexão.

Esta alternativa é semelhante à anterior, com a diferença que é voltada para usuários mais especializados, já que são necessários conhecimentos de programação na linguagem Prolog.

O usuário deve definir uma ou mais regras (procedimentos), respeitando a sintaxe de Prolog, conforme o formato definido.

Esta alternativa valida classes de conexões.

Sintaxe:

conexao_valida(Conexao) :- condições**.**

onde,

condições :: procedimentos ou expressões lógicas, válidos em Prolog.

Exemplo de sentença válida:

```
conexao_valida(Conexao) :-
    conexao(Conexao, ConjEntr, [[Lugar, Termo]|R]),
    capacidade(Lugar, LimCap),
    LimCap @< 3.
```

Este procedimento está validando apenas aquelas conexões cuja primeira porta, do conjunto de portas de saída, tem o lugar com limite de capacidade menor que três.

Os procedimentos **conexao** e **capacidade** fazem parte da LRC (Linguagem de Representação do Conhecimento) como será visto no quinto capítulo, portanto não necessitam ser definidos. Quando são utilizados procedimentos não definidos na LRC (Linguagem de Representação do Conhecimento), a definição dos mesmos é solicitada pelo EC.

d) Seleção de estados.

Este tipo de informação também serve para tornar o processo de verificação mais rápido, porém é incompreensível neste ponto da dissertação porque é necessário saber-se como é feita a verificação da alcançabilidade. A seguir é mostrada a sintaxe e alguns exemplos, e no item 5.3.3.2.2 Entrada de Conhecimento Antes da Simulação, será explicada esta alternativa.

Sintaxe:

constante_heuristica(K).

onde, K é um valor real que representa o peso da função g(n).

Exemplos de sentenças válidas:

```
constante_heuristica(1).
```

constante_heuristica(0.5).

constante_heuristica(0).

Quando nada for informado, o valor de k serà 1 (um), valor "default".

e) Definição da influência do usuàrio no processo de simulação.

Sintaxe:

simulacao(interativa).

Quando esta opção è escolhida, o usuàrio deve informar qual alteração deve ser disparada em cada passo do processo de simulação. Isto permite a observação do comportamento de uma rede de Petri, quando è feita uma sequência de disparos.

Para facilitar, ao usuàrio, a decisão de qual alteração efetuar, durante o processo de simulação, seria interessante que o mesmo pudesse visualizar o comportamento da rede de Petri e suas marcas. Devido à grande quantidade de problemas que surgem em um processo de visualização de uma rede de Petri, optou-se por não fazer isto. Assim, a cada passo da simulação são mostrados todas as alterações vàlidas para o estado corrente da rede de Petri; e após cada alteração são mostrados apenas os lugares, e respectivas marcas, que tiveram a sua marcação alterada.

Atravès desta opção de execução è possivel identificar-se outras propriedades que não aquelas vistas anteriormente. Um exemplo disto è a verificação das regiões criticas, que não è oferecida para ser feita de maneira automàtica.

Duas regiões são criticas, uma em relação à outra, se as alterações definidas pelas conexões de ambas regiões nunca devem estar habilitadas paralelamente, ou seja, não

devem estar no mesmo conjunto de passos.

Para efetuar a verificação desta propriedade, è necessàrio que o usuàrio execute a simulação com esta opção, simulação interativa. Depois informe as alterações, que devem ser executadas para fazer a verificação, passo a passo, atè entrar em uma região crítica. Neste processo deve ser observado se alterações definidas por conexões de outras regiões criticas estão habilitadas dentro de uma certa região crítica.

f) Definição da direção da pesquisa de alcançabilidade.

Em redes de Petri a pesquisa de alcançabilidade è sempre feita para frente, mas o raciocinio para tràs è interessante quando è feita alguma pesquisa no espaço de estados em que o estado inicial è mais flexivel que o estado final. Ou seja, a pesquisa tem como estado inicial um nùmero maior de possiveis marcações. Por exemplo, sò interessa a marcação de certos lugares, não importando quais marcas os demais lugares têm.

Se nada for informado, a pesquisa de alcançabilidade serà feita para frente, opção "default". Para que seja feita a pesquisa de alcançabilidade para tràs, esta informação deve estar explicita.

Sintaxe:
raciocinio(para_tras).

4.2.1.2.2 Descrição do Processo de Solicitação de Consultas

Quando se deseja fazer uma consulta sobre um modelo, uma sequência de passos deve ser seguida. E este modelo jà deve estar criado atravès do editor de redes de Petri.

No primeiro passo è informado o nome do modelo a ser consultado. Isto pode ser feito informando o nome diretamente ou por apontamento. E mostrada a relação de modelos existentes.

No segundo passo è informado o tipo de consulta que se deseja fazer. Para isto, è apresentado um menu com todas as propriedades que podem ser verificadas.

No terceiro passo são informados os parâmetros da consulta, que dependem do tipo de propriedade que se deseja verificar. Isto è feito apontando-se os objetos no modelo e digitando-se dados quando necessàrio.

Se a consulta solicitada for para verificar alguma propriedade dinâmica (ver seção 5.1 Verificação de Propriedades), vivacidade ou alcançabilidade, então um quarto passo pode ser feito opcionalmente. Neste passo são informadas as opções de execução, vistas no item 4.2.1.2.1.2 Informações de Execução. Isto è feito selecionando-se opções, apontando-se objetos no modelo e digitando-se dados quando necessàrio.

Por exemplo, para verificar se duas conexões tem definição de alterações conflitantes, deve-se seguir a seguinte sequência de passos:

1. informar o nome do modelo onde serà feita a verificação;

2. selecionar a consulta desejada: conflito;

3. selecionar as conexões para as quais serà feita a verificação, apontando-as, e fornecer a marcação informando as marcas para cada lugar da rede. Neste caso è suficiente informar as marcações dos lugares que estão ligados às conexões.

Como esta è uma propriedade estrutural, o quarto

passo è desnecessàrio .

4.2.2 Base de Redes

A BR (Base de Redes) tem por função armazenar, para cada modelo criado pelo usuàrio, todas as informações que são necessàrias para o processo de modelagem.

Um modelo deve ser armazenado, em formato textual, de tal modo que facilite a sua visualização gràfica e tambèm facilite a sua tradução na LRC (Linguagem de Representação de Conhecimento), utilizada pelo analisador e cuja sintaxe serà vista no quinto capítulo.

Cada modelo deve ter um arquivo com dois tipos de informações: informações gràficas necessàrias à visualização do modelo; e informações necessàrias ao analizador.

As informações gràficas necessàrias à visualização do modelo são as seguintes:

- posições de todos nodos;
- posições de todos arcos;
- tipo de curva de cada arco; e
- posições de todas anotações.

As informações necessàrias ao analisador são as seguintes:

- conexões (sem portas);
- lugares e respectivas capacidades e domínios;
- portas;
- anotações, em LARP;
- operações das anotações; e
- marcação inicial.

4.2.3 Analisador

O analisador è o mòdulo, da ferramenta descrita

até o momento, onde foi investido a maior parte do esforço gasto na elaboração deste trabalho. Isto foi feito porque considerou-se que o auxílio necessário à validação de um modelo é um dos mais importantes no processo de modelagem de sistemas.

Quando se deseja analisar um modelo, o mesmo deve ser carregado no analisador conforme a representação de conhecimento adotada no analisador. Portanto, torna-se necessária uma interface que traduza o modelo gerado pelo ER e que está armazenado na BR. Após este procedimento, a função do analisador é, a partir das consultas geradas no EC, fazer a análise do modelo, carregado previamente, para verificar o que foi solicitado.

A forma de realização da análise depende do tipo de consulta. Para as consultas que podem ser verificadas a partir da estrutura da rede de Petri não é necessário fazer a simulação da rede.

A liguagem escolhida para a implementação do protótipo foi o Arity Prolog, versão 5.1. De modo geral, a LC é constituída de questões, válidas na linguagem Prolog, para o analisador. Ou seja, cada consulta na LC é, sob uma visão procedural, uma chamada de procedimento em Prolog. Devido à grande flexibilidade de chamada de um procedimento da linguagem Prolog, uma consulta na LC pode ser feita de várias maneiras. Esta flexibilidade reflete-se no EC, como pode ser observado na definição da sintaxe da LC utilizada pelo EC.

O analisador é composto por dois submódulos: o controle ou motor de inferências, e a memória de trabalho.

O primeiro é constituído pelos comandos de controle, que correspondem ao programa em Prolog, e tem as

seguintes caracterìsticas principais:

- utiliza a àrvore de alcançabilidade como tècnica de anàlise;
- faz pesquisa em espaço de estados;
- utiliza a estratègia de pesquisa em espaço de estados chamada pesquisa em extensão informada ("Best-first"); e
- permite fazer manipulação simbòlica na alcançabilidade.

A memòria de trabalho serve para armazenar a Base de Dados (BD), as operações da rede de Petri na Linguagem de Representação do Conhecimento (LRC), e os Meta-Comandos de Controle (MCC). As informações armazenadas aqui são informações transitòrias, mantidas apenas durante o processo de simulação.

As informações que ficam na BD são inicialmente geradas por uma consulta que informa os estados inicial e final, e depois são geradas dinamicamente durante o processo de simulação. Estas informações são o estado objetivo (marcação final), o estado inicial (marcação inicial) e os estados intermediàrios. Os estados inicial e intermediàrios formam o espaço de estados que està sendo pesquisado. Este espaço de estados è, em outras palavras, a àrvore de alcançabilidade. A estrutura de dados utilizada são àrvores balanceadas.

O conhecimento correspondente a uma Rede Marcada, è gerado, conforme a sintaxe da LRC, atravès de uma interface entre a BR e o analisador.

Os meta-comandos de controle são as sentenças geradas, conforme a sintaxe da LC, durante uma parte do processo de solicitação de consulta. Correspondem às informações e opções de execução do usuàrio.

No próximo capítulo o analisador é descrito detalhadamente. São mostrados os elementos que compõem a sua arquitetura e as causas que motivaram a sua escolha.

5 ANALISADOR

Este é o principal capítulo deste trabalho. Aqui são estudados os métodos e técnicas que são utilizados para efetuar a análise em redes de Petri. Além disto, dentre os módulos que compõem a ferramenta objeto deste trabalho, o analisador é o mais importante e portanto onde foi dada a maior ênfase. É o analisador que faz a verificação automática de propriedades em redes de Petri.

5.1 Verificação de Propriedades

As propriedades a serem verificadas, vistas no item 4.2.1.2 Editor de Consultas, para efeito da forma de verificação, podem ser divididas basicamente em duas classes. A primeira contém as propriedades estruturais, verificáveis a partir da estrutura da rede e de sua marcação:

- conflito;
- concorrência (paralelismo); e
- ponto de bloqueio.

A segunda classe contém as propriedades dinâmicas, verificáveis através da simulação da rede:

- vivacidade ou pontos de bloqueio; e
- validade de estado futuro ou estado passado a partir de um estado inicial.

A seguir é descrito como é feita a verificação de cada uma destas propriedades.

5.1.1 Verificação de Propriedades Estruturais

a) Verificação da existência de conflito entre as alterações definidas por duas conexões para uma dada marcação.

Esta propriedade é verificada percorrendo-se todas alterações definidas pelas duas conexões, para a marcação em questão, e encontrando-se pares de alterações que, apesar de estarem habilitadas, não podem fazer parte de um mesmo passo. Esta última condição ocorre quando estas alterações estão habilitadas porèm têm marcas de entrada ou de saída comuns.

b) Verificação da existência de concorrência (paralelismo) entre as alterações definidas por duas conexões para uma dada marcação.

A verificação desta propriedade é feita percorrendo-se todas alterações definidas pelas duas conexões, para a marcação em questão, e encontrando-se pares de alterações que podem fazer parte de um único passo.

c) Verificação de bloqueio na rede de Petri com respeito a uma determinada marcação.

Uma marcação é um ponto de bloqueio quando inexistem conexões que tenham pelo menos uma alteração habilitada, isto significa que a rede fica bloqueada. A verificação desta propriedade é feita percorrendo-se todas as possíveis alterações e não encontrando alguma que esteja habilitada para esta marcação.

5.1.2 Verificação de Propriedades Dinâmicas

a) Verificação da vivacidade de marcações.

Conforme foi visto no item 4.2.1.2.1.1 Tipos de Consulta, uma marcação é viva se o seu conjunto de marcações alcançáveis não contem marcações que sejam pontos de bloqueio. Portanto, para fazer a verificação desta propriedade, é necessário percorrer todas marcações alcançáveis, e verificar, para cada marcação, se a mesma não

è um ponto de bloqueio. Este processo não è infinito graças às restrições impostas à linguagem de redes de Petri adotada (ver o terceiro capitulo).

b) Verificação da validade de estado futuro ou estado passado a partir de um estado inicial.

A verificação deste tipo de propriedade è, na verdade, a verificação da alcançabilidade, para frente ou para trás, de uma certa marcação a partir de uma marcação inicial. Para isto, no restante do capitulo è revista a questão da alcançabilidade e são apresentadas e avaliadas algumas maneiras de fazer a implementação de sua verificação.

5.2 Alcançabilidade

A maioria dos problemas e/ou questões a serem analisados em uma rede de Petri estão relacionados com marcações alcançáveis. Portanto a alcançabilidade è um instrumento poderoso de anàlise e serà fundamental no desenvolvimento do analisador.

A alcançabilidade è a propriedade de uma determinada marcação ser alcançável.

Uma marcação **m'** è alcançável a partir de uma marcação **m**, em uma rede de Petri, quando existe uma sequência válida, de alterações habilitadas, que transforma a marcação **m** da rede na marcação **m'**.

5.2.1 Arvore de Alcançabilidade

A tècnica de anàlise que tem sido mais utilizada com redes de Petri è a àrvore de alcançabilidade [PET 81], pois através da sua inspeção è possivel responder a um grande número de questões. A idèia bàsica da àrvore de

alcançabilidade é organizar todas as marcações alcançáveis em uma estrutura em árvore onde cada nodo está associado a uma marcação alcançável e cada arco está associado a uma conexão (se for uma rede de Petri de alto nível então cada arco está associado a uma alteração). Esta árvore representa o conjunto de todas marcações alcançáveis e todas as possíveis sequências de alterações de uma rede de Petri a partir de uma marcação inicial.

No caso de redes de Petri de alto nível uma conexão define um conjunto de alterações e cada arco da árvore de alcançabilidade está associado a uma alteração habilitada. Isto, normalmente, torna a árvore de alcançabilidade deste tipo de rede muito maior do que as de redes de Petri com menor poder de expressão.

A árvore de alcançabilidade é gerada disparando-se todas as alterações habilitadas para a marcação inicial. A cada alteração disparada é gerada uma nova marcação. Este procedimento é repetido para todas as novas marcações e assim sucessivamente. Cada nodo da árvore representa uma marcação e cada arco uma alteração. Veja o exemplo da árvore de alcançabilidade (parcial) na figura 5.2 para a rede de Petri da figura 5.1.

É fácil perceber que a árvore de alcaçabilidade muitas vezes é infinita, e quando isto ocorre, a questão da alcançabilidade é não decidível. A partir deste problema foram desenvolvidas outras representações da árvore de alcançabilidade visando torná-la finita para todos os casos. Porém, para conseguir isto, algumas coisas foram sacrificadas.

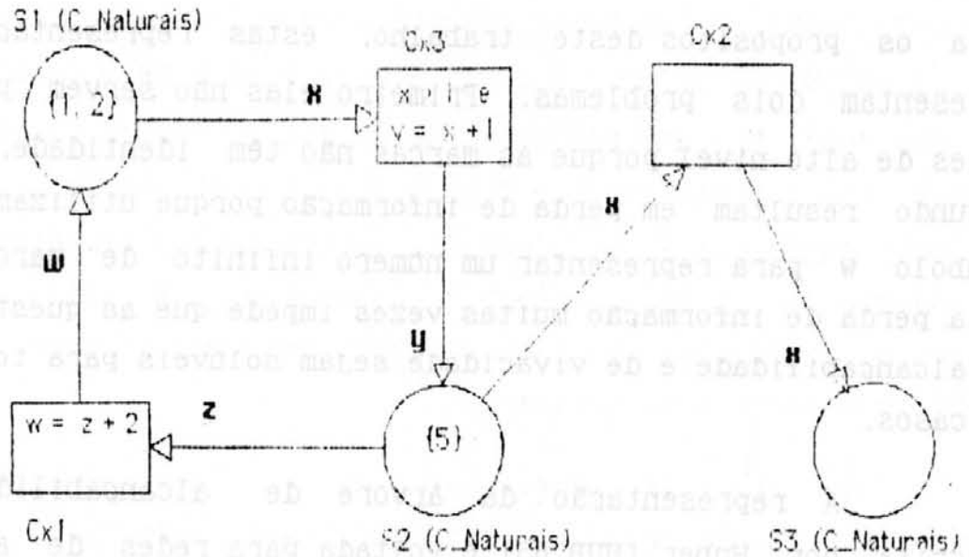


Figura 5.1 - Rede de Petri marcada

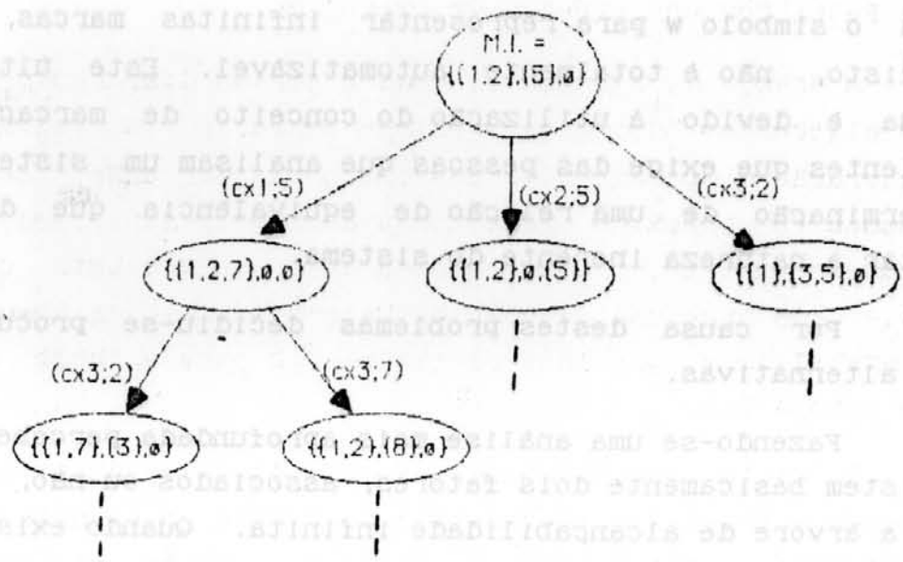


Figura 5.2 - Arvore de alcançabilidade (parcial) da rede de Petri da Figura 5.1.

As representações da árvore de alcançabilidade descrita por Peterson [PET 81] ou do grafo de cobertura descrito por Reisig [REI 86] são finitas porque não permitem que hajam marcações duplicadas e utilizam um simbolo (w) para representar uma quantidade infinita de marcas. Mas,

para os propósitos deste trabalho, estas representações apresentam dois problemas. Primeiro elas não servem para redes de alto nivel porque as marcas não têm identidade, e segundo resultam em perda de informação porque utilizam o simbolo w para representar um número infinito de marcas. Esta perda de informação muitas vezes impede que as questões de alcançabilidade e de vivacidade sejam solúveis para todos os casos.

A representação da árvore de alcançabilidade descrita por Huber [HUB 85] é voltada para redes de alto nivel e é bastante compacta porque generaliza a idéia de marcações duplicadas para marcações equivalentes. Mas aqui também existe o problema de perda de informação porque utiliza o simbolo w para representar infinitas marcas, e além disto, não é totalmente automatizável. Este último problema é devido à utilização do conceito de marcações equivalentes que exige das pessoas que analisam um sistema, a determinação de uma relação de equivalência que deve respeitar a natureza inerente do sistema.

Por causa destes problemas decidiu-se procurar outras alternativas.

Fazendo-se uma análise mais aprofundada percebe-se que existem basicamente dois fatores, associados ou não, que tornam a árvore de alcançabilidade infinita. Quando existem ciclos na rede e/ou quando um lugar tem ou pode ter um número infinito de marcas. Segue abaixo um esquema com soluções para cada um dos fatores.

a) Existência de ciclos na rede.

1. Possibilidade de gerar marcações duplicadas.

A existência de ciclos na rede permite que uma

mesma marcação possa repetir-se em vários nodos da árvore de alcançabilidade. Estas marcações repetidas têm exatamente as mesmas marcas e são denominadas marcações duplicadas.

Solução: não permitir marcações duplicadas.

2. Possibilidade de gerar um número infinito de marcas para lugares sem um limite de capacidade (para redes de Petri que permitem multiplicidade de marcas).

A existência de ciclos na rede permite que uma conexão seja disparada infinitas vezes, gerando uma árvore infinita, se não houver limite de capacidade nos lugares que estão ligados às portas de saída desta conexão.

Solução: limitar a capacidade dos lugares.

3. Existência de lugares cujo domínio tem cardinalidade infinita.

A existência de ciclos na rede associado com lugares cujo domínio tem cardinalidade infinita criam condições para a árvore ser infinita mesmo que não sejam permitidas marcações duplicadas. Isto ocorre porque podem ser geradas infinitas marcações diferentes, já que as marcas têm identidade.

Solução: limitar a cardinalidade do domínio.

b) Lugar com número infinito de marcas, na marcação inicial.

Por exemplo, informar o conjunto dos números

naturais como sendo a marcação inicial de um lugar.

Solução: limitar a capacidade dos lugares e verificar se a marcação inicial é válida, ou seja, respeita o limite de capacidade de todos os lugares.

Algumas destas soluções evidentemente limitam o poder de expressão das redes de Petri tratadas neste trabalho, mas isto justifica-se porque torna a questão da alcançabilidade decidível. Isto é, a árvore de alcançabilidade fica finita e não há perda de informação.

Apesar das limitações impostas em prol da decidibilidade, a representação da árvore de alcançabilidade proposta aqui tem um grave problema. A mesma não é compacta e pode levar muito tempo para efetuar-se a sua construção. Para minimizar este problema decidiu-se adotar técnicas de pesquisa em espaço de estados utilizadas em IA (Inteligência Artificial).

5.2.2 Simulação

O processo de geração da árvore de alcançabilidade de uma rede de Petri pode ser interpretado como sendo uma simulação sistemática desta rede.

Os estados da rede, neste processo, seriam as marcações, que estão associadas aos nodos da árvore. As operações da simulação seriam as alterações que estão associadas aos arcos da árvore. Uma alteração tem um conjunto de marcas de entrada, valorização dos termos das portas de entrada, contendo as marcas específicas que tomarão parte na transição. Completando a analogia, o espaço de estados da simulação seria a própria árvore de alcançabilidade.

Este espaço de estados pode ser interpretado como já existente, e, para verificar a alcançabilidade de um estado, é suficiente percorrê-lo, ou construí-lo, apenas parcialmente. Como pode ser notado, há um isomorfismo entre o processo de simulação da rede, visando verificar a alcançabilidade, e o processo de pesquisa em espaço de estados. Isto porque a questão da alcançabilidade é justamente encontrar um caminho (sequência de alterações) entre as marcações inicial e final. Devido a esta similaridade a representação do problema a ser analisado aqui, será feita utilizando o espaço de estados. Uma outra alternativa seria a representação pela redução de problemas, utilizando, por exemplo, grafos "e"/"ou". Esta última representação foi evitada porque não respeita as características do problema.

No estágio atual da tecnologia de simulação, segundo Gaines [GAI 86], o foco de atenção tem mudado para a utilização de conceitos de IA em pesquisa de simulação. Na geração atual de simuladores a idéia é utilizar conceitos de sistemas baseados em conhecimento. Na próxima geração o foco de atenção será a utilização de conceitos de sistemas de inferência indutiva. Neste trabalho são utilizadas idéias das áreas de solução de problemas e de sistemas baseados em conhecimentos visando tornar o processo de simulação de redes de Petri mais eficiente.

Na próxima seção serão avaliadas rapidamente algumas técnicas de pesquisa em espaço de estados utilizadas em IA, visando evitar a geração completa da árvore de alcançabilidade.

5.3 Pesquisa em Espaço de Estados

Sistemas de pesquisa em espaço de estados são usualmente descritos em termos de três principais

componentes [BAR/81]. O primeiro é a base de dados, que descreve, entre outras coisas, a situação corrente do processo de pesquisa como, por exemplo, os estados já gerados e o estado objetivo. O segundo componente é um conjunto de operadores que são usados para manipular a base de dados. O terceiro componente é a estratégia de controle cuja função é decidir o que fazer durante o processo de pesquisa.

No caso deste trabalho, a base de dados é a representação da árvore de alcançabilidade que está sendo gerada mais o estado a ser alcançado (marcação final). Os operadores são as alterações da rede de Petri, que são disparadas em cada marcação da árvore. A estratégia de controle decide que estado (marcação) será expandido e que operadores (alterações) serão utilizados.

A pesquisa em espaço de estados é feita gerando-se apenas os estados (marcações que estão associadas aos nodos da árvore de alcançabilidade) necessários para encontrar-se o estado objetivo. Neste processo de geração do espaço de estados, os nodos assumem a seguinte terminologia. Os nodos que já foram expandidos, ou seja, em que, foram aplicadas todas as alterações habilitadas para a marcação que o nodo representa, são chamados nodos internos. Os nodos que ainda não foram expandidos são denominados nodos fronteira. Os nodos cujas marcações são mortas, para as quais não existem alterações habilitadas, são denominados nodos terminais. Os nodos cujas marcações são iguais à marcação de algum outro nodo são denominados nodos duplicados.

A seguir são descritos e detalhados os componentes que fazem parte do sistema de pesquisa em espaço de estados deste trabalho.

5.3.1 Base de Dados

A base de dados armazena a representação da árvore de alcançabilidade que vai sendo construída a medida que o processo de pesquisa no espaço de estados se desenvolve. São adotadas representações para o estado objetivo, os estados inicial e intermediários, e o espaço de estados.

A escolha da estratégia de controle afeta o conteúdo e organização da base de dados. Portanto, as representações adotadas podem parecer obscuras se não forem analisadas à luz da estratégia de controle utilizada.

É utilizada a BNF, definida no terceiro capítulo, para a descrição da sintaxe das representações. São omitidos detalhes que complicam o entendimento das representações e não acrescentam vantagens relevantes para fins de uso ou análise da representação.

As convenções da BNF, utilizadas para a descrição das sentenças das representações, são repetidas abaixo:

::	é definido como;
|	alternativa;
.	fim da definição;
grifo	meta-símbolo;
negrito	símbolo terminal;
...	quantidade variável de símbolos (zero ou mais).

Como a marcação é o elemento mais complicado da sintaxe das sentenças das representações, a seguir será dada uma explicação do mesmo e apresentada a sua sintaxe.

A marcação de uma rede de Petri é o conjunto de marcas que estão presentes em todos os lugares da rede. Isto é representado através de uma lista de marcações de todos os lugares da rede. Cada lugar da rede é representado através

de uma lista contendo dois elementos: o nome do lugar e a marcação deste lugar. A marcação de um lugar é representado através de uma lista das marcas que estão presentes neste lugar.

Sintaxe:

marcação :: nome de variável em Prolog ¦
lista de marcações de lugares.

lista de marcações de lugares ::
[marcação de lugar, ... , marcação de lugar] ¦
[marcação de lugar¦Resto].

marcação de lugar :: [nome de lugar, lista marcas].

lista marcas :: nome de variável em Prolog ¦
[marca, ... , marca] ¦
[marca¦Resto].

nome de lugar :: nome de um dos lugares da rede de Petri.

marca :: elemento que faz parte da marcação de um dos lugares da rede de Petri; deve pertencer ao domínio deste lugar.

Observações:

a) **Resto** é nome de variável e, como está sendo utilizada, significa que o resto da lista em questão pode ser qualquer lista de marcas válidas, até mesmo lista vazia.

b) A utilização de nome de variável em Prolog, na construção de uma sentença, significa que esta variável pode ser qualquer coisa válida.

Exemplo:

[[s1, [1, 2, 3]], [s2, [5]], [s3, [5, 10]]]

onde,

[s1, [1, 2, 3]] e'a marcação do lugar **s1** que tem 1, 2 e 3 como marcas;
[s2, [5]] e'a marcação do lugar **s2** que tem 5 como marca; e
[s3, [5, 10]] e' a marcação do lugar **s3** que tem 5, 10 como marcas.

Outro exemplo:

[[s1, [1, 2, 3]]|Resto]

significa que o lugar **s1** tem as marcas 1, 2 e 3; e os demais lugares (resto da marcação) podem ter quaisquer marcações válidas.

5.3.1.1 Representação do Estado Objetivo

O estado objetivo é a marcação que deve ser alcançada e tem a seguinte representação:

marcacao_final(marcação).

Exemplos de sentenças válidas:

```
marcacao_final([[s1, [1]], [s2, [2, 3], [s3, []]]).
marcacao_final(MarcFinal).
marcacao_final([[s1, [1]]|Resto]).
marcacao_final([[s1, MarcLugar], [s2, [3|Resto]],
                                        [s3, []]]).
```

5.3.1.2 Representação dos Estados

Cada estado (nodo) do espaço de estados tem informações sobre o nodo, informações sobre a sequência de alterações e informações sobre a marcação corrente da rede de Petri. A seguinte representação é utilizada:

nodo([lista identificação], [tipo], valor heuristico,
num. alterações, alteração, marcação corrente).

onde,

lista identificação ::

[lista identificação nodo pai, número de ordem].

lista identificação nodo pai :: lista de identificação do nodo a partir do qual este nodo foi criado.

número de ordem :: número de ordem da expansão em que este nodo foi gerado.

tipo :: interno ¦
fronteira ¦
terminal ¦
duplicado, [lista identificação nodo duplicado].

valor heurístico :: valor, associado a cada nodo, resultado de uma função heurística;

num. alterações :: número de alterações ocorridas para chegar neste estado (marcação);

alteração :: [conexão, conj entrada].

conexão :: nome de conexão na rede de Petri.

conj entrada :: [porta entrada, ... , porta entrada].

porta entrada :: [nome de lugar, marca].

marcação corrente :: marcação.

Observações:

a) A lista de identificação do nodo raiz, o qual contem o estado inicial, é lista vazia.

b) conj entrada de uma conexão é o conjunto de portas de entrada desta conexão.

c) Em uma alteração as marcas das portas de entrada já devem estar valorizadas.

5.3.1.3 Representação do Espaço de Estados

Para a representação do espaço de estados foram estudadas algumas alternativas:

a) Utilizar uma lista para armazenar todos os caminhos que vão sendo pesquisados (construídos). Sendo que cada caminho é a lista de seus estados (nodos) intermediários. Esta representação é utilizada em [KVI 88].

Vantagem:

- Facilita a identificação de ciclos.

Desvantagens:

- Redundância: os estados anteriores, comuns a um conjunto de caminhos, são repetidos em todos estes caminhos.
- Gasta muita memória.

b) Armazenar apenas o estado atual mais a lista de operações aplicadas para chegar-se no mesmo. Esta representação é utilizada em [NIE 86].

Vantagem:

- Extrema economia de memória.

Desvantagens:

- Praticamente inviabiliza a detecção de nodos duplicados.
- Não permite que sejam considerados vários caminhos ao mesmo tempo.
- Pode entrar em "looping" mesmo que haja alguma solução.

c) Armazenar todos os estados gerados na base de dados, onde cada nodo aponta para o nodo pai.

Vantagens:

- Os estados não são repetidos, ou seja, não há redundância de dados.

- Facilita a detecção de nodos duplicados.

Desvantagem:

- O acesso aos nodos é muito lento.

d) Armazenar todos os estados gerados na base de dados e utilizar árvores balanceadas para acessar os nodos.

Vantagens:

- Os estados não são repetidos.

- Facilita a detecção de nodos duplicados.

- O acesso aos nodos é bem mais rápido que a representação anterior (c).

A representação adotada foi a última apresentada (d). Nesta representação, durante o processo de pesquisa no espaço de estados, os estados já gerados necessitam ser acessados de três diferentes maneiras.

Na primeira maneira, o nodo fronteira mais promissor deve ser acessado para efetuar a sua expansão. A chave de acesso é o valor heurístico dos nodos, e deve ser o menor valor de todos os nodos fronteira.

Na segunda, todos os nodos devem ser acessados para a verificação de existência de nodos duplicados. Como, no momento de geração de cada nodo, é feita uma análise de diferença entre a marcação do nodo e a marcação final (veja no item 5.3.3.1.1 Informação Heurística), então é suficiente verificar os nodos que tem o mesmo valor resultante desta análise de diferença. Assim a chave de acesso é o valor resultante da análise de diferença entre os nodos e o estado final. Fazendo-se isto evita-se a verificação de igualdade

de marcações para muitos nodos.

Na última maneira os nodos que fazem parte de um caminho, no espaço de estados, que chega no nodo final (estado objetivo) devem ser acessados para mostrar a solução. A chave de acesso é a lista de identificação dos nodos.

Para tornar mais eficientes as diferentes maneiras de acesso a estes estados são criadas três árvores balanceadas, cada uma com uma chave de acesso diferente. As folhas das árvores são os endereços dos nodos na base de dados. Isto é feito para evitar redundância de dados.

5.3.2 Operações

Cada operação no espaço de estados corresponde a uma alteração da rede de Petri com o conjunto das marcas de entrada já valorizadas. Como cada lugar pode ter várias marcas diferentes, podem haver diversas operações (alterações diferentes) para cada conexão.

As operações são aplicadas nos estados selecionados para serem expandidos, fazendo com que sejam gerados novos estados. Uma operação deve ser válida para poder ser aplicada em algum estado.

Para uma operação ser válida devem ser satisfeitas várias restrições:

a) a alteração deve estar habilitada;

b) as condições das anotações da conexão devem ser satisfeitas;

c) a marcação resultante da operação deve ser válida, ou seja, deve respeitar os limites de capacidade dos lugares;

d) a conexão ou alteração não pode ter sido invalidada pelo usuário.

Como pode ser observado, as operações são obtidas a partir de alguns dos elementos de um modelo (Rede Marcada) e para a descrição destes elementos é utilizada a LRC (Linguagem de Representação do Conhecimento). Estes elementos são: as conexões, as capacidades dos lugares e o UD.

Uma conexão é composta por três componentes, um conjunto de portas de entrada (conjunto de entrada), um conjunto de portas de saída (conjunto de saída) e anotações.

5.3.2.1 A Linguagem de Representação do Conhecimento

A LRC (Linguagem de Representação do Conhecimento) é a linguagem na qual deve ser traduzido todo conhecimento relativo a uma Rede Marcada. Este conhecimento é utilizado pelo Analisador para produzir as operações no espaço de estados.

Segue abaixo a sintaxe da LRC, representando os diversos elementos que compõem as redes de Petri. É utilizada a BNF definida anteriormente.

conexao(nome_conexão, [conj_ent_alt, conj_ent_res],
[conj_sai_alt, conj_sai_res]).

anotacoes(nome_conexão, [conj_ent_alt, conj_ent_res],
[conj_sai_alt, conj_sai_res]):-
condições,
ações.

onde,

nome_conexão :: nome de uma conexão na rede de Petri modelada.

conj ent alt :: conj entrada.

conj ent res :: conj entrada.

conj entrada :: [porta entrada, ... ,porta entrada].

porta entrada :: [nome de lugar, termo porta].

termo porta :: nome de variável em Prolog.

conj sai alt :: conj saída.

conj sai res :: conj saída.

conj saída :: [porta saída, ... ,porta saída].

porta saída :: [nome de lugar, termo porta].

condições :: expressões lógicas que devem ser verdadeiras para que a operação seja aplicada. É utilizada a sintaxe da LRC para a LARP, como será visto na próxima seção.

ações :: expressões ou procedimentos que definem os valores dos termos das portas de saída. É utilizada a sintaxe da LRC para a LARP, como será visto na próxima seção.

Observação: as condições e ações são obtidas a partir das anotações das conexões que estão na sintaxe da LARP como visto no terceiro capítulo. Uma conversão para a sintaxe da LRC deve ser feita.

capacidade(nome de lugar, limite de capacidade).

onde,

nome de lugar :: nome do lugar cuja capacidade deve ser limitada.

limite de capacidade :: valor numérico que indica qual o limite máximo de marcas que o lugar pode ter.

É obrigatório constar anotações para todas as conexões e capacidade para todos os lugares da rede. Isto pode ser feito automaticamente pela interface que transforma uma rede da BR na LRC, criando anotações, para as conexões sem anotações, da seguinte forma:

anotacoes(nome conexão).

e capacidade, para os lugares sem capacidade, da seguinte forma:

capacidade(nome de lugar,N).

5.3.2.1.1 Mapeamento da LARP na LRC

Tabela 5.1 - Mapeamento da sintaxe das fórmulas.

L A R P	L R C
verdadeiro	true
falso	false
relação(termo, ...)	relação(termo, ...)
(termo relação binária termo)	(termo relação binária termo)
(fórmula e fórmula)	(fórmula , fórmula)
(fórmula ou fórmula)	(fórmula ; fórmula)
não fórmula	not fórmula Obs. (a)
(fórmula impl fórmula)	(fórmula impl fórmula) Obs. (b)
paratodo var (fórmula)	não é colocado. Obs. (c)
existe var (fórmula)	não é definido.
(fórmula)	(fórmula)

Observações:

(a) é importante ressaltar que a negação na LRC não tem o poder de negação completo da lógica de primeira ordem, isto ocorre porque a LRC é um subconjunto da lógica clausal, na qual a negação é interpretada como falha [KOW 79];

(b) o impl é definido através da negação:

f1 impl f2 = not f1 ; f2

portanto, a observação (a) é válida para este operador lógico;

(c) as fórmulas da LRC são, na verdade, sentenças, ou seja, todas as variáveis são quantificadas com o quantificador em questão, e por simplicidade são omitidos.

Tabela 5.2 - Mapeamento da sintaxe das relações binárias.

L A R P	L R C
Elem	**elem**
Sub	**sub**
=	=
outros definidos de acordo com o modelo.	têm a mesma sintaxe porèm devem iniciar com letras minúsculas.

Os símbolos relacionais, binários ou não, definem relações entre entidades do UD. As definições destas relações devem ser feita por enumeração, quando da definição do UD de um modelo. A LRC tem prè-definido, ainda, os símbolos relacionais de comparação : >, <, >=, <=.

Tabela 5.3 - Mapeamento da sintaxe dos termos.

L A R P	L R C
constante	a mesma. Obs. (a)
var	a primeira letra deve ser maiúscula.
função(termo)	não é definido.
(termo função binária termo)	não é definido.
{termo, ... }	[termo, ...] Obs. (b)
{var | fórmula}	não é definido.
‹termo, ... ›	[termo, ...] Obs. (c)
(termo x termo)	não é definido.
(termo)	(termo)

Observações:

(a) não é obrigatória a delimitação por apóstrofes, porém neste caso deve iniciar com letra minúscula;

(b) é permitida a existência de duplicidade de elementos dentro do conjunto;

(c) a sintaxe é a mesma daquela de conjuntos.

Como a LRC não tem funções, para avaliar uma expressão aritmética é utilizado o operador especial **is**. Por exemplo,

```
X is 1 + 2.
```

deve atribuir 3 para X.

Dos elementos que compõem uma Rede Marcada, a LARP é o único que sofreu restrições neste trabalho. Isto porque a LRC definida não tem a mesma capacidade de expressão da LARP.

As tabelas 5.1, 5.2 e 5.3 têm como função mostrar o mapeamento entre as sintaxes das duas linguagens e mostrar as restrições. A sintaxe da LRC é muito semelhante com a da LARP, então para não ser redundante a sintaxe da LRC é mostrada nas próprias tabelas que são apresentadas e qualquer dúvida basta consultar a definição da sintaxe da LARP (terceiro capítulo).

5.3.3 Estratégia de Controle

A estratégia de controle decide que estado (marcação) será expandido durante o processo de pesquisa no espaço de estados, que operadores (alterações) serão utilizados para efetuar a expansão, e como estes operadores devem ser aplicados.

Para uma estratégia de controle ser eficiente, a sua estrutura deve respeitar as características do problema. E na sua elaboração devem ser definidos o método de pesquisa no espaço de estados, a direção do raciocínio, os níveis dos comandos de controle, entre outras coisas.

5.3.3.1 Método de Pesquisa no Espaço de Estados

Na área de IA existem vários métodos de pesquisa em espaço de estados. Apesar da literatura na área de IA não dizer explicitamente que os métodos de pesquisa em espaço de estados originaram-se da teoria dos grafos, o vocabulário básico da teoria dos grafos (arcos, nodos, caminhos, etc.) é utilizado na pesquisa em espaço de estados. Na teoria dos grafos, um grafo é um objeto abstrato. Por outro lado, um

espaço de estados é um grafo interpretado, isto é, os elementos do grafo têm uma intepretação: os arcos estão associados a operações, os nodos estão associados a estados, os caminhos estão associados a possíveis soluções, etc. Assim os algoritmos de pesquisa nas duas áreas são semelhantes, porém não são iguais devido à interpretação. Além disto alguns métodos na pesquisa em espaço de estados utilizam conhecimento, tornando os seus algoritmos bastante diferentes.

Veja em [GOL 80] e [FUR 73] a terminologia de teoria dos grafos e os algoritmos de caminhamento em grafos.

Aqui serão avaliados os quatro principais métodos de pesquisa em espaço de estados. Para uma descrição detalhada de como funcionam os métodos veja [NIL 71], [BAR 81] ou [KVI 88].

a) Pesquisa em profundidade ("depth-first").

É utilizado em [NIE 86].

Vantagens:

- Permite uma grande economia de memória, principalmente quando não é necessário guardar os nodos anteriores. Isto é possível porque é pesquisado apenas um caminho de cada vez e, se o mesmo não chegar a uma solução, é feito retrocesso ("backtracking").

- Permite identificar ciclos, porém apenas quando os nodos do caminho pesquisado são guardados.

Desvantagens:

- Quando há mais de uma solução, nem sempre a melhor solução e' encontrada.

- Entra em "looping" para caminhos (ramos da árvore) infinitos. Para evitar isto, deve haver uma

restrição como, por exemplo, um limite de profundidade de pesquisa; porèm isto pode impedir que se chegue a uma solução.

- Não utiliza heurística durante a pesquisa.

b) Pesquisa em extensão ("breadth-first").

Vantagens:

- Permite identificar ciclos, porèm, apenas quando os nodos internos são armazenados.
- Evita o "looping" se houver pelo menos uma solução.
- Quase sempre encontra a melhor solução, quando houver mais de uma solução.

Desvantagens:

- Normalmente, tem um conjunto muito grande de nodos armazenados, ou seja, o espaço de estados pesquisado cresce muito.
- E lento quando hà muitas operações.
- Não utiliza heurística durante a pesquisa.

c) Pesquisa em profundidade informado ("ordered depth-first").

Vantagens:

- Permite identificar ciclos, porèm apenas quando os nodos do caminho pesquisado são guardados.
- Utiliza heurística durante a pesquisa.

Desvantagens:

- A utilização da heurística è local, isto è, seleciona o nodo mais promissor, pela função heurística, dentre os nodos gerados a partir do último nodo expandido.
- Pode entrar em "looping" para caminhos infinitos.

d) Pesquisa em extensão informado ("best-first").

Vantagens:

- Permite identificar ciclos, porèm apenas quando os nodos internos são armazenados.

- Dependendo da função heurística adotada, evita o "looping".

- Utiliza heurística durante a pesquisa e a mesma è global, ou seja, seleciona o nodo mais promissor dentre os nodos fronteiras de todo o espaço de estados. Por causa disto, converge para um resultado mais rapidamente.

Desvantagens:

- Pode ter um conjunto muito grande de nodos armazenados, ou seja, o espaço de estados pesquisado pode crescer muito.

- O resultado encontrado não è necessariamente o melhor.

O primeiro e o terceiro mètodo ("a" e "c") tèm a grande vantagem de serem econômicos em relação à memòria, mas tèm o problema da possibilidade de entrarem em "looping" quando hà caminhos infinitos no espaço de estados. Este foi o motivo de terem sido descartados.

Os dois primeiros mètodos ("a" e "b") fazem uma pesquisa sistemàtica no espaço de estados, mas è uma pesquisa completamente causuística, ou seja, não são feitas tentativas para restringir o espaço de estados a ser pesquisado. Alèm disto, no caso do segundo mètodo, todo o espaço de estados pesquisado è armazenado, e o mesmo normalmente è muito grande para o conjunto de problemas em questão. A associação destes fatos levaram à desistência do

segundo método.

O quarto método ("d") evita os problemas mencionados nos parágrafos anteriores, se não totalmente, pelo menos em grande parte, e por isto foi o método selecionado para fazer parte da estrutura da estratégia de controle. Uma das características deste método é a utilização de heurística no processo de pesquisa no espaço de estados, o que será visto no próximo item.

5.3.3.1.1 Informação Heurística

A heurística, na pesquisa em espaço de estados, é uma informação sobre alguma propriedade do domínio específico do problema, utilizada com o intuito de guiar a pesquisa procurando torná-la mais eficiente. Como a heurística será definida agora e não pelos usuários da ferramenta, então o domínio específico do problema é o conhecimento sobre Redes de Petri de maneira geral. Uma propriedade óbvia sobre Redes de Petri, e que será utilizada aqui, é a seguinte: "uma marcação final é alcançável, quando existe uma sequência de alterações (caminho) que transforma a marcação inicial em uma marcação em que não hajam marcas diferentes entre esta e a marcação final". Assim, foi assumido que, quanto menos marcas diferentes houver entre duas marcações, mais próximas elas estão, ou seja, menor é o caminho entre elas. Aparentemente não existe outra informação heurística que seja geral, e não se tem neste ponto informações sobre os sistemas que serão modelados, ou seja, conhecimento específico da área de cada modelo.

Há basicamente três maneiras de utilizar-se a informação heurística: para decidir que nodo expandir, para decidir que sucessores serão gerados ao invés de gerar todos, e para decidir que nodos devem ser descartados, eliminados da árvore (podados) [BAR 81]. A segunda e a

terceira maneira não são interessantes porque neste tipo de problema não se sabe com certeza se os nodos, não gerados ou descartados, levam ou não a alguma solução. Assim, neste trabalho, a informação heurística será utilizada para decidir que nodo expandir em cada passo da pesquisa. Isto será feito aplicando-se uma função heurística nos estados que ainda não foram expandidos (nodos fronteira), para avaliar qual é o nodo mais promissor para chegar a uma solução.

A função heurística adotada tem a mesma forma e estrutura daquela apresentada em [BAR 81] para soluções ótimas. Além da mesma atingir o seu objetivo que é guiar a pesquisa no espaço de estados, permite também que o usuário tenha uma certa influência na sua avaliação (veja seção 5.3.3.2.2 Entrada de Conhecimento Antes da Simulação).

A função heurística utilizada, **f**, tem duas funções componentes:

$$f(n) = k.g(n) + h(n)$$

onde,

k é o peso dado pelo usuário para a função **g(n)**;

n é o nodo em que está sendo aplicada a função;

g(n) é a função componente que avalia o custo para chegar-se ao nodo atual (**n**) a partir do nodo inicial, isto é, a distância do estado inicial;

h(n) é a função componente que avalia o custo para chegar-se ao nodo final (objetivo) a partir do nodo atual (n), isto é, a distância do estado final.

O valor resultante da função **g(n)** é a quantidade de transições disparadas a partir da marcação inicial para chegar-se à marcação do nodo atual (n).

O valor da função **h(n)** é o resultado de uma análise de diferença entre a marcação do estado atual (n) e a marcação do estado final (objetivo). Isto é feito contando-se as marcas de cada lugar da marcação atual que são diferentes das marcas dos lugares correspondentes da marcação final.

A idéia da utilização de uma função componente **g(n)** surgiu a partir da necessidade de evitar-se a ocorrência de "loopings" (laços) durante o processo de pesquisa. Um processo de pesquisa entra em "looping" quando a árvore de alcançabilidade é infinita (veja 5.2.1. Arvore de Alcançabilidade). Assim, mesmo que pela análise de diferença um nodo seja mais promissor, se o caminho deste é muito longo (muitas transições foram disparadas), então será tentado outro nodo. Pois pode ocorrer situações em que, para chegar à solução, é necessário afastar-se do objetivo.

A utilização da função componente **g(n)** tem, ainda, um efeito colateral, que é o de procurar uma solução que tenha um caminho pequeno, ou seja, hajam poucas transições disparadas.

5.3.3.2 Utilização de Conhecimento

Apesar do método adotado, pesquisa em extensão informado, utilizar heurística, o mesmo é considerado um método fraco porque não impede a explosão combinatorial do espaço de estados pesquisado [KVI 88]. Para evitar este problema, a solução chave é a utilização de conhecimento específico do problema em grande quantidade e qualidade [KVI 88] [HAY 83] [CAR 88].

Uma das características do problema abordado neste trabalho é que há dois tipos de conhecimento: conhecimento sobre redes de Petri em geral e conhecimento específico da

área do sistema que está sendo modelado.

O conhecimento sobre redes de Petri está embutido na própria estrutura da estratégia de controle. Por exemplo, o conjunto de ações que deve ser executado quando é disparada uma transição, é um conhecimento a respeito de redes de Petri. Estas ações fazem parte dos comandos de controle. Outro exemplo é o conhecimento que foi colocado na elaboração da função heurística.

O conhecimento específico da área do sistema que está sendo modelado pode ser classificado em dois níveis, conforme a sua influência na estratégia de controle. O conhecimento oriundo da modelagem do sistema específico, como as transições e respectivas inscrições e a capacidade dos lugares, não influencia diretamente na estratégia de controle e dá origem às operações que podem ser aplicadas aos estados. Este conhecimento é classificado como estando no nível de modelagem. Este conhecimento já foi definido no item 5.3.2 Operações.

O conhecimento que influencia diretamente a estratégia de controle, como uma função heurística específica para um modelo ou uma definição específica da validade de conexões e/ou operações para uma verificação de alcançabilidade em um modelo, é classificado como estando no nível de controle.

O conhecimento no nível de controle são informações do usuário visando tornar o processo de pesquisa no espaço de estados mais eficiente, procurando evitar a explosão combinatorial, restringindo e orientando a expansão do espaço de estados. Quanto maior a qualidade deste tipo de conhecimento, maior a eficiência do processo de pesquisa no espaço de estado na busca de uma solução. Como o mesmo é totalmente definido pelo usuário, pode ocorrer que algumas

vezes este conhecimento, em vez de ajudar na procura de uma solução, atrapalhe e até mesmo impeça que chegue-se a alguma solução. Portanto o usuário deve saber o que está fazendo quando entra com conhecimento deste tipo.

Há dois modos do usuário entrar com esta última classe de conhecimento: durante e/ou antes da simulação.

5.3.3.2.1 Entrada de Conhecimento Durante a Simulação

Neste modo, a cada passo do processo de simulação, o usuário interfere diretamente informando qual operação deve ser executada ou se a estratégia de controle que decidirá, ou, ainda, se deve ser feito retrocesso. Este modo pode ser visto como sendo uma simulação interativa e neste caso a estratégia de controle utiliza o método de pesquisa em profundidade informado (visto no item 5.3.3.1 Método de Pesquisa no Espaço de Estados), com o retrocesso sendo controlado pelo usuário.

Caso o usuário decida deixar o sistema escolher qual será a próxima operação, então será escolhida a operação que gerar o estado mais próximo do estado objetivo. Isto é feito através de uma análise de diferenças dos próximos estados com o estado final, utilizando-se o resultado da função componente **h(n)**.

Esta maneira de efetuar-se a simulação é bastante útil quando se deseja saber como uma rede de Petri comporta-se abaixo de uma sequência específica de disparos de transições.

Para que seja possível fazer a simulação desta maneira, deve-se informar esta intenção da seguinte forma:

simulacao(interativa).

como já foi visto no quarto capítulo.

Se não for informado nada, a simulação é feita automaticamente, sem interação com o usuário.

5.3.3.2.2 Entrada de Conhecimento Antes da Simulação

Neste segundo modo, o usuário entra com o conhecimento antes da simulação, e portanto não interage com a simulação. As várias alternativas e as respectivas sintaxes para fazer isto já foram vistas no quarto capítulo. Neste item será visto o modo como este conhecimento influência na estratégia de controle.

a) Validação de alterações.

Este tipo de informação é fornecido pelo usuário antes de ser iniciada a verificação de propriedades. Abaixo são mostrados alguns exemplos de sentenças válidas que poderiam ser informadas.

```
alteracao_valida(t1,[[s1,3]]).
alteracao_valida(t1,[[s2,6]]).
alteracao_valida(t2,[[s2,4]]).
alteracao_valida(t2,[[s2,7]]).
```

Quando, por exemplo, é iniciada a verificação de alcançabilidade de alguma marcação, todas as alterações habilitadas em um certo estado são caminhos alternativos que podem ser tentados visando chegar ao estado final. Se o usuário entrar com este tipo de informação, ele está, na verdade, restringindo quais caminhos serão tentados. Assim apenas estas alterações "válidas" serão consideradas para efetuar a verificação. Isto evidentemente torna mais rápida a verificação, porém deixa a possibilidade de não terem sido tentados caminhos que levariam a solução. O "default" é todas alterações serem válidas.

b) Validação de conexões.

Este tipo de conhecimento, tem a mesma função do anterior, isto é, restringir os caminhos que serão tentados durante uma verificação de propriedade. Porèm o mesmo equivale a uma classe de alterações válidas porque, neste caso, são consideradas todas as alterações definidas para cada conexão "validada". O "default" è o caso em que todas as conexões são válidas.

Exemplo de sentença válida:

```
conexao_valida(t1).
```

c) Declaração de procedimento para validar conexões.

Conhecimento deste tipo è equivalente ao anterior, validação de conexões, no sentido que restringe o conjunto de conexões que serão tentadas para a verificação de propriedades. Porèm esta alternativa valida classes de conexões porque è considerada válida, para efetuar a verificação, toda conexão que satisfizer as condições da regra definida pelo usuário. Segue abaixo um exemplo de sentença válida para validar conexões.

```
conexao_valida(Conexao) :-
        conexao(Conexao, ConjEntr, [[Lugar, Termo]|R]),
        capacidade(Lugar, LimCap),
        LimCap @< 3.
```

d) Seleção de estados.

Utilizando este tipo de informação, o usuário tem apenas um certo grau de influência de como os estados são selecionados. Isto è feito permitindo ao usuário informar o valor de **k**, o qual faz parte da fórmula da função heurística **f(n)**. Esta função calcula o valor heurístico dos estados, como foi visto no item 5.3.3.1.1 Informação Heurística, e

tem a seguinte forma:

$$f(n) = k.g(n) + h(n).$$

A seguir são mostrados alguns exemplos de sentenças válidas para este tipo de conhecimento e explicada a sua influência na estratégia de controle.

Exemplos:

```
constante_heuristica(1).
constante_heuristica(0.5).
constante_heuristica(0).
```

Se o usuário informar **k** = 1, então as duas funções componentes terão o mesmo peso. Isto significa que o processo de escolha do estado (nodo) que deve ser expandido (ocorrência de todas alterações habilitadas e válidas) no espaço de estados (árvore de alcançabilidade) levará em conta igualmente a distância do estado inicial (marcação inicial), representada por **g(n)**, e a distância do estado final (marcação a ser alcançada), representada por **h(n)**. A figura 5.3 mostra uma árvore de alcançabilidade indicando a distância do estado inicial (**g(n)**) e a distância do estado final (**h(n)**).

Se o usuário informar **k** = 0.5, então a distância do estado inicial, representada por **g(n)**, terá metade do peso da distância do estado final, representada por **h(n)**. Isto significa que não interessa tanto o melhor resultado e sim encontrar um resultado.

Se o usuário informar **k** = 0 (zero), então a função heurística **f(n)** terá apenas uma função componente, a função que calcula a distância do estado final (**h(n)**). Isto significa que a função heurística sempre escolherá o estado mais próximo do objetivo para prosseguir na pesquisa do espaço de estados, mesmo que não seja o melhor caminho. Neste caso existe a possibilidade da simulação entrar em

"looping" porque muitas vezes é necessário afastar-se do objetivo para poder atingi-lo.

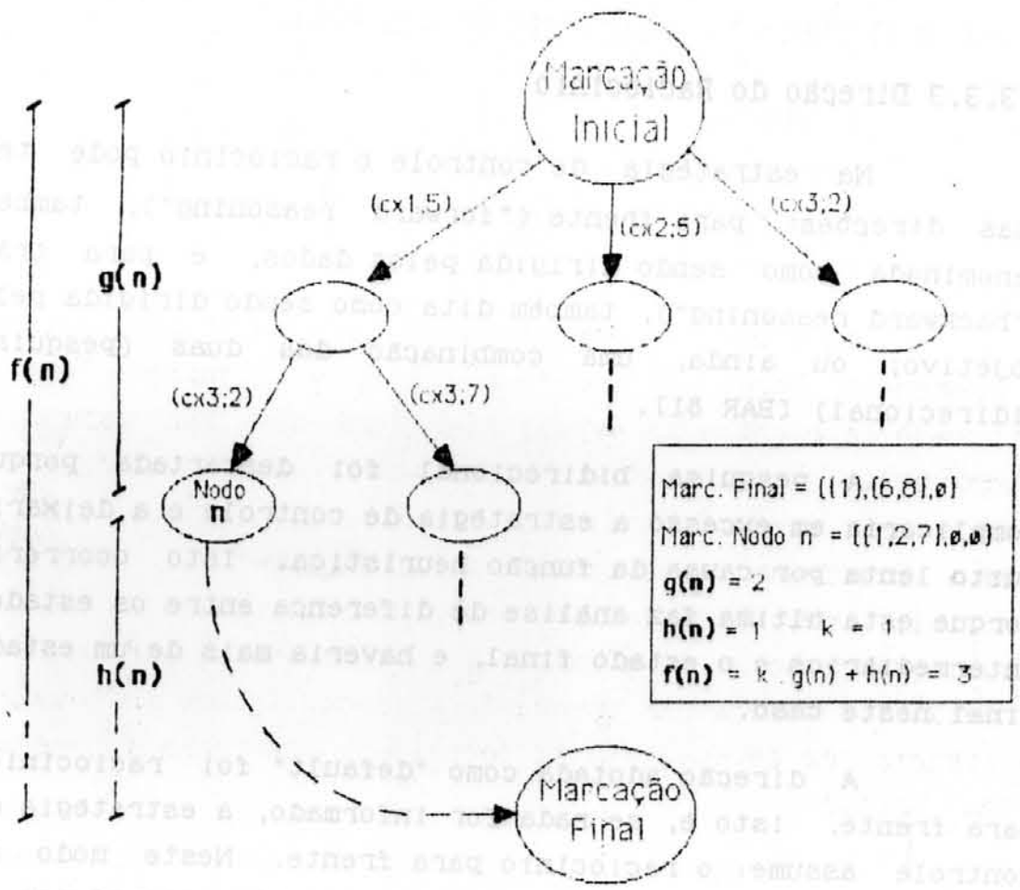


Figura 5.3 - Árvore de alcançabilidade (parcial) da Rede Marcada da figura 5.1 e os valores das funções de um dos nodos.

Por outro lado, quanto maior for o valor de **K**, mais a pesquisa, que será feita no espaço de estados, se aproxima de uma pesquisa em extensão. Ou seja, se a função componente **g(n)** tiver peso infinito, tendo influência total no resultado da função **f(n)**, então todos os estados de um mesmo nível serão expandidos antes de iniciar a expansão dos estados do próximo nível. Isto é uma pesquisa em espaço de

estados em extensão ("breadth-first"). Veja item 5.3.3.1 Método de Pesquisa no Espaço de Estados.

Quando nada for informado o valor de K será 1 (um), valor "default".

5.3.3.3 Direção do Raciocínio

Na estratégia de controle o raciocínio pode ter duas direções: para frente ("forward reasoning"), também denominada como sendo dirigida pelos dados, e para trás ("backward reasoning"), também dita como sendo dirigida pelo objetivo; ou ainda, uma combinação das duas (pesquisa bidirecional) [BAR 81].

A pesquisa bidirecional foi descartada porque complicaria em excesso a estratégia de controle e a deixaria muito lenta por causa da função heurística. Isto ocorreria porque esta última faz análise de diferença entre os estados intermediários e o estado final, e haveria mais de um estado final neste caso.

A direção adotada como "default" foi raciocínio para frente, isto é, se nada for informado, a estratégia de controle assume o raciocínio para frente. Neste modo de raciocínio, a estratégia de controle aplica operadores a partir do estado inicial até chegar no estado final. Esta é a direção de raciocínio mais natural para o problema a ser solucionado, pois a verificação de alcançabilidade é feita através da ocorrência de alterações para frente. Porém ainda é possível utilizar raciocínio para trás.

A direção de raciocínio para trás consiste da aplicação dos operadores a partir do estado final até chegar ao estado inicial. A aplicação dos operadores, neste caso, deve ser feita ao contrário. No caso de uma alteração em redes de Petri, deve-se retirar todas as marcas de saída

da marcação sucessora, e acrescentar todas as marcas de entrada à marcação precursora. O resultado disto deve ser uma marcação válida.

A mudança da direção do raciocínio, pelo usuário, é feita através da seguinte sentença:

raciocinio(para_tras).

como foi visto no quarto capítulo.

5.3.3.4 Níveis de Comandos de Controle

Existem dois níveis de comandos de controle. O primeiro contèm os comandos de nível mais baixo que, fazendo parte da estrutura da estratègia de controle, são os comandos de controle propriamente ditos.

O segundo contèm os comandos de nível mais alto, denominados meta-comandos de controle (MCC). E são aqueles que definem como será a estrutura da estratègia de controle.

Neste trabalho, os MCC são aqueles comandos que são gerados através do conhecimento que o usuário fornece para o sistema, e que alteram a estratègia de controle. Os mesmos já foram vistos nas seções anteriores e no quarto capítulo, e estão relacionados abaixo.

a) operacao_valida(nome_conexão, conj_entrada).
b) conexao_valida(nome_conexão).
c) conexao_valida(Conexao) :- condições.
d) constante_heuristica(K).
e) raciocinio(para_tras).
f) simulacao(interativa).

5.4 Eliminação de Nodos Duplicados

Dois nodos são duplicados quando os mesmos apresentam igualdade de marcações. Para evitar a ocorrência

de nodos duplicados, no momento em que um novo nodo è gerado, è feita uma anàlise de diferença entre o mesmo e todos os nodos jà gerados atè o momento. A anàlise de diferença è feita da mesma maneira como è feita para calcular a função componente **h(n)**, ou seja, comparando duas marcações para verificar a existência de marcas diferentes.

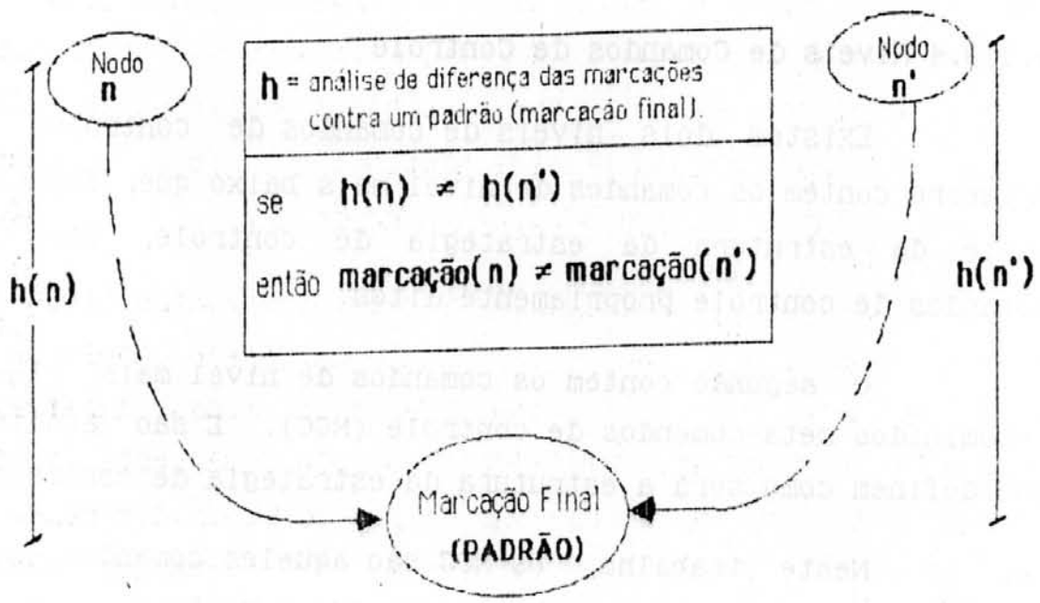


Figura 5.4 - Diferença de marcações.

E fàcil perceber que existe a possibilidade das marcações de dois nodos **n** e **n'** serem iguais, apenas quando estas marcações tiverem a mesma quantidade de marcas diferentes em relação a uma terceira marcação. Como jà è feita a anàlise de diferença entre as marcações de todos nodos e a marcação objetivo, então basta verificar a igualdade de marcações entre aquelas cujo valor, resultado da função componente **h(n)**, è igual ao valor da marcação do nodo que està sendo gerado. Isto è, um nodo **n'** pode ser duplicado em relação ao nodo gerado **n** se a seguinte condição for satisfeita:

h(n') = h(n).

Se **h(n')** é diferente de **h(n)** então há, no mínimo, uma marca de diferença na marcação destes nodos, como procura mostrar a figura 5.4.

Quando dois nodos são duplicados um deles fica com o tipo igual a "duplicado" e não é feita expansão para o mesmo. O critério adotado para decidir qual nodo deve ser o nodo com o tipo igual a "duplicado", é a distância do estado inicial, ou seja, o resultado da função componente **g(n)**. É feita a comparação entre **g(n)** e **g(n')** para decidir qual será o nodo duplicado e aquele cuja distância é maior fica como sendo o nodo com tipo igual a "duplicado". Isto é feito visando encontrar soluções com caminho menor.

Como visto anteriormente, existem quatro tipos de nodos: fronteira, interno, terminal e duplicado. Conforme o tipo de nodo que foi constatado como sendo duplicado (chamado nodo duplo) em relação ao nodo que está sendo gerado (chamado nodo novo), há um procedimento diferente a ser feito. A seguir vem um esquema de todos os casos possíveis.

1. Nodo duplo tem o tipo igual a "fronteira".
   1.1. g(nodo novo) < g(nodo duplo).
        Procedimento:
        - alterar o tipo do nodo duplo para "duplicado". Isto significa que o mesmo deixa de ser nodo fronteira.
   1.2. g(nodo novo) >= g(nodo duplo).
        Procedimento:
        - o tipo do nodo novo deve ser "duplicado", e apesar de estar sendo recém gerado não fará parte dos nodos fronteira.

2. Nodo duplo tem o tipo igual a "interno".
   2.1. g(nodo novo) < g(nodo duplo).

Neste caso, o fato do nodo duplo ter o tipo igual a "interno" significa que o mesmo tem descendentes. O procedimento completo seria, além de alterar o tipo do nodo duplo para "duplicado", passar todos os seus descendentes para o nodo novo. Isto não é feito porque teriam que ser alterados a lista de identificação e o número de transições, entre outras coisas, de todos os nodos descendentes do nodo duplo. O que poderia tornar a estratégia de controle muito lenta. Para evitar isto, é feito apenas o seguinte.

Procedimento:

- alterar o tipo do nodo duplo para "duplicado".

2.2. g(nodo novo) >= g(nodo duplo).

Procedimento:

- o tipo no nodo novo deve ser "duplicado".

3. Nodo duplo tem o tipo igual a "terminal".

3.1. g(nodo novo) < g(nodo duplo).

Procedimento:

- o tipo do nodo novo deve ser "terminal".

3.2. g(nodo novo) >= g(nodo duplo).

Procedimento:

- o tipo do nodo novo deve ser "terminal".

4. Nodo duplo tem o tipo igual a "duplicado".

Quando o tipo de um nodo é igual a "duplicado", ele sempre aponta para o nodo duplo, em relação a ele mesmo, cujo tipo é diferente de "duplicado". Este nodo é chamado nodo apontado.

4.1. g(nodo novo) < g(nodo duplo).

4.1.1. g(nodo novo) < g(nodo apontado).

Procedimento:

- alterar o tipo do nodo apontado para "duplicado".

4.1.2. g(nodo novo) >= g(nodo apontado).

Procedimento:

- o tipo no nodo novo deve ser "duplicado".

4.2. g(nodo novo) >= g(nodo duplo).

Quando isto ocorrer, por causa da propriedade da transitividade, também será verdadeira a relação g(nodo novo) >= g(nodo apontado).

Procedimento:

- o tipo no nodo novo deve ser "duplicado".

Para simplificar o procedimento de eliminação dos nodos duplicados, no quarto caso basta desconsiderar os nodos duplos que tem o tipo igual a "duplicado" quando é feita a verificação de existência de nodos duplicados. Isto é possível porque a comparação decisiva sempre pode ser feita entre o nodo novo e o nodo apontado, e o nodo apontado não tem tipo igual a "duplicado".

5.5 Alterações na Base de Dados em Decorrência da Expansão do Espaço de Estados

Como visto no item 5.3.1 Base de Dados, para tornar mais eficiente as diferentes maneiras de acesso aos nodos já gerados, são criadas três árvores balanceadas, cada uma com uma chave de acesso diferente. As folhas das árvores são os endereços dos nodos na base de dados.

A árvore cujo nome é **fronteira** armazena o endereço de todos nodos cujo tipo é fronteira. A chave de acesso aos seus elementos é o valor heurístico, calculado pela função heurística f(n). O nodo com menor valor heurístico deve ser acessado para que seja feita a sua expansão.

A segunda árvore tem o nome **todosnodos** e armazena o endereço de todos nodos gerados até o momento. A chave de acesso aos seus nodos é o valor calculado pela função

componente **h(n)**, que é o resultado da análise de diferença com o nodo final. Esta árvore é mantida para tornar mais rápida a identificação de nodos com marcação duplicada, pois basta procurar entre os nodos que tiverem o mesmo valor da função **h(n)**.

A última árvore, a qual tem como nome **arvorealc**, armazena o endereço de todos nodos gerados até o momento, como a árvore anterior. Porém a chave de acesso aos seus elementos, endereços dos nodos, é a lista de identificação do respectivo nodo. A mesma serve para permitir que sejam feitas pesquisas na árvore de alcançabilidade gerada até o momento de maneira ordenada, isto é, pela ordem de geração.

A figura 5.5 mostra um exemplo de duas árvores balanceadas, **fronteira** e **todosnodos**, cujas folhas apontam para os dados na base de dados.

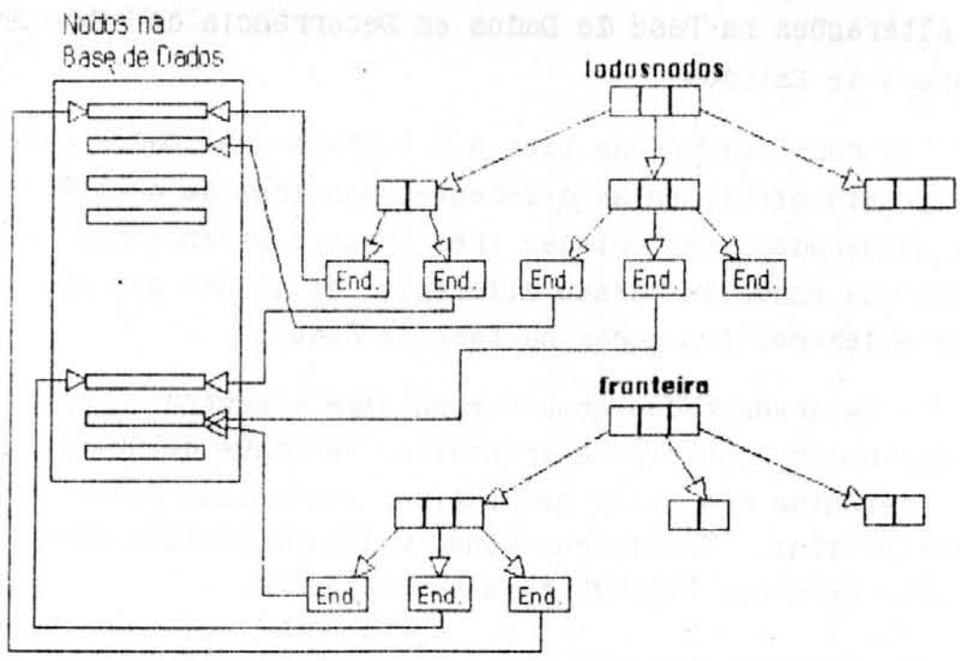


Figura 5.5 - Exemplo da estrutura de dados.

Abaixo segue um esquema de como são mantidas as árvores durante o processo de expansão do espaço de estados.

a) Quando é feita a criação inicial (geração do nodo raiz) faz:

1. Insere o nodo raiz na base de dados com o tipo igual a "fronteira";
2. Insere o endereço do nodo nas três árvores: **fronteira, todosnodos** e **arvorealc.**

b) Quando é feita a expansão do nodo selecionado (geração dos nodos intermediários) faz:

1. Remove o endereço, do nodo que está sendo expandido, da árvore **fronteira;**
2. Quando a expansão não gerar novos nodos faz:
   - Altera o tipo do nodo que está sendo expandido para "terminal";
3. Quando a expansão gerar pelo menos um nodo novo faz:
   - Altera o tipo do nodo que está sendo expandido para "interno".

c) Quando é gerado um nodo intermediário faz:

1. Quando o nodo gerado não tiver duplicado faz:
   - Insere o novo nodo na base de dados com tipo igual a "fronteira";
   - Insere o endereço do novo nodo nas três árvores: **fronteira, todosnodos** e **arvorealc;**
2. Quando o nodo gerado tiver duplicado e o nodo duplicado for tipo "terminal" faz:
   - Insere o novo nodo na base de dados com tipo igual a

"terminal";

- Insere o endereço do novo nodo nas árvores **todosnodos** e **arvorealc**.

3. Quando o nodo gerado tiver duplicado e o nodo duplicado não for tipo "terminal" faz:

3.1. Quando g(novo nodo) >= g(nodo duplicado) faz:

- Insere o novo nodo na base de dados com tipo igual a "duplicado";
- Insere o endereço do novo nodo nas árvores **todosnodos** e **arvorealc**;

3.2. Quando g(novo nodo) < g(nodo duplicado) faz:

- Insere o novo nodo na base de dados com tipo igual a "fronteira";
- Insere o endereço do novo nodo nas três árvores: **fronteira**, **todosnodos** e **arvorealc**;
- Altera o tipo do nodo duplicado para "duplicado";
- Quando o nodo duplicado tiver tipo igual a "fronteira", remove o endereço deste nodo da árvore **fronteira**.

Obs.: Os nodos com tipo igual a "duplicado" não devem ser considerados quando é feita a verificação da existência de nodos duplicados, de acordo com a seção 5.4 Eliminação de Nodos Duplicados.

A partir da descrição dos métodos, técnicas e estruturas de dados selecionados neste capítulo foi possível efetuar a implementação de um protótipo, o qual é descrito no próximo capítulo.

6 IMPLEMENTAÇAO DE UM PROTOTIPO EM PROLOG

6.1 Introdução

Como já foi dito anteriormente, o principal objetivo deste trabalho è fazer a verificação automàtica de propriedades de modelos, desenvolvidos na linguagem de modelagem redes de Petri. Assim, o protòtipo foi implementado visando atingir este objetivo. A parte da ferramenta relacionada com a interface com o usuàrio não foi implementada.

A linguagem que foi utilizada para o desenvolvimento do protòtipo foi o Prolog da Arity Corporation, versão 5.1. Foi adotado especificamente o Arity/Prolog porque, das implementações de Prolog disponiveis, esta è uma das mais flexiveis e poderosas. Alèm disto è possivel gerar còdigos objetos junto com rotinas na linguagem C [ARI 86]. Isto è interessante para a futura implementação da interface com o usuàrio, já que a linguagem Prolog è extremamente pobre neste sentido.

O equipamento utilizado foi um microcomputador compativel com o PC da IBM.

6.1.1 A Linguagem Prolog

Como o pròprio nome da linguagem diz (PROgramming in LOGic), o Prolog foi desenvolvido visando utilizar a lògica como linguagem de programação. Assim, do ponto de vista da lògica matemàtica, um programa em Prolog assemelha-se a uma teoria com o seu conjunto de axiomas, e a chamada de execução do programa seria um teorema a ser provado, nesta teoria. Para que a chamada de um programa em Prolog possa ser "provada", a pròpria linguagem tem embutida na sua estrutura um "provador automàtico de teoremas".

Estes fatos determinam todas as principais características da linguagem Prolog, como, por exemplo, apresentação clausal, orientação a processamento não-numérico, declaratividade, casamento de padrões, retrocesso automático, e até mesmo a pobreza de recursos de entrada e saída. Apesar desta última, estas e outras características a tornam uma linguagem extremamente poderosa e flexível.

A linguagem Prolog foi adotada neste trabalho porque as suas características, apontadas no parágrafo anterior, facilitariam muito a implementação da pesquisa em espaço de estados, e permitiriam que a linguagem de consulta ficasse bastante flexível. Além disto, pelo fato de ser uma linguagem declarativa e orientada ao encapsulamento, o Prolog permitiria que um usuário mais especializado facilmente incluísse novos trechos de programação para efetuar a verificação de novas propriedades, conforme as suas necessidades. Apesar de não ser um motivo determinante, poderou-se também que seria impossível implementar o protótipo em tempo hábil utilizando uma linguagem procedural, e haveria um desvio do objetivo principal. Isto porque todos os processos, que são transparentes no Prolog, como por exemplo, o gerenciamento de memória, teriam que ser implementados.

6.1.2 Estado Atual da Implementação

A implementação do protótipo está em um estágio bem avançado em relação àquilo que foi proposto.

Das consultas apresentadas na LC, foram implementadas quatro: conflito, concorrência, bloqueio e alcançabilidade. Até o presente momento está faltando apenas a consulta de vivacidade.

Das opções de execução do usuário foram implementadas quatro: validação de alterações, validação de conexões, declaração de procedimento para validar conexões, e seleção de estados. Ficaram faltando duas: simulação interativa e verificação de alcançabilidade para trás.

Todos os elementos para a representação de modelos, definidos na LRC, foram implementados, exceto domínio de lugares. Isto torna possível testar o protótipo em cima de qualquer modelo que possa ser desenvolvido utilizando a linguagem de redes de Petri definida no terceiro capítulo.

Para utilizar o protótipo no estado atual é necessário, estando no ambiente do Arity/Prolog, carregar os arquivos que contem o programa e o modelo, este último deve estar de acordo com a sintaxe da LRC. Os comandos para fazer isto são:

```
reconsult(analise).
reconsult(rede).
```

Após ter carregado o programa e a rede basta fazer consultas válidas conforme a LC. Desejando-se fornecer as informações de execução, é necessário criar um arquivo com nome **conhecim.ari** contendo estas informações conforme foi visto no item 4.2.1.2.1.2 Informações de Execução.

6.2 Descrição da Estrutura do Programa

Apesar de não ser utilizada a pesquisa em espaço de estados em profundidade, estratégia de controle da linguagem Prolog, na verificação de alcançabilidade, o programa utiliza as estruturas de controle da linguagem Prolog da maneira mais simples, dentro do possível. Isto é feito visando tornar a implementação das consultas declarativa. Entretanto em alguns pontos do programa são

utilizados comandos que alteram a estratègia de controle da linguagem, como por exemplo o operador de corte, para tornà-lo mais ràpido.

Para cada tipo de consulta da LC è definida um predicado em Prolog. Se o predicado tiver como clàusula uma regra, estão esta è definida referenciando outros predicados e assim sucessivamente, atè referenciar fatos ou regras que definem casos triviais. Isto pode ser encarado como sendo uma construção "top-down" de programa.

A seguir são descritos os predicados que são comuns a todas as consultas da LC e logo apòs os predicados que compõem cada consulta da LC.

6.2.1 Descrição dos Predicados Comuns às Consultas

Este item descreve os très predicados que são comuns a todas as consultas implementadas.

1) atualiza_conhecimento_usuario.

Esta predicado atualiza o conhecimento do usuàrio. Este conhecimento è constituido pelas informações de execução vistas nos itens 4.2.1.2.1.2 Informações de Execução, e 5.3.3.2.2 Entrada de Conhecimento Antes da Simulação. Todas estas informações são removidas atravès dos predicados que vão de "a" atè "e", depois são carregadas as novas informações que devem estar armazenadas no arquivo **conhecim.ari**, o que è feito no predicado "f". Por último è chamado o predicado validacoes_usuario, explicado mais abaixo.

```
atualiza_conhecimento_usuario:-
  a) abolish(alteracao_valida/2),
  b) abolish(conexao_valida/1),
  c) abolish(constante_heuristica/1),
  d) abolish(simulacao/1),
```

e) abolish(raciocinio/1),

f) reconsult(conhecim),

g) validacoes_usuario.

validacoes_usuario.

Este predicado encarrega-se de colocar as opções "default", caso as informações de execução não tenham sido fornecidas pelo usuário. Se nenhuma conexão foi validada, então todas conexões são consideradas válidas (predicado "a"). Se nenhuma alteração foi validada, então todas alterações são consideradas válidas (predicado "b"). Se não for informado o valor da constante heurística (K), então é assumido K=1 (predicado "c").

```
validacoes_usuario:-
  a) ifthen( not(conexao_valida(Conexao)),
             assert(conexao_valida(TodasConexoes)) ),
  b) ifthen( not(alteracao_valida(Conex, CjEnt)),
             assert(alteracao_valida(TodasConex, TodosCjEnt)) ),
  c) ifthen( not(constante_heuristica(K)),
             assert(constante_heuristica(1)) ).
```

2) marcacao_inicial_valida(marcação).

Este predicado verifica se a marcação fornecida pelo usuário (marcação inicial) é válida. Isto é feito através da verificação da capacidade (predicado "2.b" até "2.d") e da multiplicidade (predicado "2.e") de todos os lugares da marcação informada. O predicado é formado por duas cláusulas. A primeira é o caso trivial, em que a marcação (lista de lugares) é vazia. A segunda é recursiva (veja o predicado "2.f") e vai percorrendo a marcação até encontrar a marcação vazia.

O predicado "2.a" é o operador de corte do Prolog que impede que haja retrocesso [ARI 86] [BRA 86]. Aqui ele é utilizado para evitar retrocessos desnecessários, pois basta

a marcação de um lugar não ser válida para que toda a marcação seja inválida.

```
1. marcacao_inicial_valida([]).
2. marcacao_inicial_valida([[Lugar,MarcLugar]|RestoMarc]):-
   2.a) !,
   2.b) length(MarcLugar,NumMarcas),
   2.c) capacidade(Lugar,Capacidade),
   2.d) NumMarcas =< Capacidade,
   2.e) multiplicidade(MarcLugar),
   2.f) marcacao_inicial_valida(RestoMarc).
```

multiplicidade(lista de marcas).

Este predicado verifica a multiplicidade da marcação de um lugar. Não são permitidas cópias de marcas em um mesmo lugar. O predicado é definido através de duas cláusulas. A primeira é o caso trivial em que a marcação (lista de marcas) é vazia. A segunda é recursiva (predicado "2.c"); ela vai percorrendo a marcação e verificando se uma marca não é membro do resto da marcação (predicado "2.b").

```
1. multiplicidade([]).
2. multiplicidade([Prim|Resto]):-
   2.a) !,
   2.b) not(member(Prim,Resto)),
   2.c) multiplicidade(Resto).
```

member(elemento, lista).

Este predicado verifica se um elemento faz parte de uma lista.

```
1. member(Prim,[Prim|Resto]).
2. member(Elem,[Prim|Resto]):-
   a) member(Elem,Resto).
```

3) encontra_alter_habil(conexão,
conj portas alteradoras entrada,
conj portas restauradoras entrada,
conj portas alteradoras saída,

conj portas restauradoras saida,
Marcacao).

Dados uma conexão, os conjuntos de portas (alteradoras e restauradoras) de entrada e de saída desta conexão, e uma marcação; este predicado procura uma alteração habilitada. Para isto verifica se a conexão existe (predicado "a") e se è vàlida (predicado "b"). Depois faz uma valorização (predicado "c") do conjunto de portas (alteradoras e restauradoras) de entrada e verifica se esta alteração è vàlida (predicado "d"). Logo a seguir verifica se as anotações são verdadeiras para esta valorização (predicado "e") e se esta alteração està habilitada (predicado "f").

```
encontra_alter_habil(Conex, CjEntAlt, CjEntRes,
                                CjSaiAlt, CjSaiRes, Marcacao):-
  a) conexao(Conex, [CjEntAlt, CjEntRes],
                    [CjSaiAlt, CjSaiRes]),
  b) [!conexao_valida(Conex)!],
  c) valorizacao_cj_ent([CjEntAlt, CjEntRes], Marcacao),
  d) [!alteracao_valida(Conex, CjEntAlt)!],
  e) [!anotacoes(Conex, [CjEntAlt, CjEntRes],
                    [CjSaiAlt, CjSaiRes])!],
  f) [!alteracao_habilitada([CjSaiAlt, CjSaiRes], Marcacao)!].
```

valorizacao_cj_ent([conj portas alteradoras entrada,
conj portas restauradoras entrada],
Marcacao).

Este predicado efetua a valorização do conjunto de portas alteradoras de entrada (predicado "a") e do conjunto de portas restauradoras de entrada (predicado "b"). Isto è feito fazendo a unificação dos termos das portas com algumas das marcas da marcação.

```
valorizacao_cj_ent([CjEntAlt, CjEntRes], Marcacao):-
```

a) unifica_cj_marcacao(CjEntAlt,Marcacao),

b) unifica_cj_marcacao(CjEntRes,Marcacao).

unifica_cj_marcacao(conj portas entrada, marcação).

Este predicado percorre o conjunto de portas visando unificar os termos de cada porta com as marcas da marcação. Ela è composta por duas clàusulas. A primeira è o caso trivial em que o conjunto de portas està vazio (lista vazia). A segunda è o caso geral em que a clàusula vai pegando recursivamente uma a uma, as portas do conjunto de portas (predicado "2.c"). Para cada porta, encontra o respectivo lugar na marcação (predicado "2.a") e unifica os termos da porta com alguma marca da marcação (predicado "2.b").

```
1. unifica_cj_marcacao([],_).
2. unifica_cj_marcacao([Porta|RestoCjEnt],Marcacao):-
   2.a) encontra_lugar(Porta,Marcacao,Lugar),
   2.b) unifica_marca(Porta,Lugar),
   2.c) unifica_cj_marcacao(RestoCjEnt,Marcacao).
```

encontra_lugar(porta procurada, marcação da rede,
lugar encontrado na rede).

Uma porta è uma lista de dois elementos: o nome de um lugar e uma lista de termos de porta. Este predicado procura o respectivo lugar na marcação da rede. Isto è feito procurando recursivamente (predicado "2.a") atè encontrar o lugar, o que è identificado quando o nome dos dois lugares è o mesmo (clàusula "1").

```
1. encontra_lugar([NomeLugar,Termos],
                  [[NomeLugar,MarcLugar]|RestoMarc],
                                  [NomeLugar,MarcLugar]).
2. encontra_lugar(Porta,[_|RestoMarc],Lugar):-
   2.a) encontra_lugar(Porta,RestoMarc,Lugar).
```

unifica_marca(porta, lugar da marcação).

Este predicado unifica o termo da porta com alguma

marca da marcação. O predicado é recursivo para que todas as valorizações do termo sejam tentadas. É importante salientar que este predicado está funcionando apenas para portas com um só termo de porta.

```
1. unifica_marca([NomeLugar,[Marca|Resto]],
                                [NomeLugar,[Marca|RestoMarcLugar]]).
2. unifica_marca(Lugar,[NomeLugar,[Marca|RestoMarcLugar]]):-
   2.a) unifica_marca(Lugar,[NomeLugar,RestoMarcLugar]).
```

alteracao_habilitada(conj de saida, marcação).

Este predicado verifica se a alteração, obtida pela valorização dos termos das portas, está habilitada. Isto é feito através da verificação da ausência das marcas de saída na marcação considerada, ou seja, a unificação das marcas de saída com as marcas da marcação deve falhar. A presença das marcas de entrada na marcação considerada não é verificada porque a valorização dos termos das portas de entrada é feita tomando apenas marcas existentes nesta marcação.

```
alteracao_habilitada([CjSaiAlt,CjSaiRes],Marcacao):-
  a) ifthen(CjSaiAlt \== [],
  b)        not(unifica_cj_marcacao(CjSaiAlt,Marcacao)) ),
  c) ifthen(CjSaiRes \== [],
  d)        not(unifica_cj_marcacao(CjSaiRes,Marcacao)) ).
```

6.2.2 Implementação da Verificação de Conflito

conflito(conexão 1, conexão 2, marcacao).

Este predicado implementa a verificação da existência de conflito entre as alterações definidas pelas duas conexões na marcação considerada (veja item 4.2.1.2.1.1 Tipos de Consultas). Para fazer isto primeiro é atualizado o conhecimento do usuário (predicado "a", visto no item anterior) e verificado a validade da marcação inicial (predicado "b", visto no item anterior). Estes dois

predicados têm um operador especial de corte do Arity Prolog ([! predicado(s) !]) que indica que o(s) predicado(s) não deve(m) ser considerado(s) no retrocesso. Isto evita trabalho desnecessário.

Logo após encontra alterações válidas definidas pelas duas conexões (predicados "c" e "d") e testa se estas alterações podem formar um conjunto de alterações que seja um passo (predicado "e"). Caso não possam, estas alterações são conflitantes e, portanto, são listadas (predicado "f").

O último predicado ("g") obriga que sempre seja feito o retrocesso, fazendo com que sejam verificados todos os pares de alterações habilitadas para as duas conexões.

```
conflito(Conex1, Conex2, Marcacao):-
  a) [!atualiza_conhecimento_usuario!],
  b) [!marcacao_inicial_valida(Marcacao)!],
  c) encontra_alter_habil(Conex1, CjEntAlt1, CjEntRes1,
                          CjSaiAlt1, CjSaiRes1, Marcacao),
  d) encontra_alter_habil(Conex2, CjEntAlt2, CjEntRes2,
                          CjSaiAlt2, CjSaiRes2, Marcacao),
  e) ifthen( not(mesmo_passo(Marcacao, CjEntAlt1, CjSaiAlt1,
                             CjEntAlt2, CjSaiAlt2)),
  f)        (nl, write(Conex1), write(CjEntAlt1), write(' x '),
             write(Conex2), write(CjEntAlt2)) ),
  g) fail.
```

mesmo_passo(marcacao, conj entrada conexão 1,
conj saída conexão 1,
conj entrada conexão 2,
conj saída conexão 2).

Este predicado tem sucesso quando duas alterações, identificadas pelas marcas de entrada e de saída, podem fazer parte do mesmo passo. Ou seja, quando as duas alterações não possuem marcas de entrada e/ou marcas de

saída em comum.

Isto é feito tentando fazer com que as duas alterações ocorram paralelamente (predicados de "a" até "d"), o que só é possível se as mesmas não tiverem marcas de entrada em comum, senão uma desabilita a outra. Depois é verificado se a marcação resultante da ocorrência das alterações é válida (predicado "e").

```
mesmo_passo(Marcacao, CjEnt1, CjSai1, CjEnt2, CjSai2):-
  a) remove(CjEnt1, Marcacao, M1),
  b) remove(CjEnt2, M1, M2),
  c) append(CjSai1, M2, M3),
  d) append(CjSai2, M3, NovaMarc),
  e) marcacao_valida(NovaMarc).
```

remove(conj entrada, marcação, nova marcação).

Este predicado retira o conjunto de marcas de entrada, lista de portas (lugar e marca), da marcação gerando uma nova marcação sem estas marcas. Este predicado falha se alguma das marcas a serem removidas não estiver presente na marcação.

O predicado tem duas cláusulas. A primeira é o caso trivial em que o conjunto de marcas de entrada (lista de portas) esta vazio e a marcação fica inalterada. A segunda é o caso geral e percorre o conjunto de marcas de entrada pegando porta a porta (predicado "2.a") e removendo as marcas de cada porta do respectivo lugar na marcação(predicado "2.b").

```
1. remove([], Marcacao, Marcacao).
2. remove([Porta|RestoCjEnt], Marcacao, NovaMarc):-
  2.a) remove(RestoCjEnt, Marcacao, NovaMarc1),
  2.b) remove_lugar(Porta, NovaMarc1, NovaMarc).
```

remove_lugar(porta, marcação, nova marcação).

Este predicado remove as marcas de uma porta, do

respectivo lugar da marcação, retornando uma nova marcação sem estas marcas. Isto é feito através de duas cláusulas. A primeira remove as marcas (termos já valorizados) de portas da marcação do respectivo lugar da rede de Petri (predicado "1.a"). A segunda é recursiva (predicado "2.a") e vai percorrendo todos os lugares na marcação até encontrar aquele que tem nome igual ao nome de lugar na porta.

```
1. remove_lugar([NomeLugar,Termos],
                [[NomeLugar,MarcLugar]|RestoMarc],
                  [[NomeLugar,NovaMarcLugar]|RestoMarc]):-
   1.a) remove_marcacao(Termos,MarcLugar,NovaMarcLugar).
2. remove_lugar(Porta,[Lugar|RestoMarc],[Lugar|RestoNM]):-
   2.a) remove_lugar(Porta,RestoMarc,RestoNM).
```

remove_marcacao(lista de termos de porta,
marcação de lugar,
nova marcação de lugar**).**

Este predicado remove a lista de termos de porta, já valorizados, da marcação de um lugar retornando uma nova marcação de lugar sem estas marcas. Isto é feito através de duas cláusulas, onde a primeira é o caso trivial em que a lista de termos de porta está vazia e portanto a marcação do lugar permanece inalterada. A segunda é o caso geral, recursiva (predicado "2.a"), em que a lista de termos de porta é percorrida do início ao fim, e cada um destes termos é removido através do predicado "2.b".

```
1. remove_marcacao([],MarcLugar,MarcLugar).
2. remove_marcacao([Termo|RestoTermos],
                                MarcLugar,NovaMarcLugar):-
   2.a) remove_marcacao(RestoTermos,MarcLugar,
                                          NovaMarcLugar1),
   2.b) remove_marca(Termo,NovaMarcLugar1,NovaMarcLugar).
```

remove_marca(termo valorizado,

marcação de lugar,

nova marcação de lugar).

Este predicado remove a marca (termo valorizado) da marcação de lugar retornando uma nova marcação de lugar sem esta marca. O predicado tem duas cláusulas, a primeira identifica a marca como sendo o primeiro elemento da lista (marcação de lugar) e devolve o resto da lista.

A segunda é recursiva (predicado "2.a"), e é executada quando a primeira cláusula falha. Ela tenta remover a marca do resto da lista até encontrá-la.

```
1. remove_marca(Marca, [Marca|Resto], Resto).
2. remove_marca(Marca, [Marca1|Resto1], [Marca1|Resto2]):-
   2.a) remove_marca(Marca, Resto1, Resto2).
```

append(conj saída, marcação, nova marcação).

Este predicado inclui o conjunto de marcas de entrada, que é uma lista de portas (lugar e marca), da marcação, gerando uma nova marcação sem estas marcas.

O predicado tem duas cláusulas. A primeira é o caso trivial em que o conjunto de marcas de entrada (lista de portas) esta vazio e a marcação fica inalterada. A segunda é o caso geral e percorre o conjunto de marcas de entrada pegando porta a porta (predicado "2.a") e incluindo as marcas de cada porta no respectivo lugar na marcação (predicado "2.b").

```
1. append([], Marcacao, Marcacao).
2. append([Porta|RestoCjSai], Marcacao, NovaMarc):-
   2.a) append(RestoCjSai, Marcacao, NovaMarc1),
   2.b) append_lugar(Porta, NovaMarc1, NovaMarc).
```

append_lugar(porta, marcação, nova marcação).

Este predicado adiciona as marcas de uma porta, no respectivo lugar da marcação, retornando uma nova marcação com estas marcas incluídas. Isto é feito através de duas

cláusulas, a primeira remove as marcas (termos já valorizados) de portas da marcação do respectivo lugar da rede de Petri (predicado "1.a"); a segunda é recursiva (predicado "2.a") e percorre todos os lugares na marcação até encontrar aquele que tem nome igual ao nome de lugar na porta.

```
1. append_lugar([NomeLugar,Termos],
                    [[NomeLugar,MarcLugar]|RestoMarc],
                   [[NomeLugar,NovaMarcLugar]|RestoMarc]):-
   1.a) concatena(Termos,MarcLugar,NovaMarcLugar).

2. append_lugar(Porta,[Lugar|RestoMarc],[Lugar|RestoNM]):-
   2.a) append_lugar(Porta,RestoMarc,RestoNM).
```

concatena(lista 1, lista 2, nova lista**).**

Este predicado retorna uma lista (nova lista) contendo o resultado da concatenação de duas listas (lista 1 e lista 2). O mesmo tem duas cláusulas, onde a primeira é o caso trivial em que a concatenação da lista vazia com uma lista é a própria lista. A segunda cláusula é recursiva: percorre a primeira lista e concatena cada um de seus elementos com a terceira lista até aquela ficar vazia, quando então concatena com a segunda lista através da primeira cláusula.

```
1. concatena([],Lista,Lista).
2. concatena([Elem|RestoLista],Lista,[Elem|RestoNovLista]):-
   2.a) concatena(RestoLista,Lista,RestoNovLista).
```

marcacao_valida(marcação**).**

Este predicado verifica se a marcação em questão é válida, isto é, se a capacidade de todos os seus lugares é respeitada. Isto é feito através de duas cláusulas onde a primeira é o caso trivial, em que uma marcação vazia é válida; e a segunda percorre recursivamente todos os lugares

da marcação (predicado "2.e") e verifica se a capacidade de cada lugar è respeitada. Ou seja, encontra o número de marcas presentes no lugar (predicado "2.b") e a capacidade do respectivo lugar (predicado "2.c") e testa se o número de marcas encontrado è menor ou igual à capacidade (predicado "2.e").

```
1. marcacao_valida([]).
2. marcacao_valida([[Lugar,MarcLugar]|RestoMarc]):-
   2.a) !,
   2.b) length(MarcLugar,NumMarcas),
   2.c) capacidade(Lugar,Capacidade),
   2.d) NumMarcas =< Capacidade,
   2.e) marcacao_valida(RestoMarc).
```

6.2.3 Implementação da Verificação de Concorrência

concorrencia(conexão_1, conexão_2, marcação).

Este predicado verifica a existência de concorrência (paralelismo) entre as alterações definidas pelas duas conexões informadas, para a marcação considerada (veja item 4.2.1.2.1.1 Tipos de Consultas). Para isto è feita a mesma sequência de passos do item anterior (6.2.2 Implementação da Verificação de Conflito): atualização do conhecimento do usuário, verificação da validade da marcação inicial, procura por alterações válidas definidas pelas duas conexões, teste da possibilidade destes pares de alterações formarem um passo, listagem das alterações que são concorrentes, e por último o predicado obrigando que sempre seja feito o retrocesso, o que faz com que sejam verificados todos os pares de alterações habilitadas para as duas conexões.

A diferença do item anterior está no predicado "e" que testa a possibilidade destes pares de alterações formarem um passo. Para que sejam conflitantes, um par de

alterações não podem estar em um mesmo passo; e para que sejam concorrentes, um par de alterações podem estar no mesmo passo.

```
concorrencia(Conex1, Conex2, Marcacao):-
  a) [!atualiza_conhecimento_usuario!],
  b) [!marcacao_inicial_valida(Marcacao)!],
  c) encontra_alter_habil(Conex1, CjEntAlt1, CjEntRes1,
                          CjSaiAlt1, CjSaiRes1, Marcacao),
  d) encontra_alter_habil(Conex2, CjEntAlt2, CjEntRes2,
                          CjSaiAlt2, CjSaiRes2, Marcacao),
  e) ifthen( mesmo_passo(Marcacao, CjEntAlt1, CjSaiAlt1,
                                   CjEntAlt2, CjSaiAlt2),
  f)      (nl, write(Conex1), write(CjEntAlt1), write(' x '),
                      write(Conex2), write(CjEntAlt2)) ),
  g) fail.
```

6.2.4 Implementação da Verificação de Bloqueio

bloqueio(marcação).

Este predicado verifica se a rede de Petri fica bloqueada com relação à marcação informada (veja item 4.2.1.2.1.1 Tipos de Consultas). Uma marcação bloqueia a rede quando todas alterações possíveis, definidas por cada uma das conexões da rede, estão desabilitadas para esta marcação. Neste caso a rede fica "parada".

O predicado tem duas cláusulas. A primeira percorre todas as alterações definidas por cada conexão da rede e procura encontrar uma que esteja habilitada e produza uma marcação válida, quando então escreve uma mensagem informando que a marcação não é um ponto de bloqueio e qual foi a alteração encontrada. A segunda cláusula é executada quando a primeira falhar, e escreve uma mensagem informando que a marcação é um ponto de bloqueio.

Na primeira cláusula, todas as alterações possíveis são verificadas, graças ao retrocesso do Prolog: quando um predicado falha, é feito um retrocesso para o predicado anterior e é tentada outra alternativa; se todas alternativas para este predicado foram tentadas então novamente é feito retrocesso, até não houverem mais alternativas. Para uma explicação mais detalhada do retrocesso veja [BRA 86].

A primeia cláusula atualiza o conhecimento do usuário (predicado "1.a"), verifica se a marcação informada é válida (predicado "1.b"), encontra as alterações habilitadas (predicado "1.c"), para cada alteração habilitada efetua a ocorrência da mesma (predicados "1.d" e "1.e") e verifica se a marcação resultante é válida (predicado "1.f"), e, por último, se encontrar uma alteração válida escreve a mensagem correspondente (predicado "1.g").

```
1. bloqueio(Marcacao):-
   a) [!atualiza_conhecimento_usuario!],
   b) [!marcacao_inicial_valida(Marcacao)!],
   c) encontra_alter_habil(Conex, CjEntAlt, CjEntRes,
                           CjSaiAlt, CjSaiRes, Marcacao),
   d) remove(CjEntAlt, Marcacao, NovaMarcW),
   e) append(CjSaiAlt, NovaMarcW, NovaMarc),
   f) [!marcacao_valida(NovaMarc)!],
   g) nl,
      write('Nao e'' bloqueio. Foi encontrada a alteracao:'),
      nl,
      write(Conex),
      write(CjEntAlt).

2. bloqueio(Marcacao):-
        nl,write('A marcacao e'' um ponto de bloqueio.').
```

6.2.5 Implementação da Verificação de Alcançabilidade

alcancabilidade(marcação inicial, marcação final).

Este predicado verifica se a marcação final è alcançàvel a partir da marcação inicial, e, se for o caso, lista a sequência de alterações que "atingem" a marcação final (veja item 4.2.1.2.1.1 Tipos de Consultas).

Para isto, o predicado faz uma sèrie de passos iniciais: verifica a validade da marcação inicial (predicado "a"), limpa a memòria (predicado "b"), atualiza o conhecimento do usuàrio (predicado "c"), grava a marcação final na base de dados (predicado "d") e calcula o valor heurìstico para a função componente **h(n)** da marcação inicial (predicado "e"). Por último, o predicado verifica se o objetivo foi atingido (as duas marcações foram informadas iguais), quando, então, o predicado tem sucesso; caso contràrio a àrvore de alcançabilidade começa a ser gerada gravando o nodo raiz (predicado "g") e inicia-se a pesquisa da alcançabilidade (predicado "h").

O ponto e vìrgula (;) no predicado "f" significa um "ou".

```
alcancabilidade(MarcInicial,MarcFinal):-
  a) marcacao_inicial_valida(MarcInicial),
  b) limpa_memoria,
  c) atualiza_conhecimento_usuario,
  d) assert(marcacao_final(MarcFinal)),
  e) funcao_componente_h(MarcInicial,Heur_h),
  f) (verifica_objetivo(Heur_h);
  g)  grava_fronteira(nodo([1],[fronteira],Heur_h,0,[],
                                              MarcInicial)),
  h)  percorre_arvore).
```

limpa_memoria.

Este predicado remove todos elementos das três

árvores balanceadas: **fronteira**, **todosnodos** e **arvorealc**. E, além disto, remove da base de dados todas as cláusulas com nome **marcacao_final** e aridade um.

limpa_memoria:-

a) removeallb(fronteira),

b) removeallb(todosnodos),

c) removeallb(arvorealc),

d) abolish(marcacao_final/1).

funcao_componente_h(marcação, valor heurístico**)**.

Este predicado calcula o valor heurístico da função componente **h(n)** (veja o item 5.3.3.1.1 Informação Heurística) para a marcação considerada. Isto é feito acessando a marcação final (predicado "a") e fazendo uma análise de diferença entre as duas marcações (predicado "b").

funcao_componente_h(Marcacao,Heur_h):-

a) marcacao_final(MarcacaoFinal),

b) analisa_difer(Marcacao,MarcacaoFinal,Heur_h).

analisa_difer(marcação 1, marcação 2, valor heurístico**)**.

Este predicado calcula o valor heurístico através da análise de diferença das duas marcações. Isto é feito através de duas cláusulas. A primeira é o caso trivial, onde a diferença entre duas marcações vazias é zero. A segunda cláusula percorre recursivamente todos os lugares da segunda marcação (predicado "2.c") e para cada um deste lugares pega o respectivo lugar na primeira marcação (predicado "2.a"). Depois faz uma análise de diferença das marcações destes lugares (predicado "2.b") e vai somando estas diferenças (predicado "2.d").

Não interessa a ordem dos lugares nas duas marcações, porém todos os lugares devem estar presentes.

1. analisa_difer([],[],0).

```
2. analisa_difer(Marcacao,
                 [[NomeLugarF,MarcLugarF]|RestoMarcF],
                 Heur_h):-
   2.a) [!pega_lugar(NomeLugarF,Marcacao,MarcSemLugar,
                                          MarcLugar)!],
   2.b) [!analisa_dif_lugar(MarcLugar,MarcLugarF,
                                          HeurLugar)!],
   2.c) analisa_difer(MarcSemLugar,RestoMarcF,HeurResto),
   2.d) Heur_h is HeurLugar + HeurResto.
```

pega_lugar(nome de lugar, marcação,
marcação sem lugar,
marcação do lugar).

Este predicado procura o lugar da marcação que tem o nome de lugar informado, e retorna a marcação deste lugar e a marcação sem este lugar. Isto è feito recursivamente atravès de duas clàusula: a primeira tem sucesso quando encontra e a segunda segue procurando no resto da marcação.

```
1. pega_lugar(NomeLugar,
              [[NomeLugar,MarcLugar]|RestoMarc],
                          RestoMarc,MarcLugar).

2. pega_lugar(NomeLugar,[Lugar|RestoMarc1],
                        [Lugar|RestoMarc2],MarcLugar):-
   2.a) pega_lugar(NomeLugar,RestoMarc1,RestoMarc2,MarcLugar).
```

analisa_dif_lugar(marcação de lugar 1,
marcação de lugar 2,
heuristica do lugar).

Este predicado retorna o valor heuristico do lugar fazendo uma anàlise de diferença entre os dois lugares informados. O predicado tem très clàusulas, sendo que as duas primeiras são os casos triviais: a diferença entre duas marcações (listas de marcas) iguais è zero, e a diferença entre uma lista vazia e outra lista è o número de elementos

desta (predicado "2.a"). A terceira cláusula percorre recursivamente todas as marcas da primeira lista (predicado "3.b") e para cada uma destas marcas verifica se ela está na segunda lista de marcas (predicado "3.a") retornando a heurística a nível de marca que pode ser zero ou um.

```
1. analisa_dif_lugar(MarcLugar,MarcLugar,0).
2. analisa_dif_lugar([],Resto,Tam):-
   2.a) length(Resto,Tam).
3. analisa_dif_lugar([Marca|Resto],MarcLugar,HeurLugar):-
   3.a) [!elimina_marca(Marca,MarcLugar,MLSemMarca,
                                        HeurMarca)!],
   3.b) analisa_dif_lugar(Resto,MLSemMarca,HeurRestoLugar),
   3.c) HeurLugar is HeurMarca + HeurRestoLugar.
```

elimina_marca(marca a ser eliminada,
marcação de lugar,
marcação de lugar sem a marca,
heurística da marca**)**.

Este predicado procura uma marca na marcação de lugar e se encontrar retorna esta marcação sem a respectiva marca. O predicado retorna também o valor heurístico a nível de marca que é 0 (zero) se encontrar a marca, ou é 1 (um) caso contrário.

```
1. elimina_marca(Marca,[],[],1).
2. elimina_marca(Marca,[Marca|Resto],Resto,0).
3. elimina_marca(Marca,[OutroMarca|Resto1],
                        [OutroMarca|Resto2],Heur):-
   3.a) elimina_marca(Marca,Resto1,Resto2,Heur).
```

verifica_objetivo(valor heurístico**)**.

Este predicado verifica se o objetivo (marcação alcançável) foi atingido. Isto ocorre quando não houver diferenças entre a marcação atual e a marcação final, ou

seja o valor heurístico da função componente **h(n)** é igual a 0 (zero).

verifica_objetivo(Heur_h):-

a) Heur_h=:=0.

grava_fronteira(nodo a ser gravado).

Este predicado insere o nodo na base de dados (predicado "b") e o endereço do nodo informado nas três árvores balanceadas: **todosnodos** (predicado "c"), **fronteira** (predicado "g") e **arvorealc** (predicado "1").

grava_fronteira(Nodo):-

a) arg(3,Nodo,Heur_h),

b) recordz(nodo,Nodo,Refnum),

c) recordb(todosnodos,Heur_h,Refnum),

d) arg(4,Nodo,Heur_g),

e) constante_heuristica(K),

f) Heur_f is Heur_h + (Heur_g * K),

g) recordb(fronteira,Heur_f,Refnum),

h) arg(1,Nodo,LIdent),

i) recordb(arvorealc,LIdent,Refnum).

percorre_arvore.

Este predicado efetua a pesquisa no espaço de estados utilizando o método de pesquisa em extensão informado ("best-first") como visto no quinto capítulo.

A estrutura de controle do predicado ficou um tanto complexa devido ao fato da estrutura de dados ser alterada durante a pesquisa no espaço de estados e de não ser possível fazer retrocesso em estruturas de dados que tenham sido alteradas. O predicado tem, basicamente, a seguinte estrutura de controle:

- o predicado em questão nunca falha devido ao predicado de repetição ("a"), pois o mesmo sempre sucede tentando outra alternativa;

- a pesquisa na árvore de alcançabilidade encerra

por dois motivos:

1) quando não há mais nodos para selecionar (o teste do predicado "c" é falso); ou

2) quando encontrou um caminho que torna a marcação alcançável (o predicado "k" sucede).

Todos os nodo que são selecionados (predicado "b") são expandidos, o que é feito como segue. Primeiro é removido o fato que indica que o nodo foi expandido (predicado "d") e o contador de nodos expandidos é inicializado com 1 (um) (predicado "e"). A seguir, se o nodo não tiver nodos filhos (predicado "g"), então o tipo do nodo é alterado para **terminal** (predicado "h"). Isto é feito apenas no retrocesso, após a tentativa de expansão do nodo (predicados de "i" a "k"). Todas as alterações válidas são procuradas via retrocesso (predicado "j"), e para cada uma destas é tentada a sua ocorrência (predicado "k").

```
percorre_arvore:-
  a) repeat,
  b) seleciona_nodo(Nodo),
  c) ifthen(Nodo \= fim,
  d) (    abolish(expandido),
  e)      ctr_set(0,1),
  f)      arg(6,Nodo,Marcacao),
  g)      ifthen(sem_filhos,
  h)          altera_tipo_nodo(Nodo,[terminal])),
  i)      assert(expandido),
  j)      encontra_alter_habil(Conex,CjEntAlt,CjEntRes,
                                   CjSaiAlt,CjSaiRes,Marcacao),
  k)      ocorre_alter(Conex,Nodo,[CjEntAlt,CjEntRes],
                                       CjEntAlt,CjSaiAlt)   )).
```

seleciona_nodo(nodo_selecionado**).**

Este predicado seleciona e retorna o nodo com

menor valor heurístico que ainda não tenha sido expandido. Isto é feito lendo um endereço de nodo da árvore balanceada **fronteira** utilizando como chave uma variável não instanciada (predicado "1.a"), e acessando o nodo na base de dados (predicado "1.c"). A seguir o endereço deste nodo é removido da árvore fronteira (predicado "1.d") e o seu tipo é alterado para **interno** (predicados "1.e" e "1.f").

O predicado tem duas cláusulas. Quando a primeira falhar significa que não há mais nodos fronteiras para serem expandidos, logo a árvore de alcançabilidade já foi completamente gerada. E assim uma mensagem informando que a marcação não é alcançável é escrita (predicado 2.a") e o conteúdo da variável **Nodo** é instanciado com a constante "fim" (cabeça da segunda cláusula) indicando o final da pesquisa.

O operador de corte (predicado "1.b") foi colocado para evitar que a árvore **fronteira** seja lida no retrocesso. Isto não deve ser feito porque esta árvore é atualizada durante o processo de expansão do nodo e, portanto, ocorreria erro de execução.

```
1. seleciona_nodo(Nodo):-
  1.a) retrieveb(fronteira,Heur,Refnum),
  1.b) !,
  1.c) [!instance(Refnum,Nodo)!],
  1.d) [!removeb(fronteira,Heur,Refnum)!],
  1.e) [!argrep(Nodo,2,[interno],NodoInt)!],
  1.f) [!replace(Refnum,NodoInt)!].
2. seleciona_nodo(fim):-
  2.a) nl,nl,write('marcacao nao alcancavel'),nl.
```

sem_filhos.

Este predicado sucede quando o nodo considerado foi expandido (predicado "a") e não foram gerados nodos filhos para ele (predicado "b").

O predicado "b" verifica se o contador zero (global) tem como valor 1 (um). O mesmo é utilizado para contar o número de nodos filhos gerados mais um.

sem_filhos:-
  a) expandido,
  b) ctr_is(0,1).

altera_tipo_nodo(nodo, tipo).

Este predicado substitui o tipo do nodo na estrutura armazenada em **Nodo**, para o tipo informado (predicado "a"). Depois acessa o esdereço do nodo na base de dados (predicados "b" e "c") e substitui a estrutura alterada na base de dados através do predicado "d".

altera_tipo_nodo(Nodo, Tipo):-
  a) argrep(Nodo, 2, Tipo, NodoAlterado),
  b) arg(1, Nodo, LIdent),
  c) retrieveb(arvorealc, LIdent, Refnum),
  d) replace(Refnum, NodoAlterado).

ocorre_alter(conexão, nodo, conj_entrada,
conj_alterador_entrada,
conj_alterador_saida**).**

Este predicado tenta fazer com que a alteração informada (identificada pela conexão e pelos conjuntos alteradores de entrada e de saída) ocorra sobre a marcação do nodo em expansão. Isto é feito removendo as marcas alteradoras de entrada da marcação (predicado "a"), incluindo as marcas alteradoras de saída na marcação (predicado "b"), e verificando se a marcação resultante é válida (predicado "c"). Após a ocorrência da alteração, é executada uma série de passos visando atualizar a base de dados (predicados de "d" até "m") e é feita a verificação do objetivo (predicado "n") para saber se a nova marcação é igual a marcação a ser alcançada. Se o objetivo foi atingido

então é impressa a sequência de alterações pela qual se verifica a alcançabilidade (predicados "o" e "p"), e a base de dados é limpa (predicado "q"). Neste último caso o predicado sucede e encerra-se a pesquisa na árvore de alcançabilidade.

```
ocorre_alter(Conex,
             nodo(LIdent, Tipo, Heur_h, NAlter, _, Marcacao), CjEnt,
                                                  CjEntAlt, CjSaiAlt):-
  a) remove(CjEntAlt, Marcacao, NovaMarcW),
  b) append(CjSaiAlt, NovaMarcW, NovaMarc),
  c) [!marcacao_valida(NovaMarc)!],
  d) ctr_inc(0, Ident),
  e) [!  funcao_componente_h(NovaMarc, Heur_hNM)  !],
  g) [!concatena(LIdent, [Ident], NLIdent)!],
  h) inc(NAlter, NNAlter),
  i) Alteracao = [Conex, CjEnt],
  j) NodoGerado = nodo(NLIdent, [fronteira], Heur_hNM,
                                  NNAlter, Alteracao, NovaMarc),
  k) [!ifthenelse(marc_dupl(NovaMarc, Heur_hNM, NodoDuplo),
  l)    duplicado(NodoGerado, NodoDuplo),
  m)    grava_fronteira(NodoGerado)   ) !],
  n) verifica_objetivo(Heur_hNM),
  o) cls, write('Sequencia de Alteracoes'), nl,
  p) mostra_alcanc(NLIdent),
  q) limpa_memoria.
```

marc_dupl(marcação, valor_heuristico, nodo).

Este predicado sucede se a marcação é duplicada em relação à marcação de algum nodo já gerado (predicado "e") e neste caso retorna o nodo cuja marcação é duplicada. Isto é feito acessando todos os nodos, através do retrocesso, que tem o mesmo valor heurístico da função componente **h(n)**. O predicado falha quando todas as alternativas foram tentadas e nenhuma tem marcação igual. Os nodos que tem tipo igual a **duplicado** não são considerados (predicado "c").

```
marc_dupl(Marcacao, Heur_h, Nodo):-
  a) retrieveb(todosnodos, Heur_h, Refnum),
  b) instance(Refnum, Nodo),
  c) not(arg(2, Nodo, [duplicado, _])),
  d) arg(6, Nodo, MarcN),
  e) iguais(Marcacao, MarcN).
```

iguais(marcação 1, marcação 2**).**

Este predicado verifica a igualdade de duas marcações. Isto é feito testando se a análise de diferença entre as duas marcações é igual a zero (predicado "a").

```
iguais(Marcacao, MarcN):-
  a) [!  analisa_difer(Marcacao, MarcN, 0)  !].
```

duplicado(nodo duplo, nodo gerado**).**

Este predicado testa se o nodo duplicado é do tipo **terminal** (predicado "a"), e neste caso insere o nodo gerado na estrutura de dados com tipo igual a **terminal** (predicado "b"), senão altera a estrutura de dados (predicado "c") conforme foi visto na seção 5.5 Alterações na Base de Dados em Decorrência da Expansão do Espaço de Estados.

```
duplicado(NodoDuplo, NodoGerado):-
  a) ifthenelse(arg(2, NodoDuplo, [terminal]),
  b)     terminal(NodoGerado),
  c)     nao_terminal(NodoDuplo, NodoGerado)).
```

terminal(nodo gerado**).**

Este predicado insere o nodo gerado na base de dados com tipo igual a **terminal** (predicados "a" e "b") e insere o endereço deste nodo nas árvores balanceadas **todosnodos** (predicados "c" e "d") e **arvorealc** (predicados "e" e "f").

```
terminal(NodoGerado):-
  a) argrep(NodoGerado, 2, [terminal], NodoTerm),
  b) recordz(nodo, NodTerm, Refnum),
```

c) arg(3,NodoTerm,Heur_h),

d) recordb(todosnodos,Heur_h,Refnum),

e) arg(1,NodoTerm,LIdent),

f) recordb(arvorealc,LIdent,Refnum).

nao_terminal(nodo duplo,nodo gerado**).**

Este predicado compara o número de alterações do nodo duplicado com o do nodo gerado (predicado "d"). Aquele que tiver o maior número de alterações é colocado na árvore de alcançabilidade como sendo um nodo duplicado.

nao_terminal(NodoDuplo,NodoGerado):-

a) arg(4,NodoDuplo,NAlterD),

b) arg(4,NodoGerado,NAlterG),

c) arg(1,NodoDuplo,LIdent),

d) ifthenelse(NAlterG >= NAlterD,

e) nodo_gerado_duplo(NodoGerado,LIdent),

f) nodo_duplo_duplo(NodoDuplo,NodoGerado)).

nodo_gerado_duplo(nodo gerado,
lista identificação nodo duplo**).**

Este predicado insere o nodo gerado na base de dados com o tipo igual a **duplicado** e com a lista de identificação do nodo duplo (predicados "a" e "b"). A seguir insere o endereço deste nodo nas árvores balanceadas **todosnodos** e **arvorealc**.

nodo_gerado_duplo(NodoGerado,LIdentD):-

a) argrep(NodoGerado,2,[duplicado,LIdentD],NodoDuplicado),

b) recordz(nodo,NodoDuplicado,Refnum),

c) arg(3,NodoDuplicado,Heur_h),

d) recordb(todosnodos,Heur_h,Refnum),

e) arg(1,NodoDuplicado,LIdent),

f) recordb(arvorealc,LIdent,Refnum).

nodo_duplo_duplo(nodo duplo,nodo gerado**).**

Este predicado insere o nodo gerado na base de dados com o tipo igual a **fronteira** e insere o endereço deste

nodo nas três árvores balanceadas: **fronteira**, **todosnodos** e **arvorealc** (predicado "a"). A seguir altera o tipo do nodo duplo para **duplicado** (predicados "d" e "e") e se ele tinha tipo igual a **fronteira** então o remove da árvore balanceada **fronteira** (predicado "c").

```
nodo_duplo_duplo(NodoDuplo,NodoGerado):-
  a) grava_fronteira(NodoGerado),
  b) ifthen(arg(2,NodoDuplo,[fronteira]),
  c)   retira_front(NodoDuplo)),
  d) arg(1,NodoGerado,LIdent),
  e) altera_tipo_nodo(NodoDuplo,[duplicado,LIdent]).
```

retira_front(nodo a ser retirado).

Este predicado remove o endereço, do nodo a ser retirado, da árvore balanceada **fronteira** através do predicado "g". Para isto é necessário saber a chave de acesso, valor heurístico; e o conteúdo do elemento da árvore, endereço do nodo na base de dados (veja o item 5.3.1.3 Representação do Espaço de Estados e a seção 5.5 Alterações na Base de Dados em Decorrência da Expansão do Espaço de Estados). Os predicados "a" e "f" encontram o endereço do nodo na base de dados; e os predicados que vão de "b" até "e" encontram o valor heurístico, chave de acesso da árvore **fronteira**.

```
retira_front(Nodo):-
  a) arg(1,Nodo,LIdent),
  b) arg(3,Nodo,Heur_h),
  c) arg(4,Nodo,Heur_g),
  d) constante_heuristica(K),
  e) Heur_f is Heur_h + (Heur_g * K),
  f) retriveb(arvorealc,LIdent,Refnum),
  g) removeb(fronteira,Heur_f,Refnum).
```

mostra_alcanc(lista de identificação).

Este predicado mostra a sequência de alterações

que devem ocorrer a partir da marcação inicial para chegar na marcação final. Através da lista de identificação do nodo que contem a marcação final é montada a lista das alterações (predicado "a"). Depois esta lista de alterações é escrita (predicado "b").

mostra_alcanc(LIdent):-

a) monta_lista_alter(ListaAlter,LIdent),

b) escreve_lista(ListaAlter).

monta_lista_alter(lista alterações, lista identificação**).**

Este predicado monta a lista de alterações que devem ocorrer para alcançar a marcação final. Isto é feito utilizando a lista de identificação do nodo porque através desta é possivel acessar todos os nodos da árvore de alcançabilidade que formam o caminho do nodo inicial até chegar ao nodo que contem a marcação final, cuja lista de indentificação é informada.

O predicado tem duas cláusulas. A primeira é o caso trivial onde a lista de identificação tem apenas um elemento indicando que o nodo é o raiz (inicial) e portanto não houve alteração para gerá-lo. A segunda cláusula acessa recursivamente os nodos anteriores (predicado "2.g") até chegar ao nodo raiz e volta concatenando a lista de alterações com a alteração, recuperada pelo predicado "2.e", de cada nodo acessado (predicado "2.h"). O predicado "2.f" encontra a lista de identificação do nodo anterior deletando o último elemento da lista de identificação do nodo em questão. O nodo é acessado através de dois passos: recuperação do seu endereço (**Refnum**) na árvore **arvorealc** (predicado "2.a") utilizando a chave de acesso (lista de identificação), e recuperação do nodo na base de dados através do predicado "2.b". Se o nodo é do tipo duplicado (predicado "2.c") então ele não é considerado e continua a montagem da lista de alterações com a lista de identificação

do nodo duplo (predicado "2.d").

```
1. monta_lista_alter([], [P|[]]).
2. monta_lista_alter(ListaAlter, LIdent):-
   2.a) retrieveb(arvorealc, LIdent, Refnum),
   2.b) instance(Refnum, Nodo),
   2.c) ifthenelse(arg(2, Nodo, [duplicado, LIdentDupl]),
   2.d)   monta_lista_alter(ListaAlter, LIdentDupl),
   2.e)   ( arg(5, Nodo, Alteracao),
   2.f)     del_ult_elem(LIdent, NLIdent),
   2.g)     monta_lista_alter(NListaAlter, NLIdent),
   2.h)     concatena(NListaAlter, [Alteracao], ListaAlter) )).
```

del_ult_elem(lista, nova lista).

Este predicado remove o último elemento da lista retornando uma nova lista sem o mesmo. Isto è feito através de três cláusulas em que a primeira e a segunda são os casos triviais. A primeira retorna lista vazia se a lista informada for vazia. A segunda retorna lista vazia se a lista informada tiver apenas um elemento. A terceira percorre recursivamente a lista informada, atè chegar no final da lista, e passa para a outra lista todos os elementos exceto o último.

```
1. del_ult_elem([], []).
2. del_ult_elem([E], []).
3. del_ult_elem([P|R1], [P|R2]):-
   3.a) del_ult_elem(R1, R2).
```

escreve_lista(lista).

Este predicado, utilizando duas cláusulas, escreve todos elementos de uma lista na forma vertical, isto è, um elemento em cada linha. A primeira apenas tem sucesso quando a lista está vazia e è a condição de término. A segunda cláusula percorre recursivamente todos os elementos da lista e para cada elemento o escreve e salta para a próxima linha.

```
1. escreve_lista([]).
2. escreve_lista([P|Resto]):-
   2.a) write(P),nl,
   2.b) escreve_lista(Resto).
```

7 EXEMPLOS DE UTILIZAÇAO DO PROTOTIPO

Este capitulo apresenta dois exemplos procurando mostrar o potencial da ferramenta, proposta neste trabalho, na compreensão e validação de modelos.

Como foi visto no capitulo seis, não foi implementada a interface com o usuário. Portanto é necessário ter-se um arquivo contendo o modelo (**rede.ari**), escrito diretamente na LRC, e um arquivo contendo as opções de execução (**conhecim.ari**).

Para cada exemplo apresentado é mostrada a representação do mesmo na LRC, e solicitadas algumas consultas de verificação de propriedades na LC. Após cada consulta aparece o respectivo resultado.

7.1 Primeiro Exemplo

Na figura 7.1 é apresentado um modelo na linguagem de Redes Marcadas sem maiores pretensões em relação ao significado, porèm bastante útil para ilustrar as consultas.

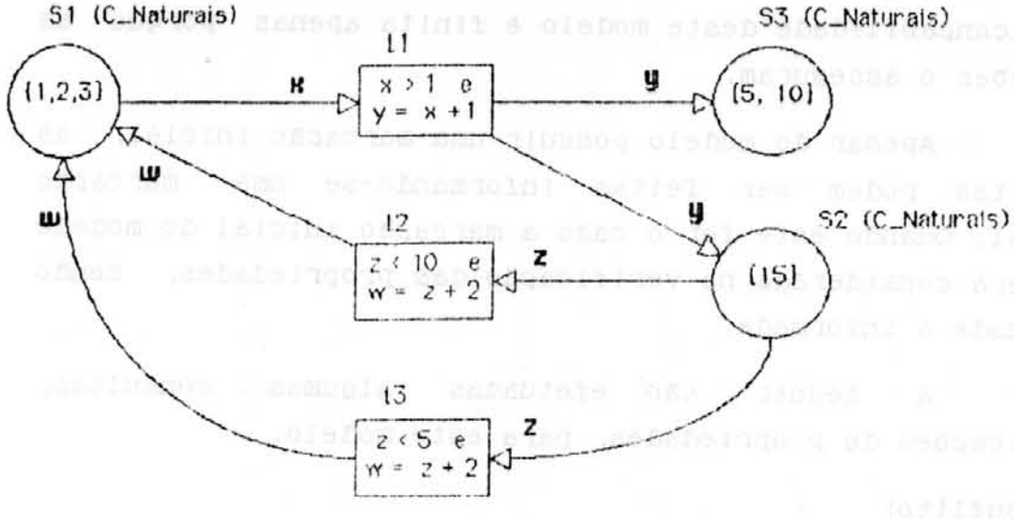


Figura 7.1 - Exemplo de Rede Marcada

A representação deste modelo (rede) na LRC é a seguinte:

```
conexao(t1,[[[s1,[X]]],[]],[[[s2,[Y]],[s3,[Y]]],[]]).
anotacoes(t1,[[[s1,[X]]],[]],[[[s2,[Y]],[s3,[Y]]],[]]):-
                                        X>1, Y is X+1.

conexao(t2,[[[s2,[Z]]],[]],[[[s1,[W]]],[]]).
anotacoes(t2,[[[s2,[Z]]],[]],[[[s1,[W]]],[]]):-
                                        Z<10, W is Z+2.

conexao(t3,[[[s2,[Z]]],[]],[[[s1,[W]]],[]]).
anotacoes(t3,[[[s2,[Z]]],[]],[[[s1,[W]]],[]]):-
                                        Z<5, W is Z+2.

capacidade(s1,N).
capacidade(s2,N).
capacidade(s3,N).
```

Para entender esta representação veja a sintaxe da LRC no item 5.3.2.1 A Linguagem de Representação do Conhecimento.

A capacidade dos lugares é infinita, o que deixa margem para que a alcançabilidade seja indecidível. A árvore de alcançabilidade deste modelo é finita apenas porque as anotações o asseguram.

Apesar do modelo possuir uma marcação inicial, as consultas podem ser feitas informando-se uma marcação inicial. Quando este for o caso a marcação inicial do modelo não será considerada na verificação das propriedades, sendo utilizada a informada.

A seguir são efetuadas algumas consultas, verificações de propriedades, para este modelo.

a) Conflito:

A consulta abaixo verifica se as conexões t2 e t3

definem alterações conflitantes, as quais serão listadas, para a marcação informada.

```
?- conflito(t2, t3, [[s1, [1, 2, 3]], [s2, [4, 5, 6]], [s3, []]]).
t2[[s2, [4]]] x t3[[s2, [4]]]
```

Existe apenas um par de alterações em conflito porque há apenas uma alteração definida pela conexão **t3** que está habilitada (veja as anotações).

b) Concorrência:

Este tipo de consulta verifica se as duas conexões informadas definem alterações concorrentes, listando-as. Isto é feito para a marcação informada.

```
?- concorrencia(t2, t3, [[s1, [1, 2, 3]], [s2, [4, 5, 6]], [s3, []]]).
t2[[s2, [5]]] x t3[[s2, [4]]]
t2[[s2, [6]]] x t3[[s2, [4]]]

?- concorrencia(t1, t2, [[s1, [1, 2, 3]], [s2, [4, 5, 6]], [s3, []]]).
t1[[s1, [2]]] x t2[[s2, [4]]]
t1[[s1, [2]]] x t2[[s2, [5]]]
t1[[s1, [2]]] x t2[[s2, [6]]]
t1[[s1, [3]]] x t2[[s2, [4]]]
t1[[s1, [3]]] x t2[[s2, [5]]]
t1[[s1, [3]]] x t2[[s2, [6]]]
```

c) Bloqueio:

As consultas deste tipo verificam se a marcação informada bloqueia a rede.

```
?- bloqueio([[s1, [1, 2, 3]], [s2, [4, 5, 6]], [s3, []]]).
Nao e' bloqueio. Foi encontrada a alteracao:
t1[[s1, [2]]]

?- bloqueio([[s1, [1]], [s2, []], [s3, [3, 5]]]).
A marcacao e' um ponto de bloqueio.
```

```
?- bloqueio([[s1,[1]],[s2,[9]],[s3,[3,5]]]).
Nao e' bloqueio. Foi encontrada a alteracao:
t2[[s2,[9]]]

?- bloqueio([[s1,[1]],[s2,[19,10]],[s3,[3,5]]]).
A marcacao e' um ponto de bloqueio.
```

d) Alcançabilidade:

Neste tipo de consulta é verificado se a segunda marcação informada é alcançável tomando como marcação inicial a primeira marcação informada. Se a marcação for alcançável, então é listada a sequência de alterações que devem ocorrer para atingi-la. Normalmente é apresentada a melhor solução se houver alguma, considerando que a melhor solução é aquela com o menor número de ocorrências de alterações.

```
-?alcancabilidade([[s1,[1,2,3]],[s2,[]],[s3,[]]],
                  [[s1,[1,6]],[s2,[3]],[s3,[3,4]]]).
Sequencia de Alteracoes
[t1,[[[s1,[2]]],[]]]
[t1,[[[s1,[3]]],[]]]
[t1,[[[s1,[4]]],[]]]

-?alcancabilidade([[s1,[1,2,3]],[s2,[]],[s3,[]]],
                                        [[s1,[1]]:_]).
Sequencia de Alteracoes
[t1,[[[s1,[2]]],[]]]
[t1,[[[s1,[3]]],[]]]

-?alcancabilidade([[s1,[1,2,3]],[s2,[]],[s3,[]]],
                  [[s1,[]],[s2,[2,3,4]],[s3,[2,3,4]]]).
marcacao nao alcancavel

-?alcancabilidade([[s1,[1,9,10]],[s2,[]],[s3,[]]],
                  [[s1,[1,6]],[s2,[3]],[s3,[3,4]]]).
marcacao nao alcancavel
```

```
-?alcancabilidade([[s1, [1, 9, 10]], [s2, []], [s3, []]],
                                                [[s1, [1]]|_]).
Sequencia de Alteracoes
[t1, [[[s1, [9]]], []]]
[t1, [[[s1, [10]]], []]]
```

7.2 Segundo Exemplo: Mercado de Trabalho

Na figura 7.2 é apresentado um modelo do mercado de trabalho na linguagem de Redes Marcadas.

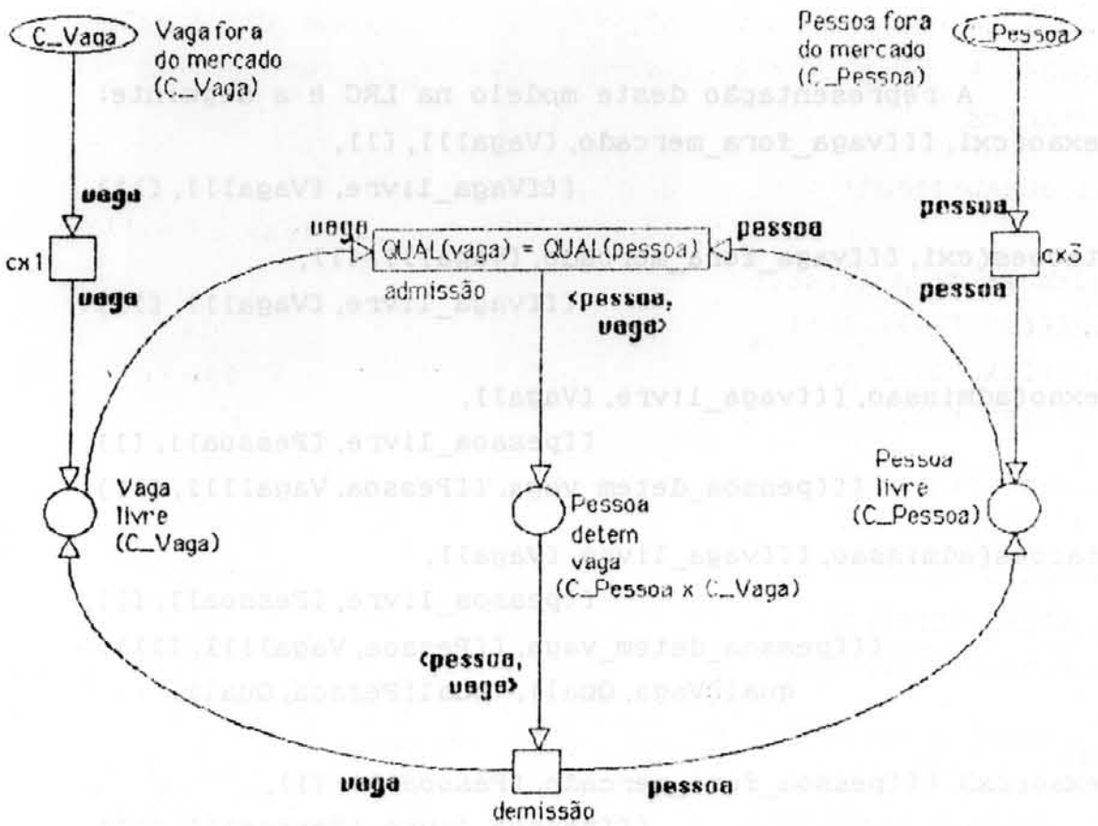


Figura 7.2 - Segundo exemplo: modelo do mercado de trabalho.

O UD deste modelo é o seguinte:

C_Vaga = {professor, analista, gerente}

C_Pessoa = {alvaro, vera, renata, ana, jose}

```
qual(professor, saber).
qual(analista, experiencia).
qual(gerente, bom_senso).
qual(alvaro, saber).
qual(ana, experiencia).
qual(vera, saber).
qual(vera, experiencia).
```

Observação: como na LRC não tem funções, utilizou-se a relação **qual** para poder fazer o mapeamento do modelo na LRC.

A representação deste modelo na LRC é a seguinte:

```
conexao(cx1, [[[vaga_fora_mercado, [Vaga]]], []],
                                [[[Vaga_livre, [Vaga]]], []]).

anotacoes(cx1, [[[vaga_fora_mercado, [Vaga]]], []],
                                [[[vaga_livre, [Vaga]]], []]).


conexao(admissao, [[[vaga_livre, [Vaga]],
                                [[pessoa_livre, [Pessoa]], []],
                [[[pessoa_detem_vaga, [[Pessoa, Vaga]]]], []]).

anotacoes(admissao, [[[vaga_livre, [Vaga]],
                                [[pessoa_livre, [Pessoa]], []],
                [[[pessoa_detem_vaga, [[Pessoa, Vaga]]]], []]):-
                    qual(Vaga, Qual), qual(Pessoa, Qual).

conexao(cx3, [[[pessoa_fora_mercado, [Pessoa]]], []],
                            [[[pessoa_livre, [Pessoa]]], []]).

anotacoes(cx3, [[[pessoa_fora_mercado, [Pessoa]]], []],
                            [[[pessoa_livre, [Pessoa]]], []]).
```

```
conexao(demissao, [[[pessoa_detem_vaga, [[Pessoa, Vaga]]]], []],
        [[[vaga_livre, [Vaga]], [pessoa_livre, [Pessoa]]], []]).

anotacoes(demissao, [[[pessoa_detem_vaga, [[Pessoa, Vaga]]]],
                                                    []],
        [[[vaga_livre, [Vaga]], [pessoa_livre, [Pessoa]]], []]).

qual(professor, saber).
qual(analista, experiencia).
qual(gerente, bom_senso).
qual(alvaro, saber).
qual(ana, experiencia).
qual(vera, saber).
qual(vera, experiencia).

capacidade(vaga_fora_mercado, N).
capacidade(vaga_livre, N).
capacidade(pessoa_detem_vaga, N).
capacidade(pessoa_fora_mercado, N).
capacidade(pessoa_livre, N).
```

Para entender esta representação veja a sintaxe da LRC no item 5.3.2.1 A Linguagem de Representação do Conhecimento.

A capacidade dos lugares é ilimitada, porèm como a cardinalidade dos dominios de lugares è limitada e não è permitido multiplicidade, então a àrvore de alcançabilidade è finita.

A seguir são efetuadas algumas consultas, verificação de propriedades, para este modelo.

a) Conflito:

A consulta abaixo verifica se existem pessoas procurando a mesma vaga, na marcação da rede considerada, e informa que existem dois casos.

```
?- conflito(admissao, admissao,
            [[vaga_fora_mercado, _], [pessoa_fora_mercado, _],
             [vaga_livre, [professor]],
             [pessoa_livre, [alvaro, vera, ana]],
             [pessoa_detem_vaga, [[jose, gerente]]]      ]).

admissao[[pessoa_livre, [alvaro]], [vaga_livre, [professor] x
     admissao[[pessoa_livre, [vera]], [vaga_livre, [professor]

admissao[[pessoa_livre, [vera]], [vaga_livre, [analista] x
      admissao[[pessoa_livre, [ana]], [vaga_livre, [analista]

admissao[[pessoa_livre, [vera]], [vaga_livre, [professor] x
     admissao[[pessoa_livre, [vera]], [vaga_livre, [analista]
```

b) Concorrência:

Esta consulta verifica se existem pessoas procurando vagas diferentes.

```
?- concorrencia(admissao, admissao,
            [[vaga_fora_mercado, _], [pessoa_fora_mercado, _],
             [vaga_livre, [professor, analista, gerente]],
             [pessoa_livre, [alvaro, vera, ana]],
             [pessoa_detem_vaga, [[jose, gerente]]]      ]).

admissao[[pessoa_livre, [alvaro]], [vaga_livre, [professor] x
     admissao[[pessoa_livre, [vera]], [vaga_livre, [analista]

admissao[[pessoa_livre, [alvaro]], [vaga_livre, [professor] x
      admissao[[pessoa_livre, [ana]], [vaga_livre, [analista]

admissao[[pessoa_livre, [vera]], [vaga_livre, [professor] x
     admissao[[pessoa_livre, [ana]], [vaga_livre, [analista]
```

c) Bloqueio:

A consulta verifica se o mercado de trabalho está bloqueado para a marcação considerada. E a resposta é

afirmativa porque nenhuma alteração pode ocorrer.

```
?- bloqueio([[vaga_fora_mercado, []], [pessoa_fora_mercado, []],
             [vaga_livre, [gerente]],
             [pessoa_livre, [alvaro, vera, ana]],
             [pessoa_detem_vaga, []]        ]).
```

A marcacao e' um ponto de bloqueio.

d) Alcançabilidade:

A consulta abaixo verifica se **vera**, que está fora do mercado de trabalho, pode vir a conseguir um emprego de **professor**, na atual conjuntura (marcação inicial).

```
-?alcancabilidade([[vaga_fora_mercado, [professor]],
                   [pessoa_fora_mercado, [vera]],
                   [vaga_livre, [gerente]],
                   [pessoa_livre, [jose, ana]],
                   [pessoa_detem_vaga, []]  ],

                        [[vaga_fora_mercado, [professor]],
                        [pessoa_fora_mercado, [vera]],
                        [vaga_livre, [gerente]],
                        [pessoa_livre, [jose, ana]],
                        [pessoa_detem_vaga, []]  ]).
Sequencia de Alteracoes
[cx1, [[[vaga_fora_mercado, [professor]]], []]]

[cx3, [[[pessoa_fora_mercado, [vera]]], []]]

[admissao, [[[vaga_livre, [professor]],
                                  [[pessoa_livre, [vera]], []]]
```

8 CONCLUSAO

8.1 Avaliações

Uma ferramenta de apoio à modelagem de sistemas utilizando redes de Petri foi proposta, alèm disto, um protòtipo para verificar algumas propriedades de modelos foi implementado em Prolog.

Procurou-se, na proposta da ferramenta deste trabalho, auxiliar o usuàrio em todas as fases do processo de modelagem de sistemas. Uma maior ènfase foi dada na verificação de propriedades de modelos.

A utilização de Redes Marcadas na modelagem de sistemas apresenta algumas dificuldades aos usuàrios não habituados com a lògica matemàtica e o formalismo [PER 89], entretanto espera-se que as vantagens oferecidas, como a verificação de propriedades, supere estes problemas.

O protòtipo mostrou-se importante para a compreensão e validação de modelos, jà que o mesmo verifica propriedades ùteis para este fim, como visto no sètimo capìtulo. Entretanto, para que o mesmo possa ser facilmente utilizado è necessàrio que os outros mòdulos da ferramenta, relacionados com a interface com o usuàrio, seja implementado.

A utilização de Prolog como linguagem de implementação ocasionou algumas dificuldades, como jà esperado. A implementação dos mòdulos relativos à interface è impossìvel em Prolog e o tempo de resposta não è muito bom. A alternativa para a implementação da interface è a linguagem C que pode ser utilizada com o Arity/Prolog [ARI 86], entretanto existem problemas não solucionados de incompatibilidade entre as duas linguagens. Apesar destes problemas, a linguagem Prolog mostrou-se extremamente ùtil

devido às suas qualidades (vide capitulo seis).

8.2 Possiveis Extensões

Apesar de a ferramenta proposta neste trabalho ser bastante ampla e complexa, pois envolve àreas diversas como computação gràfica, inteligência artificial e engenharia de software, existem muitas questões não abordadas que seriam interessantes e importantes para uma ferramenta. Abaixo segue uma relação de possiveis extensões que poderiam ser feitas.

Para ampliar o leque de auxilios na fase de validação, seria importante que fosse feita a formalização de outras propriedades, visando implementà-las, como por exemplo a verificação de regiões criticas.

No que se refere ao poder de modelagem da linguagem, seriam interessantes a definição e implementação de uma LRC que não impusesse restrições à LARP, a implementação das redes marcadas extendidas, e a possibilidade de ter-se multiplicidade nos lugares.

Relacionado com o auxilio à fase de criação de modelos, poder-se-ia trabalhar na questão de tratamento de versões de modelos, e tambèm no suporte a alguma metodologia de modelagem como a que està sendo proposta ou desenvolvida em [PER 89]. Para esta ùltima questão seria necessàrio auxiliar o usuàrio em coisas como: equivalência de redes, composição de redes e modelagem em vàrios niveis de abstração.

Apêndice A: Listagem do Programa que Implementa o Protótipo

```
/*-----------------------------------------------------------------

        Programa que implementa um protótipo
    para a verificação de propriedades em mo-
    delos criados com Redes Marcadas.

    Autor : Alvaro Guarda     -     CPGCC / UFRGS
-----------------------------------------------------------------*/

/*-----------------------------------------------------------------------------------------

            Verificação de Alcançabilidade

-----------------------------------------------------------------------------------------*/

alcancabilidade(MarcInicial,MarcFinal):-
        marcacao_inicial_valida(MarcInicial),
        limpa_memoria,
        atualiza_conhecimento_usuario,
        assert(marcacao_final(MarcFinal)),
        funcao_componente_h(MarcInicial,Heur_h),
        (verifica_objetivo(Heur_h);
        grava_fronteira(nodo([],[fronteira],Heur_h,0,[],MarcInicial)),
        percorre_arvore).


marcacao_inicial_valida([]).

marcacao_inicial_valida([[Lugar,MarcLugar]|RestoMarc]):-
        !,
        length(MarcLugar,NumMarcas),
        capacidade(Lugar,Capacidade),
        NumMarcas =< Capacidade,
        multiplicidade(MarcLugar),
        marcacao_inicial_valida(RestoMarc).
```

```
multiplicidade([]).

multiplicidade([Prim|Resto]):-
        !,
        not(member(Prim,Resto)),
        multiplicidade(Resto).


member(Prim,[Prim|Resto]).

member(Elem,[Prim|Resto]):-
        member(Elem,Resto).


 limpa_memoria:-
        removeallb(fronteira),
        removeallb(todosnodos),
        removeallb(arvorealc),
        abolish(marcacao_final/1).


atualiza_conhecimento_usuario:-
        abolish(alteracao_valida/2),
        abolish(conexao_valida/1),
        abolish(constante_heuristica/1),
        abolish(simulacao/1),
        abolish(raciocinio/1),
        reconsult(conhecim),
        validacoes_usuario.
```

```
validacoes_usuario:-
        ifthen( not(conexao_valida(Conexao)),
                assert(conexao_valida(TodasConexoes)) ),

        ifthen( not(alteracao_valida(Conex,CjEnt)),
                assert(alteracao_valida(TodasConex,TodosCjEnt)) ),

        ifthen( not(constante_heuristica(K)),
                assert(constante_heuristica(1)) ).



funcao_componente_h(Marcacao,Heur_h):-
        marcacao_final(MarcacaoFinal),
        analisa_difer(Marcacao,MarcacaoFinal,Heur_h).



analisa_difer([],[],0).

analisa_difer(Marcacao,[[NomeLugarF,MarcLugarF]|RestoMarcF],Heur_h):-
        [!pega_lugar(NomeLugarF,Marcacao,MarcSemLugar,MarcLugar)!],
        [!analisa_dif_lugar(MarcLugar,MarcLugarF,HeurLugar)!],
        analisa_difer(MarcSemLugar,RestoMarcF,HeurResto),
        Heur_h is HeurLugar + HeurResto.



pega_lugar(NomeLugar,[[NomeLugar,MarcLugar]|RestoMarc],RestoM arc,MarcLugar).

pega_lugar(NomeLugar,[Lugar|RestoMarc1],[Lugar|RestoMarc2],MarcLugar):-
        pega_lugar(NomeLugar,RestoMarc1,RestoMarc2,MarcLugar).
```

```
analisa_dif_lugar(MarcLugar,MarcLugar,0).

analisa_dif_lugar([],Resto,Tam):-
        length(Resto,Tam).

analisa_dif_lugar([Marca|Resto],MarcLugar,HeurLugar):-
        [!elimina_marca(Marca,MarcLugar,MLSemMarca,HeurMarca)!],
        analisa_dif_lugar(Resto,MLSemMarca,HeurRestoLugar),
        HeurLugar is HeurMarca + HeurRestoLugar.



elimina_marca(Marca,[],[],1).

elimina_marca(Marca,[Marca|Resto],Resto,0).

elimina_marca(Marca,[OutroMarca|Resto1],[OutroMarca|Resto2],Heur):-
        elimina_marca(Marca,Resto1,Resto2,Heur).



verifica_objetivo(Heur_h):-
        Heur_h=:=0.
          \


grava_fronteira(Nodo):-
        arg(3,Nodo,Heur_h),
         recordz(nodo,Nodo,Refnum),
        recordb(todosnodos,Heur_h,Refnum),
        arg(4,Nodo,Heur_g),
        constante_heuristica(K),
        Heur_f is Heur_h + (Heur_g * K),
        recordb(fronteira,Heur_f,Refnum),
        arg(1,Nodo,LIdent),
        recordb(arvorealc,LIdent,Refnum).
```

```
percorre_arvore:-
        repeat,
        seleciona_nodo(Nodo),   /* escolhe estado mais promissor  */
        ifthen(Nodo \= fim,
            (   abolish(expandido),  /* inicializa como nao expandido */
                ctr_set(0,1),   /*  num. de nodos filhos + 1   */
                arg(6,Nodo,Marcacao),

                ifthen(sem_filhos,  altera_tipo_nodo(Nodo,[terminal])),
                assert(expandido),  /* indica que houve expansao */
                encontra_alter_habil(Conex,CjEntAlt,CjEntRes,CjSaiAlt,CjSaiRes,Marcacao),
                ocorre_alter(Conex,Nodo,[CjEntAlt,CjEntRes],CjEntAlt,CjSaiAlt)   )).


seleciona_nodo(Nodo):-
        retrieveb(fronteira,Heur,Refnum),  !,
        [!instance(Refnum,Nodo)!],
        [!removeb(fronteira,Heur,Refnum)!],

        [!argrep(Nodo,2,[interno],Nodo(nt)!],
        [!replace(Refnum,NodoInt)!].


seleciona_nodo(fim):-
        nl,nl,write('marcacao nao alcancavel'),nl.


sem_filhos:-
        expandido,
        ctr_is(0,1).            /*   sem filhos   */


altera_tipo_nodo(Nodo,Tipo):-
        argrep(Nodo,2,Tipo,NodoAlterado),
        arg(1,Nodo,LIdent),
        retrieveb(arvorealc,LIdent,Refnum),
        replace(Refnum,NodoAlterado).
```

```
encontra_alter_habil(Conex,CjEntAlt,CjEntRes,CjSaiAlt,CjSaiRes,Marcacao):-
        conexao(Conex,[CjEntAlt,CjEntRes],[CjSaiAlt,CjSaiRes]),
        [!conexao_valida(Conex)!],
        valorizacao_cj_ent([CjEntAlt,CjEntRes],Marcacao), /*todas possiveis*/
        [!alteracao_valida(Conex,CjEntAlt)!],
        [!anotacoes(Conex,[CjEntAlt,CjEntRes],[CjSaiAlt,CjSaiRes])!],
        [!alteracao_habilitada([CjSaiAlt,CjSaiRes],Marcacao)!].


/*-------------------------------------------------------
     Sao gerados apenas as alteracoes cuja valorizacao
     seja possivel na marcacao corrente.
-------------------------------------------------------*/
valorizacao_cj_ent([CjEntAlt,CjEntRes],Marcacao):-
        unifica_cj_marcacao(CjEntAlt,Marcacao),
        unifica_cj_marcacao(CjEntRes,Marcacao).


unifica_cj_marcacao([],_).

unifica_cj_marcacao([Porta|RestoCjEnt],Marcacao):-
        encontra_lugar(Porta,Marcacao,Lugar),
        unifica_marca(Porta,Lugar),
        unifica_cj_marcacao(RestoCjEnt,Marcacao).



encontra_lugar([NomeLugar,Termos],[[NomeLugar,MarcLugar]|RestoMarc],
                                    [NomeLugar,MarcLugar]).

encontra_lugar(Porta,[_|RestoMarc],Lugar):-
        encontra_lugar(Porta,RestoMarc,Lugar).
```

```
unifica_marca([NomeLugar,[Marca|Resto]],[NomeLugar,[Marca|RestoMarcLugar]]).

unifica_marca(Lugar,[NomeLugar,[Marca|RestoMarcLugar]]):-
        unifica_marca(Lugar,[NomeLugar,RestoMarcLugar]).



alteracao_habilitada([CjSaiAlt,CjSaiRes],Marcacao):-
        ifthen(CjSaiAlt \== [],
               not(unifica_cj_marcacao(CjSaiAlt,Marcacao)) ),
        ifthen(CjSaiRes \== [],
               not(unifica_cj_marcacao(CjSaiRes,Marcacao)) ).



ocorre_alter(Conex,nodo(LIdent,Tipo,Heur_h,NAlter,_,Marcacao),CjEnt,CjEntAlt,CjSaiAlt):-
        remove(CjLntAlt,Marcacao,NovaMarcW),
        append(CjSaiAlt,NovaMarcW,NovaMarc),
        L!marcacao_valida(NovaMarc)!],
        ctr_inc(0,Ident),          /*  num. de nodos filhos gerados  */
        [!  funcao_componente_h(NovaMarc,Heur_hNM)  !],
        [!concatena(LIdent,[Ident],NLIdent)!],
        inc(NAlter,NNAlter),
        Alteracao = [Conex,CjEnt],
        NodoGerado = nodo(NLIdent,[fronteira],Heur_hNM,NNAlter,Alteracao,NovaMarc),
        [!ifthenelse(marc_dupl(NovaMarc,Heur_hNM,NodoDuplo),
                     duplicado(NodoGerado,NodoDuplo),
                     grava_fronteira(NodoGerado))!],
        verifica_objetivo(Heur_hNM),
        cls,write('Sequencia de Alteracoes'),nl,
        mostra_alcanc(NLIdent),
        limpa_memoria.
```

```
remove([],Marcacao,Marcacao).

remove([Porta|RestoCjEnt],Marcacao,NovaMarc):-
        remove(RestoCjEnt,Marcacao,NovaMarc1),
        remove_lugar(Porta,NovaMarc1,NovaMarc).



remove_lugar([NomeLugar,Termos],[[NomeLugar,MarcLugar]|RestoMarc],[[NomeLugar,NovaMarcLugar]|RestoMarc]):-
        remove_marcacao(Termos,MarcLugar,NovaMarcLugar).

remove_lugar(Porta,[Lugar|RestoMarc],[Lugar|RestoNM]):-
        remove_lugar(Porta,RestoMarc,RestoNM).



remove_marcacao([],MarcLugar,MarcLugar).

remove_marcacao([Termo|RestoTermos],MarcLugar,NovaMarcLugar):-
        remove_marcacao(RestoTermos,MarcLugar,NovaMarcLugar1),
        remove_marca(Termo,NovaMarcLugar1,NovaMarcLugar).


remove_marca(Marca,[Marca|Resto],Resto).
remove_marca(Marca,[Marca1|Resto1],[Marca1|Resto2]):-
        remove_marca(Marca,Resto1,Resto2).



append([],Marcacao,Marcacao).

append([Porta|RestoCjSai],Marcacao,NovaMarc):-
        append(RestoCjSai,Marcacao,NovaMarc1),
        append_lugar(Porta,NovaMarc1,NovaMarc).
```

```
append_lugar([NomeLugar,Termos],[[NomeLugar,MarcLugar]|RestoMarc],[[NomeLugar,NovaMarcLugar]|RestoMarc]):-
        concatena(Termos,MarcLugar,NovaMarcLugar).

append_lugar(Porta,[Lugar|RestoMarc],[Lugar|RestoNM]):-
        append_lugar(Porta,RestoMarc,RestoNM).


concatena([],Lista,Lista).

concatena([Elem|RestoLista],Lista,[Elem|RestoNovaLista]):-
        concatena(RestoLista,Lista,RestoNovaLista).


marcacao_valida([]).

marcacao_valida([[Lugar,MarcLugar]|RestoMarc]):-
        !,
        length(MarcLugar,NumMarcas),
        capacidade(Lugar,Capacidade),
        NumMarcas =< Capacidade,
        marcacao_valida(RestoMarc).


marc_dupl(Marcacao,Heur_h,Nodo):-
        retrieveb(todosnodos,Heur_h,Refnum),
        instance(Refnum,Nodo),
        not(arg(2,Nodo,[duplicado,_])),
        arg(6,Nodo,MarcN),
        iguais(Marcacao,MarcN).


iguais(Marcacao,MarcN):-
        [!  analisa_difer(Marcacao,MarcN,0)  !].
```

```
duplicado(NodoDuplo,NodoGerado):-
        ifthenelse(arg(2,NodoDuplo,[terminal]),
                terminal(NodoGerado),
                nao_terminal(NodoDuplo,NodoGerado)).


terminal(NodoGerado):-
        argrep(NodoGerado,2,[terminal],NodoTerm),
        recordz(nodo,NodTerm,Refnum),
        arg(3,NodoTerm,Heur_h),
        recordb(todosnodos,Heur_h,Refnum),
        arg(1,NodoTerm,LIdent),
        recordb(arvorealc,LIdent,Refnum).


nao_terminal(NodoDuplo,NodoGerado):-
        arg(4,NodoDuplo,NAlterD),
        arg(4,NodoGerado,NAlterG),
        arg(1,NodoDuplo,LIdent),
        ifthenelse(NAlterG >= NAlterD,
                nodo_gerado_duplo(NodoGerado,LIdent),
                nodo_duplo_duplo(NodoDuplo,NodoGerado)).


nodo_gerado_duplo(NodoGerado,LIdentD):-
        argrep(NodoGerado,2,[duplicado,LIdentD],NodoDuplicado),
        recordz(nodo,NodoDuplicado,Refnum),
        arg(3,NodoDuplicado,Heur_h),
        recordb(todosnodos,Heur_h,Refnum),
        arg(1,NodoDuplicado,LIdent),
        recordb(arvorealc,LIdent,Refnum).
```

```
nodo_duplo_duplo(NodoDuplo,NodoGerado):-
        grava_fronteira(NodoGerado),
        ifthen(arg(2,NodoDuplo,[fronteria]),
              retira_front(NodoDuplo)),
        arg(1,NodoGerado,LIdent),
        altera_tipo_nodo(NodoDuplo,[duplicado,LIdent]).


retira_front(Nodo):-
        arg(1,Nodo,LIdent),
        arg(3,Nodo,Heur_h),
        arg(4,Nodo,Heur_g),
        constante_heuristica(K),
        Heur_f is Heur_h + (Heur_g * K),
        retriveb(arvorealc,LIdent,Refnum),
        removeb(fronteira,Heur_f,Refnum).


mostra_alcanc(LIdent):-
        monta_lista_alter(ListaAlter,LIdent),
        escreve_lista(ListaAlter).


monta_lista_alter([],[]:[]]).

monta_lista_alter(ListaAlter,LIdent):-
       retrieveb(arvorealc,LIdent,Refnum),
       instance(Refnum,Nodo),
       ifthenelse(arg(2,Nodo,[duplicado,LIdentDupl]),
                 monta_lista_alter(ListaAlter,LIdentDupl),
                 (   arg(5,Nodo,Alteracao),
                     del_ult_elem(LIdent,NLIdent),
                     monta_lista_alter(NListaAlter,NLIdent),
                     concatena(NListaAlter,[Alteracao],ListaAlter) )).
```

```
del_ult_elem([],[]).

del_ult_elem([L],[]).

del_ult_elem([P|R1],[P|R2]):-
        del_ult_elem(R1,R2).



escreve_lista([]).

escreve_lista([P|Resto]):-
        write(P),nl,
        escreve_lista(Resto).



/*-------------------------------------------------------------------------------

              Verificação de Conflito

-------------------------------------------------------------------------------*/


conflito(Conex1,Conex2,Marcacao):-
        [!atualiza_conhecimento_usuario!],
        [!marcacao_inicial_valida(Marcacao)!],

        encontra_alter_habil(Conex1,CjEntAlt1,CjEntRes1,
                                CjSaiAlt1,CjSaiRes1,Marcacao),
        encontra_alter_habil(Conex2,CjEntAlt2,CjEntRes2,
                                CjSaiAlt2,CjSaiRes2,Marcacao),

        ifthen( not(mesmo_passo(Marcacao,CjEntAlt1,CjSaiAlt1,
                                        CjEntAlt2,CjSaiAlt2)),
                (nl,write(Conex1),write(CjEntAlt1),write(' x '),
                   write(Conex2),write(CjEntAlt2)) ),
        fail.
```

```
mesmo_passo(Marcacao,CjEnt1,CjSai1,CjEnt2,CjSai2):-
        remove(CjEnt1,Marcacao,M1),
        remove(CjEnt2,M1,M2),
        append(CjSai1,M2,M3),
        append(CjSai2,M3,NovaMarc),
        marcacao_valida(NovaMarc).


/*-------------------------------------------------------------------------------

            Verificação de Concorrência

-------------------------------------------------------------------------------*/


concorrencia(Conex1,Conex2,Marcacao):-
       [!atualiza_conhecimento_usuario!],
       [!marcacao_inicial_valida(Marcacao)!],

       encontra_alter_habil(Conex1,CjEntAlt1,CjEntRes1,CjSaiAlt1,CjSaiRes1,Marcacao),
       encontra_alter_habil(Conex2,CjEntAlt2,CjEntRes2,CjSaiAlt2,CjSaiRes2,Marcacao),

       ifthen( mesmo_passo(Marcacao,CjEntAlt1,CjSaiAlt1,CjEntAlt2,CjSaiAlt2),
              (nl,write(Conex1),write(CjEntAlt1),write(' x '),
                 write(Conex2),write(CjEntAlt2)) ),
       fail.
```

```
/*-------------------------------------------------------------------------------

            Verificação de Bloqueio

-------------------------------------------------------------------------------*/


bloqueio(Marcacao):-
        [!atualiza_conhecimento_usuario!],
        [!marcacao_inicial_valida(Marcacao)!],

        encontra_alter_habil(Conex,CjEntAlt,CjEntRes,CjSaiAlt,CjSaiRes,Marcacao),

        remove(CjEntAlt,Marcacao,NovaMarcW),
        append(CjSaiAlt,NovaMarcW,NovaMarc),

        [!marcacao_valida(NovaMarc)!],
        nl,write('Nao e'' bloqueio. Foi encontrada a alteracao: '),
        nl,write(Conex),write(CjEntAlt).


bloqueio(Marcacao):-
        nl,write('A marcacao e'' um ponto de bloqueio.').

/*-------------------------------------------------------------------------------*/
```

Apêndice B: Índice de Referência aos Autores

Neste apêndice estão relacionados os primeiros autores das obras referênciadas, em ordem alfabética, e as páginas onde são referenciados.

BIBLIOGRAFIA

[ARI 86] ARITY CORPORATION. **IV The Arity/Prolog Programming Language.** Concord, 1986.

[BAR 81] BARR, A.; FEIGENBAUM, E.A. (Ed.) **The Handbook of Artificial Inteligence.** Massachusetts, Addison-Wesley, 1981. v.1.

[BER 82] BERG, H.K.; BOEBERT, W.E.; FRANTA, W.R.; MOHER, T.G. **Formal Methods of Verification and Specification.** Englewood Cliffs, Prentice-Hall, 1982.

[BRA 86] BRATKO, I. **Prolog programming for Artificial Inteligence.** Massachusetts, Addison-Wesley, 1986.

[CAR 88] CARNOTA, R.; TESZKIEWICZ, A. **Sistemas Expertos y Representacion del Conocimiento.** Curitiba, III EBAI - Escola Brasileiro-Argentina de Informática, 1988.

[EMS 70] EMSHOFF, J.R.; SISSON, R.L. **Design and Use of Computer Simulation Models.** New York, Macmillan, 1970.

[FRA 77] FRANTA, W.R. **The Process View of Simulation.** New York, North-Holland, 1977.

[FUR 73] FURTADO, A.L. **Teoria dos Grafos: Algoritmos.** Rio de Janeiro, Livros Tècnicos e Cientificos, 1973.

[GAI 86] GAINES, B.R. From Differential Analyzers to Knowledg-Based Systems: The Changing Technologies of Simulation. IN: JSST CONFERENCE ON RECENT ADVANCES IN SIMULATION OF COMPLEX SYSTEMS, Tokio, July 15-17, 1986. **Proceedings**. Tokio, Japan Society for Simulation Technology, 1986. p.271-276.

[GOL 80] GOLUMBIC, M.C. **Algorithmic Graph Theory and Perfect Graphs**. New York, Academic Press, 1980.

[HAY 83] HAYES-ROTH, F.; WATERMAN, D.A.; LENAT, D.B. (Ed.) **Building expert systems**. Massachusetts, Addison-Wesley, 1983.

[HEU 89] HEUSER, C.A. **Modelagem Conceitual de Sistemas**. Santiago del Estero, IV EBAI - Escola Brasileiro-Argentina de Informática, 1989.

[HUB 86] HUBER, P.; JENSEN A.M.; JEPSEN L.O.; JENSEN K. Reachability trees for High-Level Petri Nets. **Theoretical Computer Science**, Amsterdam, 45(3):261-292, 1986.

[KVI 88] KVITA, A.M. **Resolucion de problemas con Inteligencia Artificial**. Curitiba, III EBAI - Escola Brasileiro-Argentina de Informática, 1988.

[KOW 79] KOWALSKI, R. **Logic for Problem Solving**. New York, North-Holland, 1979.

[MEL 89] MELO, W.L.M. **Proposta de um Editor Diagramàtico Generalizado.** Porto Alegre, PGCC da UFRGS, 1989.

[MEN 89] MENNABARRETO, R.; CECCATO, L. **SQ1: Editor Analisador de Modelos Q1.** Porto Alegre, PUC/RS, 1989.

[NIE 86] NIEHUIS, S.; VICTOR, F. **Modellierung und Simulation von Pr/T - Netzen in Prolog.** Bereich, GMD, 1986.

[NIL 71] NILSSON, N.J. **Problem-Solving Methods in Artificial Intelligence.** New York, McGraw-Hill, 1971.

[OBE 87] OBERWEIS, A.; SCHONTHALER, F.; SEIB, J.; LAUSEN, G. Database Supported Analyses Toll for Predicate/Transition Nets. **Petri Net Newsletter**, Bonn, 28, p.21-23, Dec. 1987.

[OLI 86] OLIVEIRA, F.M. **SISREDE: Editor Gràfico para Redes de Petri.** Porto Alegre, PGCC da UFRGS, 1986.

[PER 89] PERES, E.M. **Integrando Aspectos Formais e Informais em uma Linguagem de Anotação de Redes de Petri.** Porto Alegre, CPGCC da UFRGS, 1989.

[PET 81] PETERSON, J.L. **Petri net theory and the modeling of systems.** Englewood Cliffs, Prentice-Hall, 1981.

[REI 86] REISIG, W. **Petri Nets An Introduction.** Berlin, Springer-Verlag, 1986.

[SHN 87] SHNEIDERMAN, B. **Designing the User Interface: Strategies for Effective Human-Computer Interaction.** Massachusetts, Addison-Wesley, 1987.

[TUR 87] TURSKI, W.M.; MAIBAUM, T.S.E. **The Specification of Computer Programs.** Massachusetts, Addison-Wesley, 1987.

OUTRAS FONTES

[HEU 88] HEUSER, C.A. **Modelagem de Sistemas com Redes de Petri**, 1988. (apostila)