UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PÓS-GRADUAÇÃO EM CIENCIA DA COMPUTAÇÃO

UM PROTOTIPO DE SISTEMA ESPECIALISTA PARA PROJETO LOGICO DE BLOCOS OPERACIONAIS DE CIRCUITOS DIGITAIS

por

SANDRA LUZIA CORTINOVI

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

Prof. Flávio Rech Wagner
Orientador

Prof. Antônio Carlos da Rocha Costa
Co-orientador

Porto Alegre, dezembro de 1986.

CATALOGAÇÃO NA FONTE

CORTINOVI, SANDRA LUZIA

Um Protótipo de sistema especialista para projeto lógico de blocos operacionais de circuitos digitais. Porto Alegre, PGCC da UFRGS, 1986.
1 v.
Diss. (mestr. ci. comp.) UFRGS-PGCC, Porto Alegre, BR-RS, 1986.

Dissertação: Inteligencia Artificial
Sistemas Especialistas: Projeto de circuitos
Linguagem: Prolog

AGRADECIMENTOS

A Universidade Federal do Rio Grande do Sul, através do seu Centro de Processamento de Dados, pelo apoio recebido enquanto aluna do Curso de Pós-Graduação em Ciência da Computação e durante o desenvolvimento dessa dissertação.

Ao meu colega de trabalho e amigo Roberto Porto Löw, que muito me ensinou e colaborou para a realização desse trabalho.

Ao Prof. Neron Arruda Leonel, pelo incentivo e interesse demonstrado.

Aos meus orientadores e a todos os que não mediram esforços para que este trabalho pudesse ser realizado da melhor forma possível.

SUMARIO

LISTA DE ABREVIATURAS 9

LISTA DE SINAIS 10

LISTA DE FIGURAS 11

LISTA DE TABELAS 14

RESUMO 15

ABSTRACT 16

1 INTRODUÇÃO 17

1.1 Organização do texto 18

2 SISTEMAS ESPECIALISTAS: UMA VISÃO GERAL 20

2.1 Evolução dos enfoques de pesquisa 20

2.2 Requisitos de um sistema especialista 22

2.3 O problema da representação do conhecimento 24

2.3.1 Regras de produção 25

2.3.2 Redes semânticas ou associativas 25

2.3.3 Quadros 26

2.3.4 Lógica formal 26

2.4 Aquisição de conhecimento 27

2.4.1 Aprendizado através do que é dito 28

2.4.2 Aprendizado induzido por exemplos 30

2.4.3 Aprendizado por observação e descoberta 30

2.5 Prolog e programação em lógica 31

3 O PROBLEMA: PROJETO DE CIRCUITOS DIGITAIS 34

3.1 Níveis de projeto 34

3.2 Características do nível RT 36

3.3 Formas de descrição no nível RT 36

3.3.1 Descrição comportamental 37

3.3.2 Descrição funcional 38

3.3.3 Descrição estrutural 38

3.4 O PROTO no contexto de projeto de circuitos digitais ... 38

4 DESCRIÇÃO GERAL DO PROTO 40

4.1 O método de solução de problemas por transformação de redes ... 40

4.2 Arquitetura 43

4.2.1 Arquitetura do subsistema de projeto lógico 43

4.2.1.1 Módulo de controle 43

4.2.1.2 Modulo compilador da descrição algorítmica do circuito ... 45

4.2.1.3 Módulo interpretador de regras 45

4.2.1.4 Módulo compilador de estilos 46

4.2.1.5 Rotinas de mapeamento direto 47

4.2.2 Arquitetura do subsistema de aquisição de conhecimento ... 47

4.2.2.1 Módulo de controle 49

4.2.2.2 Módulo de construção de bases de conhecimento 49

4.2.2.3 Módulo de edição de bases de conhecimento 50

4.3 Organização do conhecimento 51

4.3.1 Identificação da base 51

4.3.2 Regras 52
4.3.3 Estilos 53
4.3.4 Elementos 54
4.3.5 Relações 57
4.3.6 Marcas 59
4.3.7 Testes 60

5 A TAREFA: PROJETO LOGICO DE BLOCOS OPERACIONAIS 62
5.1 Controle de sessão 62
5.2 Opção a: construção de rede comportamental 63
5.2.1 A descrição do circuito 64
5.2.2 A rede comportamental 70
5.2.3 O processo de construção da rede comportamental .. 74
5.2.3.1 Declarações 74
5.2.3.1.1 Portadores de entrada 74
5.2.3.1.2 Portadores de saída 75
5.2.3.1.3 Portadores permanentes 75
5.2.3.2 Comandos 76
5.2.3.2.1 Comando de ligação 76
5.2.3.2.2 Comando de atribuição 78
5.2.3.2.3 Comando se 80
5.2.3.2.4 Comando enquanto 82
5.2.3.2.5 Bloco de comandos 83
5.2.3.3 Separação da rede comportamental 85
5.2.3.4 Mensagens de erro 88
5.3 Opção b: construção de rede estrutural através do interpretador de regras 88
5.3.1 A rede estrutural 90

5.3.2 O processo de construção da rede estrutural 93
5.3.2.1 Interpretação das regras de construção de uma rede estrutural básica 95
5.3.2.1.1 Regra: map_permanentes 96
5.3.2.1.2 Regra: map_constantes 96
5.3.2.1.3 Regra: map_entradas 97
5.3.2.1.4 Regra: map_ligadores_entrada .. 97
5.3.2.1.5 Regra: map_extratores 98
5.3.2.1.6 Regra: map_operadores 98
5.3.2.1.7 Regra: map_modif_simples 99
5.3.2.1.8 Regra: map_modif_multiplo1 ... 100
5.3.2.1.9 Regra: map_modif_multiplo2 ... 101
5.3.2.1.10 Regra: map_saidas 101
5.3.2.2 A rede resultante 102
5.3.3 Monitoração da execução das regras 102
5.4 Opção c: construção de rede estrutural através de rotina de mapeamento direto 108
5.5 Opção d: compilação de rotina de mapeamento direto 110
5.6 Opção e: projeto lógico através do interpretador de regras 111
5.7 Opção f: projeto lógico através de rotina de mapeamento direto 114
6 AQUISIÇAO DE CONHECIMENTO: CONSTRUÇAO E EDIÇAO DE BASES DE CONHECIMENTO 118
6.1 Controle de sessão 118
6.2 Opção a: construção de base de conhecimento 119
6.2.1 A descrição da base 119

6.2.2 O processo de construção de base de conhecimento 133

6.2.2.1 Identificação da base 133

6.2.2.2 Identificação de itens de conhecimento estruturais 134

6.2.2.3 Geração final da base 135

6.2.2.4 Mensagens de erro 136

6.2.3 A base de conhecimento resultante 138

6.3 Opção b: edição de base de conhecimento 150

6.3.1 Consultas à base de conhecimento 151

6.3.2 Inclusão de itens de conhecimento 153

6.3.3 Exclusão de itens de conhecimento 154

6.3.4 Alteração de itens de conhecimento 155

6.3.5 Exemplo de edição 157

7 CONCLUSÃO 165

ANEXO : Listagem do interpretador de regras 167

BIBLIOGRAFIA 175

LISTA DE ABREVIATURAS

BC	Base de Conhecimento
CPD	Centro de Processamento de Dados
Fig.	Figura
Nível RT	Nível de Transferência entre Registradores
Rede-TR	Rede de Transformação de Representações
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE SINAIS

Sinal	Significado
=	Precedência igual
>	Precedência maior
<	Precedência menor
>=	Precedência maior ou igual
=<	Precedência igual ou menor
+	Sinal de adição
-	Sinal de subtração
*	Sinal de multiplicação
/	Sinal de divisão

LISTA DE FIGURAS

Figura 4.1 Exemplo de aplicação do método de transformação de redes 42
Figura 4.2 Arquitetura do subsistema de projeto lógico 44
Figura 4.3 Arquitetura do subsistema de aquisição de conhecimento 48
Figura 5.1 Algoritmo de um circuito que realiza divisão de inteiros positivos através de subtrações sucessivas 65
Figura 5.2 Modelos em rede comportamental de alguns componentes de circuitos lógicos 73
Figura 5.3 Exemplo de compilação de declarações de entradas, saídas e registradores de um circuito 75
Figura 5.4 Exemplos de compilação de comandos de ligação 77
Figura 5.5 Representação comportamental do comando de atribuição 79
Figura 5.6 Exemplos de compilação de comandos de atribuição 79
Figura 5.7 Representação comportamental do comando se 80
Figura 5.8 Exemplos de compilação de comandos se 81
Figura 5.9 Exemplo de compilação de comando enquanto 82
Figura 5.10 Representação comportamental do comando enquanto 83
Figura 5.11 Exemplo de compilação de um bloco de comandos 84
Figura 5.12 Representação comportamental de bloco de comandos 84

Figura 5.13 Rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1: divisão de inteiros positivos através de subtrações sucessivas 87
Figura 5.14 Modelos em rede comportamental e estrutural de alguns componentes de circuitos lógicos 91
Figura 5.15 O processo de construção da rede estrutural (final ou parcial): bloco operacional estrutural + bloco de controle comportamental 94
Figura 5.16 Regra: map_permanentes 96
Figura 5.17 Regra: map_constantes 97
Figura 5.18 Regra: map_entradas 97
Figura 5.19 Regra: map_ligadores_entrada 98
Figura 5.20 Regra: map_extratores 98
Figura 5.21 Regra: map_operadores 99
Figura 5.22 Regra: map_modif_simples 99
Figura 5.23 Regra: map_modif_multiplo1 100
Figura 5.24 Regra: map_modif_multiplo2 101
Figura 5.25 Regra: map_saidas 101
Figura 5.26 Rede final PROTO/ESTRUTURAL/DIVISAO1,gerada no estilo basico a partir da rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1 103
Figura 5.27 Rede PROTO/ESTRUTURAL/DIVISAO1/2, gerada a partir da rede PROTO/COMPORTAMENTAL/DIVISAO1, estilo estendido 109
Figura 5.28 Rede comportamental PROTO/COMPORTAMENTAL/MULT1: multiplicação de inteiros através de somas sucessivas 113

Figura 5.29 Rede final PROTO/ESTRUTURAL/MULT1: gerada no estilo basico a partir da rede comportamental PROTO/COMPORTAMENTAL/MULT1 114

Figura 5.30 Rede final PROTO/ESTRUTURAL/MULT1/2: gerada no estilo estendido a partir da rede comportamental PROTO/COMPORTAMENTAL/MULT1 117

LISTA DE TABELAS

TABELA 5.1 Desempenho comparativo dos processos de execução das regras 116

TABELA 5.2 Desempenhos na tarefa de construção de rede estrutural 116

RESUMO

O PROTO é um protótipo de sistema especialista voltado para a solução de problemas que possam ser resolvidos pelo método de transformação de redes. A aplicação que motivou o seu desenvolvimento foi o projeto lógico de blocos operacionais de circuitos digitais no nível RT.

O PROTO compõe-se de dois subsistemas: um subsistema de projeto lógico, que gera uma rede estrutural do bloco operacional de um circuito a partir de sua descrição algorítmica-comportamental; e um subsistema de aquisição de conhecimento, que permite a construção, exame e modificação das bases de conhecimento do sistema.

ABSTRACT

PROTO is the prototype of a rule-based system oriented for solving problems that can be soved by net transformation method. The application it was developed for is the logical design of digital circuits′ data path at the RT-level.

PROTO is composed by two subsystems: the first one generates the structural net describing the circuit′s data path, on the basis of its behavioral algorithmic specification. The second one is a knowledge acquisition subsystem, that allows the creation, examination and changing of the knowledge base.

1. INTRODUÇÃO

Nos últimos anos tem havido uma atividade muito intensa na área de Inteligência Artificial, notadamente no que tange ao campo de Sistemas Especialistas. Este busca o desenvolvimento de sistemas altamente complexos, que incorporem experiência e processos de tomada de decisão de especialistas na solução de problemas computacionais não passíveis de descrições simples. Em algumas áreas de conhecimento, como a Medicina, a Química, a Matemática Simbólica, a Análise Eletrônica e a Engenharia de Software, sistemas especialistas foram e são desenvolvidos com sucesso, o que mostra a viabilidade deste campo de pesquisa.

O modo clássico de abordar problemas computacionais consiste na definição precisa de algoritmos solucionadores. Os sistemas especialistas trazem um enfoque completamente distinto. Procuram relacionar um conjunto de informações relativas ao problema, o conhecimento sobre determinada área, com o mecanismo dedutivo que permite utilizá-las na solução do problema.

Este trabalho apresenta o PROTO, que é um protótipo de sistema especialista voltado para a solução de problemas passíveis de serem resolvidos através de um método de transformação de redes /COS 84/. Neste método não determinístico uma rede é gerada através da aplicação de regras de transformação de redes sobre uma rede inicial. Este método possui a característica especial de dispensar retrocessos.

A aplicação que motivou o desenvolvimento do PROTO é o

projeto lógico de blocos operacionais de circuitos digitais no nível RT. Dentro desse contexto, o PROTO engloba os seguintes subsistemas: um subsistema de projeto lógico, que utiliza o método de transformação de redes, apoiado por uma base de conhecimento, para gerar a rede estrutural do bloco operacional de um circuito a partir da descrição algorítmica do seu comportamento; e um subsistema de aquisição de conhecimento, que possibilita a construção, exame e modificação das bases de conhecimento.

O PROTO foi desenvolvido na linguagem Prolog existente no equipamento Burroughs B6700 do CPD da UFRGS /LOW 85/. Atualmente encontra-se implementado no equipamento Burroughs A10F dessa instituição.

1.1 Organização do texto

O presente texto está dividido em sete capítulos, sendo esta introdução considerada como o primeiro.

O segundo capítulo apresenta uma introdução ao campo de Sistemas Especialistas, mencionando também algumas características da linguagem Prolog.

O terceiro capítulo introduz alguns conceitos relativos ao problema do projeto de circuitos digitais, com particular atenção ao nível RT.

O quarto capítulo descreve o método de solução de problemas por transformação de redes, a arquitetura do PROTO e a representação do conhecimento adotada.

O quinto capítulo detalha o subsistema de projeto lógico, explicando cada um de seus módulos.

O sexto capítulo apresenta em detalhe o subsistema de aquisição de conhecimento, fornecendo exemplos de construção e de manuseio de uma base de conhecimento.

Finalmente, o sétimo capítulo contém algumas conclusões e avaliações.

2. SISTEMAS ESPECIALISTAS: UMA VISÃO GERAL

M. L. Minsky define Inteligência Artificial como a "ciência de desenvolver máquinas que façam coisas que poderiam exigir inteligência se realizadas pelo homem" /BOD 77/. Esta definição não implica que uma máquina terá inteligência por si própria (a habilidade de aprender, raciocinar e compreender), mas somente que ela pode exibir comportamentos que pareçam ser orientados por inteligência. A distinção é importante, pois aponta diretamente para os tipos de pesquisa que estão sendo desenvolvidas na área de Inteligência Artificial. O objetivo não é criar máquinas que possam pensar, e sim usar máquinas para modelar o pensamento. A orientação está na tentativa de definição de sistemas computacionais que imitem eficientemente o raciocínio humano.

Sistemas especialistas, também chamados de sistemas peritos ou sistemas de conhecimento, constituem um dos ramos de pesquisa da área de Inteligência Artificial. Seu objetivo é desenvolver sistemas de desempenho elevado que incorporem conhecimento, experiência e processos de tomada de decisão de especialistas numa dada área profissional. Eles processam conhecimento e tentam imitar algumas formas de raciocínio humano na solução dos problemas. O suporte para processar conhecimento encontra-se em bases de conhecimento que contém fatos, idéias, relações lógicas e regras relativas a uma área de especialização.

2.1 Evolução dos enfoques de pesquisa

A idéia que orientou os primeiros trabalhos no campo de

sistemas de Inteligência Artificial baseava em mecanismos gerais os procedimentos de solução de problemas implementados nos sistemas. Esta abordagem é representada pela "Máquina da Teoria Lógica" /NEW 63/, pelo "Solucionador Geral de Problemas" /NEW 63a/ e pelos primeiros trabalhos em Prova Automática de Teoremas /GRE 69/. Entretanto, este enfoque não abriu um espaço para o conhecimento de como realizar a tarefa eficientemente, tendo como conseqüência sistemas demasiadamente lentos, que ainda hoje não podem ser suportados eficientemente pelos computadores correntes.

A seguir surgiu o enfoque no qual se deveria coletar conhecimento declarativo (sobre a natureza da tarefa) e "procedural" (sobre como realizá-la) para uma determinada tarefa e se estar pronto para escrever um sistema especialista para cada problema específico /FEI 83/. Esta abordagem teve o mérito de permitir a construção dos primeiros sistemas peritos viáveis, tendo sido a principal responsável pelo interesse comercial atual no campo de sistemas baseados em conhecimento. Entretanto, exige um trabalho muito intenso por parte dos profissionais especialistas na tarefa e do engenheiro de conhecimento, cuja função é coletar e codificar a experiência desses profissionais.

Nos últimos anos a ênfase da pesquisa tem recaído sobre as ferramentas de apoio à construção de sistemas especialistas. Essas vão desde linguagens de propósitos gerais como LISP, Prolog e OPS5 /FOR 81/, até sistemas altamente especializados, como o Emicyn /BUI 83/. Esses últimos são chamados "shells" de

sistemas especialistas. Cada um deles cobre uma classe de sistemas, mas nenhum é tão geral quanto os solucionadores gerais de problemas propostos há quase duas décadas. Juntos esses "shells" suportam uma grande variedade de tarefas, de tal forma que podem ser agrupados em "kits de ferramentas" para sistemas peritos: sistemas de propósitos gerais para execução e construção de sistemas especialistas, como o Expert System Environment/VM /HIR 86/ e o KEE /FIK 85/. Embora a maioria desses sistemas ainda deva ser preparada por engenheiros de conhecimento, o tempo dispendido na construção dos mesmos é reduzido.

2.2 Requisitos de um sistema especialista

Espera-se que um sistema especialista atue como um assistente inteligente em algumas tarefas ou solucione problemas de uma dada área profissional que poderiam ser solucionados por um especialista humano.

Como o conhecimento especializado pode alterar-se com o tempo, deseja-se que um sistema perito seja flexível na integração de novo conhecimento ao conhecimento existente, apoiando, portanto, a tranferência de conhecimento. Deseja-se também que ele seja capaz de fornecer seu conhecimento numa forma fácil de ser lida.

Caso o sistema seja usado como apoio à tomada de decisões, espera-se que possua a habilidade de fornecer explicações adequadas sobre suas respostas.

Devido ao fato do conhecimento humano especializado ser

freqüentemente incompleto ou parcialmente compreensível, um sistema especialista deve ser capaz de raciocinar com conhecimento inexato. Este conhecimento pode ser declarativo, "procedural" ou ambos.

Como, em geral, deverá ser usado por utilizadores ocasionais, um sistema especialista deve ser capaz de manusear sentenças simples em alguma língua natural.

Em resumo, um sistema especialista deve ser capaz de:

a) solucionar problemas que poderiam ser resolvidos por um especialista humano;

b) ser flexível na integração de novo conhecimento ao conhecimento existente;

c) assistir a elucidação, estruturação e transferência do conhecimento;

d) fornecer o seu conhecimento numa forma fácil de ser lida;

e) fornecer explicações a respeito de suas soluções;

f) raciocinar com conhecimento deduzido ou inexato sobre a natureza da tarefa ou como realizá-la eficientemente; e

g) manusear sentenças simples em alguma língua natural.

Os principais componentes encontrados na maioria dos sistemas especialistas são:

a) conhecimento na forma de fatos e regras;

b) um mecanismo de inferência para raciocínio com fatos e regras;

c) um gerador de explicações;

d) um mecanismo de aquisição de conhecimento; e

e) um processador de língua natural, normalmente limitado.

2.3 O problema da representação do conhecimento

Caso se deseje usar conhecimento em sistemas computacionais deve-se primeiro escolher uma forma de representá-lo. Segundo Richard Duda e Edward Shortliffe /DUD 83/ um formalismo ideal para representação de conhecimento deve:

a) representar os conceitos e intenções do especialista fielmente;

b) estar apto para ser interpretado pelo programa (solucionador do problema) correta e efetivamente;

c) suportar explicações que exprimam uma linha de raciocínio que um observador possa entender e criticar;

d) facilitar o processo de descoberta de lacunas e erros na base de conhecimento; e

e) permitir a separação entre domínio do conhecimento e interpretador de tal forma que a base de conhecimento possa ser

ampliada ou corrigida sem a necessidade de alterações nesse último.

Estes critérios podem colocar necessidades conflitantes no projeto do sistema. Os dois primeiros levam a representações complexas específicas para cada situação, enquanto os demais sugerem um formalismo único e uniforme, fácil de ser interpretado.

Os principais modelos de representação de conhecimento são regras de produção, redes semânticas ou associativas, quadros ("frames") e a lógica formal.

2.3.1 Regras de produção

Permitem representar situações ou condições que ajudam a definir a solução de problemas através de indicação de transformações no conjunto de informações armazenadas relativas ao problema. Os sistemas Emycin /BUI 83/ e Sylog /WAL 84/ utilizam este modelo.

2.3.2 Redes semânticas ou associativas

Permitem representar o conhecimento através de modelos que são formulados como grafos, com os nodos representando objetos e/ou conceitos e as arestas representando suas relações. No sistema Prospector /REB 81/ o conhecimento é descrito dessa forma, sendo que os nodos representam evidências ou hipóteses geológicas e os arcos representam ligações causais entre os

nodos.

Neste tipo de organização é possível testar facilmente se as informações a respeito de um indivíduo conhecido são novas, redundantes, inconsistentes ou deriváveis a partir de relações prévias.

2.3.3 Quadros

Permitem estender as redes semânticas através da adição de atributos fixos e/ou variáveis.

Utiliza a noção de hierarquia, na qual normalmente as propriedades dos níveis superiores são inerentes aos níveis inferiores. Nos quadros os níveis superiores são fixos e representam informações que são sempre verdadeiras sobre as situações que estão representando. Os níveis inferiores são formados por terminais que podem ser preenchidos pelos nodos. Pode-se pensar num quadro como semelhante a uma forma a ser preenchida, a qual pode ter um lugar numa hierarquia de formas.

2.3.4 Lógica formal

Utiliza a linguagem matemática do cálculo de predicados de primeira ordem. O método lógico representa o conhecimento através de sentenças lógicas.

Para pesquisadores como Green /GRE 69/, Colmerauer et alli /COL 73/ e Kowalski /KOW 79/ este método fornece boas representações para o tipo de conhecimento necessário à construção de

sistemas coloquiais /COE 80/. Além disso, sugerem que as regras de inferência e os axiomas lógicos desses sistemas sejam implementados como programas, e que as propriedades, as limitações e o desempenho desses programas sejam analisados por técnicas da lógica.

Embora as diversas notações apresentadas anteriormente (regras de produção, redes semânticas e quadros), tenham sido projetadas para situações distintas, há evidências crescentes que cada uma pode ser escrita através da lógica formal /GUE 86/. Disso se conclui que, desde que possa ser suportada eficientemente, a lógica tende a ser a notação comum para representação de conhecimento.

2.4 Aquisição de conhecimento

A identificação e codificação do conhecimento é uma das atividades mais árduas e complexas na construção de sistemas especialistas. Tentativas de construção de bases de conhecimento freqüentemente levam à descoberta de lacunas no domínio do assunto e falhas nas técnicas de representação disponíveis. Mesmo quando um formalismo adequado de representação do conhecimento é utilizado, os profissionais especialistas freqüentemente tem dificuldade de escrever o seu conhecimento. Portanto, o processo de construção de bases de conhecimento exige interações entre especialistas e engenheiros de conhecimento que consomem períodos consideráveis de tempo. Enquanto uma equipe experiente pode desenvolver um pequeno protótipo de sistema em poucos meses, o esforço exigido para produzir um sistema completo para avaliações

mais sérias é normalmente medido em anos.

Kitakami et alli /KIT 83/ definem a aquisição de conhecimento como " a função de coleta de conhecimento através de assimilação e acomodação numa base de conhecimento ", ou seja, através de um processo de aprendizado de máquina.

Desde que se tenha escolhido representar o conhecimento numa forma fácil de ser examinada e alterada, é possível estabelecer diversos métodos pelos quais um sistema computacional pode adquiri-lo. Estes métodos normalmente são agrupados nas seguintes classes gerais: aprendizado através do que é dito, aprendizado induzido por exemplos e aprendizado por observação e descoberta /MIC 83/.

2.4.1 Aprendizado através do que é dito

Neste método são fornecidos ao sistema fatos e regras gerais sobre como manipular estes fatos, os quais constituem uma base de conhecimento K que sugere as respostas desejadas. Normalmente K é muito menor que a lista explícita de respostas.

Esta é a forma mais simples de aquisição de conhecimento, podendo ser surpreendentemente útil, especialmente se houver um sistema de apoio nos pontos de omissão ou de regras incorretas. Este processo tem sido chamado de transferência interativa de experiência.

Em princípio espera-se que o processo de aquisição de conhecimento verifique o que é dito. Basicamente há três casos

que podem ocorrer quando se apresenta ao sistema novos itens de conhecimento:

a) O novo item é dedutível a partir do conhecimento existente. Desconsiderando os aspectos de eficiência, o sistema deve rejeitar o item, enviando uma mensagem apropriada ao utilizador.

b) O novo item é inconsistente em relação ao conhecimento corrente. Ou o item será rejeitado ou o conhecimento será alterado antes de sua aceitação. Alternativamente pode-se desejar que o sistema adicione o item automaticamente como uma exceção do conhecimento ou que mantenha um diálogo sobre quais decisões tomar com a pessoa que o fornece.

c) O novo item não é nem dedutível nem inconsistente. O item é acrescentado ao conhecimento. Entretanto, não pode haver redundâncias, ou seja, caso se adicione uma regra geral que cubra um certo conjunto de fatos, deve-se removê-lo antes da adição da regra.

A subtração de itens de conhecimento pode acarretar alguns problemas teóricos e práticos, já que as pessoas não esquecem automaticamente fatos ou conhecimentos quando estes deixam de ser verdadeiros. De fato, Kowalski e Sergot /KOW 85/ propuseram que, ao invés de excluir fisicamente um item, deve-se adicionar à base de conhecimento o fato de o item ter deixado de ser verdadeiro num dado momento. Entretanto, esta proposta pode trazer problemas práticos, levando à necessidade de arquivos

muito extensos para armazenar o conhecimento.

2.4.2 Aprendizado induzido por exemplos

E um fato aceito a dificuldade que especialistas numa determinada área profissional têm freqüentente em escrever explicitamente seu conhecimento. O normal é explicarem a realização de uma tarefa através de um conjunto de exemplos.

No método de aprendizado induzido a partir de exemplos, a parte de aquisição de conhecimento do sistema consiste numa coleção de boas e más recomendações que deve induzir uma base de conhecimento K a sugerir as boas recomendações, abster-se das más recomendações e fornecer respostas corretas para exemplos distintos dos da coleção original. Com esse objetivo, o mecanismo de aquisição de conhecimento necessita de um critério de orientação para detectar a melhor base de conhecimento K. Logicamente é um processo de indução: dado E, descobrir um K conveniente tal que K implique E, sendo E o conjunto de exemplos. A dificuldade reside no fato de ser muito mais difícil capturar um conceito geral do que reconhecer a conveniência para um caso particular.

2.4.3 Aprendizado por observação e descoberta

No aprendizado por observação e descoberta equipa-se o sistema de aquisição de conhecimento com uma base de conhecimento inicial mínima K, alguns operadores O para adição de informações a K, e algumas orientações D sobre quais operadores aplicar em determinadas circunstâncias. Quando executado, esse sistema de

aquisição de conhecimento aplica O a K, orientado por D. Caso seja feita uma boa escolha de K, O e D, o sistema gerará uma grande base de conhecimento K´, a qual pode conter algumas conjecturas úteis.

Esta abordagem tem a característica de ser a nível lógico um processo de segunda ordem, pois inclusive nomes de funções e predicados devem ser representados por variáveis. Isto exige a manipulação de funções ou predicados gerais, unificáveis com quaisquer outros.

2.5 Prolog e programação em lógica

Embora a linguagem Prolog tenha sido primeiro desenvolvida para suportar processamento de língua natural, ela trabalha essencialmente como uma parte eficientemente executável da lógica matemática, o que é de interesse para a modelagem de raciocínio em sistemas especialistas.

Por volta de 1981, Prolog foi adotada pelos japoneses como base para seu Projeto de Computadores de V Geração. Os objetivos desse projeto de dez anos incluem trabalhos fundamentais tanto em hardware como em software para bases de conhecimento avançadas.

Prolog possui um certo aspecto híbrido, pois contém algumas características declarativas da lógica matemática computacional e alguns aspectos da programação convencional.

Muitos dos mecanismos necessários a um "shell" de

sistema especialista estão na linguagem. Por exemplo, Prolog possui uma associação lógica de padrões generalizados chamada unificação, a qual normalmente não é vista ou invocada explicitamente quando se escreve um programa. Talvez devido à falta de características convencionais, Prolog possui uma semântica muito prática e elegante que parece ser a chave para muito de seu apelo.

Devido a sua relação com a lógica matemática, a linguagem Prolog possui uma base teórica bem como um corpo de experiências práticas associadas. De fato, a teoria e a prática podem unir-se muito bem, e pode ser útil estendê-las para incluir bases de dados relacionais /BOA 86/.

Como uma conseqüência da sua semântica simples, Prolog pode trabalhar muito bem como sua própria metalinguagem, ou seja, é muito fácil descrever a semântica da linguagem Prolog em Prolog. Enquanto um interpretador Prolog escrito numa linguagem convencional pode ocupar até 10000 bytes, um interpretador Prolog escrito em Prolog ocupa cerca de 100 bytes. Isto pode parecer apenas uma curiosidade acadêmica, até que se observe que muitos mecanismos de inferência para sistemas especialistas são de alguma forma similares ao mecanismo de inferência da linguagem Prolog.

Paga-se um preço em eficiência para esta elegância conceitual, pois os mecanismos de inferência construídos em Prolog são geralmente mais lentos do que se tivessem sido escritos diretamente. Entretanto, quando suportada por um computador

rápido, o desempenho é adequado para muitas tarefas, as vantagens de flexibilidade são esmagadoras, e é possível compilar alguns aspectos do mecanismo de inferência, se necessário. O mecanismo de inferência consiste de regras Prolog que funcionam como metainformação: são regras sobre como usar regras para uma determinada tarefa. Caso se tenha informação procedural sobre como realizar um grupo de tarefas eficientemente, então pode-se codificar esta em metaregras que definem o mecanismo de inferência. Dessa forma, na abordagem de metalinguagem para sistemas especialistas em Prolog, existe um local natural para conhecimento procedural, sendo que o conhecimento da tarefa pode permanecer declarativo.

Mencionou-se que Prolog é uma linguagem simples, com poucas características. De fato, a principal parte da linguagem Prolog carece de muitas construções padrões em outras linguagens de programação. Isto tende a guiar o programador em direção a um estilo mais declarativo, distanciando-se do estado de transição do nível de máquina.

3. O PROBLEMA: PROJETO DE CIRCUITOS DIGITAIS

Projetar circuitos digitais é uma tarefa complexa, em geral exigindo a aplicação de uma gama muito grande de métodos e técnicas especializadas. A automatização desses procedimentos é uma forma de torná-los rápidos e, principalmente, confiáveis e reproduzíveis. A confiabilidade diz respeito à correção do projeto final. A rapidez e a reprodutibilidade dizem respeito à facilidade de tentar alternativas de estruturas, procedimentos ou componentes.

Diversos trabalhos na área de projeto automatizado de circuitos digitais seguindo abordagens distintas têm sido desenvolvidos nos últimos anos /THO 81/. Alguns deles adotam métodos não determinísticos, apoiados por bases de conhecimento, como o sistema SOCRATES /COH 85/ e o sistema experimental CSU /MUE 85/.

Neste capítulo apresenta-se uma classificação de níveis de projeto de circuitos, enfatizando o nível RT e definindo os conceitos de descrições comportamental, funcional e estrutural neste nível. Apresenta-se também os objetivos do PROTO no contexto do problema de projeto de circuitos digitais.

3.1 Níveis de projeto

Barbacci /BAR 75/ define cinco níveis maiores na hierarquia de projeto de sistemas digitais. São eles:

(a) Nível de Sistema (nível PMS): E o nível mais alto de descrição, avaliando as propriedades flagrantes de sistemas

computacionais. Seus elementos são processadores, memórias, interruptores, unidades periféricas, etc, e seus parâmetros são custos, capacidades de memória, taxas de fluxo de informação, etc.

b) Nível de Programação: Neste nível os componentes básicos são o ciclo de interpretação, as instruções de máquina e as operações (as quais são definidas no nível RT). O comportamento do processador é determinado pela natureza e seqüência de suas operações. Esta seqüência é dada por um programa (conjunto de bits na memória primária) e um conjunto de regras de interpretação. Dessa forma, caso se especifique a natureza das operações e as regras de interpretação, o comportamento real do processador depende unicamente das condições iniciais e do programa específico.

c) Nível de Transferência entre Registradores (nível RT ou funcional): Neste o fluxo de dados e o controle operam em passos discretos. Uma combinação de circuitos de chaveamento é usada para formar registradores e outras operações de dados. Os conteúdos de elementos (registradores) são combinados (transformados) de acordo com algumas regras e então armazenados (transferidos) em outro registrador. As regras de transformação podem ser quaisquer, desde transferências simples até expressões aritméticas e lógicas complexas.

(d) Nível de circuitos de chaveamento (sub-níveis seqüencial e combinacional): Neste nível a estrutura do sistema é fornecida através de uma coleção de portas lógicas e flip-flops,

e o seu comportamento através de operações booleanas. A temporização é levada a um grau mais refinado, sendo a unidade de tempo usualmente na ordem de um retardo de uma porta lógica.

e) Nível de circuito eletrônico: Este nível descreve as portas lógicas como algumas combinações de diodos, transistores, resistores, etc, conforme as leis dos circuitos eletrônicos. A maioria das propriedades discretas dos dois níveis anteriores são perdidas, e a temporização é levada a um grau altamente refinado onde o comportamento transitório é uma consideração importante.

3.2 Características do nível RT

O nível RT é uma generalização do nível de circuitos de chaveamento. Os elementos estruturais são coleções de subsistemas idênticos aos pertencentes ao nível de chaveamento; por exemplo, registradores são formados de flip-flops e portas lógicas, carregados por sinais de relógio. O comportamento é descrito através de transformações (funções) e transferências entre registradores. O elemento chave que coloca este nível fora do nível abaixo é o aparecimento do controle (a habilidade de realizar estas transformações e transferências de forma seletiva) como uma entidade explícita.

3.3 Formas de descrição no nível RT

Caso uma descrição das instruções de máquina seja fornecida de tal forma que cada passo possa ser interpretado como uma ação de hardware, obtem-se uma descrição no nível RT. Neste,

a descrição é representada como uma seqüência de transformações de informações transferidas de portador (registrador ou barramento) para portador por redes funcionais (operadores) e/ou barramentos ou outros caminhos de dados. Entretanto, há muitas formas nas quais uma descrição RT pode ser fornecida, dependendo do seu propósito e diferindo em grau de detalhe e clareza da informação apresentada. Giloi et alli /GIL 78/ apresentam uma definição das três maiores formas de descrição RT: comportamental, funcional e estrutural.

3.3.1 Descrição comportamental

Esta forma apresenta somente a descrição dos efeitos de cada instrução sobre os portadores, sendo que os conteúdos da memória são descritos e nenhuma temporização é fornecida. Comandos de controle fornecem somente as informações necessárias para o entendimento do comportamento dos algoritmos descritos, não implicando na implementação real. O mesmo é verdadeiro para os operadores funcionais, representando transformações lógicas, aritméticas, de reestruturação e especiais. Variáveis intermediárias, para as quais não existe correspondência de hardware, podem ser introduzidas. Os caminhos de dados necessários para que a transferência se realize estão implícitos. O domínio e a abrangência dos operadores não são especificados, assumindo-se que satisfazem as exigências impostas pelo contexto no qual são usados.

3.3.2 Descrição funcional

A maior diferença entre uma descrição comportamental e uma descrição funcional é que, nesta última, o domínio e a abrangência dos operadores são declarados na descrição e detalhes de temporização são especificados para todas as operações e processos. Além disso, os operadores devem ser definidos de tal forma que sua implementação em hardware esteja implícita.

3.3.3 Descrição estrutural

Nesta forma de descrição as construções de controle para sincronização e ramificação são gradualmente substituídas por uma seção de controle de hardware. Todos os caminhos de dados, incluindo as facilidades de chaveamento, multiplexação e demultiplexação necessárias, devem ser especificados. Conseqüentemente, a semântica de uma atribuição neste nível de descrição é a de uma conexão. Temporização é introduzida explicitamente através do uso de sinais de relógio, substituindo pelo menos parcialmente as construções de controle da descrição funcional.

3.4 O PROTO no contexto de projeto de circuitos digitais

O PROTO trata de uma fase específica do projeto de circuitos digitais: o projeto lógico de blocos operacionais a partir de uma descrição algorítmica comportamental-RT do circuito, gerando uma descrição estrutural-RT correspondente.

As descrições do circuito (comportamental e estrutural) são representadas através de grafos, sendo que os nodos represen-

tam os componentes do circuito (registradores, linhas, operadores, etc) e os arcos representam as interconexões entre esses componentes. Esta representação está baseada num formalismo denominado Redes de Transformação de Representações (redes-TR) /COS 82/, o qual é capaz de modelar circuitos lógicos. A partir dos modelos comportamentais se pode obter, por meio de regras bem definidas associadas a um método de transformação de redes, modelos estuturais de circuitos lógicos.

4. DESCRIÇÃO GERAL DO PROTO

O PROTO é um protótipo de sistema especialista orientado para a solução de problemas que possam ser expressos como problemas de transformação de redes /COS 84/, desenvolvido em particular para o problema de projeto lógico de blocos operacionais de circuitos digitais, que é uma das tarefas para o qual este método melhor se adapta.

4.1 O método de solução de problemas por transformação de redes

Este método soluciona problemas que possam ser expressos como problemas de transformação de redes. Está inspirado no método de resolução de problemas por ajustamento de padrões /McD 81/.

O método de solução de problemas por transformação de redes caracteriza a transformação como um mapeamento de uma rede inicial para uma rede final. Esta transformação é realizada por um processo de construção paulatina da rede final, onde são detectadas todas as instâncias de subredes unificáveis com os esquemas de subredes presentes nas regras de transformação, resultando as subredes que compõem a rede final. Um esquema de subrede é um conjunto de padrões parcialmente instanciados, (compostos por constantes e variáveis), que define uma subrede genérica.

O método supõe que os valores substituídos nas variáveis dos esquemas de subrede dependem apenas do local de

aplicação das regras, e não da ordem das mesmas (condição de correspondência). Além disso, é possível definir uma ordem de aplicação das regras de transformação de tal modo que a aplicação de quaisquer regras não exija a revisão das decisões tomadas por outras regras (condição de propagação). Estas duas condições de aplicabilidade são responsáveis pela grande atração do método, que é a característica de dispensar retrocessos.

A figura 4.1 apresenta um exemplo de aplicação deste método. Nela são mostradas graficamente três regras de transformação que atuam sobre uma pequena rede inicial, gerando uma rede final correspondente. Para cada regra, é apresentado o esquema de subrede que é condição para sua aplicação (à esquerda do símbolo "==>") e o esquema de subrede resultante, além de todas as instâncias de subrede detectadas na rede inicial. As linhas tracejadas informam os nodos da rede inicial que foram mapeados para a rede final. Neste exemplo existe apenas um nível de mapeamento: nodos da rede inicial para nodos da rede final. Entretanto, o método admite diversos níveis de mapeamento a partir de uma rede inicial (nível i), gerando subredes de níveis intermediários (nível i+1, nível i+2, ..., nível n-1), até os nodos que formam a rede final (nível n). Além disso, a forma da rede final depende do conjunto de regras aplicadas, sendo que conjuntos distintos podem determinar redes compostas por nodos distintos e com diferentes topologias a partir de uma mesma rede inicial. A simbologia adotada no exemplo da figura 4.1 é descrita nas seções 5.2.2 e 5.3.1.

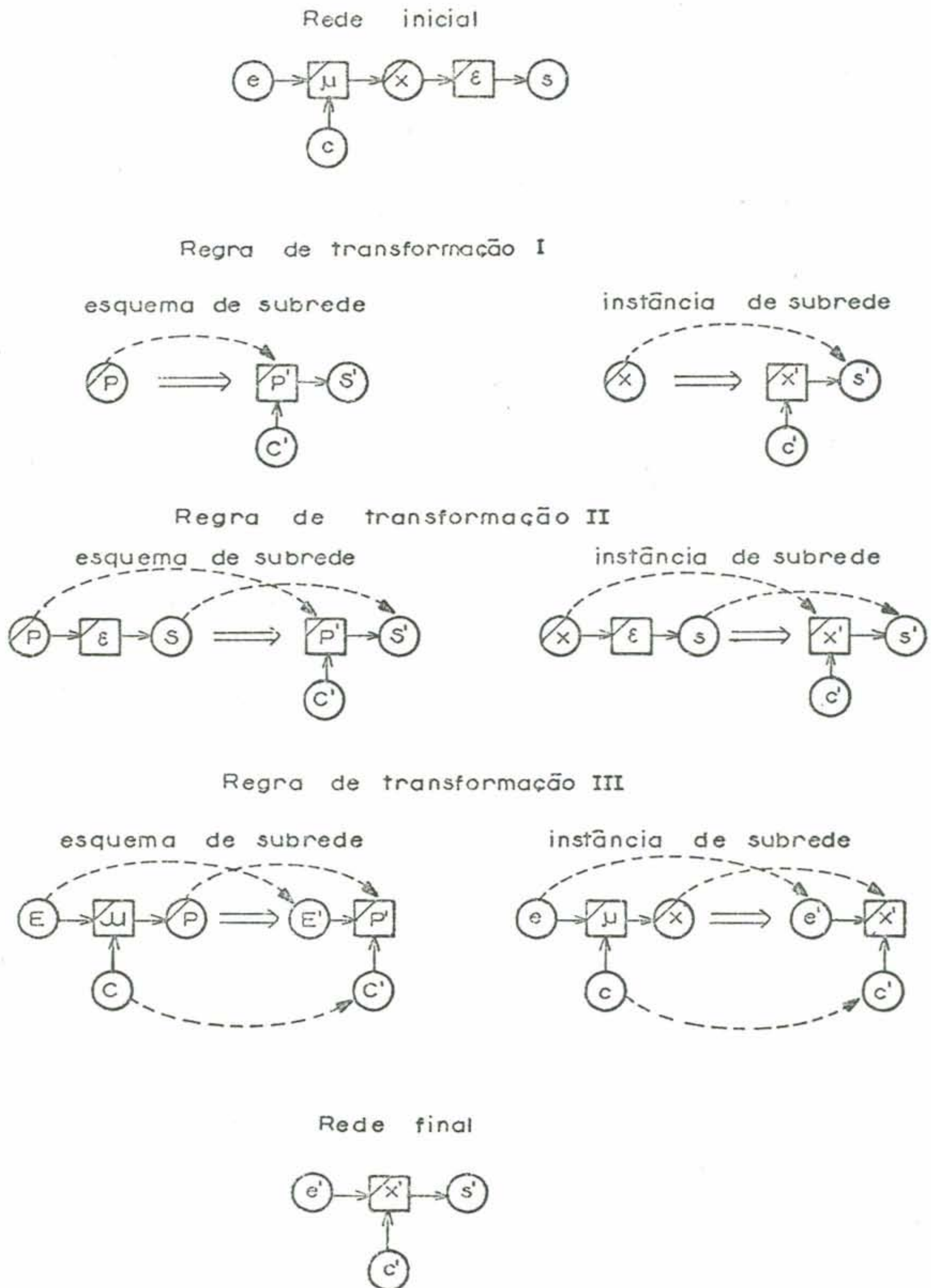


Fig. 4.1 - Exemplo de aplicação do método de transformação de redes.

4.2 Arquitetura

O PROTO é composto por dois subsistemas distintos: o subsistema de projeto lógico e o subsistema de aquisição de conhecimento. O primeiro é responsável pela tarefa de projeto lógico de blocos operacionais de circuitos digitais, enquanto o segundo suporta a construção e edição das bases de conhecimento a serem usadas pelo subsistema de projeto lógico.

4.2.1 Arquitetura do subsistema de projeto lógico

Este subsistema é formado por quatro módulos: módulo de controle, módulo de compilação da descrição algorítmica do circuito, módulo interpretador de regras e módulo compilador de estilos de mapeamento, além de rotinas de mapeamento direto controladas pelo módulo de controle. Em linhas gerais apresenta a arquitetura mostrada na figura 4.2.

4.2.1.1 Módulo de controle

A opção por um controle centralizado foi necessária para que as várias funções possíveis do subsistema possam ser solicitadas de modo interativo, em uma única sessão. Essas funções estão localizadas em diferentes módulos do subsistema. Cabe ao módulo de controle obter as intenções do utilizador em relação às tarefas desejadas durante uma sessão, invocando os módulos que as realizem.

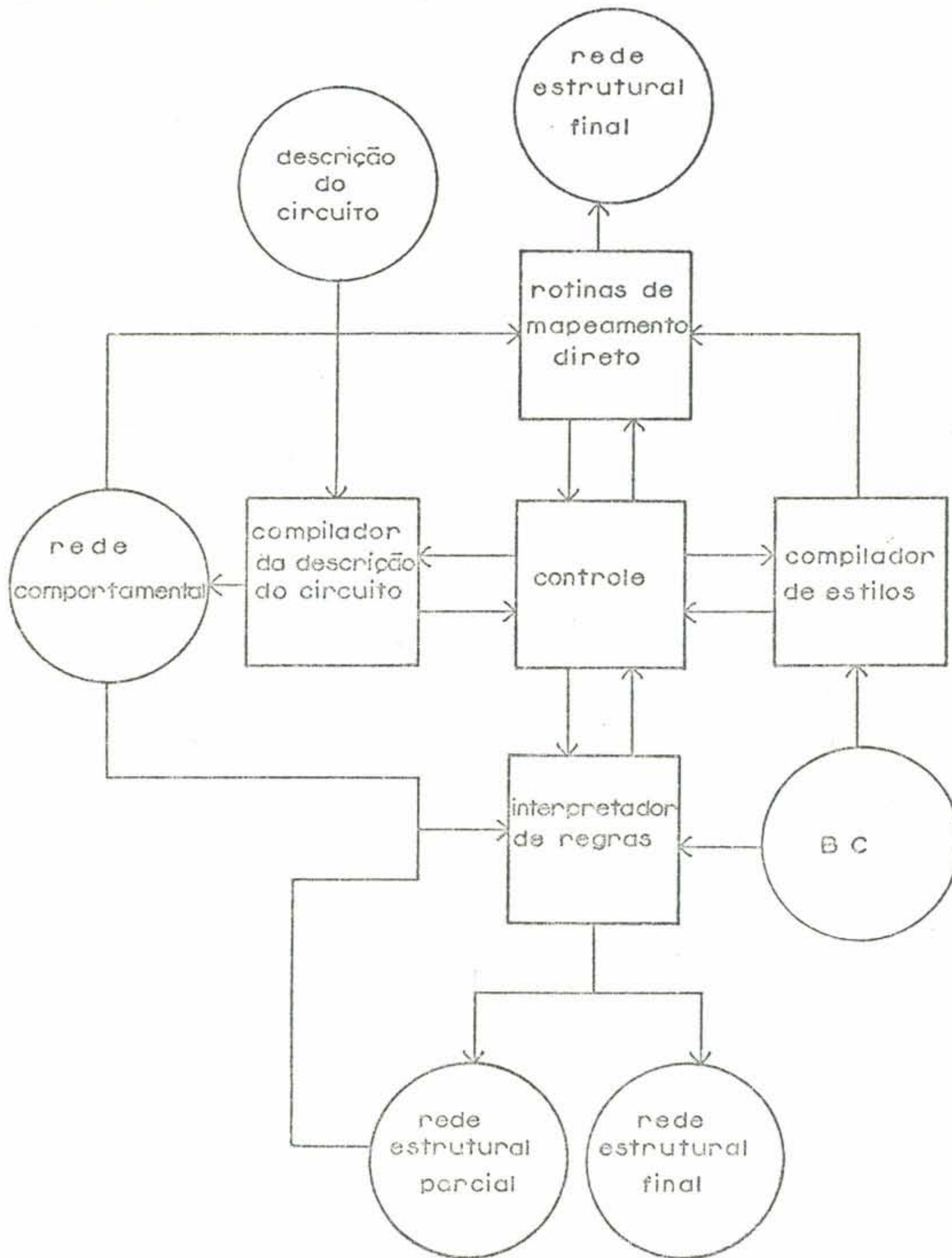


Fig. 4.2 - Arquitetura do subsistema de projeto lógico.

4.2.1.2 Módulo compilador da descrição algorítmica do circuito

Este módulo tem por função gerar a rede comportamental do circuito a partir da sua descrição algorítmica.

A construção da rede comportamental é feita por um processo de compilação em que a descrição do circuito é convertida em uma rede-TR. Este processo utiliza uma gramática de cláusulas definidas que descreve as construções admitidas na linguagem de descrição do circuito e um conjunto de sub-processos de transformação dessas estruturas nos predicados que constituem a rede comportamental.

Gramáticas de cláusulas definidas são uma extensão das gramáticas livres de contexto. São reconhecidas pela linguagem Prolog, possuindo a forma C-->P, com o significado: "uma possível forma para C é P" /CLO 81/.

A linguagem de descrição do circuito, a rede comportamental e o processo de compilação estão detalhados, respectivamente, nas seções 5.2.1, 5.2.2 e 5.2.3.

4.2.1.3 Módulo interpretador de regras

Este módulo tem por objetivo gerar uma rede estrutural do bloco operacional de um circuito a partir de uma rede inicial (comportamental ou estrutural parcial) do mesmo. Isto é feito por meio de um processo de interpretação e execução das regras de transformação de redes contidas numa base de conhecimento, selecionadas por um estilo de aplicação. A base de conhecimento e o

estilo de aplicação das regras são indicados pelo utilizador. Os procedimentos contidos no interpretador combinados com o conteúdo da base de conhecimento implementam o método de solução de problemas por transformação de redes, mantendo-se o interpretador de regras independente em relação ao domínio do conhecimento sobre o problema.

Este módulo também apresenta um mecanismo que permite a monitoração da execução das regras que mostra, para cada regra, as instâncias de subredes detectadas que satisfazem suas condições e as ações por ela realizadas. Este mecanismo é opcional, sendo útil durante a etapa de depuração e validação das regras de mapeamento de redes de uma base de conhecimento.

A utilização do interpretador de regras, os predicados que descrevem um estilo de geração de rede estrutural e um exemplo do processo de interpretação das regras são apresentados na seção 5.3.

4.2.1.4 Módulo compilador de estilos

O objetivo deste módulo é a construção de rotinas específicas para mapeamento direto de redes comportamentais para redes estruturais. Isto é feito através de um processo de compilação de todas as estruturas de uma base de conhecimento necessárias à transformação de um estilo (seqüência) de aplicação de regras contido nessa base em uma rotina Prolog. O compilador de estilos também incorpora às rotinas de mapeamento direto geradas todos os procedimentos necessários para estas terem para o utili-

zador o mesmo comportamento do interpretador de regras, exceto a possibilidade de monitoração da execução das regras e a necessidade de indicar uma base de conhecimento.

4.2.1.5 Rotinas de mapeamento direto

São programas Prolog gerados pelo compilador de estilos. Cada rotina de mapeamento direto implementa o método de solução de problemas por transformação de redes diretamente, ou seja, gera uma rede estrutural do bloco operacional de um circuito a partir de sua rede comportamental, dispensando consultas a uma base de conhecimento externa, pois todo o conhecimento necessário à realização da tarefa de projeto lógico de circuitos encontra-se nela programado.

A possibilidade de utilização dessas rotinas foi incorporada ao sistema com o intuito de liberar certos utilizadores do conhecimento e manuseio das bases de conhecimento do PROTO, além de melhorar o desempenho da tarefa de projeto lógico. Isto torna o interpretador de regras reservado apenas aos utilizadores que contrõem e mantém as bases de conhecimento, auxiliando-os nas etapas de depuração e validação do conhecimento.

4.2.2 Arquitetura do subsistema de aquisição de conhecimento

Este subsistema apóia as atividades de construção e manutenção das bases de conhecimento do PROTO. Permite a construção de novas bases, bem como a ampliação, supressão, alteração e

exame do conhecimento existente. Compõe-se de três módulos: módulo de controle, módulo de construção e módulo de edição de bases de conhecimento. Sua arquitetura é apresentada graficamente na figura 4.3.

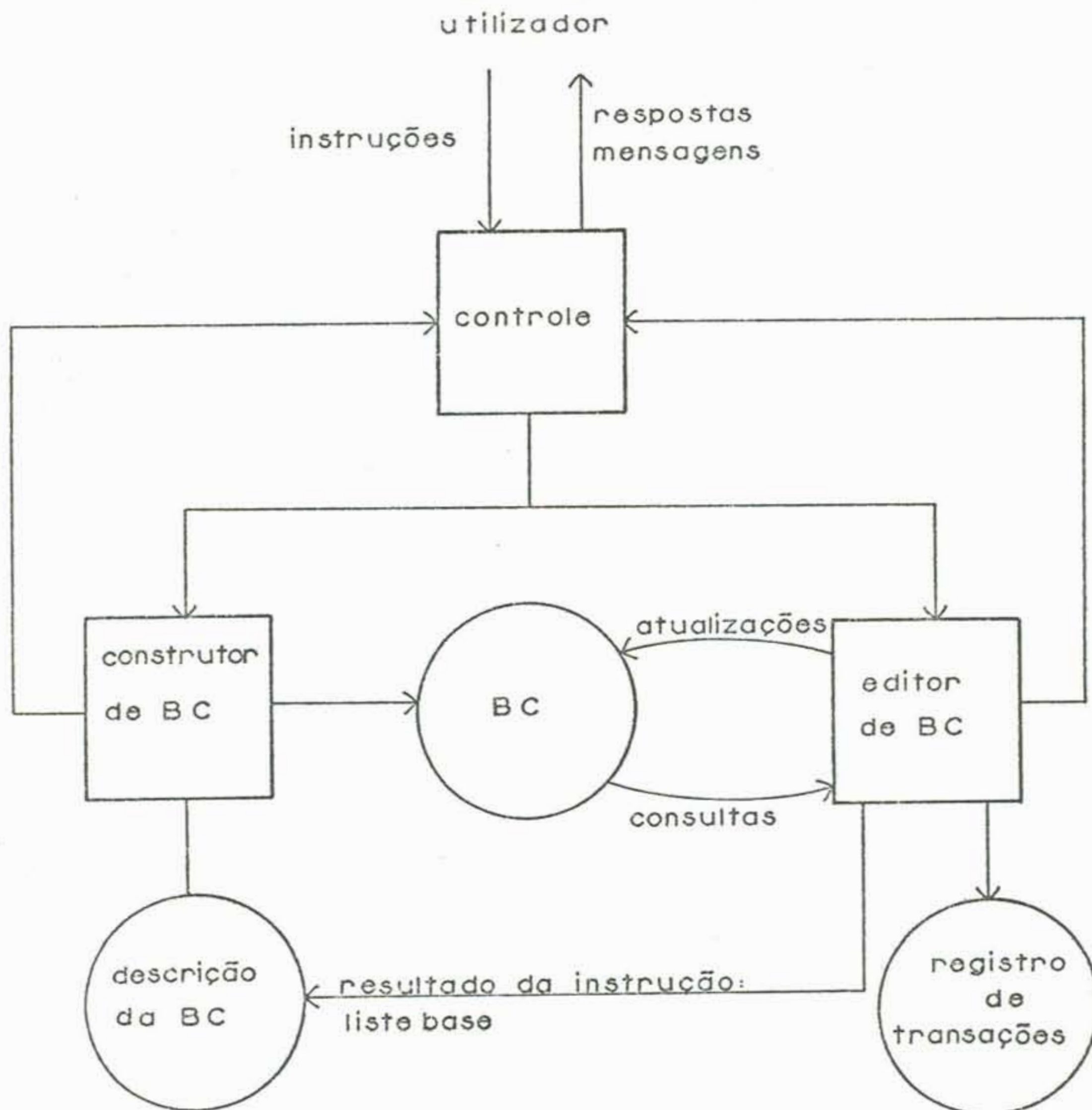


Fig. 4.3 - Arquitetura do subsistema de aquisição de conhecimento.

A responsabilidade final sobre a validade do conhecimento contido nas bases é do utilizador desse subsistema, pois se optou por uma forma simples de aquisição de conhecimento que utiliza o enfoque do aprendizado pelo que é dito. São funções do subsistema:

a) analisar a sintaxe do conhecimento fornecido;

b) adaptar os itens de conhecimento fornecidos pelo utilizador à representação interna da base de conhecimento; e

c) apontar incongruências semânticas nos itens de conhecimento que não estiverem de acordo com os demais itens da base.

4.2.2.1 Módulo de controle

Este módulo tem funções equivalentes ao módulo de controle do subsistema de projeto lógico. Controla a execução dos demais módulos de tal forma que um utilizador pode construir e/ou editar diversas bases de conhecimento em uma única execução do subsistema de aquisição de conhecimento.

4.2.2.2 Módulo de construção de bases de conhecimento

O módulo de construção de bases de conhecimento é um compilador de descrições de bases de conhecimento escritas numa linguagem específica.

Uma descrição de base de conhecimento contém o nome da base a ser construída, o arquivo onde se encontra o conhecimento

pré-definido do sistema (os componentes da rede comportamental, a relação básica de mapeamento de redes e alguns procedimentos Prolog de apoio à definição das regras), os predicados que descrevem a(s) rede(s) estrutural(is), as regras de transformação de redes e os estilos de aplicação dessas regras. O módulo de construção de bases de conhecimento analisa essas informações e gera as representações internas correspondentes. Pode também explicitar informações deduzidas a partir do conteúdo global da base gerada. Optou-se por explicitar o conhecimento deduzido para melhorar o desempenho das operações de consulta e consistência das bases. Acredita-se que um melhor desempenho nessas operações compensa a presença de algumas informações redundantes.

O processo de compilação somente gera a base de conhecimento se não forem detectados erros na sua descrição. Os erros encontrados são apresentados imediatamente ao utilizador.

4.2.2.3 Módulo de edição de bases de conhecimento

Este módulo é um processo interativo que suporta operações de modificação e consulta às bases de conhecimento do PROTO. As operações de modificação englobam inclusões, exclusões e alterações de itens de conhecimento. As operações de consulta permitem o exame de itens de conhecimento isolados e/ou conjuntos de itens de conhecimento relacionados.

Cada operação de modificação é registrada em um arquivo de transações que apresenta a situação de cada item modificado antes e depois da realização da operação. Sua finalidade é, no

momento, apenas documentacional, mas pode vir a ser utilizada no futuro por um subsistema de recuperação de bases de conhecimento.

Por último, o módulo pode fornecer uma descrição da base de conhecimento no formato de entrada para o módulo de construção.

4.3 Organização do conhecimento

As bases de conhecimento do PROTO compõem-se de um conjunto de **regras** de transformação de redes, dos **estilos** de aplicação dessas regras e de uma caracterização das construções que formam as redes sobre as quais as regras atuam. Essas construções são os **elementos** (nodos) das redes, as possíveis **relações** (arcos) entre esses elementos e **marcas** (inscrições) que os caracterizam em determinadas situações.

As bases de conhecimento também podem apresentar rotinas escritas em Prolog arroladas pelo nome genérico de **testes**. Estas podem ser referenciadas pelas regras, bem como auxiliar a definição de elementos e relações.

4.3.1 Identificação da base

Uma base de conhecimento é identificada internamente através do seguinte predicado:

```
base_conhecimento(<nome da base>,
                  <nome do conhecimento pré-definido>,
                  <objetivos da base>).
```

O conhecimento pré-definido é um arquivo que contém:

a) os predicados que descrevem os elementos, relações e marcas que formam a rede comportamental;

b) a definição da relação básica de mapeamento de elementos de redes; e

c) um conjunto de testes (procedimentos Prolog) de apoio à confecção das regras de transformação de redes.

Os objetivos da base são um conjunto de sentenças em língua natural que têm caráter documentacional. São representados através de uma lista de termos (palavras, números, caracteres de pontuação e símbolos especiais).

4.3.2 Regras

As **regras de transformação de redes** englobam um conjunto de predicados interpretáveis pelo subsistema de projeto lógico que, a partir de esquemas de subredes de nível n, geram instâncias de subredes de nível n+1 ou complementam instâncias de subredes de nível n.

Os predicados que definem uma regra são os seguintes:

```
regra(<nome da regra>,
      <objetivos da regra>).

condicao(<nome da regra>,
         <lista de parâmetros>,
         <esquema de subrede de nível n>).
```

```
acao(<nome da regra>,
     <lista de parâmetros>,
     <seqüência de ações>).
```

Um esquema de subrede de nível n pode ser formado por padrões gerais de elementos da rede de nível n, relações entre esses elementos, marcas sobre alguns elementos e testes Prolog sobre variáveis e/ou padrões que auxiliam sua definição.

A seqüência de ações pode conter os seguintes predicados:

a) criar(P) : cria um elemento de rede de nível n+1 cuja forma obedece ao padrão dado por P; e/ou

b) marcar(P) : cria um predicado P, sendo P um padrão de marca ou relação sobre/entre elementos de nível n ou elementos de nível n+1.

A lista de parâmetros é uma lista de variáveis comuns às condições e ações da regra, possibilitando o uso do mecanismo de unificação da linguagem Prolog durante o processo de interpretação das regras.

4.3.3 Estilos

As possíveis seqüências de aplicação das regras de transformação de redes são definidas por predicados **estilo**. Cada estilo apresenta um subconjunto não vazio de nomes de regras da base de conhecimento, de tal forma ordenado que atenda à condição de propagação do método de solução de problemas por transformação

de redes. Um predicado **estilo** apresenta a seguinte forma geral:

```
estilo(<nome do estilo>,
       <objetivos do estilo>,
       <seqüência de nomes de regra>).
```

4.3.4 Elementos

São chamados **elementos** os predicados que descrevem os nodos das redes comportamental e estrutural. Um elemento encontra-se definido numa base de conhecimento através do seu padrão geral, do seu nível de definição (nível da rede a partir do qual está definido), dos estilos que o referenciam, de suas origens e das regras que o utilizam para resultar instâncias de subredes.

```
elemento(<nome do elemento>,
         <objetivos do elemento>).

padrao_elem(<nome do elemento>,
            <padrão geral>,
            <lista de valores de tipo>).

nivel_elem(<nome do elemento>,
           <nível de definição>).

estilos(<nome do elemento>,
        <lista de estilos>).

origem(<nome do elemento>,
       <lista de origens>).
```

resulta(<nome do elemento>,
<lista de resultantes>).

O padrão geral de um elemento é um predicado unificável com todas as instâncias possíveis deste elemento nas redes. Pode apresentar dois formatos:

a) P([Id,N]) ou

b) P([Id,N],T).

Sendo: P - nome de predicado Prolog;

Id- variável que representa o identificador do elemento;

N - variável que identifica o nível de geração do elemento; e

T - variável ou constante (letras, números, símbolos, predicados, etc) que informa o tipo do elemento.

A lista de valores de tipo associada ao padrão geral do elemento é vazia no caso (a) e possui pelo menos um elemento no caso (b).

O predicado **nivel_elem** informa o menor nível de rede a partir do qual o elemento está definido. Elementos da rede comportamental possuem nível 0. Elementos gerados a partir do mapeamento de elementos de nível 0 possuem nível de definição 1. Elementos resultantes apenas do mapeamento de elementos de nível 1 possuem nível de definição 2. E assim sucessivamente. E

importante ressaltar que nada impede que um elemento com nível de definição i componha uma subrede de nível j, j > i.

O predicado **estilos** informa todos os estilos que referenciam o elemento. A lista de estilos pode conter:

a) um ou mais nomes de estilo; ou

b) o termo **todos**, quando, por definição, o elemento é referenciado por todos os estilos presentes na base.

A lista de origens do predicado **origem** pode conter:

a) um ou mais nomes de regras de transformação de redes que o podem criar;

b) o termo **compilacao**, quando for um elemento de nível comportamental (nível 0); ou

c) o termo **indeterminado**, quando, por definição, não é possível delimitar um subconjunto de regras que o podem originar.

De forma semelhante, a lista de resultantes associado ao predicado **resulta** pode apresentar:

a) um ou mais nomes de regras que o referenciam nas suas condições; ou

b) o termo **indeterminado**, quando, por definição, não é conveniente delimitar um conjunto de regras que o referenciem.

As listas de estilos, origens e resultantes podem ser vazias, desde que não haja na base de conhecimento,

respectivamente, nenhum estilo, condição de regra ou ação **criar** que referencie o elemento.

4.3.5 Relações

As **relações** descrevem as conexões entre os diversos elementos de um grafo. Correspondem aos arcos que unem os nodos e definem a topologia da rede. Cada relação presente numa base de conhecimento do PROTO é definida através do seguinte conjunto de predicados:

```
relacao(<nome da relação>,
        <objetivos da relação>,
        <tipo da relação>).

padrao_rel(<nome da relação>,
           <forma geral>).

arg_rel(<nome da relação>,
        1,
        <descrição do argumento>).

        . . . . .

arg_rel(<nome da relação>,
        <n>,
        <descrição do argumento>).
```

As relações podem ser de três tipos: **b**, que identifica as relações consideradas básicas pelo sistema, como a relação básica de mapeamento entre elementos de redes; **c**, que identifica

as relações que definem a topologia da rede comportamental; ou **e**, que identifica as relações da rede estrutural. As relações de tipo **b** e **c** fazem parte do conhecimento pré-definido do sistema.

O padrão geral de uma relação obedece ao formato P(<lista de argumentos>), onde P é um nome de predicado Prolog e a lista de argumentos é composta por duas ou mais variáveis, separadas por vírgula. Cada argumento dessa lista possui um predicado **arg-rel** associado que o descreve.

Um argumento de uma relação pode representar um elemento ou um índice. A descrição de um argumento que representa um elemento apresenta o seguinte formato:

```
arg_rel(<nome da relação>,
        <i>,
        elemento(<padrão>),
        nivel(<relação de nível>).
```

Sendo que i é a ordem do argumento na lista de argumentos, padrão é um predicado unificável com um padrão geral de elemento, e relação de nível é uma expressão relacional da forma <variável> <operador relacional> <valor>, onde variável representa o nível do elemento, operador relacional pode ser =, >, <, >= ou =<, e valor pode ser uma constante inteira sem sinal ou uma outra variável.

Argumentos que representam índices são descritos por predicados que seguem o seguinte modelo:

```
arg_rel(<nome da relação>,
        <i>,
        indice,
        valores(<lista de valores de índice>).
```

A lista de valores de índice apresenta uma ou mais constantes (numéricas ou alfanuméricas ou símbolos especiais) e/ou cabeças de cláusulas Prolog declaradas na base de conhecimento como testes.

4.3.6 Marcas

São chamadas **marcas** as inscrições presentes nos grafos que caracterizam funções especiais de certos elementos (por exemplo, entrada no circuito, saída do circuito, início de bloco). Cada marca é definida na base de conhecimento pelo seguinte grupo de predicados:

```
marca(<nome da marca>,
      <objetivo da marca>,
      <tipo da marca>).

padrao_marca(<nome da marca>,
             <forma geral>).

arg_marca(<nome da marca>,
          <padrão do argumento>,
          <lista de tipos do argumento>,
          <relação de nível>).
```

O tipo de uma marca pode ser:

a) **c**, quando atua sobre elementos da rede comportamental; ou

b) **e**, quando atua sobre elementos da(s) rede(s) estruturais.

A forma geral de uma marca segue o formato P(A), onde P é um nome de predicado Prolog e A é uma variável que representa um identificador de elemento.

O padrão do argumento é um predicado unificável com um padrão geral de elemento ou uma variável.

A lista de tipos do argumento contém todas as instâncias do tipo T de um padrão de elemento P([E,N],T) sobre as quais a marca pode atuar. Esta é obrigatoriamente vazia quando o padrão do argumento é da forma P([E,N]).

4.3.7 Testes

São denominados **testes** os procedimentos Prolog presentes e/ou referenciados numa base de conhecimento. Eles podem ser usados nas definições de esquemas de subredes que formam as condições das regras, de tipos de elementos e argumentos do tipo índice de relações. Cada um desses procedimentos é armazenado na base associado a um predicado teste, o qual apresenta a seguinte estrutura:

```
teste(<nome do teste>,
      <objetivo do teste>,
      <forma geral do teste>).
```

A forma geral de um teste é a cabeça de cláusula ou a função Prolog unificável com todas as possíveis invocações do teste, de forma que o símbolo funcional desse seja o próprio nome do teste.

As bases de conhecimento apresentam um predicado teste para cada procedimento Prolog referenciado, inclusive para os predicados e operadores definidos pela linguagem Prolog. Este fato é devido à necessidade de agilizar as operações de consistência efetuadas pelos módulos de construção e edição de bases de conhecimento, pois dispensa a necessidade de um módulo analisador de termos Prolog, além de fornecer às bases de conhecimento um certo grau de independência da implementação.

5. A TAREFA: PROJETO LOGICO DE BLOCOS OPERACIONAIS

O projeto lógico de blocos operacionais de circuitos digitais constitui-se na tarefa que orientou o desenvolvimento do PROTO. Esta é realizada pelo subsistema de projeto lógico, o qual implementa o método de solução de problemas por transformação de redes.

As seções seguintes apresentam cada um dos componentes desse subsistema, descrevendo as tarefas realizadas, as informações solicitadas e as mensagens e resultados fornecidos ao utilizador.

5.1 Controle de sessão

Para iniciar uma sessão do subsistema de projeto lógico o utilizador deve enviar ao processador Prolog a seguinte seqüência de instruções:

```
?-consult('PROTO/PROJETO/LOGICO').
?-projeto_logico.
```

A primeira instrução carrega o módulo de controle do subsistema na memória acessível ao Prolog. A segunda instrução invoca a execução desse módulo, o qual define os operadores especiais do subsistema e envia ao utilizador o seguinte conjunto de opções:

```
Indique a tarefa desejada:
(a) Construcao de rede comportamental.
(b) Construcao de rede estrutural atraves do
    interpretador de regras.
```

(c) Construcao de rede estrutural atraves de rotina de mapeamento direto.
(d) Compilacao de rotina de mapeamento direto.
(e) Projeto logico atraves do interpretador de regras.
(f) Projeto logico atraves de rotina de mapeamento direto.
(g) Termino de sessao.

O utilizador deve selecionar uma das tarefas enviando a letra correspondente. Qualquer resposta distinta da letras (a) à (g) implica no envio da mensagem "Opcao invalida.", seguida do conjunto de opções.

O significado e comportamento dos módulos ativados por cada opção são detalhados a seguir. Após o término de todas as tarefas vinculadas à opção selecionada, o sistema novamente envia o grupo de opções ao utilizador.

5.2 Opção a: construção de rede comportamental

Esta opção invoca o módulo compilador da rede comportamental, o qual gera a rede comportamental de um circuito digital simples a partir da descrição algorítmica do comportamento do mesmo fornecida pelo utilizador.

A seguir apresenta-se o fluxo de mensagens entre o sistema e o utilizador para a construção da rede comportamental de um circuito que realiza divisão de números inteiros positivos através de subtrações sucessivas. Este algoritmo é mostrado na figura 5.1. Mensagens precedidas por "S:" são enviadas do sistema para o utilizador, enquanto mensagens precedidas por "U:" são deste último para o sistema.

```
S: Forneca a origem do algoritmo.
   Caso seja arquivo, forneca seu nome.
   Caso seja terminal, forneca user.
U: 'PROTO/ALGORITMO/DIVISAO1'.
S: Processo de compilacao iniciado.
S: Forneca um nome para a rede comportamental.
U: 'DIVISAO1'
S: Rede comportamental gerada -->
   PROTO/COMPORTAMENTAL/DIVISAO1
```

Observe que o nome da rede comportamental deve obedecer às normas para nomes de arquivos do equipamento Burroughs A10F, sendo que o PROTO acrescenta o diretório PROTO/COMPORTAMENTAL a este. Quaisquer modificações no nome de uma rede comportamental podem ser feitas apenas fora das sessões do subsistema.

Caso o utilizador houvesse fornecido uma descrição algorítmica com problemas sintáticos, o sistema enviaria uma mensagem de erro adequada quando da detecção da primeira incorreção, interromperia o processo de compilação e enviaria a seguinte mensagem antes do retorno ao módulo de controle:

```
S: Detectados erros de sintaxe.
   Rede comportamental nao gerada.
```

5.2.1 A descrição do circuito

A descrição do circuito é um algoritmo escrito numa linguagem composta por um conjunto pequeno de declarações e comandos, capaz de descrever o comportamento de circuitos digitais simples. Esta linguagem permite a declaração dos componentes (portadores) de entrada, saída e armazenamento (registradores) de informações do circuito, bem como comandos de atribuição de valores a registradores (comando de atribuição), ligação de

resultados às saídas do circuito (comando de ligação), de escolhas condicionais de caminhos (comando se) e de interações condicionais (comando enquanto).

```
entr a,b.
sai c,d,e.
reg x,y,w,z.
inicio.
se a < 0.
entao w:=1.
senao se b =< 0.
      entao w:=1.
      senao inicio.
            w:=0.
            z:=0.
            x:=a.
            y:=b.
            enquanto x > y.
            fazer inicio.
                  x:=x - y.
                  z:=z + 1.
                  fim.
            c<=z.
            d<=x.
            fim.
e<=w.
fim.
#
```

Fig. 5.1 - Algoritmo de um circuito que realiza divisão de inteiros positivos através de subtrações sucessivas.

Um algoritmo escrito nessa linguagem traz explicitamente as seguintes informações:

a) os portadores permanentes (registradores) do bloco operacional do circuito;

b) os transformadores aritméticos e lógicos do bloco operacional;

c) a interface do circuito com o ambiente; e

d) o processo executado pelo circuito.

Estão implícitas na descrição algorítmica as seguintes informações:

a) a interconexão de portadores e transformadores;

b) os tipos de sinais manipulados pelo circuito;

c) a interface entre o bloco operacional e o bloco de controle; e

d) os possíveis estados iniciais e finais do circuito.

A linguagem é apresentada a seguir através de diagramas de sintaxe. Ítens entre os símbolos < e > são definidos por outros diagramas ou por textos explicativos entre chaves.

```
<descrição do circuito>:

!---> <declarações> --> <bloco de comandos> ---> # -->!

<declarações>:

  -----------------------------------------------------
  !                                                   !
  V                                                   !
!------> entr ---> <lista de port-entr> ---> . ------>!
    !                                                ^
    !--> sai ---> <lista de port-sai> ---> . --->!
    !                                            !
    !--> reg ---> <lista de port-perm> --> . --->!

<lista de port-entr>:

          ---------------- , <--------------
          !                                !
          V                                !
!-------------------> <port-entr> -------------------->!
```

<lista de port-sai>:

```
                ---------------- , <--------------
                !                                !
                V                                !
!---------------------------> <port-sai> ---------------------->!
```

<lista de port-perm>:

```
                ---------------- , <--------------
                !                                !
                V                                !
!---------------------------> <port-perm> --------------------->!
```

<bloco de comandos>:

```
!---> inicio. ---> <grupo de comandos> ---> fim. ---->!
```

<grupo de comandos>:

```
                 ----------------------------------
                 !                                !
                 V                                !
!---------------------------> <comando> ---------------------->!
```

<comando>:

```
!-----------------> <comando de ligação> ---------------->!
          !                                     ^
          !-------> <comando de atribuição> --->!
          !                                     !
          !-------> <comando se> -------------->!
          !                                     !
          !-------> <comando enquanto> -------->!
          !                                     !
          !-------> <bloco de comandos> ------->!
```

<comando de ligação>:

```
!---> <port-sai> ---> <= ---> <expressão> ---> . ---->!
```

<comando de atribuição>:

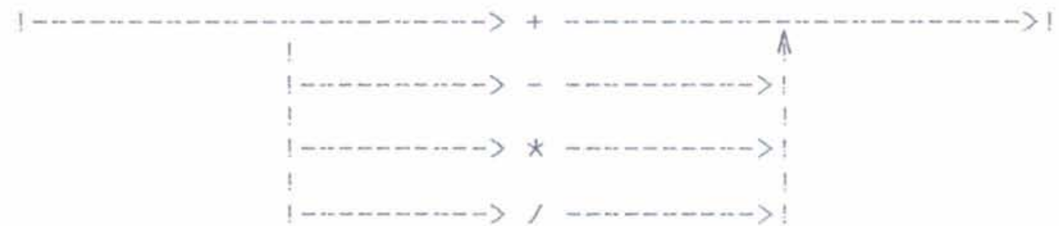
```
!---> <port-perm> ---> := ---> <expressão> ---> . --->!
```

```
<comando se>:

!--------> se ---------> <condição> -------> . ---------->

>--> entao --> <comando> ----------------------------->!
                          !                            ^
                          !--> senao --> <comando> ->!

<comando enquanto>:

!------> enquanto ------> <condição> ------> . ------->

>------> fazer ----------> <comando> --------------->!

<expressão>:

         ------ <operador aritmético> <------
         v                                   !
!--------------> <constante inteira> --------------->!
              !                            ^
              !------> <port-entr> ------->!
              !                            !
              !------> <port-perm> ------->!

<condição>:

         ------- <operador relacional> <------
         v                                   !
!--------------> <constante inteira> --------------->!
              !                            ^
              !------> <port-entr> ------->!
              !                            !
              !------> <port-perm> ------->!

<constante inteira>:

            -----------------------------------
            v                                 !
!--------------------> <dígito> --------------------->!

<port-entr>:

                   ----------------------------
                   v                          !
!---> <letra> ---------------------------------------->!
                        !                 ^
                        !---> <letra> ---->!
                        !                  !
                        !---> <dígito> --->!
```

<port-sai>:

<port-perm>:

<operador aritmético>:



<operador relacional>:

<letra> : {Letra minúscula.}

<dígito> : {0, 1, 2, 3, 4, 5, 6, 7, 8 ou 9}.

5.2.2 A rede comportamental

A rede comportamental é uma rede-TR, que é um formalismo capaz de modelar circuitos lógicos.

As redes-TR apresentam três tipos de portadores: permanentes, temporários e de controle, e quatro tipos de transformadores: modificadores, extratores, de controle e operadores. O PROTO refina esses conceitos, introduzindo também os seguintes elementos: portadores de entrada, portadores de saída, portadores constantes, transformadores ligadores (=>) e transformadores complexos (se e enquanto).

Esses elementos são representados graficamente da seguinte forma:

a) círculos simples:

Representam portadores temporários, de entrada, de saída ou de controle (linhas, barramentos).

b) retângulos simples:

Representam operadores combinacionais (aritméticos, lógicos, multiplexadores, etc) e transformadores complexos (se e enquanto).

c) círculos cortados:

Representam portadores permanentes e portadores constantes (flip-flops, registradores, memórias).

d) retângulos cortados:

Representam operações de acesso a informações armazena-

das permanentemente. Há dois tipos de operadores a nível comportamental: μ , que é o operador de modificação da informação armazenada, e ε , que é o operador de extração da informação armazenada em portadores permanentes e constantes. μ modifica a informação armazenada em um portador permanente, tornando-a igual à informação disponível no portador temporário a que μ está ligado. ε copia a informação armazenada no portador temporário que está a ele conectado como saída.

(e) arcos: ⟶

Representam conexões ou relações de acesso entre portadores e transformadores e definem a topologia da rede.

(f) inscrições:

São palavras e sinais colocados na rede para especificar a semântica de seus elementos.

O PROTO representa esses elementos e suas conexões através do seguinte conjunto de predicados:

(a) port([P,N],Tipo)

Onde: P - identicação do portador;

N - nível do portador; e

Tipo - perm, temp, contr, entr, sai ou const(<valor inteiro sem sinal>).

(b) transf([T,N],Tipo)

Onde: T - identificação do transformador;

N - nível do transformador; e

Tipo - mu, epsi, contr, =>, +, -, *, /, =, >, <,

>=, =<, se ou enq.

(c) port_entr(T,P,I)

Onde: T - identificação de transformador;

P - identificação de portador; e

I - índice da entrada, contr ou cond.

(d) port_sai(T,P,I)

Onde: T - identificação de transformador;

P - identificação de portador; e

I - índice da saída.

(e) inicio_bloco(P)

Onde P é um portador de controle que ativa um **bloco de comandos**.

(f) fim_bloco(P)

Onde P é um portador de controle que caracteriza o fim de um **bloco de comandos**.

(g) entao(P)

Onde P é um portador de controle que ativa o comando associado à cláusula **entao** de um comando **se**.

(h) senao(P)

Onde P é um portador de controle que ativa o comando associado à cláusula **senao** de um comando **se**.

(i) fazer(P)

Onde P é um portador de controle que ativa o comando associado à cláusula **fazer** de um comando **enquanto**.

(j) fim_enq(P)

Onde P é um portador de controle que caracteriza o fim de um comando **enquanto**.

Os predicados gerais de (a) e (b) representam elementos (nodos) da rede, os predicados gerais de (c) e (d) representam relações (arcos) e os predicados de (e) até (j) são marcas (inscrições).

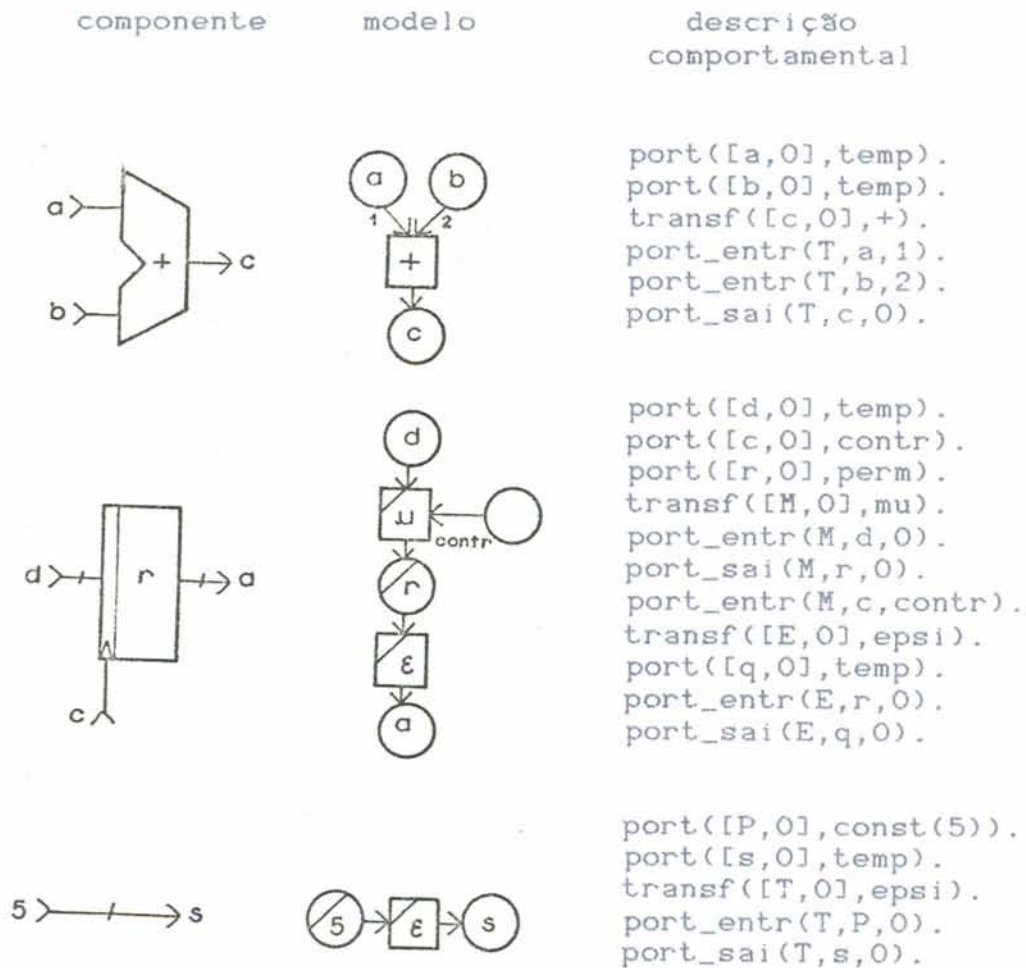


Fig. 5.2 - Modelos em rede comportamental de alguns componentes de circuitos lógicos.

5.2.3 O processo de construção da rede comportamental

A construção da rede comportamental é feita por um processo de compilação em que a descrição algorítmica do circuito é convertida em uma rede-TR. Este processo compõe-se de uma gramática de cláusulas definidas, que analisa a descrição algorítmica, e um conjunto de subprocessos de geração dos predicados que descrevem a rede comportamental correspondente.

A seguir apresenta-se o resultado da compilação de cada tipo de estrutura presente na linguagem de descrição de circuitos.

5.2.3.1 Declarações

São de três tipos: portadores de entrada, portadores de saída e portadores permanentes do tipo registrador de leitura e escrita.

5.2.3.1.1 Portadores de entrada

A declaração de portadores de entrada de dados no circuito é feita pela frase:

entr <lista de portadores de entrada>.

Cada portador P identificado na lista de portadores de entrada dá origem a um elemento descrito pelo predicado port([P,0],entr).

5.2.3.1.2 Portadores de saída

A declaração de portadores de saída de dados do circuito é feita pela frase:

sai <lista de portadores de saída>.

Cada portador P identificado na lista de portadores de saída dá origem a um elemento descrito pelo predicado port([P,0],sai).

5.2.3.1.3 Portadores permanentes

A declaração de portadores permanentes do circuito é feita pela frase:

reg <lista de portadores permanentes>.

Cada portador P identificado na lista de portadores permanentes dá origem a um elemento descrito pelo predicado port([P,0],perm).

Trecho de algorítimo	Trecho da rede comportamental

entr a,b.
sai c.
reg x,y,z.

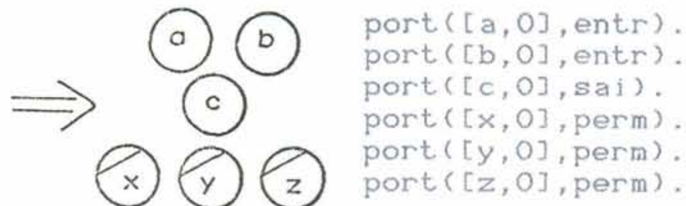


Fig. 5.3 - Exemplo de compilação de declarações de entradas, saídas e registradores de um circuito.

5.2.3.2 Comandos

A linguagem de descrição de circuitos admite cinco tipos de comandos: comandos de **ligação**, comandos de **atribuição**, comandos **se**, comandos **enquanto** e **blocos de comandos**.

As expressões associadas aos comandos de ligação e atribuição, bem como as condições presentes nos comandos se e enquanto, podem ter como operandos portadores permanentes, portadores de entrada e/ou constantes. A cada um desses operandos correspondente como resultados da compilação, respectivamente, instâncias de subredes com operações de extração, ligação e extração de informações. Da mesma forma, expressões com mais de um operando ou condições geram transformadores binários do tipo transf([T,O],OP), onde OP representa um operador aritmético ou relacional, com primeira entrada um portador temporário P1, segunda entrada um portador temporário P2 e saída única um portador temporário P.

5.2.3.2.1 Comando de ligação

Os comandos de **ligação** ligam explicitamente o resultado de expressões a portadores de saída. Apresentam a seguinte forma geral:

<port-sai> <= <expressão>

O operador de ligação **<=** é representado pelo predicado transf([T,O],=>). O resultado da compilação de um comando de ligação é caracterizado na rede comportamental pelo seguinte

grupo de predicados:

```
transf([T,0],=>).

port_entr(T,P,0).

port_sai(T,S,0).
```

A figura seguinte apresenta alguns exemplos de compilação desse comando.

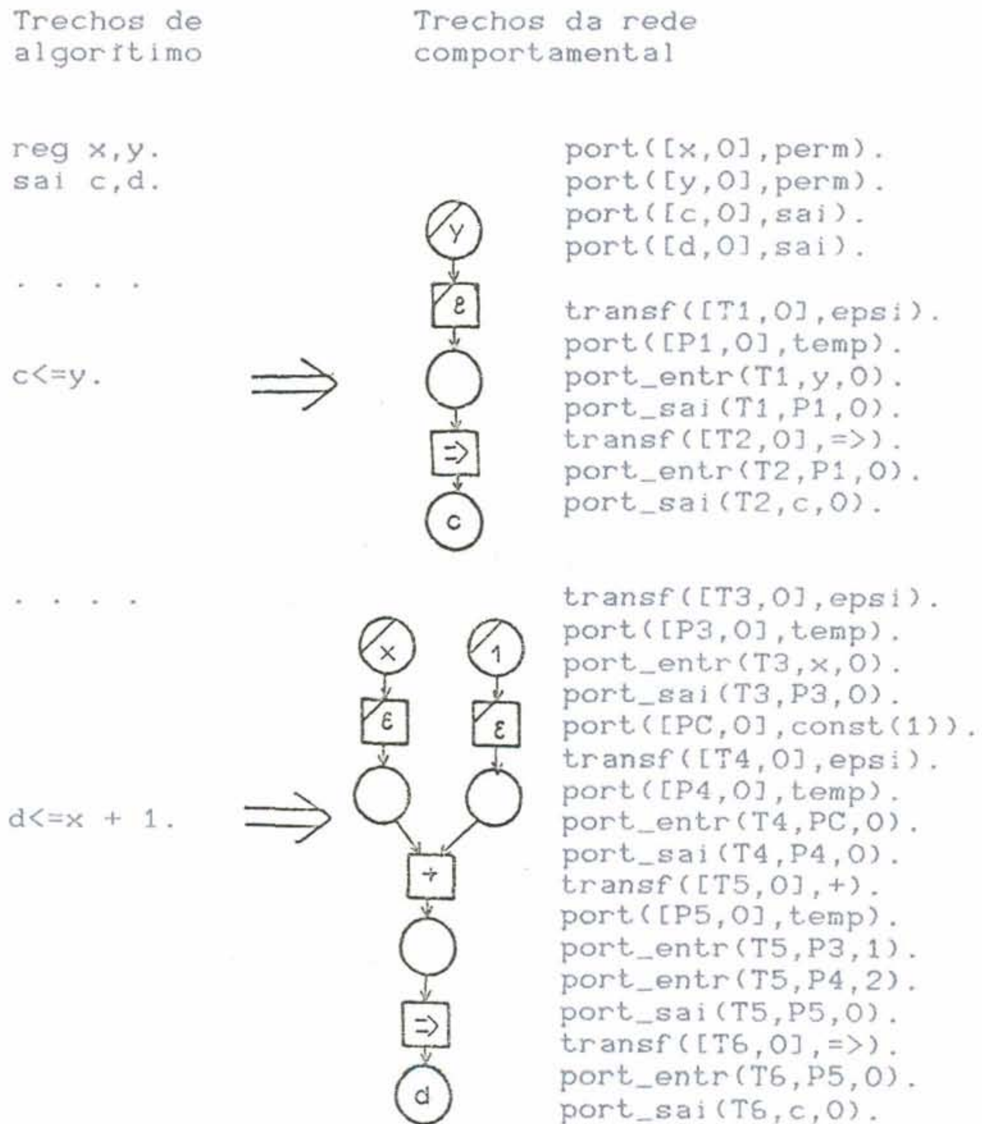


Fig. 5.4 - Exemplos de compilação de comandos de ligação.

O operador de ligação também é gerado quando do uso de portadores de entrada como operandos de expressões e condições. Nesse caso, embora não esteja explicitado no algoritimo, sua presença na rede comportamental indica fluxo de entrada de informações externas ao circuito. Portanto, a presença de predicados transf([T,O],=>) na rede comportamental pode caracterizar tanto entrada quanto saída de dados do circuito, perfeitamente dicerníveis na rede.

5.2.3.2.2 Comando de atribuição

Os comandos de atribuição possibilitam a atribuição de resultados de expressões a portadores permanentes. São da forma:

<port-perm>:=<expressão>.

O operador de atribuição := é representado na rede comportamental através do predicado transf([T,O],mu).

A figura a seguir mostra a representação comportamental genérica de um comando de atribuição. Nesta figura o predicado port([C,O],contr) descreve um portador de controle resultante da compilação de um comando anterior ao comando de atribuição, o qual pode ser um comando se, um comando enquanto, um bloco de comandos ou outro comando de atribuição. Por exemplo, caso houvesse um comando de atribuição imediatamente após este comando genérico (desconsiderando-se comandos de ligação, que não geram descrições de controle), o portador de controle C2 desempenharia para este outro comando a mesma função do portador de controle C.

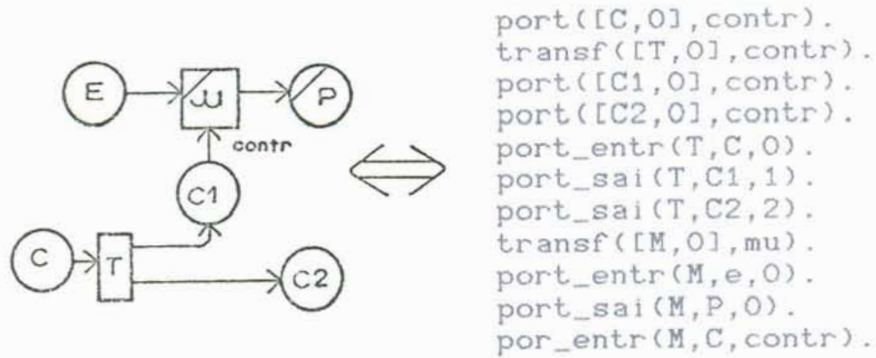


Fig. 5.5 - Representação comportamental do comando de atribuição.

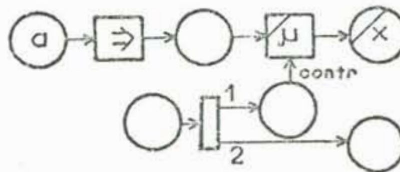
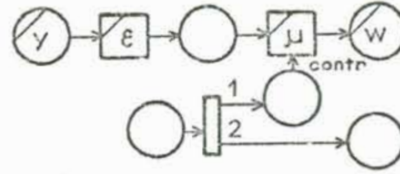
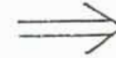
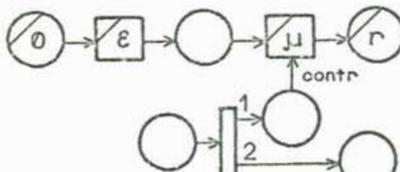
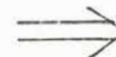
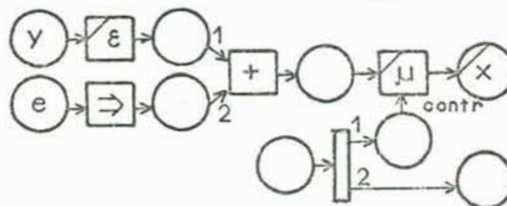
Trechos de algorítimos		Trechos de redes comportamentais
entr a. reg x. . . . . . x:=a.		
reg w,y. . . . . . w:=y.		
reg r. . . . . . r:=0.		
entr e. reg x,y. . . . . . x:=y + e.		

Fig. 5.6 - Exemplos de compilação de comandos de atribuição.

5.2.3.2.3 Comando se

Os comandos se permitem a execução condicional de operações. Podem ser de duas formas:

a) se <condição>.
   entao <comando>.

b) se <condição>.
   entao <comando>.
   senao <comando>.

A condição é uma expressão relacional da forma <operando1> <operador relacional> <operando2>, e o comando pode ser qualquer comando da linguagem, inclusive outro comando se.

A figura a seguir apresenta o esquema de subrede comportamental que caracteriza a compilação de um comando se. Nesta o portador temporário P é resultante da compilação da condição associada, e o portador de controle C é proveniente da compilação de comando anterior ao comando se.

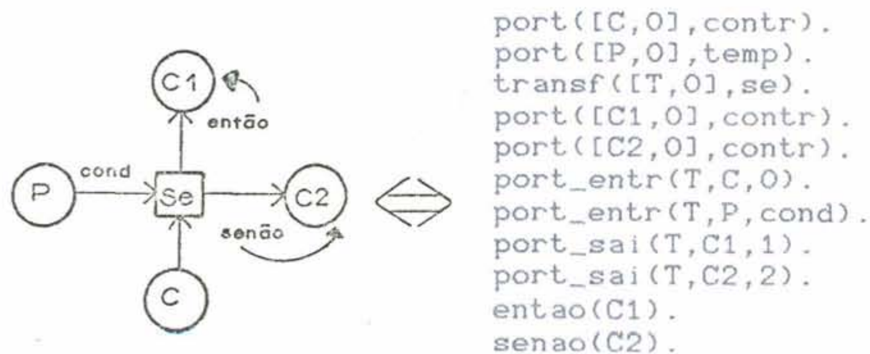
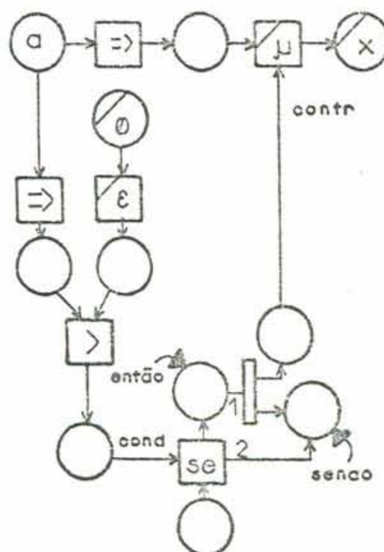


Fig. 5.7 - Representação comportamental do comando se.

Os exemplos da figura 5.8 mostram o resultado da compilação de comandos se.

Trechos de algoritmos | Trechos de redes comportamentais

```
entr a.
reg x.

. . . .

se a > 0.
entao x:=a.
```




```
entr e.
reg y.

. . . .

se e=< 0.
entao y:=y + 1.
senao y:=e.
```

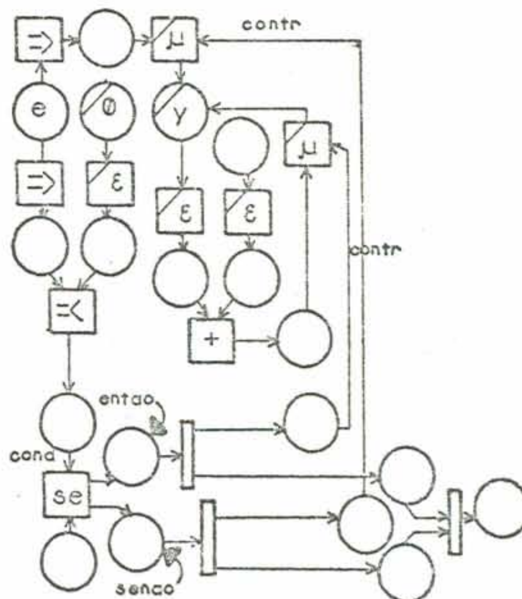


Fig. 5.8 - Exemplos de compilação de comandos se.

5.2.3.2.4 Comandos enquanto

Os comandos **enquanto** possibilitam a execução interativa de um comando ou bloco de comandos. São da forma:

enquanto <condição>.

fazer <comando>.

A compilação de comandos enquanto pode ser visualizada através do exemplo mostrado na figura seguinte.

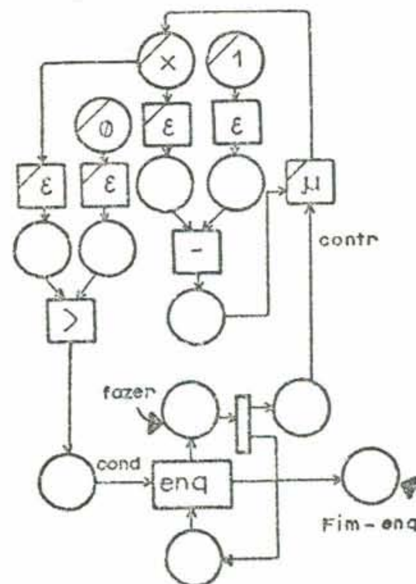
Trecho do algoritmo	Trecho da rede comportamental
reg x. . . . . enquanto x > 0. fazer x:=x - 1.	

Fig. 5.9 - Exemplo de compilação de comando enquanto.

Um comando **enquanto** está caracterizado na rede comportamental por um predicado transf([T,0],enq), ao qual estão conec-

tados um portador temporário P que traz o resultado da condição, um portador de controle C que ativa o transformador T, um portador de controle C1 que ativa o comando associado à cláusula **fazer** e um portador de controle que C2 é ativado quando a condição deixar de ser verdadeira. A figura 5.10 apresenta o esquema de subrede que caracteriza um comando enquanto.

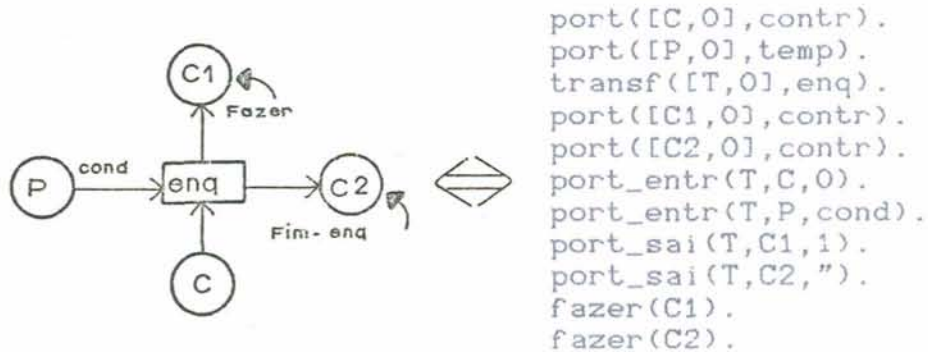


Fig. 5.10 - Representação comportamental do comando enquanto.

5.2.3.2.5 Bloco de comandos

Blocos de comandos têm a forma:

inicio.

<grupo de comandos>

fim.

Um exemplo de compilação dessa estrutura é mostrado na figura seguinte.

Algoritmo

```
reg x,y.
inicio.
x:=0.
y:=0.
fim.
```

Rede comportamental

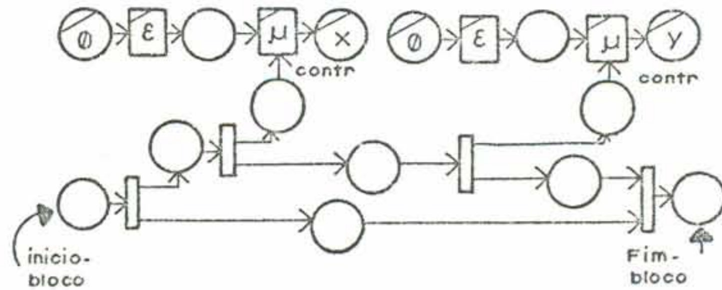
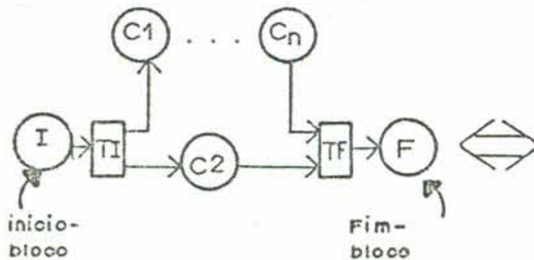


Fig. 5.11 - Exemplo de compilação de um bloco de comandos.

Qualquer descrição algorítmica apresenta pelo menos um bloco de comandos, o qual gera apenas elementos do bloco de controle da rede comportamental. A representação comportamental gráfica e em forma de predicados dessa estrutura é apresentada na figura 5.12.




```
port([I,0],contr).

transf([TI,0],contr).
port([C1,0],contr).
port([C2,0],contr).
port_entr(TI,I,1).
port_sai(TI,C1,1).
port_sai(TI,C2,2).
inicio_bloco(I).

port([Cn,0],contr).

transf([TF,0],contr).
port([F,0],contr).
port_entr(TF,Cn,1).
port_entr(TF,C2,2).
port_sai(TF,F,0).
fim_bloco(F).
```

Fig. 5.12 - Representação comportamental de bloco de comandos.

5.2.3.3 Separação da rede comportamental

Uma rede comportamental é logicamente separada em dois blocos: bloco operacional e bloco de controle. Isto é feito pelo processo de compilação por meio da marca **interface(P)** sobre todos os portadores P que constituem a ligação entre estes dois blocos.

São constituintes do bloco operacional:

(a) portadores permanentes;

(b) portadores temporários;

(c) portadores de entrada;

(d) portadores de saída;

(e) transformadores aritméticos (+,-, * e /);

(f) transformadores relacionais (=, >, <, >= e =<);

(g) transformadores de ligação (=>);

(h) transformadores de extração (epsi);

(i) transformadores de modificação (mu);

(j) arcos que ligam os elementos de (a) até (i).

São constituintes do bloco de controle:

(k) portadores de controle;

(l) transformadores de controle;

(m) transformadores se;

(n) transformadores enquanto; e

(o) arcos que ligam os elementos de (k) à (n).

Constituem a interface entre os blocos operacional e de controle:

(p) portadores de controle que ativam modificadores;

(q) portadores temporários que são condições de de transformadores se; e

(r) portadores temporários que são condições de transformadores enquanto.

A figura 5.13 apresenta a representação gráfica da rede comportamental gerada pela compilação do algoritmo da figura 5.1. Nesta, a linha tracejada indica os elementos que constituem a interface entre os blocos operacional e de controle, de forma que acima desta linha encontra-se a rede comportamental do bloco operacional do circuito e abaixo está a rede comportamental do bloco de controle.

Todos os identificadores P e T dos predicados port([P,0],Tipo) e transf([T,0],Tipo), exceto os da forma port([P,0],entr), port([P,0],sai) e port([P,0],perm), são constantes inteiras. Isto ocorre porque o processo de compilação incrementa o valor V contido num predicado valor_id(V) em uma unidade quando da criação de um elemento da rede, usando este valor V como seu identificador. O valor inicial de V é 0. Este predicado é inserido na rede comportamental como um parâmetro

para o processo que construirá uma rede estrutural a partir dela, informando o valor da identificação do último elemento comportamental gerado.

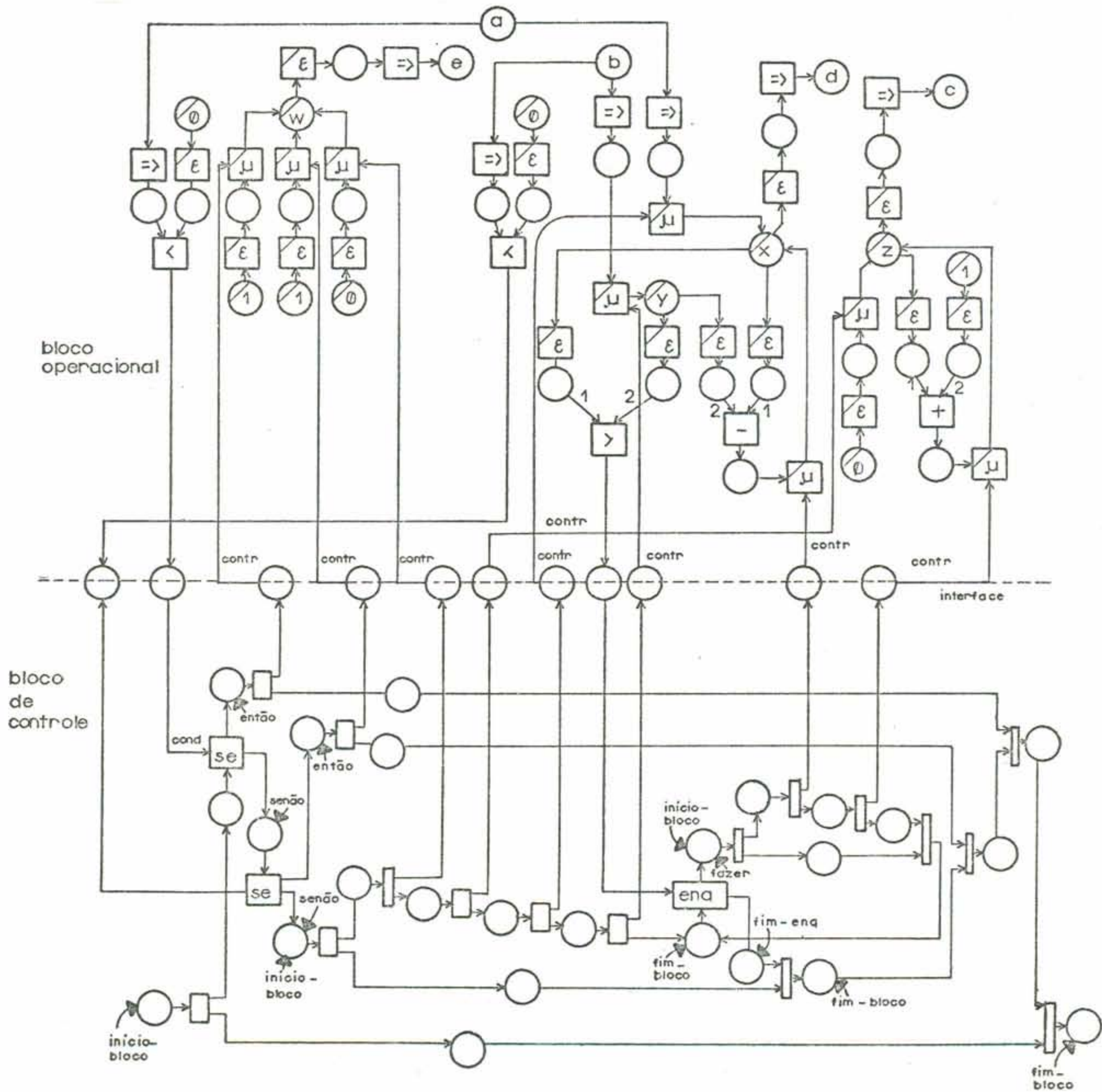


Fig. 5.13 - Rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1: divisão de inteiros positivos através de subtrações sucessivas.

5.2.3.4 Mensagens de erro

As possíveis mensagens de erro enviadas ao utilizador pelo processo de construção da rede comportamental são as seguintes:

(a) Identificador invalido.

Quando a entrada, saída ou registrador do circuito não obedecer às regras para definição de identificadores.

(b) Identificador duplicado.

Quando o identificador já foi usado na declaração de entrada, saída ou registrador anterior.

(c) Comando invalido.

(d) Expressao invalida.

(e) Condicao invalida.

(f) Operador relacional invalido.

5.3 Opção b: construção de rede estrutural através do interpretador de regras

Esta opção invoca o módulo interpretador de regras, o qual gera uma rede estrututural do bloco operacional de um circuito a partir da rede comportamental ou estrutural intermediária do mesmo. Com este objetivo, o interpretador de regras reconhece e executa um conjunto de regras de transformação de redes contido numa base de conhecimento. O estilo de aplicação das regras e a base de conhecimento também são indicados pelo

utlizador, conforme é mostrado no exemplo seguinte.

```
S: Forneca a rede de entrada.
U: 'PROTO/COMPORTAMENTAL/DIVISAO1'.
S: Forneca o nome da base de conhecimento.
U: 'PROTO/BC'.
S: Indique o tipo da rede de saida:
   (a) Rede final.
   (b) Rede parcial.
U: a.
S: Forneca o estilo de geracao da rede estrutural.
U: 'L'.
S: estilo: basico
           Sequencia de regras que efetuam o mapeamento
           rede  comportamental  -->  rede  estrutural
           basica.
   estilo: estendido
           Sequencia  de regras que estendem  o  estilo
           basico  ao  nivel  de deslocadores  de  bits
           (direita   e   esquerda),    incrementadores,
           decrementadores e sinais de reset.
   Forneca o estilo de geracao da rede estrutural.
U: basico.
S: Voce quer monitorar a execucao das regras?
   Responda sim ou nao.
U: nao.
S: Construcao da rede iniciada.
S: Forneca um nome para a rede final.
U: 'DIVISAO1'.
S: Rede final gerada ---> PROTO/ESTRUTURAL/DIVISAO1.
```

Este exemplo de execução ilustra as seguintes características do interpretador de regras:

(a) Possibilidade de listar todos os estilos de geração da rede estrutural presentes na base de conhecimento através da resposta 'L' à solicitação do estilo.

(b) Possibilidade de monitorar a execução das regras.

(c) Opção de gerar redes parciais, as quais podem ser entradas no interpretador numa execução futura. A distinção entre redes parciais e redes finais encontra-se na representação dos

seu elementos (nodos). Elementos de redes parciais têm seus níveis de mapeamento explicitados, ao passo que nas redes finais esta informação é suprimida.

(d) O nome da rede gerada ao final do processo ser o nome fornecido pelo utilizador acrescido do diretório PROTO/ESTRUTURAL.

5.3.1 A rede estrutural

A rede estrutural é, basicamente, uma rede-TR. Entretanto, os predicados que a descrevem e a simbologia gráfica adotada diferem dos utilizados para a rede comportamental.

Graficamente tem-se os seguintes elementos:

Representam linhas ou sinais de entrada, saída, controle, condição, etc.

Representam operadores em geral: aritméticos, relacionais, seletores, etc.

Representam elementos de armazenamento de informações: flip-flops, registradores, memórias.

Representam conexões ou relações de acesso a elementos da rede, definindo a sua topologia.

(e) inscrições:

São marcas presentes na rede que especificam a semântica de seus elementos.

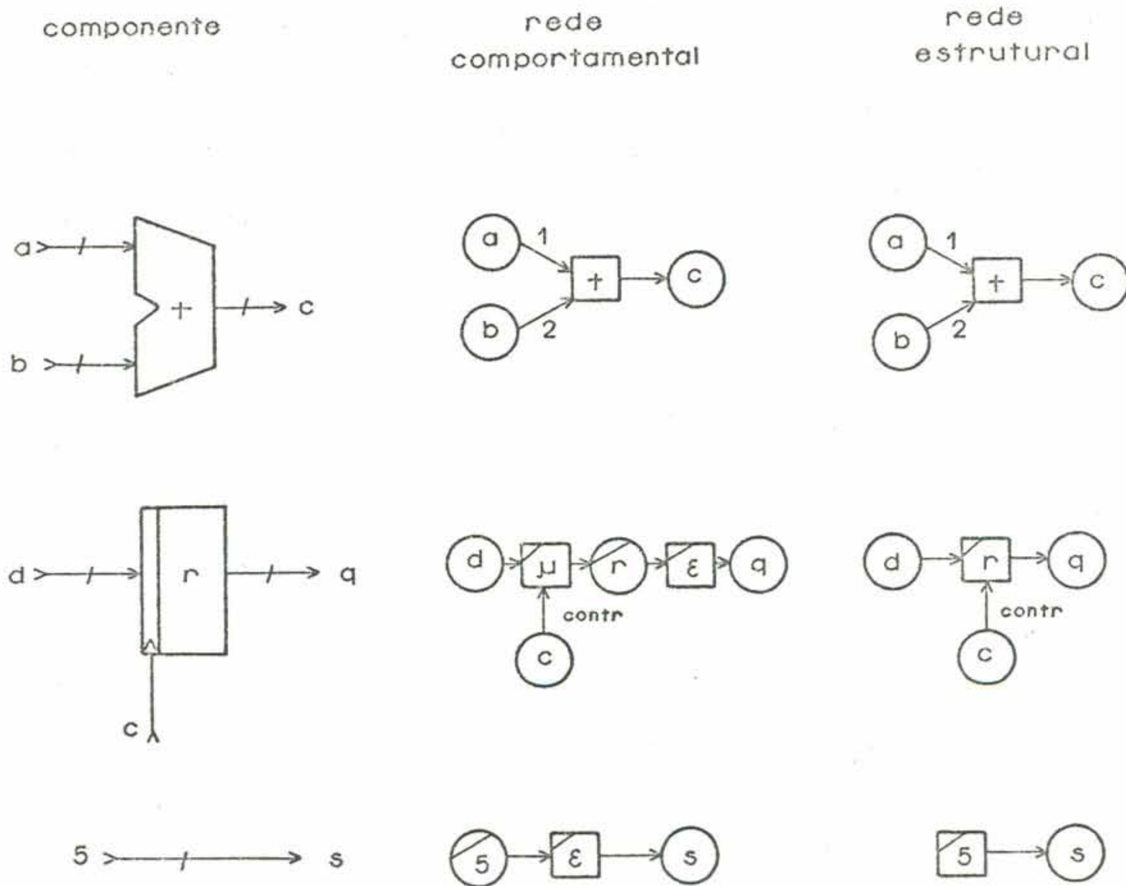


Fig. 5.14 - Modelos em rede comportamental e estrutural de alguns componentes de circuitos lógicos.

Os predicados presentes na base PROTO/BC que descrevem a rede estrutural são os seguintes:

(a) reg([R,N])

Especifica um registrador de leitura e escrita R, gera-

do no nível N de mapeamento.

(b) reg([R,N],const(C))

Especifica um registrador de apenas leitura R, gerado no nível N de mapeamento, que armazena a informação constante C.

(c) lin([L,N])

Especifica uma linha ou conjunto de linhas de sinal ou dados L gerada no nível N de mapeamento.

(d) oper([O,N],T)

Representa um operador O, gerado no nível N, com tipo especificado por T. Nesta base T pode ser: +, -, *, /, =, >, <, >=, =<, sel, +1, -1, -> ou <-.

(e) entr(E,L,I)

Indica a relação: L é linha de entrada no elemento E, sendo o tipo da informação trazida por L dada pelo índice I. I pode ser: valor inteiro sem sinal (quando dados), contr, cond, sel ou reset. Quando L for entrada única de dados, o valor de I será 0.

(f) sai(E,L,I)

Indica a relação: L é linha de saída do elemento E, sendo o índice da saída dado pelo valor inteiro sem sinal I. Quando L for saída única de E, o valor de I será 0.

(g) entrada_circ(L)

Inscrição que caracteriza a linha L como uma entrada de dados no circuito.

(h) saida_circ(L)

Inscrição que caracteriza a linha L como uma saída de dados do circuito.

(i) sel(S)

Inscrição que indica sinal de seleção sobre o seletor S.

5.3.2 O processo de construção da rede estrutural

A construção da rede estrutural do bloco operacional do circuito é feita através de um processo que reconhece e executa a seqüência de regras de transformação de redes contidas no estilo escolhido pelo utilizador. É um processo de construção paulatina da rede final, onde cada regra é aplicada sobre todas as instâncias de subrede de nível n que satisfazem sua condição, gerando instâncias de subrede de nível n+1 ou complementando instâncias de subrede de nível n.

O bloco operacional completo da rede comportamental é mapeado para uma rede estrutural de nível 1, chamada de rede estrutural básica. A partir dessa pode haver uma superposição de subredes de níveis superiores a 1, não havendo necessariamente mapeamentos completos da rede estrutural básica para essas redes complementares. Dessa forma, a rede estrutural do bloco operacional do circuito pode ser formada por elementos de diversos níveis n, n >= 1.

O mapeamento entre os diversos níveis de subredes é feito pelo predicado básico de mapeamento **map(E,E')**, onde E é um

elemento de nível n e E´ é um elemento de nível n+1.

O processo de construção da rede estrutural tambem pode gerar elementos pertencentes à rede comportamental do bloco de controle. Estes elementos são resultantes da criação de subredes do bloco operacional que requerem modificações na estrutura do controle. Normalmente são originados do mapeamento de portadores de controle que constituem a interface entre os blocos operacional e de controle do circuito, sendo descritos pelos predicados port([P,N],contr) e transf([T,N],contr), com N >= 1.

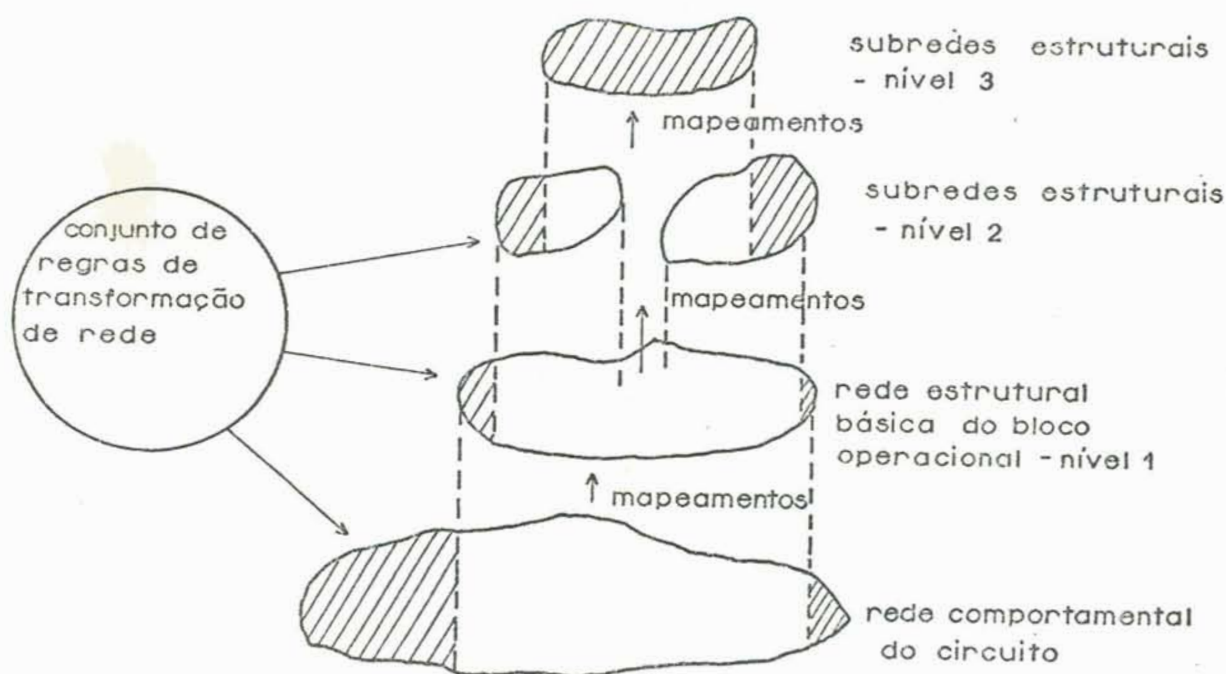


Fig. 5.15 - O processo de construção da rede estrutural (final ou parcial): bloco operacional estrutural + bloco de controle comportamental.

5.3.2.1 Interpretação das regras de construção de uma rede estrutural básica

Nesta seção apresenta-se a seqüência de aplicação das regras de transformação de redes do estilo **basico**, presente na base de conhecimento PROTO/BC, e a rede final PROTO/ESTRUTURAL/DIVISAO1, gerada a partir da rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1.

O estilo **basico** apresenta a seguinte seqüência de regras:

(a) map_permanentes;

(b) map_constantes;

(c) map_entradas;

(d) map_ligadores_entrada;

(e) map_extratores;

(f) map_operadores;

(g) map_modif_simples;

(h) map_modif_multiplo1;

(i) map_modif_multiplo2; e

(j) map_saidas.

Cada uma dessas regras é descrita a seguir em função das condições para sua aplicação e das subredes resultantes. Nas

figuras, as linhas tracejadas indicam os mapeamentos entre os elementos, e as linhas pontilhadas, quando aparecem, separam os predicados que constituem as condições dos predicados resultantes da aplicação da regra.

5.3.2.1.1 Regra: map_permanentes

Gera instancias de subredes com registradores de leitura e escrita, mapeados de portadores permanentes.

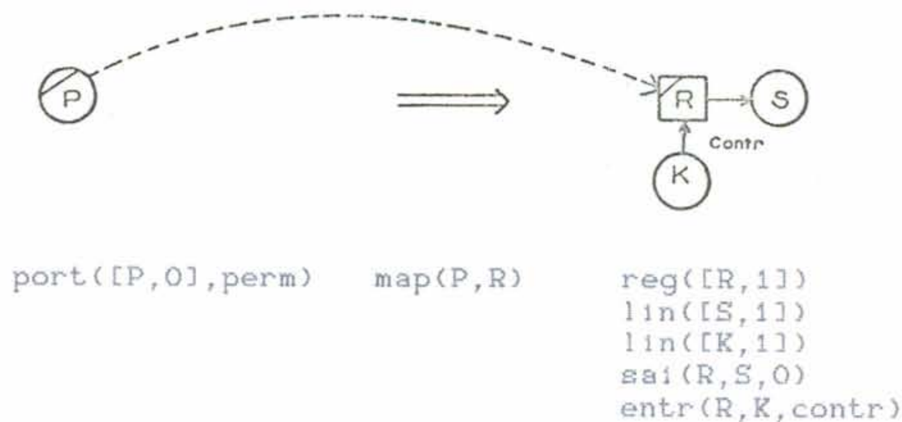


Fig. 5.16 - Regra: map_permanentes.

5.3.2.1.2 Regra: map_constantes

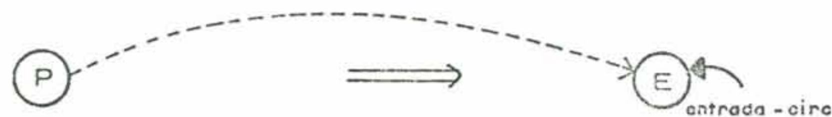
Gera subredes com registradores constantes, mapeados a partir de portadores do tipo port([P,0],const(C)).

```
port([P,0],const(C))  map(P,R)  reg([R,1],const(C))
                                lin([S,1])
                                sai(R,S,0)
```

Fig. 5.17 - Regra: map_constantes.

5.3.2.1.3 Regra: map_entradas

Gera linhas a partir do mapeamento de portadores de entrada, marcando-as como entrada do circuito.

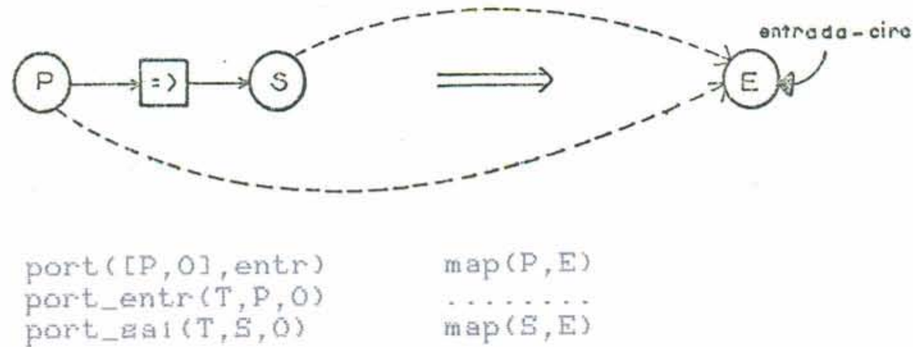



```
port([P,0],entr)     map(P,E)     lin([E,1])
                                  entrada_circ(E)
```

Fig. 5.18 - Regra: map_entradas.

5.3.2.1.4 Regra: map_ligadores_entrada

Mapeia instâncias de subredes com operações de ligação de entradas ao circuito para as linhas correspondentes marcadas como entrada do circuito.

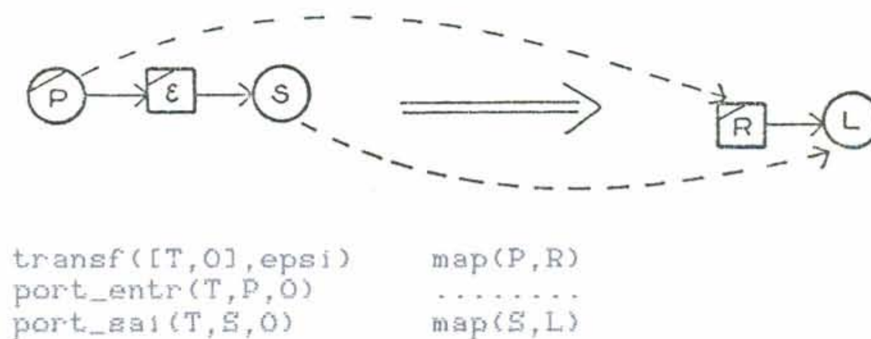



```
port([P,0],entr)       map(P,E)
port_entr(T,P,0)       .........
port_sai(T,S,0)        map(S,E)
```

Fig. 5.19 - Regra: map_ligadores_entrada.

5.3.2.1.5 Regra: map_extratores

Mapeia subredes com operações de extração de informação de portadores permanentes e constantes para as saídas dos registradores correspondentes.

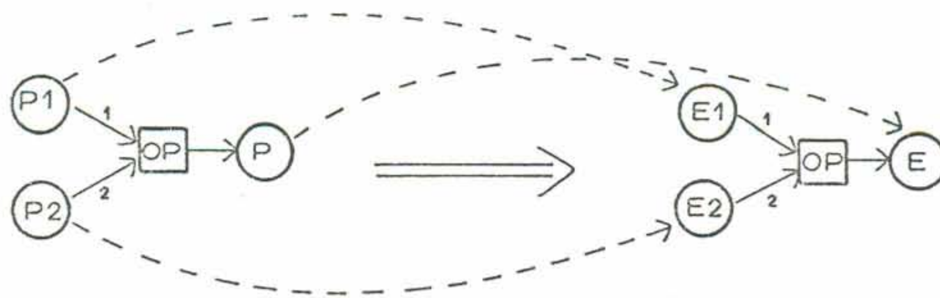



```
transf([T,0],epsi)     map(P,R)
port_entr(T,P,0)       .........
port_sai(T,S,0)        map(S,L)
```

Fig. 5.20 - Regra: map_extratores.

5.3.2.1.6 Regra: map_operadores

Gera instâncias de subredes com operações aritméticas ou relacionais, mapeadas de suas correspondentes a nível comportamental.

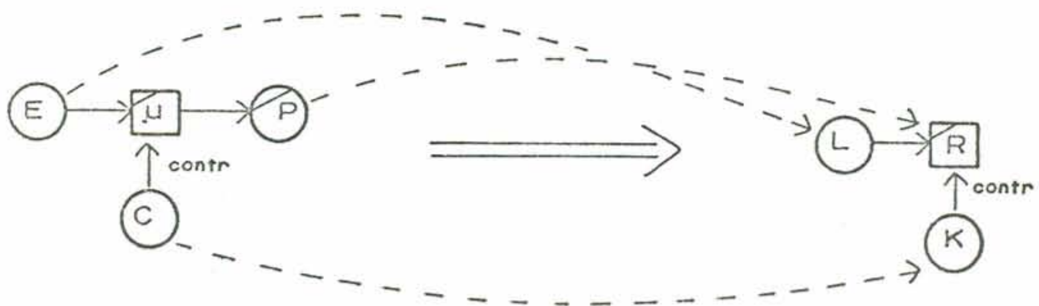



```
transf([T,O],OP)        map(P1,E1)      oper([O,1],OP)
operador(OP)            map(P2,E2)      lin([E,1])
port_entr(T,P1,1)       ..........      entr(O,E1,1)
port_entr(T,P2,2)       map(P,E)        entr(O,E2,2)
port_sai(T,P,O)                         sai(T,E,O)
```

Fig. 5.21 - Regra: map_operadores.

5.3.2.1.7 Regra: map_modif_simples

Mapeia subredes que apresentam apenas uma operação de modificação por portador permenente.

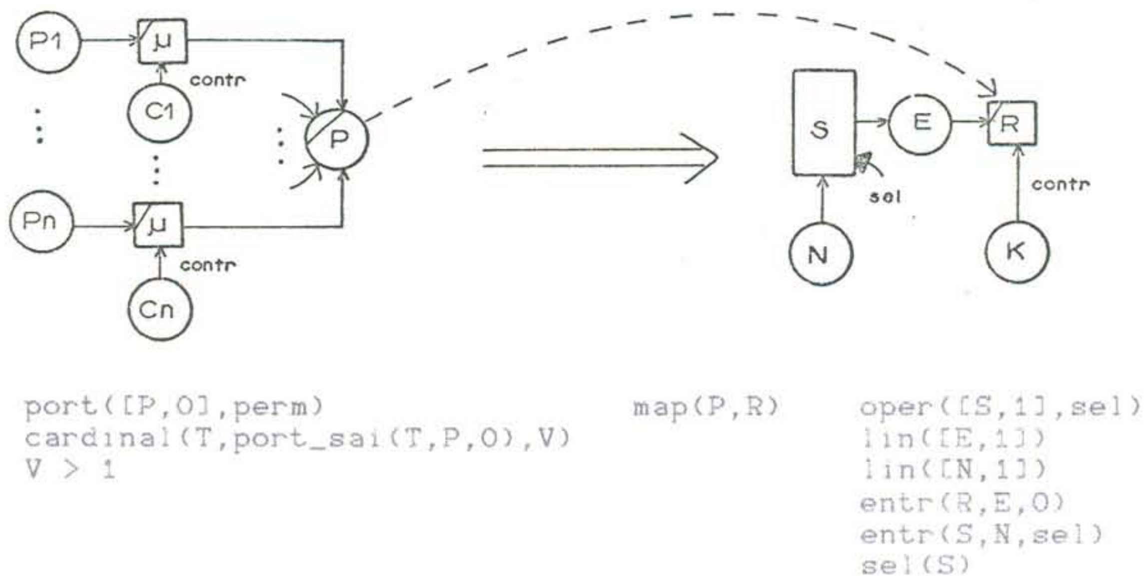



```
port([P,O],perm)                    map(P,R)     entr(R,K,contr)
cardinal(T,port_sai(T,P,O),1)       map(P,L)     ...............
port_sai(T,P,O)                     ........     entr(R,L,O)
port_entr(T,E,O)                    map(C,K)
port_entr(T,C,contr)
```

Fig. 5.22 - Regra: map_modif_simples.

5.3.2.1.8 Regra: map_modif_multiplo1

Gera seletores de entrada correspondendentes às subredes comportamentais que apresentam mais de uma operação de modificação por portador permanente.




```
port([P,0],perm)                    map(P,R)    oper([S,1],sel)
cardinal(T,port_sai(T,P,0),V)                   lin([E,1])
V > 1                                           lin([N,1])
                                                entr(R,E,0)
                                                entr(S,N,sel)
                                                sel(S)
```

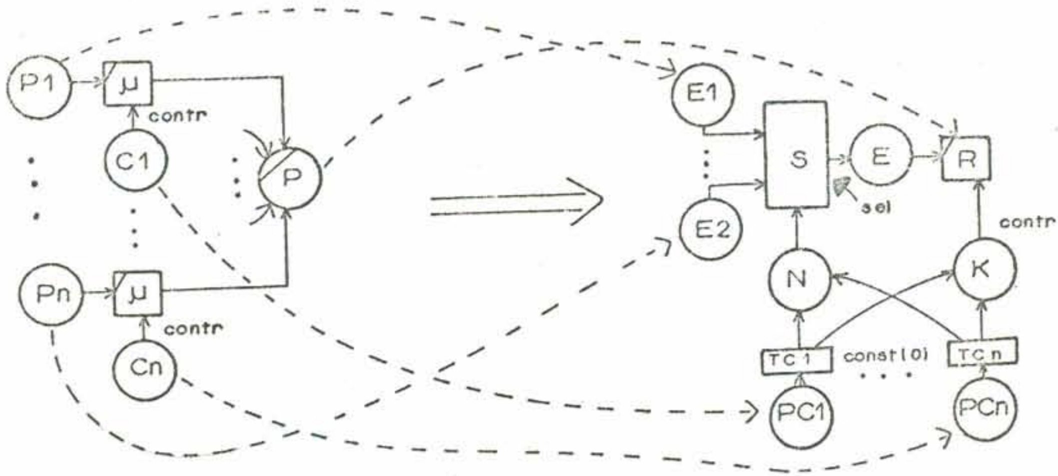
Fig. 5.23 - Regra: map_modif_multiplo1

5.3.2.1.9 Regra: map_modif_multiplo2

Esta regra liga as entradas aos seletores gerados pela regra anterior, acrescentando elementos ao bloco de controle.

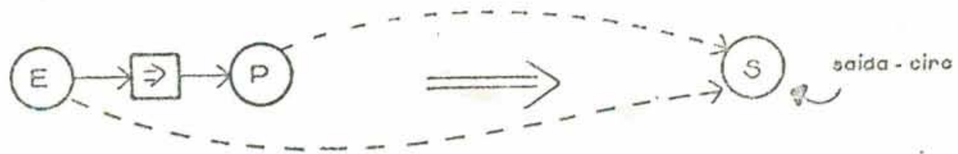
5.3.2.1.10 Regra: map_saidas

Mapeia instâncias de subrede com operações de ligação de portadores temporários a portadores de saída para as linhas de saída do circuito correspondentes.




```
port([P,0],perm)          map(P,R)       entr(R,E,0)
ordem_inicial(0)          map(Pi,Ei)     sai(S,E,0)
port_sai(T,P,0)           ...........    entr(R,K,contr)
port_entr(T,Pi,0)         map(Ci,Pc)     entr(S,N,sel)
port_entr(T,Ci,contr)     map(Ci,Tc)     ......................
ordem(0)                                 transf([Tc,1],contr)
                                         port([Pc,1],contr)
                                         port_entr(Tc,Pc,0)
                                         port_sai(Tc,K,0)
                                         port_sai(Tc,N,const(0))
                                         entr(S,Ei,const(0))
```

Fig. 5.24 - Regra: map_modif_multiplo2.




```
port([P,0],sai)       map(E,S)      saida_circ(S)
port_sai(T,P,0)       ........
port_entr(T,E,0)      map(P,S)
```

Fig. 5.25 - Regra: map_saidas.

5.3.2.2 A rede resultante

A rede gerada pelo processo de interpretação de regras apresenta o bloco operacional do circuito a nível estrutural, e o bloco de controle a nível comportamental. Este último pode apresentar acréscimos de elementos em relação ao seu correspondente na rede comportamental. A rede mostarada na figura 5.26 ilustra bem esta situação. Estes acréscimos dependem do estilo de aplicação das regras escolhido pelo utilizador. Por exemplo, a rede final gerada a partir da rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1 no estilo **estendido**, acrescenta menos elementos ao bloco de controle que sua correspondente no estilo **basico** (ver figura 5.27). Contudo, ambas reduzem o número de elementos presentes bloco operacional do circuito em relação ao seu equivalente comportamental. Isto é uma constante no processo de geração de redes estruturais: uma rede estrutural é descrita sempre por conjunto menor de predicados que a sua correspondente a nível comportamental, pois os elementos estruturais possuem significados funcionais implícitos.

5.3.3 Monitoração da execução das regras

Quando o utilizador indica ao interpretador de regras que deseja monitorar a execução do processo de construção da rede estrutural, são listadas toda as instâncias de subrede que satisfazem às condições de cada regra e todas as instâncias de subredes geradas. O formato das informações para cada regra interpretada é o seguinte:

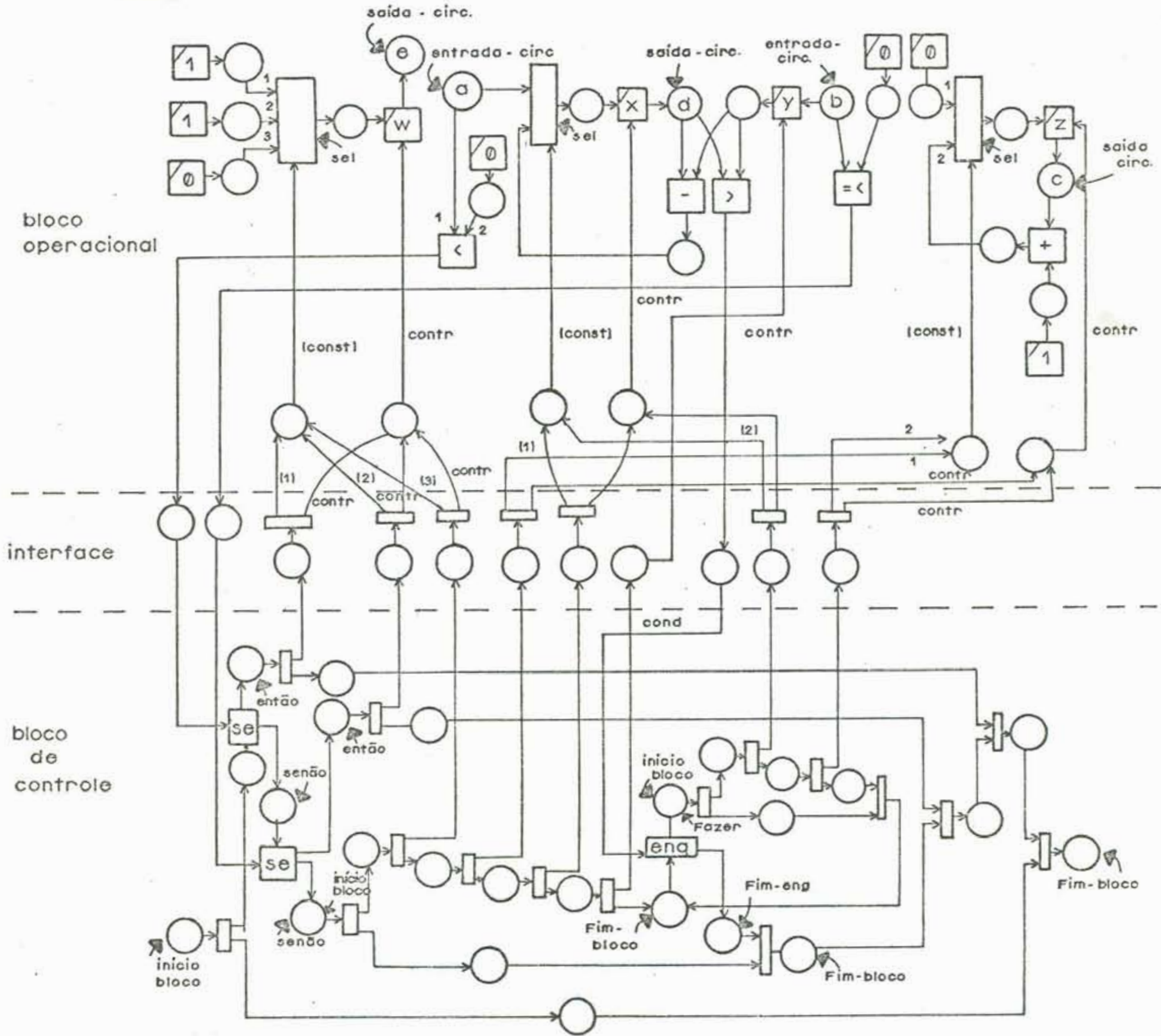


Fig. 5.26 - Rede final PROTO/ESTRUTURAL/DIVISAO1: gerada no estilo basico a partir da rede comportamental PROTO/COMPORTAMENTAL/DIVISAO1.

regra:<nome da regra>.

condicao:<forma geral da condição>.

acoes:<forma geral das ações>.

instancia de subrede:<primeira condição detectada>.

procedimentos:<ações efetuadas>.

. . . .

instancia de subrede:<n-ésima condição detectada>.

procedimentos:<ações efetuadas>.

A sessão do subsistema de projeto lógico apresentada a seguir exemplifica o funcionamento da opção de monitoração.

```
U: projeto_logico.
S: Indique a tarefa desejada:
   (a) Construcao de rede comportamental.
   (b) Construcao   de   rede   estrutural  atraves  do
       interpretador de regras.
   (c) Construcao de rede estrutural atraves de  rotina
       de mapeamento direto.
   (d) Compilacao de rotina de mapeamento direto.
   (e) Projeto logico com interpretador de regras.
   (f) Projeto logico com rotina de mapeamento direto.
   (g) Termino da sessao.
U: a.
S: Forneca a origem do algoritmo.
   Caso seja arquivo, forneca seu nome.
   Caso seja terminal, forneca user.
U: user.
S: Forneca a descricao do circuito.
   Lembre-se de encerra-la com #.
U: entr a,b.
   reg x.
   sai c.
   inicio.
   x:=a + b.
   c<=x.
   fim.
   #
S: Processo de compilacao iniciado.
S: Forneca um nome para a rede comportamental.
U: 'SOMA'.
S: Rede comportamental gerada -->
   PROTO/COMPORTAMENTAL/SOMA
S: Indique a tarefa desejada:
   (a) Construcao de rede comportamental.
```

```
   (b) Construcao de rede estrutural atraves do
       interpretador de regras.
   (c) Construcao de rede estrutural atraves de rotina
       de mapeamento direto.
   (d) Compilacao de rotina de mapeamento direto.
   (e) Projeto logico com interpretador de regras.
   (f) Projeto logico com rotina de mapeamento direto.
   (g) Termino da sessao.
U: b.
S: Forneca a rede de entrada.
U: 'PROTO/COMPORTAMENTAL/SOMA'.
S: Forneca a base de conhecimento.
U: 'PROTO/BC'.
S: Indique o tipo da rede de saida:
   (a) Rede final.
   (b) Rede parcial.
U: b.
S: Forneca o estilo de geracao da rede estrutural.
U: basico.
S: Deseja monitorar a execucao das regras?
   Responda sim ou nao.
U: sim.
S: Construcao da rede iniciada.
S: Regra: map_permanentes
   Condicoes: port([P,0],perm)
   Acoes: criar(reg([R,1]))
          criar(lin([S,1]))
          criar(lin([C,1]))
          marcar(map(P,R))
          marcar(sai(R,S,0))
          marcar(entr(R,C,contr))
   Instancia de subrede: port([x,0],perm)
   Procedimentos: criar(reg([20,1]))
                  criar(lin([21,1]))
                  criar(lin([22,1]))
                  marcar(map(x,20))
                  marcar(sai(20,21.0))
                  marcar(entr(20,21,contr))
   Regra: map_constantes
   Condicoes: port([P,0],const(C))
   Acoes: criar(reg([R,1],const(C)))
          criar(lin([S,1]))
          marcar(map(P,R))
          marcar(sai(R,S,0))
   Regra: map_entradas
   Condicoes: port([P,0],entr)
   Acoes: criar(lin([E,1]))
          marcar(entrada_circ(E))
          marcar(map(P,E))
   Instancia de subrede: port([b,0],entr)
   Procedimentos: criar(lin([23,1]))
                  marcar(entrada_circ(23))
                  marcar(map(b,23))
   Instancia de subrede: port([a,0],entr)
   Procedimentos: criar(lin[24,1]))
```

```
                 marcar(entrada_cir(24))
                 marcar(map(a,24))
Regra: map_ligadores_entrada
Condicoes: port([P,0],entr)
           map(P,E)
           port_entr(T,P,0)
           port_sai(T,S,0)
Acoes: marcar(map(S,E))
Instancia de subrede: port([b,0],entr)
                      map(b,23)
                      port_entr(7,b,0)
                      port_sai(7,8,0)
Procedimentos: marcar(map(8,23))
Instancia de subrede: port([a,0],entr)
                      map(a,24)
                      port_entr(5,a,0)
                      port_sai(5,6,0)
Procedimentos: marcar(map(6,24))
Regra: map_extratores
Condicoes: transf([T,0],epsi)
           port_entr(T,P,0)
           port_sai(T,S,0)
           map(P,R)
           sai(R,L,0)
Acoes: marcar(map(S,L))
Instancia de subrede: transf([15,0],epsi)
                      port_entr(15,x,0)
                      port_sai(15,16,0)
                      map(x,20)
                      sai(20,21,0)
Procedimentos: marcar(map(16,21))
Regra: map_operadores
Condicoes: transf([T,0],OP)
           operador(OP)
           port_entr(T,P1,1)
           port_entr(T,P2,2)
           port_sai(T,P,0)
           map(P1,E1)
           map(P2,E2)
Acoes: criar(oper([O,1],OP))
       criar(lin([S,1]))
       marcar(entr(O,E1,1))
       marcar(entr(O,E2,2))
       marcar(sai(O,S,0))
       marcar(map(P,S))
Instancia de subrede: transf([9,0],+)
                      operador(+)
                      port_entr(9,6,1)
                      port_entr(9,8,2)
                      port_sai(9,10,0)
                      map(6,24)
                      map(8,23)
Procedimentos: criar(oper([25,1],+))
               criar(lin([26,1]))
               marcar(entr(25,24,1))
```

```
                marcar(entr(25,23,2))
                marcar(sai(25,26,0))
                marcar(map(10,26))
Regra: map_modif_simples
Condicoes: port([P,0],perm)
           cardinal(T,por_sai(T,P,0),1)
           port_sai(T,P,0)
           port_entr(T,E,0)
           port_entr(T,C,contr)
           map(P,R)
           map(E,L)
           entr(R,K,contr)
Acoes: marcar(entr(R,L,0))
       marcar(map(C,K))
Instancia de subrede: port([x,0],perm)
                      cardinal(T,port_sai(T,x,0),1)
                      port_sai(14,x,0)
                      port_entr(14,10,0)
                      port_entr(14,12,contr)
                      map(x,20)
                      map(10,26)
                      entr(20,22,contr)
Procedimentos: marcar(entr(20,26,0))
               marcar(map(12,22))
Regra: map_modif_multiplo1
Condicoes: port([P,0],perm)
           cardinal(T,port_sai(T,P,0),V)
           V>1
           map(P,R)
Acoes: criar(oper([S,1],sel))
       criar(lin([E,1]))
       criar(lin([N,1]))
       marcar(entr(R,E,0))
       marcar(sai(S,E,0))
       marcar(sel(S))
       marcar(entr(S,N,sel))
Regra: map_modif_multiplo2
Condicoes: port([P,0],perm)
           map(P,R)
           entr(R,E,0)
           sai(S,E,0)
           entr(S,N,sel)
           entr(R,K,contr)
           ordem_inicial(O)
           port_sai(T,P,0)
           port_entr(T,Pi,0)
           port_entr(T,Ci,contr)
           map(Pi,Ei)
           ordem(O)
Acoes: criar(port([Pc,1],contr))
       criar(transf([Tc,1],contr))
       marcar(map(Ci,Pc))
       marcar(map(Ci,Tc))
       marcar(port_entr(Tc,Pc,0))
       marcar(port_sai(Tc,N,const(O)))
```

```
        marcar(port_sai(Tc,K,0))
        marcar(entr(S,Ei,0))
   Regra: map_saidas
   Condicoes: port([P,0],sai)
              port_sai(T,P,0)
              port_entr(T,E,0)
              map(S,E)
   Acoes: marcar(map(P,S))
          marcar(saida_circ(S))
   Instancia de subrede: port([c,0],sai)
                         port_sai(17,c,0)
                         por_entr(17,16,0)
                         map(16,21)
   Procedimentos: marcar(map(c,21))
                  marcar(saida_circ(21))
S: Forneca um nome para a rede final.
U: 'SOMA'.
S: Rede gerada --> PROTO/ESTRUTURAL/SOMA
S: Indique a tarefa desejada:
   (a) Construcao de rede comportamental.
   (b) Construcao de rede estrutural atraves do
       interpretador de regras.
   (c) Construcao de rede estrutural atraves de rotina
       de mapeamento direto.
   (d) Compilacao de rotina de mapeamento direto.
   (e) Projeto logico com interpretador de regras.
   (f) Projeto logico com rotina de mapeamento direto.
   (g) Termino da sessao.
U: g.
S: Fim de sessao - Subsistema de Projeto Logico
```

5.4 Opção c: construção de rede estrutural através de rotina de mapeamento direto.

A escolha dessa opção leva ao fluxo de mensagens entre o sistema e o utilizador semelhante ao mostrado a seguir.

```
S: Forneca o nome da rotina de mapeamento direto.
U: 'PROTO/MAPDIR/ESTENDIDO'.
S: Forneca o nome da rede comportamental.
U: 'PROTO/COMPORTAMENTAL/DIVISAO1'.
S: Mapeamento iniciado.
S: Forneca um nome para a rede final.
U: 'DIVISAO1/2'.
S: Rede final gerada --> PROTO/ESTRUTURAL/DIVISAO1/2.
```

Cada rotina de mapeamento direto implementa um estilo

de aplicação das regras de transformação de rede contidas numa base de conhecimento. A rotina solicitada pelo utilizador no exemplo é resultante da compilação do estilo estendido da base PROTO/BC, sendo a rede final PROTO/ESTRUTURAL/DIVISAO1/2 apresentada na figura 5.27.

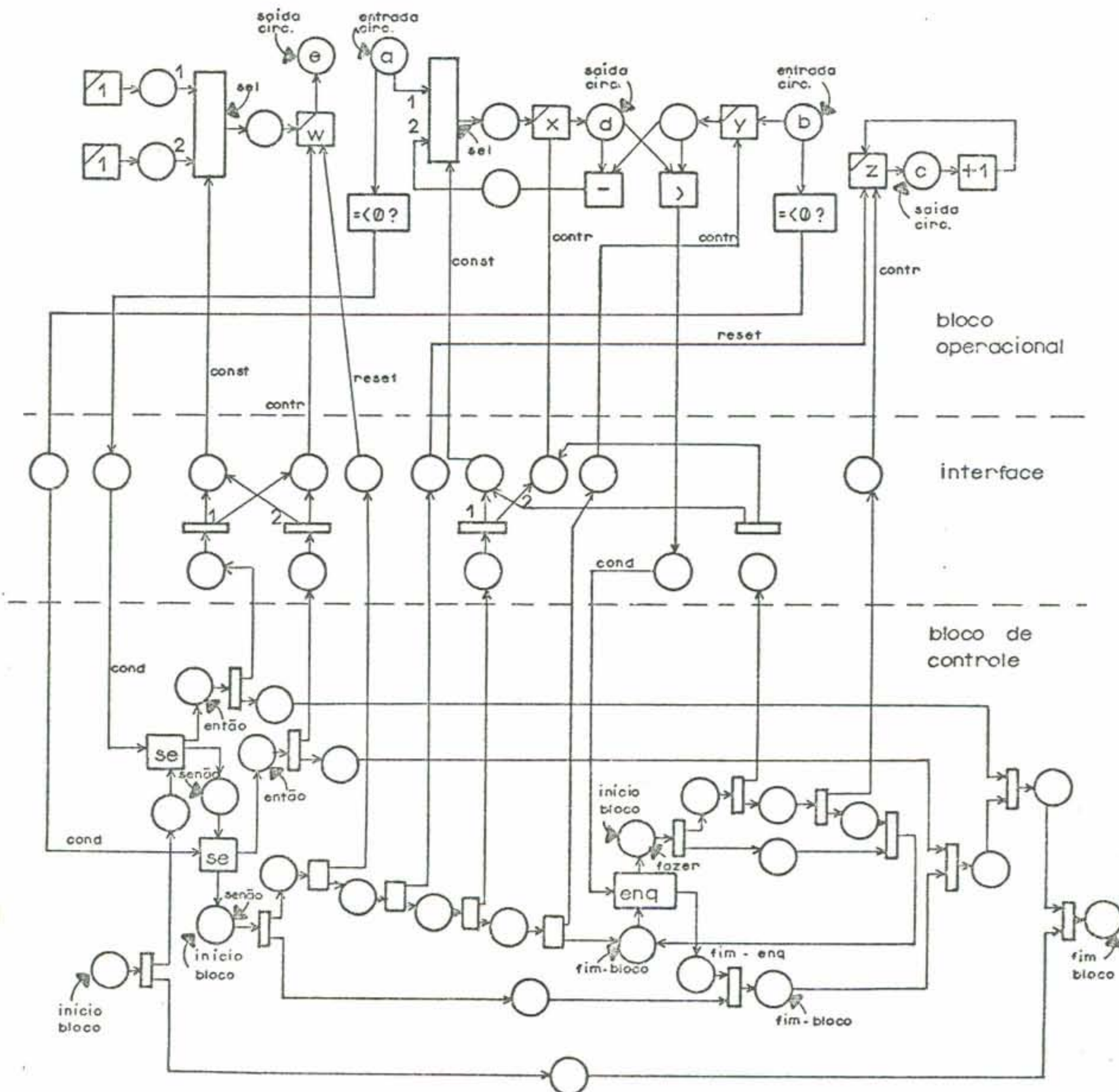


Fig. 5.27 - Rede PROTO/ESTRUTURAL/DIVISAO1/2: gerada a partir da rede PROTO/COMPORTAMENTAL/DIVISAO1, estilo estendido.

O uso de uma rotina de mapeamento direto ao invés do interpretador de regras melhora o desempenho do processo de construção da rede estrutural, pois dispensa a consulta a uma base de conhecimento. Entretanto, tais rotinas não admitem monitoração de suas execuções.

5.5 Opção d: compilação de rotina de mapeamento direto

Esta opção ativa o módulo compilador de estilos, o qual transforma as estruturas envolvidas num estilo de aplicação de regras de transformação de redes de uma base de conhecimento em um programa Prolog. Ambos, estilo e base de conhecimento, são escolhidos pelo utilizador.

Cada rotina de mapeamento direto apresenta os seguintes procedimentos:

(a) Um procedimento chamado **gera_estrutural(Arq)**, que obtém, se necessário, a rede comportamental, invoca o processo de mapeamento direto e o processo que gera a rede final.

(d) Um procedimento chamado **processo**, que é o resultado da compilação do predicado estilo da base de conhecimento. Invoca seqüencialmente as regras contidas na seqüência de regras desse predicado.

(c) Procedimentos que implementam as regras de transformação de redes. Dada uma regra de nome **r** com condição **c** e ação **a**, o procedimento Prolog correspondente é o seguinte:

```
r:-c,a,fail.
r.
```

(d) Predicados **padrao(E,P)**, onde E pode ser **elemento**, **relacao** ou **marca**, e P é, respectivamente, o padrão geral de um elemento, relação ou marca da rede estrutural.

(e) Predicados **argumento_marca(P,A,LT)**, onde P é um padrão geral de marca, A é o padrão geral do argumento da marca e LT é uma lista com os tipos deste argumento sobre os quais a marca pode atuar.

(f) Os procedimentos Prolog referenciados pelas regras.

(g) Procedimentos que geram o arquivo com a rede final. Estes procedimentos utilizam os predicados **padrao(E,P)** e **argumento_marca(P,A,LT)** para reconhecer os elementos da rede estrutural. Os predicados que descrevem a rede comportamental, bem como a relação básica de mapeamento de redes, são referenciados explicitamente por estas rotinas.

O exemplo a seguir ilustra o fuxo de mensagens durante a execução desse módulo. Convém ressaltar que o nome da rotina de mapeamento direto é sempre formado pelo diretório PROTO/MAPDIR seguido do nome do estilo convertido para letras maiúsculas.

```
S: Forneca o nome da base de conhecimento.
U: 'PROTO/BC'.
S: Forneca o estilo de geracao de rede estrutural.
U: estendido.
S: Compilacao iniciada.
S: Rotina gerada --> PROTO/MAPDIR/ESTENDIDO
```

5.6 Opção e: projeto lógico através do interpretador de regras

Esta opção invoca seqüencialmente os módulos compilador

da rede comportamental e interpretador de regras, sendo a entrada desse último a rede comportamental gerada pelo primeiro. O fluxo de mensagens entre o sistema e o utilizador apresentado a seguir exemplifica o comportamento dessa opção.

```
S: Forneca a origem do algoritmo.
   Caso seja arquivo, forneca seu nome.
   Caso seja terminal, forneca user.
U: user.
S: Forneca a descricao do circuito.
   Lembre-se de encerra-la com #.
U: entr a,b.
   sai c.
   reg x,y,z.
   inicio.
   x:=a.
   y:=b.
   z:=0.
   enquanto x > 0.
   fazer inicio.
         x:=x - 1.
         z:=z + y.
         fim.
   c<=z.
   fim.
   #
S: Processo de compilacao iniciado.
S: Forneca um nome para a rede comportamental.
U: ´MULT1´.
S: Rede comportamental gerada -->
   PROTO/COMPORTAMENTAL/MULT1
S: Forneca a base de conhecimento.
U: ´PROTO/BC´.
S: Indique o tipo de rede de saida:
   (a) Rede final.
   (b) Rede parcial.
U: a.
S: Forneca o estilo de geracao da rede estrutural.
U: basico.
S: Voce quer monitorar a execucao das regras?
   Responda sim ou nao.
U: nao.
S: Construcao da rede iniciada.
S: Forneca um nome para a rede final.
U: ´MULT1´.
S: Rede final gerada --> PROTO/ESTRUTURAL/MULT1
```

A rede comportamental PROTO/COMPORTAMENTAL/MULT1 e a rede final PROTO/ESTRUTURAL/MULT1, geradas no fluxo sistema-

utilizador acima, estão representadas, respectivamente, nas figuras 5.28 e 5.29.

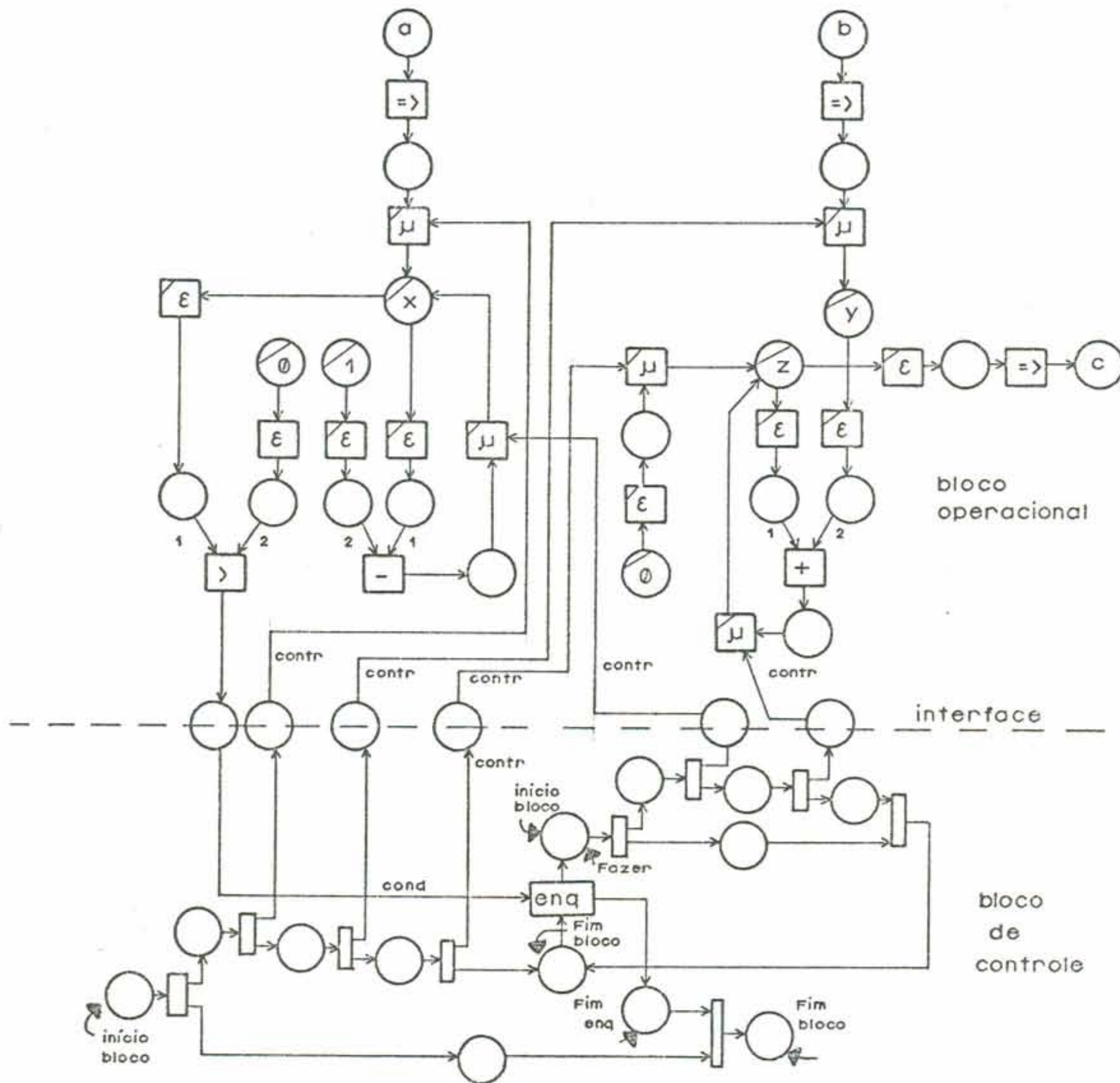


Fig 5.28 - Rede comportamental PROTO/COMPORTAMENTAL/MULT1: multiplicação de inteiros positivos através de somas sucessivas.

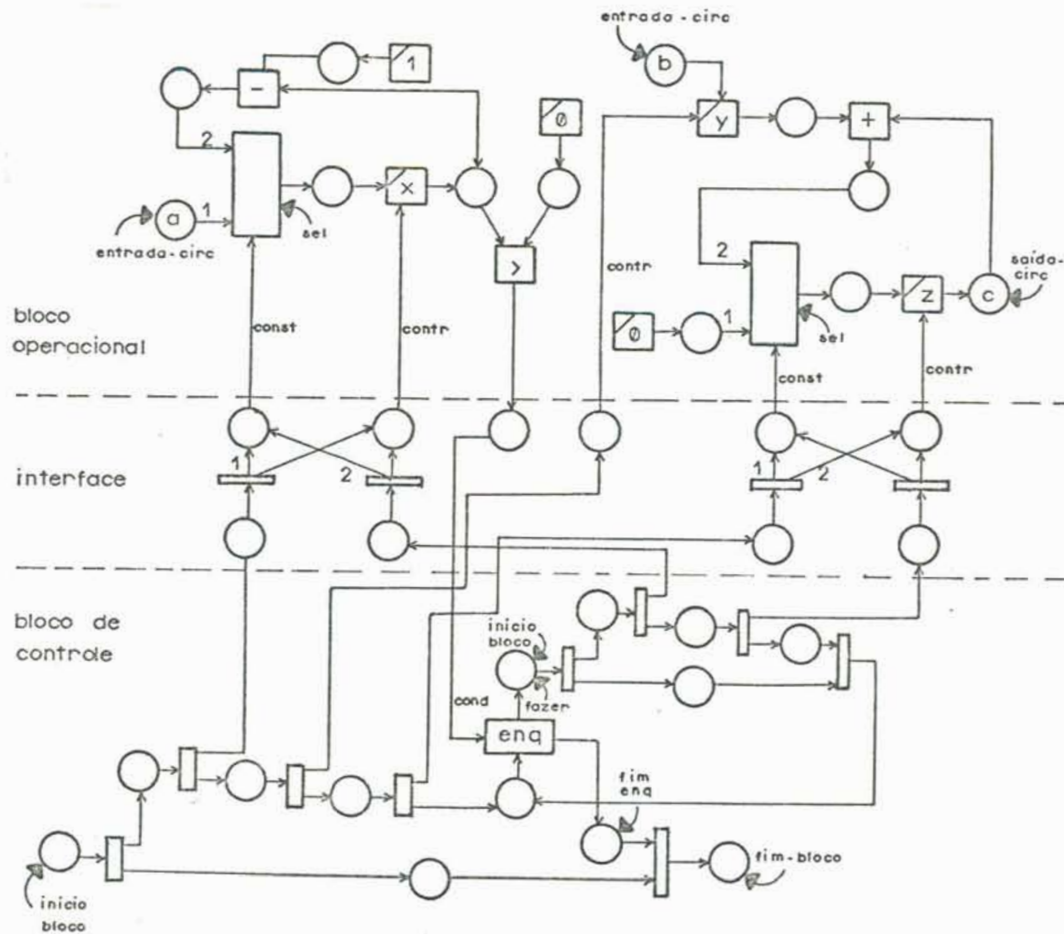


Fig. 5.29 - Rede final PROTO/ESTRUTURAL/MULT1 : gerada no estilo basico a partir da rede comportamental PROTO/COMPORTAMENTAL/MULT1.

5.7 Opção f: projeto lógico através de rotina de mapeamento direto

Esta opção invoca seqüencialmente o compilador da rede comportamental e uma rotina de mapeamento direto escolhida pelo utilizador. A entrada para a rotina de mapeamento direto é a rede comportamental gerada pelo compilador. O fluxo de mensagens a

seguir exemplifica o comportamento dessa opção.

```
S: Forneca o nome da rotina de mapeamento direto.
U: 'PROTO/MAPDIR/ESTENDIDO'.
S: Forneca a origem do algoritmo.
   Caso seja arquivo, forneca seu nome.
   Caso seja terminal, forneca user.
U: user.
S: Forneca a descricao do circuito.
   Lembre-se de encerra-la com #.
U: entr a,b.
   sai c.
   reg x,y,z.
   inicio.
   x:=a.
   y:=b.
   z:=0.
   enquanto x > 0.
   fazer inicio.
         x:=x - 1.
         z:=z + y.
         fim.
   c<=z.
   fim.
   #
S: Processo de compilacao iniciado.
S: Forneca um nome para a rede comportamental.
U: 'MULT1'.
S: Rede comportamental gerada -->
   PROTO/COMPORTAMENTAL/MULT1
S: Mapeamento iniciado.
S: Forneca um nome para a rede final.
U: 'MULT1/2'.
S: Rede final gerada --> PROTO/ESTRUTURAL/MULT1/2
```

A rede PROTO/ESTRUTURAL/MULT1/2 gerada pela rotina de mapeamento direto PROTO/MAPDIR/ESTENDIDO é mostrada graficamente na figura 5.30.

Comparações entre os desempenhos do interpretador de regras no estilo **estendido** e a rotina de mapeamento direto PROTO/MAPDIR/ESTENDIDO para uma mesma rede de entrada (a rede comportamental PROTO/COMPORTAMENTAL/MULT1) são apresentadas nas tabelas seguintes. A tabela 5.2 apresenta os tempos de processa-

mento de cada módulo para o processo isolado de execução das regras de transformação de redes, enquanto a tabela 5.2 mostra o desempenho desses módulos para a tarefa de construção da rede estrutural. Estas justificam a afirmação anterior a respeito do melhor desempenho das rotinas de mapeamento direto em relação ao interpretador de regras.

TABELA 5.1 Desempenho comparativo dos processos de execução das regras

Processo de execução das regras	Tempo proces. (seg.)
Interpretador de regras	5.4
Rotina de mapeamento direto	4.0

TABELA 5.2 Desempenhos na tarefa de construção de rede estrutural

Tarefa de construção de rede estrutural	Tempo proces. (seg.)	Tempo E/S (seg.)
Interpretador de regras	22.1	1.7
Rotina de mapeamento direto	13.8	0.9

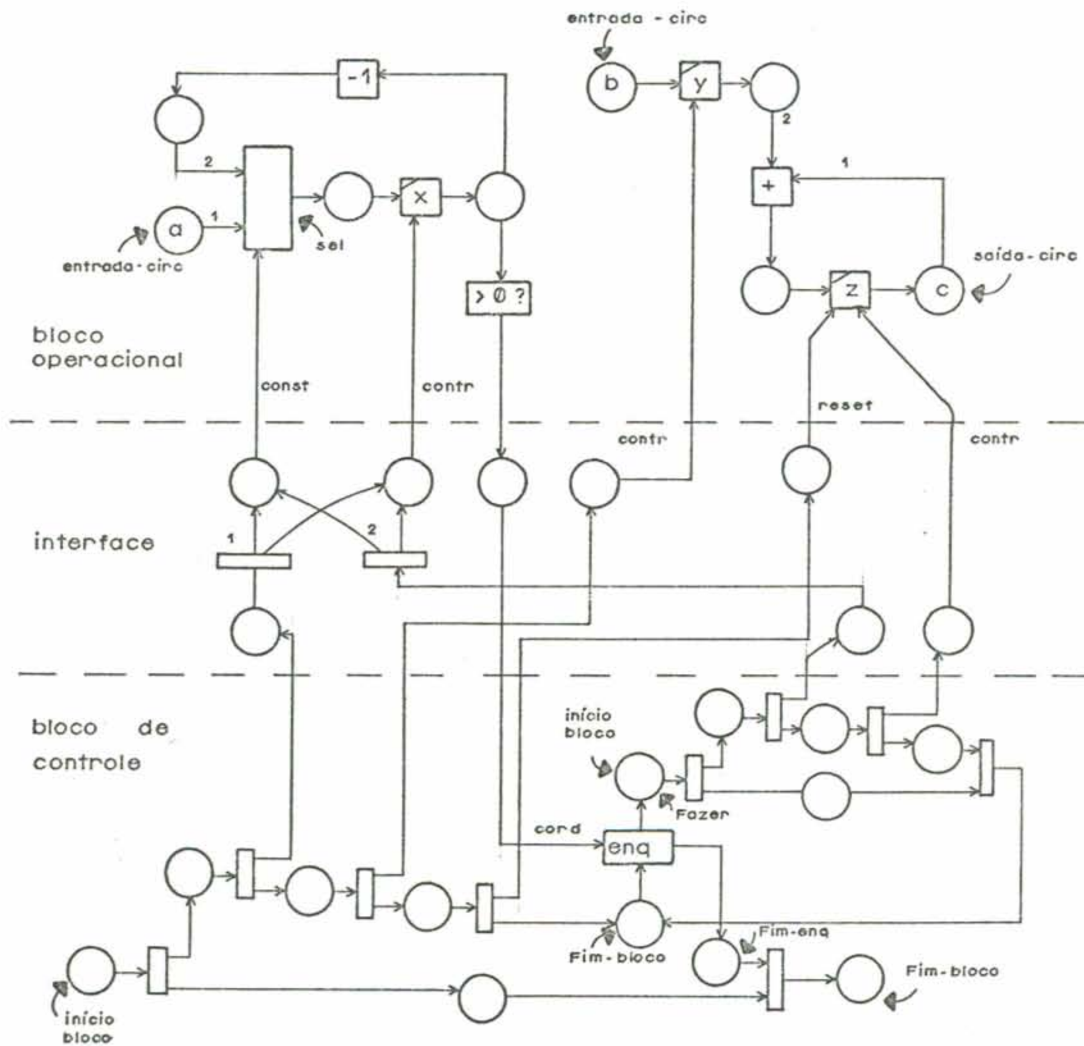


Fig. 5.30 - Rede final PROTO/ESTRUTURAL/MULT/2 : gerada no estilo estendido a partir da rede comportamental PROTO/COMPORTAMENTAL/MULT1.

6. AQUISIÇÃO DE CONHECIMENTO: CONSTRUÇÃO E EDIÇÃO DE BASES DE CONHECIMENTO

As atividades de construção e manipulação das bases de conhecimento do PROTO são suportadas pelo subsistema de aquisição de conhecimento. Este possibilita a construção de novas bases de conhecimento e o exame, inclusão, alteração e supressão de itens de conhecimento contidos nas bases do PROTO.

Neste capítulo apresenta-se as funções realizadas por este subsistema, as informações esperadas do utilizador e as mensagens e resultados fornecidos a este. Pressupõem-se que o utilizador possua um amplo domínio na tarefa de projeto lógico de circuitos digitais e do método de solução de problemas por transformação de redes.

6.1 Controle de sessão

Cada execução desse subsistema constitui uma sessão de aquisição de conhecimento, sendo iniciada através do envio ao interpretador Prolog da seguinte seqüência de instruções:

```
?-consult('PROTO/AQUISICAO/CONHECIMENTO').
?-aquisicao_conhecimento.
```

A primeira instrução coloca na memória acessível ao Prolog o módulo de controle e rotinas comuns aos módulos de construção e edição de bases de conhecimento. A segunda instrução causa a execução do módulo de controle, o qual define os operadores especiais necessários e envia o seguinte conjunto de

opções ao utilizador:

```
Indique a opcao desejada:
(a) Construcao de base de conhecimento.
(b) Edicao de base de conhecimento.
(c) Termino da sessao.
```

O utilizador deve fornecer a opção que corresponde à tarefa desejada: letras a, b ou c. Sempre que uma atividade de contrução ou edição for concluída, o conjunto de escolhas acima será novamente enviado.

6.2 Opção a: construção de base de conhecimento

Esta opção invoca o módulo de construção de bases de conhecimento, o qual gera uma base a partir de um texto previamente preparado que contém a sua descrição. O fluxo de mensagens a seguir exemplifica a interação entre o sistema e o utilizador durante a construção de uma base de conhecimento.

```
S: Forneca o nome do texto com a descricao da base.
U: 'PROTO/TXT/BC'.
S: Processo de compilacao iniciado.
S: Processo de compilacao concluido.
   Base gerada --> PROTO/BC
```

O texto PROTO/TXT/BC e a base resultante PROTO/BC relativas a este exemplo são apresentados nas seções seguintes.

6.2.1 A descrição da base

O texto de entrada do compilador de bases de conhecimento é uma descrição de base de conhecimento escrita numa linguagem específica. Esta linguagem permite as seguintes

definições:

(a) o nome da base de conhecimento, o arquivo com o conhecimento pré-definido que deve ser incorporado à base resultante, e um conjunto de sentenças em língua natural que descrevem seus objetivos;

(b) um conjunto de procedimentos (testes) Prolog adicionais necessários às condições das regras de transformação de redes;

(c) os elementos que podem compor a(s) redes(s) estrutural(is);

(d) as marcas (inscrições) que podem caracterizar esses elementos;

(e) as relações possíveis entre os elementos estruturais;

(f) as regras de transformação de redes; e

(g) os estilos de aplicação dessas regras.

A sintaxe desse texto é apresentada através dos diagramas de sintaxe a seguir. Nestes, itens entre os símbolos < e > são explicados por outros diagramas ou por definições entre chaves.

```
<texto>:

                    ----------------------------------
                    V                                !
!--> <id.base> ---------------------------------------> fim --> # -->!
                        !                        ^
                        !--> <def.teste> ---->!
                        !                     !
                        !--> <def.elemento> ->!
                        !                     !
                        !--> <def marca> ---->!
                        !                     !
                        !--> <def.relação> -->!
                        !                     !
                        !--> <def.regra> ---->!
                        !                     !
                        !--> <def.estilo> --->!


<id.base>:

!-----> base: ------> <nome da base> --------> ; ---------------->
>-----> pre-definido: ----> <nome dos pré-definidos> ----> ; --->
>-----> objetivo: ------> <objetivos> ----------> # ----------->!


<def.teste>:

!------> teste: ------> <nome de teste> -------> ; ------------->
>------> objetivo: ------> <objetivos> -------> ; -------------->
>------> padrao: ------> <padrão geral de teste> ------> ; ----->

          ------------------------------------------------
          V                                              !
>------------------------------------------------------------> # ------>!
               !                           ^
               !---> <cláusula Prolog> --->!
```

```
<def.elemento>:

!-------> elemento: ------> <nome> ------> ; --------------------->

>-------> objetivo: ------> <objetivos> ------> ; --------------->

>-------> padrao: ------> <padrão geral do elemento> ------------>

>---------------------------------------------------------> ; ----->
        !                                                  ^
        !-----> -> -----> <lista de tipos> ----->!

>-------> nivel: ---------> <inteiro> ---------------------------->

>------------------------------------------------------------> # --->!
      !                                                    ^
      !----> ; ---> estilos: ---> todos ----------->!
      !                                   !          !
      !   ---------------------------------          !
      !   !                                          !
      !---v-> ; ---> origem: ---> indeterminado ---->!
      !                                        !     !
      !   -------------------------------------      !
      !   !                                          !
      !---v-> ; ---> resulta: ---> indeterminado --->!


<def.marca>:

!-------> marca: ------> <nome> ------> ; ----------------------->

>-------> objetivo: ------> <objetivos> ------> ; --------------->

>-------> padrao: ------> <padrão de marca> ------> ; ----------->

                         ----> <padrão de elemento> ----
                         !                             !
>-------> argumento: ----                               ---------->
                         !                             !
                         ---------> <variável> ---------

>-------------------------------------------------------> ; ------>
         !                                       ^
         !----> -> ----> <lista de tipos> ---->!

>----> <var.nível> ----> <op.rel> -----> <variável> ----> # --->!
```

```
<def. relação>:

!-------> relacao: -------> <nome> -------> ; ----------------------->

>-------> objetivo: -------> <objetivos> -------> ; ---------------->

>-------> padrao: -------> <padrão geral de relação> -----> ; ----->

          ----> <descrição de argumento elemento> ----
          !                                          !
          ----------------------------------------   !
                                                 !   !
>------> argumento: ----> <ordem arg.> ----> ;----   -----> # --->!
   ^                                             !   !   !
   !      ----------------------------------------   !   !
   !      !                                          !   !
   !      ----> <descrição de argumento índice> ---->!   !
   !                                                     !
   --------------------------- ; <------------------------

<def.regra>:

!-------> regra: -------> <nome> -------> ; ------------------------>

>-------> objetivo: -------> <objetivos> -------> ; ---------------->

>-------> condicoes: -------> <lista de condições> -------> ; ------>

        ----> criar: ----> <lista de elementos> ----> ; ----
        !                                                  V
>------------------------------------------------------------------->

>------> marcar: -----> <lista de relações e marcas> ---> # ---->!

<def.estilo>:

>----------> estilo: ----------> <nome> ----------> ; -------------->

>----------> objetivo: ---------> <objetivos> ---------> ; --------->

>----------> regras: --------> <lista de regras> ------> # -------->!

<padrão geral de teste>:

!---> <nome de teste> ----------------------------------------->!
                            !                                  ^
                            !       ------- , <------          !
                            !       !               !          !
                            !---> ( V--> <variável> ---> ) --->!
```

ou

```
   ---> <variável> ---                          ---> <variável> ---
   !                 V                          !                 V
!-----------------------> <nome de teste> ----------------------->!
```

<padrão geral de elemento>:

```
!-----> <nome> -----> ( -----> [ -----> <variável> -----> , ----->

>-----> <variável> --> ] ------------------------------> ) --->!
                        !                              A
                        !       ---> <variável>---     !
                        !       !                !     !
                        !--> , --                -->!
                                !                !
                                --> <constante> --
```

<padrão de marca>:

```
!-------> <nome> -------> ( ------> <variável> ------> ) ------>!
```

<padrão geral de relação>:

```
                                       -----------------------
                                       V                     !
!--> <nome> --> ( --> <variável> ---> , --> <variável> ---> ) -->
```

<descrição argumento elemento>:

```
                      ----> <padrão de elemento> ---
                      !                            !
!-----> elemento: -----                            -----> ; ---->
                      !                            !
                      ------> <variável> -----------

                                      ----> <inteiro> -----
                                      !                   !
>-----> <var.nível> ----> <op.rel> ----                   ----->!
                                      !                   !
                                      ---> <variável> ----
```

<descrição argumento índice>:

```
!----------> indice: ----------> <lista de índices> ----------->!
```

<padrão de elemento>:

```
!-----> <nome> -----> ( -----> [ -----> <variável> -----> , ---->

  ---> <inteiro> ---
  !                !
>--                ---> ] ------------------------------> ) --->!
  !                !        !                          ^
  ---> <variável> --        !        --> <variável> --- !
                            !        !              ! !
                            !--> , --               ---
                                     !              !
                                     --> <constante> --
```

<padrão de relação>:

```
!----> <nome> ---> ( ----> <variável> ---> , ---> <variável> --->

>-----------------------------------------------------> ) ----->!
               !                                      ^
               !        ----> <inteiro> ----          !
               !        !                  !          !
               !----> , ---                ---->!
                        !                  !
                        ---> <variável> ----
```

<padrão de teste>:

```
!----------------------> <nome de teste> ----------------------->!
  !                   ^                     !                  ^
  !--> <constante> -->!                     !--> <constante> ->!
  !                   !                     !                  !
  !--> <variável> --->!                     !--> <variável> -->!
```

ou

```
                                    ----------- , <----------
                                   v                        !
!---> <nome de teste> ---> ( -------> <constante> -------> ) ---->!
                                 !                   ^
                                 !--> <variável> --->!
```

<lista de tipos>:

```
                  ---------------- , <---------------
                 v                                  !
!------------------------> <constante> ------------------------->!
                    !                        ^
                    !---> <padrão de teste> --->!
```

<lista de índices>:

```
                  ---------------- , <-----------------
                  V                                    !
!-----------------------------> <constante> ---------------------------->!
                    !                        ^
                    !----> <padrão de teste> ----!
```

<lista de regras>:

```
                    ----------------- , <-------------
                    V                                !
!---------------------------> <nome de regra> ------------------------->!
```

<lista de condições>:

```
                ------------------- , <-------------------
                V                                        !
!------------------------> <padrão de elemento> ---------------------->!
                    !                           ^
                    !---> <padrão de teste> ---->!
                    !                           !
                    !---> <padrão de marca> ---->!
                    !                           !
                    !---> <padrão de relação> -->!
```

<lista de elementos>:

```
                    ----------------- , <--------------
                    V                                 !
!-------------------------> <padrão de elemento> -------------------->!
```

<lista de relações e marcas>:

```
                  ------------------- , <--------------
                  V                                   !
!-------------------------> <padrão de relação> --------------------->!
                      !                      ^
                      !---> <padrão de marca> ---->!
```

<nome da base>:

```
!---------------------> <nome de arquivo> ------------------------>!
```

<nome dos pré-definidos>:

```
!---------------------> <nome de arquivo> ------------------------>!
```

<nome de arquivo>:

```
                   ------------------------------------------------
                   v                                               !
!---> <letra maiusc.> --------------------------------------------------->!
                      !                                          ^
                      !--> / ---                                 !
                      !         v                                !
                      !-----------> <letra maiusc.>-->!
                      !         v                                !
                      !----------------> <dígito> ---->!
```

<nome>:

```
              ----------------------------------------------------
              v                                                  !
!---> <letra> ----------------------------------------------------------->!
                   !                                  ^
                   !---> _ ---                        !
                   !            v                     !
                   !---------------> <letra> ---->!
                   !            v                     !
                   !---------------> <digíto> -->!
```

<nome de teste>:

```
!----------------------------> <nome> ------------------------------->!
           !                                    ^
           !---> <símbolo funcional Prolog> --->!
```

<inteiro>:

```
              --------------------------------------------
              v                                          !
!---------------------------> <dígito> ----------------------------->!
```

<variável>:

```
                   ------------------------------------------------
                   v                                               !
!---> <letra maiusc.> --------------------------------------------------->!
                      !                                          ^
                      !--> _ ---                                 !
                      !         v                                !
                      !-----------> <letra maiusc.> -->!
                      !         v                                !
                      !---------------> <dígito> ------->!
                      !         v                                !
                      !---------------> <letra> -------->!
```

<op.rel>:

<objetivos>: (Conjunto de sentenças em língua natural que descrevem os objetivos da estrutura. Essas sentenças po- apresentar quaisquer caracteres, exceto os caracteres # e ;).

<ordem arg.>: (Valor inteiro que informa a posição do argumento na lista de argumentos de uma relação.)

<var.nível>: (Variável do padrão do elemento que representa o nível de criação deste.)

<constante>: (Átomo Prolog ou valor inteiro.)

<cláusula Prolog>: (Fato "p." ou regra "p:-q" da linguagem Prolog.)

<símbolo funcional Prolog>: (Símbolo que denota um operador da linguagem Prolog.)

A seguir apresenta-se o texto PROTO/TXT/BC, o qual define as construções estruturais do estilo **basico** da base de conhecimento PROTO/BC.

```
base: PROTO/BC;
pre-definido: PROTO/PREDEFINIDO/BC;
objetivo:  Esta base apresenta o conhecimento minimo
           para o mapeamento da rede comportamental
           para a rede estrutural basica.
#
elemento: registrador;
objetivo: Elemento de nivel estrutural que representa
          um registrador de leitura e escrita.;
padrao: reg([R,N]);
nivel: 1;
resulta:indeterminado
#
elemento: registrador_constante;
objetivo: Elemento de nivel estrutural que representa
          um registrador de apenas leitura, com valor
          constante C.
padrao: reg([R,N],const(C));
nivel: 1
#
elemento: linha_de_sinal;
objetivo: Elemento estrutural que representa uma linha
          ou conjunto de linhas de sinal.;
padrao: lin([L,N]);
nivel: 1;
estilos: todos;
origem: indeterminado;
resulta: indeterminado
#
elemento: operador;
objetivo: Elemento de nivel estrutural que representa
          um operador aritmetico, relacional ou um
          seletor.;
padrao: oper([O,N],OP) -> +,-,*,/,=,>,<,>=,=<,sel;
nivel: 1
#
marca: entrada_circuito;
objetivo: Identifica uma linha L como uma entrada de
          de dados no circuito.;
padrao: entrada_circ(L);
argumento: lin([L,N] ; N>0
#
marca: saida_circuito;
objetivo: Identifica uma linha L como uma saida de
          de dados do circuito.;
padrao: saida_circ(L);
argumento: lin([L,N]) ; N>0
#
```

```
marca: selecao;
objetivo: Caracteriza  o  operador  S  como receptor de
          sinal de selecao.;
padrao: sel(S);
argumento: oper([S,N],sel) ; N>0
#
relacao: linha_de_entrada;
objetivo: Define  a  relacao: L e' linha de  entrada do
          elemento E, sendo o indice  da  entrada  dado
          por I.;
padrao: entr(E,L,I);
argumento: 1; elemento:E; N>0;
argumento: 2; elemento:lin([L,N]); N>0;
argumento: 3; indice:const(I),contr,sel
#
relacao: linha_de_saida;
objetivo: Define a relacao:  L  e' linha  de  saida  do
          elemento  E, sendo o indice da saida dado por
          I.;
padrao: sai(E,L,I);
argumento: 1; elemento:E; N>0;
argumento: 2; elemento:lin([L,N]); N>0;
argumento: 3; indice: const(I)
#
regra: map_permanentes;
objetivo: Mapear  portadores  permanentes para subredes
          com registradores de leitura e escrita.;
condicoes: port([P,0],perm);
criar: reg([R,1]),
       lin([S,1]),
       lin([C,1]);
marcar: map(P,R),
        sai(R,S,0),
        entr(R,C,contr)
#
regra: map_constantes;
objetivo: Mapear  portadores  constantes  para subredes
          com registradores de apenas leitura.;
condicoes: port([P,0],const(C));
criar: reg([R,1],const(C)),
       lin([S,1]);
marcar: map(P,R),
        sai(R,S,0)
#
regra: map_entradas;
objetivo: Mapear portadores de entrada.;
condicoes: port([P,0],entr);
criar: lin([E,1]);
marcar: map(P,E),
        entrada_circ(E)
#
regra: map_ligadores_entrada;
objetivo: Mapear ligadores de entrada.;
condicoes: port([P,0],entr),
           map(P,E),
```

```
            port_entr(T,P,0),
            port_sai(T,S,0);
marcar: map(S,E)
#
regra: map_extratores;
objetivo: Mapear operadores que extraem informacoes de
          portadores permanentes e constantes.;
condicoes: transf([T,0],epsi),
            port_entr(T,P,0),
            port_sai(T,S,0),
            map(P,R),
            sai(R,L,0);
marcar: map(S,L)
#
regra: map_operadores;
objetivo: Mapear operadores aritmeticos e relacionais.;
condicoes: transf([T,0],OP),
            operador(OP),
            port_entr(T,P1,1),
            port_entr(T,P2,2),
            port_sai(T,P,0),
            map(P1,E1),
            map(P2,E2);
criar: oper([O,1],OP),
       lin([S,1]);
marcar: entr(O,E1,1),
        entr(O,E2,2),
        sai(O,S,0),
        map(P,S)
#
regra: map_modif_simples;
objetivo: Mapear subredes em que ha um operador de
          modificacao por portador permanente.;
condicoes: port([P,0],perm),
            cardinal(T,port_sai(T,P,0),1),
            port_sai(T,P,0),
            port_entr(T,E,0),
            port_entr(T,C,contr),
            map(P,R),
            map(E,L),
            entr(R,K,contr);
marcar: entr(R,L,0),
        map(C,K);
#
regra: map_modif_multiplo1;
objetivo: Criar seletores de entrada.;
condicoes: port([P,0],perm),
            cardinal(T,port_sai(T,P,0),V),
            V > 1,
            map(P,R);
criar: oper([S,1],sel),
       lin([E,1]),
       lin([N,1]);
marcar: entr(R,E,0),
        sai(S,E,0),
```

```
        sel(S),
        entr(S,N,sel)
#
regra: map_modif_multiplo2;
objetivo: Ligar entradas ao seletor.;
condicoes: port([P,O],perm),
           map(P,R),
           entr(R,E,O),
           sai(S,E,O),
           entr(S,N,sel),
           entr(R,K,contr),
           ordem_inicial(O),
           port_sai(T,P,O),
           port_entr(T,Pi,O),
           port_entr(T,Ci,contr),
           map(Pi,Ei),
           ordem(D);
criar: port([Pc,1],contr),
       transf([Tc,1],contr);
marcar: map(Ci,Pc),
        map(Ci,Tc),
        port_entr(Tc,Pc,O),
        port_sai(Tc,K,O),
        port_sai(Tc,N,const(D)),
        entr(S,Ei,O)
#
regra: map_saidas;
objetivo: Mapear portadores de saida.;
condicoes: port([P,O],sai),
           port_sai(T,P,O),
           port_entr(T,E,O),
           map(E,S);
marcar: map(P,S),
        saida_circ(S)
#
estilo: basico;
objetivo: Sequencia de regras  que  fazem  o mapeamento
          rede  comportamental   -->   rede  estrutural
          basica.;
regras: map_permanentes,
        map_constantes,
        map_entradas,
        map_ligadores_entrada,
        map_extratores,
        map_operadores,
        map_modif_simples,
        map_modif_multiplo1,
        map_modif_multiplo2,
        map_saidas
#
fim
#
```

6.2.2 O processo de construção de bases de conhecimento

O processo de construção de bases de conhecimento utiliza uma gramática de cláusulas definidas, que descreve a sintaxe do texto de entrada, e um conjunto de subprocessos, que transformam as informações contidas nesse texto nos predicados correspondentes da base de conhecimento. Pode ser subdividido em três etapas: identificação da base, identificação dos itens de conhecimento estruturais e geração final da base.

Quaisquer incorreções detectadas no texto de entrada são apontadas imediatamente ao utilizador. Estas podem ser de dois tipos:

(a) incorreções na identificação da base, que causam a interrupção do processo de análise do texto; e

(b) incorreções na definição de componentes estruturais, as quais não interrompem o processo de análise sintática, embora inibam a construção da base de conhecimento.

6.2.2.1 Identificação da base

A primeira atividade realizada por este processo é a identificação da base de conhecimento B a ser construída, do arquivo P com o conhecimento pré-definido e dos objetivos O desta base. Esta atividade tem como resultados a criação do predicado base_conhecimento(B,P,O) e a incorporação dos itens de conhecimento contidos em P à base de conhecimento B em construção.

Um arquivo de informações pré-definidas distingue-se das bases de conhecimento devido à omissão dos predicados de identificação e finalização de base (respectivamente, base_conhecimento e fim_base), e por apresentar os procedimentos Prolog, relativos aos predicados teste, associados a predicados que os caracterizam inequivocamente. Estes predicados apresentam a seguinte forma geral:

```
procedimento_teste(<nome do teste>,
                   <valor de ordem do procedimento>,
                   <fato ou regra Prolog>).
```

De fato, estes predicados identificam todos os procedimentos Prolog pertencentes a uma base de conhecimento durante sua manipulação, quer a nível de construção como de edição.

6.2.2.2 Identificação de itens de conhecimento estruturais

Nesta etapa são reconhecidas as definições de elementos, marcas, relações, regras, estilos e testes presentes no texto de entrada, gerando como resultado os predicados correspondentes que as descrevem na base de conhecimento. Além da análise sintática referente à gramática que descreve essas definições, esta etapa verifica as seguintes condições:

(a) qualquer nome de elemento, marca, relação, regra, estilo ou teste deve ser único no(s) predicados(s) que o(s) descreve(m);

(b) qualquer padrão geral de elemento, marca, relação ou teste deve ser único;

(c) qualquer teste referenciado por elementos, relações e/ou regras deve estar definido na base de conhecimento em construção; e

(d) qualquer regra referenciada por um ou mais estilos deve estar definida na base de conhecimento.

As seguintes informações são explicitadas pelo sistema durante esta etapa:

(a) as regras que criam um determinado elemento são incorporadas à lista de origens deste, desde que essa última não contenha a informação **compilacao** ou **indeterminado**.

(b) as regras que utilizam um elemento como parte de sua condição são incorporadas à sua lista de resultantes, desde que essa lista não contenha a informação **indeterminado**.

6.2.2.3 Geração final da base

E a última etapa do processo de construção de bases de conhecimento, executada somente se não forem detectados erros nas etapas anteriores. Nesta, são explicitados todos os estilos que referenciam um determinado elemento (quando este não possuir o termo **todos** na sua lista de estilos), sendo gerado o arquivo B que contém a base de conhecimento.

6.2.2.4 Mensagens de erro

As incorreções detectadas no texto de entrada do módulo de construção de bases de conhecimento são apontados ao utilizador através de um pequeno conjunto de mensagens de erro. Deste conjunto, as seguintes mensagens são exclusivas aos erros na identificação da base:

(a) Nome da base invalido.

(b) Nome de arquivo invalido --> pre-definidos.

(c) Arquivo inexistente --> <arquivo de pré-definidos>.

(d) Espera-se arquivo de pre-definidos.

(e) Espera-se objetivos da base.

(f) Identificacao da base incorreta.

As seguintes mensagens referem-se a erros na definição das demais construções (elementos, marcas, relações, testes, regras ou estilos):

(a) Nome de <construção> incorreto.

(b) Nome de <contrução> duplicado --> <nome>.

(c) Espera-se objetivos <da construção> --> <nome>.

(d) Padrao invalido --> <construção> <nome>.

(e) Padrao geral duplicado --> <construção> <nome>.

(f) Espera-se padrao <da construção> <nome>.

(g) Clausula tipo invalida --> elemento <nome>.

(h) Nivel de definicao invalido --> elemento <nome>.

(i) Espera-se nivel de definicao do elemento <nome>.

(j) Clausula estilos incorreta --> elemento <nome>.

(k) Clausula origem incorreta --> elemento <nome>.

(l) Clausula resulta incorreta --> elemento <nome>.

(m) Padrao de argumento incorreto --> marca <nome>.

(n) Tipos de argumento incorretos --> marca <nome>.

(o) Espera-se nivel do argumento --> marca <nome>.

(p) Espera-se argumento da marca <nome>.

(q) Descricao de argumento invalida --> relacao <nome>.

(r) Valor de indice invalido.

(s) Condicao incorreta na regra <nome>.

(t) Condicao invalida --> <padrão>.

(u) Elemento invalido --> criar <padrão>.

(v) Padrao invalido --> marcar <padrão>.

(w) Regra invalida no estilo <nome>.

(x) Espera-se regras do estilo <nome>.

(y) Clausula invalida nesta situacao.

(z) Definicao de construcao invalida.

6.2.3 A base de conhecimento resultante

Nesta seção, apresenta-se a base de conhecimento PROTO/BC, construída a partir do texto PROTO/TXT/BC.

```
base_conhecimento('PROTO/BC',
                  'PROTO/PREDEFINIDO/BC',
                  ['Esta',base,apresenta,o,conhecimento
                   ,minimo,para,o,mapeamento,da,rede,
                   comportamental,para,a,rede,
                   estrutural,basica,.]).
teste(operador,
      ['Verifica',se,o,tipo,de,um,transformador,e,'',
       aritmetico,ou,relacional,.],
      operador(OP)).
operador(+).
operador(-).
operador(*).
operador(/).
operador(=).
operador(>).
operador(<).
operador(>=).
operador(=<).
teste(ordem_inicial,
      ['Fornece',um,valor,inicial,'V',para,o,predicado,
       valor_ordem,',',o,qual,sera,incrementado,pelo,
       predicado,ordem,.],
      ordem_inicial(V)).
ordem_inicial(V):-(retract(valor_ordem(_)) ; true),
                  asserta(valor_ordem(V)),!.
teste(ordem,
      ['Incrementa',o,valor,associado,ao,predicado,
       valor_ordem,em,uma,unidade,',',devolvendo-o,
       na,variavel,'V',.,'Este',teste,exige,o,uso,
       previo,do,predicado,ordem_inicial,.],
      ordem(V)).
ordem(V):-retract(valor_ordem(VA)),
          V is VA + 1,
          asserta(valor_ordem(V)).
teste(cardinal,
      ['Devolve',em,'N',o,numero,de,instancias,de,um,
       argumento,'A',num,predicado,'P',.,'A',nao,deve,
       estar,instanciada,.,'Os',demais,argumentos,de,
       'P',devem,estar,instanciados,.],
      cardinal(A,P,N)).
cardinal(A,P,N):-setof(A,P,L),!,
                 length(L,N).
```

```
teste(const,
      [´Verifica´,se,´C´,representa,um,valor,inteiro,
       sem,sinal,.],
      const(C)).
const(C):-name(C,[H:T]),
          digito(H),
          numero(T).
digito(X):-X > 47,X < 58.
numero([]).
numero([H:T]).-digito(H),numero(T).
teste(>,
      [´Verifica´,se,´X´,e,´´,maior,que,´Y´,.],
      X>Y).
teste(<,
      [´Verifica´,se,´X´,e,´´,menor,que,´Y´,.],
      X<Y).
teste(=,
      [´Verifica´,se,´X´,e,´´,igual,a,´Y´,.].
      X=Y).
teste(>=,
      [´Verifica´,se,´X´,e,´´,maior,ou,igual,a,´Y´,.],
      X>=Y).
teste(=<,
      [´Verifica´,se,´X´,e,´´,menor,ou,igual,a,´Y´,.],
      X=<Y).
teste(=\=,
      [´Verifica´,se,o,inteiro,´X´,e,´´,diferente,do,
       ´inteiro´,´Y´,.],
      X=\=Y).
teste(\+,
      [´Sucede´,quando,´P´,for,insatisfazivel,.],
      \+P).
elemento(portador_permanente,
         [´Elemento´,da,rede,comportamental,resultante,
          da,compilacao,da,estrutura,reg,identificador,
          .,´Armazena´,informacoes,modificaveis,.]).
padrao_elem(portador_permanente,
            port([P,N],perm),[perm]).
nivel_elem(portador_permanente,
           0).
estilos(portador_permanente,
        [todos]).
origem(portador_permanente,
       [compilacao]).
resulta(portador_permanente,
        [map_permanentes,
         map_modif_simples,
         map_modif_multiplo1,
         map_modif_multiplo2]).
elemento(portador_constante,
         [´Elemento´,da,rede,comportamental,resultante,
          o,uso,de,constante,inteira,no,algoritmo,que,
          descreve,o,circuito,.,´Armazena´,informacao,
          constante,.]).
padrao_elem(portador_constante,
```

```
             port([P,N],const(C)),[const(C)]).
nivel_elem(portador_constante,
            0).
estilos(portador_constante,
         [todos]).
origem(portador_constante,
        [compilacao]).
resulta(portador_constante,
         [map_constantes]).
elemento(portador_temporario,
          ['Elemento',da,rede,comportamental,utilizado,
           para,identificar,linhas,de,dados,.]).
padrao_elem(portador_temporario,
             port([P,N],temp),[temp]).
nivel_elem(portador_temporario,
            0).
estilos(portador_temporario,
         [todos]).
origem(portador_temporario,
        [compilacao]).
resulta(portador_temporario,
         [indeterminado]).
elemento(portador_entrada,
          ['Elemento',da,rede,comportamental,resultante,
           da,compilacao,de,declaracoes,de,entrada,.,
           'Indica',entrada,sde,dados,no,circuito,.]).
padrao_elem(portador_entrada,
             port([P,N],entr),[entr]).
nivel_elem(portador_entrada,
            0).
estilos(portador_entrada,
         [todos]).
origem(portador_entrada,
        [compilacao]).
resulta(portador_entrada,
         [map_entradas,
          map_ligadores_entrada]).
elemento(portador_saida,
          ['Elemento',da,rede,comportamental,resultante,
           da,compilacao,da,estrutura,sai,identificador,
           .,'Indica',saida,de,dados,do,circuito,.]).
padrao_elem(portador_saida,
             port([P,N],sai),[sai]).
nivel_elem(portador_sai,
            0).
estilos(portador_saida,
         [todos]).
origem(portador_saida,
        [compilacao]).
resulta(portador_saida,
         [map_saidas]).
elemento(portador_controle,
          ['Elemento',de,nivel,comportamental,que,porta,
           informacao,de,controle,.]).
padrao_elem(portador_controle,
```

```
            port([P,N],contr),[contr]).
nivel_elem(portador_controle,
           0).
estilos(portador_controle,
        [todos]).
origem(portador_controle,
       [compilacao]).
resulta(portador_controle,
        [indeterminado]).
elemento(transf_modificacao,
         ['Elemento',da,rede,comportamental,oriundo,
          da,compilacao,de,comandos,de,atribuicao,.,
          'Representa',o,operador,de,atribuicao,.=,
          .]).
padrao_elem(transf_modificacao,
            transf([T,N],mu),[mu]).
nivel_elem(transf_modificacao,
           0).
estilos(transf_modificacao,
        [todos]).
origem(transf_modificacao,
       [compilacao]).
resulta(transf_modificacao,
        [indeterminado]).
elemento(transf_extracao,
         ['Elemento',da,rede,comportamental,oriundo,
          da,compilacao,de,expressoes,que,envolvam,
          portadores,permanentes,ou,constantes,.,
          'Representa',o,operador,de,extracao,de,
          informacao,armazenada,permanentemente,.]).
padrao_elem(transf_extracao,
            transf([T,N],epsi),[epsi]).
nivel_elem(transf_extracao,
           0).
estilos(transf_extracao,
        [todos]).
origem(transf_extracao,
       [compilacao]).
resulta(transf_extracao,
        [map_extratores]).
elemento(transf_controle,
         ['Elemento',de,nivel,comportamental,oriundo,
          da,compilacao,de,todos,os,comandos,exceto,
          o,comando,de,ligacao,.]).
padrao_elem(transf_controle,
            transf([T,N],contr),[contr]).
nivel_elem(transf_controle,
           0).
estilos(transf_controle,
        [todos]).
origem(transf_controle,
       [compilacao]).
resulta(transf_controle,
        [indeterminado]).
elemento(transf_ligacao,
```

```
          ['Elemento',da,rede,comportamental,resultante,
           da,compilacao,de,comandos,de,ligacao,e,do,
           uso,de,portadores,de,entrada,como,operandos,
           de,expressoes,.]).
padrao_elem(transf_ligacao,
            transf([T,N],=>),[=>]).
nivel_elem(transf_ligacao,
           0).
estilos(transf_ligacao,
        [todos]).
origem(transf_ligacao,
       [compilacao]).
resulta(transf_ligacao,
        [indeterminado]).
elemento(transf_aritmetico,
          ['Elemento',da,rede,comportamental,originado,
           da,compilacao,de,espressoes,aritmeticas,.]).
padrao_elem(transf_aritmetico,
            transf([T,N],OP),[+,-,*,/]).
nivel(transf_aritmetico,
      0).
estilos(transf_aritmetico,
        [todos]).
origem(transf_aritmetico,
       [compilacao]).
resulta(transf_aritmetico,
        [indeterminado]).
elemento(transf_relacional,
          ['Elemento',da,rede,comportamental,oriundo,
           da,compilacao,de,esxpressoes,relacionais,.]).
padrao_elem(transf_relacional,
            transf([T,N],OP),[=,>,<,>=,=<]).
nivel_elem(transf_relacional,
           0).
estilos(transf_relacional,
        [todos]).
origem(transf_relacional,
       [compilacao]).
resulta(transf_relacional,
        [indeterminado]).
elemento(transf_se,
          ['Elemento',de,nivel,comportamental,oriundo,
           da,compilacao,de,comandos,se,.]).
padrao_elem(transf_se,
            transf([T,N],se],[se]).
nivel_elem(transf_se,
           0).
estilos(transf_se,
        [todos]).
origem(transf_se,
       [compilacao]).
resulta(transf_se,
        [indeterminado]).
elemento(transf_enq,
          ['Elemento',de,nivel,comportamental,oriundo,
```

```
          da,compilacao,de,comandos,enquanto,.]).
padrao_elem(transf_enq,
            transf([T,N],enq),[enq]).
nivel_elem(transf_enq,
           0).
estilos(transf_enq,
        [todos]).
origem(transf_enq,
       [compilacao]).
resulta(transf_enq,
        [indeterminado]).
elemento(registrador,
         ['Elemento',de,nivel,estrutural,que,
          representa,um,registrador,de,leitura,
          e,escrita,.]).
padrao_elem(registrador,
            reg([R,N]),[]).
nivel_elem(registrador,
           1).
estilos(registrador,
        [basico]).
origem(registrador,
       [map_permanentes]).
resulta(registrador,
        [indeterminado]).
elemento(registrador_constante,
         ['Elemento',de,nivel,estrutural,que,
          representa,um,registrador,de,apenas,
          leitura,.]).
padrao_elem(registrador_contante,
            reg([R,N],cont(C)),[const(C)]).
nivel_elem(registrador_constante,
           1).
estilos(registador_constante,
        [basico]).
origem(registador_constante,
       [map_constantes]).
resulta(registrador_constante,
        []).
elemento(linha_de_sinal,
         ['Elemento',de,nivel,estrutural,que,
          representa,um,linha,ou,conjunto,de,
          linhas,de,sinal,.]).
padrao_elem(linha_de_sinal,
            lin([L,N]),[]).
nivel_elem(linha_de_sinal,
           1).
estilos(linha_de_sinal,
        [todos]).
origem(linha_de_sinal,
       [indeterminado]).
resulta(linha_de_sinal,
        [indeterminado]).
elemento(operador,
         ['Elemento',de,nivel,estrutural,que,
```

```
        representa,um,operador,aritmetico,´,´,
        relacional,ou,um,seletor,.]).
padrao_elem(operador,
        oper([O,N],OP),[+,-,*,/,=,>,<,>=,=<]).
nivel_elem(operador,
        1).
estilos(operador,
        [basico]).
origem(operador,
        [map_operadores,
         map_modif_multiplo1]).
resulta(operador,
        []).
marca(inicio_bloco,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       .,'Indica',inicio,de,um,bloco,de,comandos,.],
      c).
padrao_marca(inicio_bloco,
             inicio_bloco(P)).
arg_marca(inicio_bloco,
          port([P,N],contr),[contr],
          N=0).
marca(fim_bloco,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       .,'Indica',fim,de,um,bloco,de,comandos,.],
      c).
padrao_marca(fim_bloco,
             fim_bloco(P)).
arg_marca(fim_bloco,
          port([P,N],contr),[contr],
          N=0).
marca(entao,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       que,sao,saidas,entao,de,transformadores,se,.],
      c).
padrao_marca(entao,
             entao(P)).
arg_marca(entao,
          port([P,N],contr),[contr],
          N=0).
marca(senao,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       que,sao,saidas,senao,de,transformadores,se,.],
      c).
padrao_marca(senao,
             senao(P)).
arg_marca(senao,
          port([P,N],contr),[contr],
          N=0).
marca(fazer,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       que,sao,saidas,fazer,de,transformadores,enq,.],
      c).
padrao_marca(fazer,
             fazer(P)).
```

```
arg_marca(fazer,
          port([P,N],contr),[contr],
          N=0).
marca(fim_enq,
      ['Marca',aplicavel,sobre,portadores,de,controle,
       que,sao,saidas,fim,de,transformadores,enq,.],
      c).
padrao_marca(fim_enq,
             fim_enq(P)).
arg_marca(fim_enq,
          port([P,N],contr),[contr],
          N=0).
marca(interface,
      ['Marca',aplicavel,sobre,elementos,de,controle,
       '(',nivel,0,')',que,sao,a,interface,entre,os,
       blocos,operacional,e,de,controle,.],
      b).
padrao_marca(interface,
             interface(P)).
arg_marca(interface,
          P,[contr],
          N=0).
marca(entrada_circuito,
      ['Identifica',uma,linha,'L',como,entrada,de,
       dados,do,circuito,.],
      e).
padrao_marca(entrada_circuito,
             entrada_circ(L)).
arg_marca(entrada_circuito,
          lin([L,N]),[],
          N>0).
marca(saida_circuito,
      ['Identifica',uma,linha,'L',como,saida,de,dados,
       do,circuito,.],
      e).
padrao_marca(saida_circuito,
             saida_circ(L)).
arg_marca(saida_circuito,
          lin([L,N]),[],
          N>0).
marca(selecao,
      ['Caracteriza',o,operador,'S',como,receptor,de,
       sinal,de,selecao,.],
      e).
padrao_marca(selecao,
             sel(S)).
arg_marca(selecao,
          oper([S,N],sel),[sel],
          N>0).
relacao(entrada_de_transformador,
        ['E','',utilizada,na,definicao,da,topologia,da,
         rede,comportamental,.,'Significa',':','P',e,
         '',portador,de,entrada,do,transformador,'T',
         ',',sendo,o,indice,da,entrada,dado,por,'I',.],
        c).
```

```
padrao_rel(entrada_de_transformador,
           port_entr(T,P,I)).
arg_rel(entrada_de_transformador,
        1,
        elemento(transf([T,N],Tipo)),
        nivel(N>=0)).
arg_rel(entrada_de_transformador,
        2,
        elemento(port([P,N],Tipo)),
        nivel(N>=0)).
arg_rel(entrada_de_transformador,
        3,
        indice,
        valores([const(I),contr,cond])).
relacao(saida_de_transformador,
        ['E','',utilizada,na,definicao,da,topologia,da,
         rede,comportamental,.,'Significa',':','P',e,
         '',portador,de,saida,do,transformador,'T',',',
         sendo,o,indice,da,saida,dado,pelo,inteiro,'I',
         .],
        c).
padrao_rel(saida_de_transformador,
           port_sai(T,P,I)).
arg_rel(saida_de_transformador,
        1,
        elemento(transf([T,N],Tipo)),
        nivel(N>=0)).
arg_rel(saida_de_transformador,
        2,
        elemento(port([P,N],Tipo)),
        nivel(N>=0)).
arg_rel(saida_de_transformador,
        3,
        indice,
        valores([const(I)])).
relacao(mapear,
        ['Relacao',basica,para,mapeamento,de,redes,que,
         mapeia,um,elemento,de,nivel,'N',para,outro,de,
         nivel,'N',+,1,.],
        b).
padrao_rel(mapear,
           map(E,En)).
arg_rel(mapear,
        1,
        elemento(Padrao),
        (N>=0)).
arg_rel(mapear,
        2,
        elemento(Padrao),
        (M>N)).
relacao(linha_de_entrada,
        ['Define',a,relacao,':','L',e,'',linha,de,
         entrada,no,elemento,'E',',',sendo,o,indice,
         da,entrada,dado,por,'I',.],
        e).
```

```
padrao_rel(linha_de_entrada,
           entr(E,L,I)).
arg_rel(linha_de_entrada,
        1,
        elemento(Padrao),
        nivel(N>O)).
arg_rel(linha_de_entrada,
        2,
        elemento(lin([L,N])),
        nivel(N>O)).
arg_rel(linha_de_entrada,
        3,
        indice,
        valores([const(I),contr,sel])).
relacao(linha_de_saida,
        ['Define',a,relacao,':','L',e,'',linha,de,
         saida,do,elemento,'E',',',sendo,o,indice,
         da,saida,dado,por,'I',.],
        e).
padrao_rel(linha_de_saida,
           sai(E,L,I)).
arg_rel(linha_de_saida,
        1,
        elemento(Padrao),
        nivel(N>O)).
arg_rel(linha_de_saida,
        2,
        elemento(lin([L,N])),
        nivel(N>O)).
arg_rel(linha_de_saida,
        3,
        indice,
        valores([const(I)])).
regra(map_permanentes,
      ['Mapear',portadores,permanentes,para,subredes,
       com,registradores,de,leitura,e,escrita,.]).
condicao(map_permanentes,
         [P],
         (port([P,O],perm))).
acao(map_permanentes,
     [P],
     (criar(reg([R,1])),
      criar(lin([S,1])),
      criar(lin([K,1])),
      marcar(map(P,R)),
      marcar(sai(R,S,O)),
      marcar(entr(R,K,contr)))).
regra(map_constantes,
      ['Mapear',portadores,constantes,para,subredes,
       com,registradores,de,apenas,leitura,.]).
condicao(map_constantes,
         [P,C],
         (port([P,O],const(C)))).
acao(map_constantes,
     [P,C],
```

```
      (criar(reg([R,1],const(C))),
       criar(lin([S,1])),
       marcar(map(P,R)),
       marcar(sai(R,S,0)))).
regra(map_entradas,
       ['Mapear',portadores,de,entrada,.]).
condicao(map_entradas,
          [P],
          (port([P,0],entr))).
acao(map_entradas,
      [P],
      (criar(lin([E,1])),
       marcar(map(P,E)),
       marcar(entrada_circ(E)))).
regra(map_ligadores_entrada,
       ['Mapear',ligadores,de,entrada,.]).
condicao(map_ligadores_entrada,
          [S,E],
          (port([P,0],entr),
           map(P,E),
           port_entr(T,P,0),
           port_sai(T,S,0))).
acao(map_ligadores_entrada,
      [S,E],
      (marcar(map(S,E)))).
regra(map_extratores,
       ['Mapear',operadores,que,extraem,informacoes,de,
        portadores,permanentes,e,constantes,.]).
condicao(map_extratores,
          [S,L],
          (transf([T,0],epsi),
           port_entr(T,P,0),
           port_sai(T,S,0),
           map(P,R),
           sai(R,L,0))).
acao(map_extratores,
      [S,L],
      (marcar(map(S,L)))).
regra(map_operadores,
       ['Mapear',operadores,aritmeticos,e,relacionais,
        .]).
condicao(map_operadores,
          [OP,E1,E2,P],
          transf([T,0],OP),
          operador(OP),
          port_entr(T,P1,1),
          port_entr(T,P2,2),
          port_sai(T,P,0),
          map(P1,E2),
          map(P2,E2))).
acao(map_operadores,
      criar(oper([O,1],OP)),
      criar(lin([S,1])),
      marcar(entr(O,E1,1)),
      marcar(entr(O,E2,2)),
```

```
     marcar(sai(O,S,O)),
     marcar(map(P,S)))).
regra(map_modif_simples,
      ['Mapear',subredes,em,que,ha,um,operador,de,
       modificacao,por,portador,permanente,.]).
condicao(map_modif_simples,
         [R,L,C,K],
         (port([P,O],perm),
          cardinal(T,port_sai(T,P,O),1),
          port_sai(T,P,O),
          port_entr(T,E,O),
          port_entr(T,C,contr),
          map(P,R),
          map(E,L),
         entr(R,K,contr))).
acao(map_modif_simples,
     [R,L,C,K],
     (marcar(entr(R,L,O)),
      marcar(map(C,K)))).
regra(map_modif_multiplo1,
      ['Criar',seletores,de,entrada,.]).
condicao(map_modif_multiplo1,
         [R],
         (port([P,O],perm,
          cardinal(T,port_sai(T,P,O),V),
          V>1,
          map(P,R))).
acao(map_modif_multiplo1,
     [R],
     (criar(oper([S,1],sel)),
      criar(lin([E,1])),
      criar(lin([N,1])),
      marcar(entr(R,E,O)),
      marcar(sai(S,E,O)),
      marcar(sel(S)),
      marcar(entr(S,N,sel)))).
regra(map_modif_multiplo2,
      ['Ligar',entradas,ao,seletor,.]).
condicao(map_modif_multiplo2,
         [S,N,K,Ci,Ei,O],
         (port([P,O],perm)),
          map(P,R),
          entr(R,E,O),
          sai(S,E,O),
          entr(S,N,sel),
          entr(R,K,contr),
          ordem_inicial(O),
          port_sai(T,P,O),
          port_entr(T,Pi,O),
          port_entr(T,Ci,contr),
          map(Pi,Ei),
          ordem(O))).
acao(map_modif_multiplo2,
     [S,N,K,Ci,Ei,O],
     (criar(port([Pc,1],contr)),
```

```
        criar(transf([Tc,1],contr)),
        marcar(map(Ci,Pc)),
        marcar(map(Ci,Tc)),
        marcar(port_entr(Tc,Pc,0)),
        marcar(port_sai(Tc,K,0)),
        marcar(port_sai(Tc,N,cont(0))),
        marcar(entr(S,Ei,0)))).
regra(map_saidas,
        ['Mapear',portadores,de,saida,.]).
condicao(map_saidas,
           [P,S],
           (port([P,0],sai),
            port_sai(T,P,0),
            port_entr(T,E,0),
            map(E,S))).
acao(map_saidas,
      [P,S],
      (marcar(map(P,S)),
       marcar(saida_circ(S)))).
estilo(basico,
         ['Sequencia',de,regras,que,fazem,o,mapeamento,
          rede,comportamental,'-->',rede,estrutural,
          basica,.],
         [map_permanentes,
          map_constantes,
          map_entradas,
          map_ligadores_entrada,
          map_extratores,
          map_operadores,
          map_modif_simples,
          map_modif_multiplo1,
          map_modif_multiplo2,
          map_saidas]).
fim_base.
```

6.3 Opção b: edição de base de conhecimento

Esta opção invoca o editor de bases de conhecimento. Este é um módulo interativo que permite as operações de consulta, inclusão, exclusão e alteração de itens de conhecimento contidos na base indicada pelo utilizador. Além disso, admite a possibilidade de gerar o texto que a descreve, o qual pode permitir a sua construção no caso dessa base ser indevidamente destruída.

Quaisquer modificações sobre itens de conhecimento da

base são registradas no arquivo PROTO/TRANSA/<nome da base>, que apresenta a situação de cada item modificado antes e após a modificação, discriminando-a. Cabe ressaltar que só são admitidas operações de modificação (inclusão, exclusão e alteração) sobre elementos, marcas e relações de níveis estruturais.

As seções seguintes descrevem as possíveis operações solicitadas pelo utlizador, tanto de exame quanto de modificação da base. Também é apresentado um exemplo de interação entre o utilizador e o editor de bases.

6.3.1 Consultas à base de conhecimento

Uma consulta é identificada pelo termo **liste** (ou **l**) seguido do(s) item(ns) desejado(s). O utilizador pode obter:

(a) A relação de todos os itens de conhecimento de uma determinada classe (elemento, marca, relação, regra, estilo ou teste) juntamente com seus objetivos. O diagrama seguinte apresenta a sintaxe dessa classe de consulta.

```
!------> liste ----------> elementos -------> . ----->!
    !          ^     !                       ^
    !---> l ------!  !---> marcas ----->!
                     !                  !
                     !---> relacoes --->!
                     !                  !
                     !---> regras ----->!
                     !                  !
                     !---> estilos ---->!
                     !                  !
                     !---> testes ----->!
```

A relação que é resposta a este tipo de consulta apresenta a seguinte forma geral:

```
<nome do primeiro item>;

objetivo: <objetivos do primeiro item>

          . . . . .

<nome do n-ésimo item>;

objetivo: <objetivos do n-ésimo item>
```

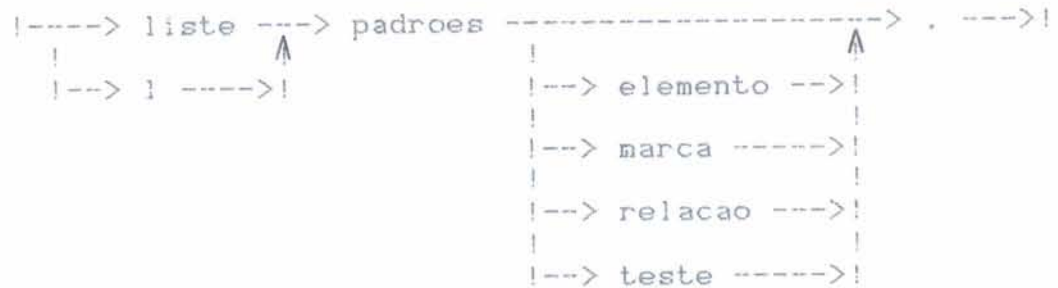
(b) A definição de um item de conhecimento específico.

```
!-----> liste -------> elemento ----> <nome> --> . -->!
   !            !  !                  !
   !--> l ---->!   !--> marca ---->!
                   !                  !
                   !--> relacao -->!
                   !                  !
                   !--> regra ---->!
                   !                  !
                   !--> estilo --->!
                   !                  !
                   !--> teste ---->!
```

São fornecidos ao utilizador todos os dados sobre o item solicitado, sendo que são considerados dados seus objetivos, padrão, condições, etc. O seguinte trecho de interação ilustra essa classe de consulta.

```
U: liste elemento portador_controle.
S: objetivo: Elemento de nivel comportamental que porta
             informacao de controle.;
   padrao: port([P,N],contr);
   nivel: 0;
   estilos: todos;
   origem: indeterminado;
   resulta: indeterminado
   #
```

(c) A relação de todos os padrões possíveis de elementos, relações, marcas e/ou testes presentes na base. A sintaxe desse tipo de consulta é apresentada no diagrama a seguir.



Esta relação é fornecida ao utilizador no seguinte formato:

<nome do primeiro item> : <padrão do primeiro item>

. . . . .

<nome do n_ésimo item> : <padrão do n-ésimo item>

(d) Um determinado procedimento Prolog associado a um teste:

```
!----> liste ---> procedimento ---> <nome do teste> -->
  !          ^
  !--> l ---->!

>-----> <número de ordem do procedimento> ----> . --->!
```

Por exemplo, para listar a segunda cláusula Prolog do teste const, envia-se:

```
U: liste procedimento const 2.
S: digito(X):-X>47,X<58.
```

6.3.2 Inclusão de itens de conhecimento

Uma inclusão de item de conhecimento é identificada pelo termo **inclui** (ou apenas **i**) seguido da definição do item, tal como é mostrado no próximo diagrama sintático.

```
!----> inclui -------> teste --------> <nome> ---> . --->
  !            ^  !                ^
  !--> i ----->!  !--> elemento -->!
                  !                !
                  !--> marca ----->!
                  !                !
                  !--> relacao --->!
                  !                !
                  !--> regra ----->!
                  !                !
                  !--> estilo ---->!

>---------------> <dados do item> --------> # -------->!
```

Os **dados do item** obedecem à sintaxe do texto de descrição da base apresentada na seção 6.2.1. Além disso, as condições para a aceitação do item e os ajustes efetuados na base devido à sua inclusão são idênticos aos descritos para o módulo de construção de bases de conhecimento.

6.3.3 Exclusão de itens de conhecimento

A exclusão de um item de conhecimento é identificada pelo termo **exclui**, ou simplesmente **e**, seguido da identificação do item a ser suprimido.

```
!----> exclui -------> elemento ---> <nome> ---> . --->!
  !            ^  !              ^
  !--> e ----->!  !--> marca --->!
                  !              !
                  !--> relacao ->!
                  !              !
                  !--> regra --->!
                  !              !
                  !--> estilo -->!
                  !              !
                  !--> teste --->!
```

A exclusão de itens das classes teste, elemento, marca ou relação são aceitas imediatamente pelo editor apenas se nenhum outro item da base os referenciar. Em caso contrário, uma mensa-

gem explicativa é enviada ao utilizador juntamente com a solicitação de confirmação da operação. A confirmação causa a supressão do item e de todas as suas referências na base.

A exclusão de um item da classe regra leva ao procedimento descrito acima apenas se ele for referenciado por algum estilo. De outra forma, a operação comporta-se exatamente como na exclusão de itens da classe estilo, com a supressão do item e de todas as referências a este na base.

6.3.4 Alteração de itens de conhecimento

Alterações em itens de conhecimento das classes elemento, marca, relação, regra ou estilo são identificadas pelo termo **altere**, ou apenas **a**, seguido da identificação do item e dos dados a serem alterados.

```
!----> altere -------> elemento ---> <nome> ---> . --->
 !            ^    !              ^
 !--> a ----->!    !--> marca --->!
                   !              !
                   !--> relacao ->!
                   !              !
                   !--> regra --->!
                   !              !
                   !--> estilo -->!

          -------------- ; <-------------
          !                             !
          V                             !
>-------------> <dado do item> --------------> # ----->!
```

Neste tipo de operação cada novo **dado do item** substitui integralmente o anterior, sendo realizados os ajustes necessários quando a alteração refletir-se em outros itens da base. Entretanto, alterações em padrões de elementos ou relações que modificam

o número de argumentos desses predicados somente são efetuadas mediante confirmação da operação por parte do utilizador, sendo que as referências ao padrão antigo são removidas sem serem substituídas pelo novo padrão.

Alterações em itens de conhecimento da classe teste podem ser feitas somente sobre seus objetivos ou procedimentos Prolog associados. Alteração sobre os objetivos de um teste obedecem às mesmas regras descritas acima. Já alterações sobre os procedimentos Prolog exige o número de ordem desse, sendo possível:

(a) Substituir integralmente um procedimento.

```
!-----> altere -----> teste -----> <nome> ----> . ---->
   !            ^
   !---> a ----->!

>-------> procedimento ------> <número de ordem> ------>

>--------> <fato ou regra Prolog> --------> # ------->!
```

(b) Suprimir um procedimento.

```
!-----> altere -----> teste -----> <nome> ----> . ---->
   !            ^
   !---> a ----->!

>---> procedimento ---> <número de ordem> ---> # ---->!
```

(c) Acrescentar um procedimento com número de ordem j.

```
!-----> altere -----> teste -----> <nome> ----> . ---->
   !            ^
   !---> a ----->!

>--> procedimento --> <ordem i> --> - --> <ordem j> -->

>----> : ----> <fato ou regra Prolog> ----> # ------->!
```

Nas situações (a) e (b) os números de ordem que seguem

os procedimentos suprimidos ou acrescentados são ajustados automaticamente pelo sistema.

6.3.5 Exemplo de edição

Nas páginas seguintes apresenta-se o uso do editor de bases de conhecimento para inclusão do estilo **estendido** na base PROTO/BC.

```
S: Forneca o nome da base a ser editada.
U: 'PROTO/BC'.
S: Voce pode:
   - Examinar a base de conhecimento (liste);
   - Incluir informacoes na base (inclua);
   - Excluir informacoes da base (exclua);
   - Alterar informacoes (altere);
   - Obter  um  texto com a descricao  da  base  (liste
     base).
U: liste elemento operador.
S: objetivo:  Elemento   de    nivel   estrutural   que
              representa   um   operador    aritmetico,
              relacional ou um seletor de entradas.;
   padrao: oper([O,N],OP) -> +,-,*,/,=,>,<,>=,=<,sel;
   nivel:1;
   estilo: basico;
   origem: map_operadores,
           map_modif_multiplo1
   #
U: altere elemento operador.
   objetivo: Elemento  estrutural  que  representa  um
             operador  aritmetico, relacional,  seletor
             de entradas, incrementador, decrementador,
             deslocador de bits ou comparador com 0.;
   padrao: oper([O,N],OP) -> +,-,*,/,=,>,<,>=,=<, sel,
           +1,-1,<-,->,'=0?','>0?','<0?','>=0?','=<0?';
   #
S: Elemento alterado.
U: inclui teste relacional.
   objetivo: Verifica  se o  tipo  OP1  do  operador e'
             relacional, associando-o ao comparador OP2
             correspondente.;
   padrao: relacional(OP1,OP2);
   relacional(=,'=0?').
   relacional(>,'>0?').
   relacional(<,'<0?').
   relacional(>=,'>=0?').
   relacional(=<,'=<0?').
   #
```

```
S: Teste incluido.
U: inclui teste soma_sub.
   objetivo: Verifica se o tipo OP1 do operador e´ + ou
             -, associando-o com o OP2 correspondente.;
   padrao: soma_sub(OP1,OP2);
   soma_sub(+,+1).
   soma_sub(-,-1).
   #
S: Teste incluido.
U: inclui teste mult_div.
   objetivo: Verifica se o tipo OP1 do operador e´ * ou
             /, associando-o com o OP2 correspondente.;
   padrao: mult_div(OP1,OP2);
   mult_div(*,<-).
   mult_div(/,->).
   #
S: Teste incluido.
U: inclui teste valor_reset.
   objetivo: Predicado que diz o numero de entradas de
             um seletor S mapeadas para sinais de
             reset.;
   padrao. valor_reset(S,V)
   #
S: Teste incluido.
U: inclui teste reset_inicial.
   objetivo: Clausula que cria o predicado  valor_reset
             para um seletor S com valor associado 0.;
   padrao: reset_inicial(S);
   reset_inicial(S):-asserta(valor_reset(S,0)).
   #
S: Teste incluido.
U: inclui teste incr_reset.
   objetivo: Clausula que incrementa o valor V do
             predicado valor_reset de um seletor S.;
   padrao: incr_reset(S);
   incr_reset(S):-retract(valor_reset(S,VI)),
                  VF is VI + 1,
                  asserta(valor_reset(S,VF)).
   #
S: Teste incluido.
U: altere relacao linha_de_entrada.
   argumento: 3; indice: const(I),contr,sel,reset
   #
S: Relacao alterada.
U: inclui regra map_incr_decr.
   objetivo: Mapear somas ou subtracoes com  constantes
             1 para, respectivamente,  operacoes de
             incremento ou decremento.;
   condicoes: reg([R,1],const(1)),
              sai(R,L1,O),
              oper([O,1],OP1),
              incr_decr(OP1,OP2),
              entr(O,L1,I1),
              entr(O,L2,I2),
              I1=\=I2,
```

```
              sai(O,L,O);
   criar: oper([O2,2],OP2),
          lin([E,2]);
   marcar: entr(O2,E,O),
           sai(O2,L,O),
           map(R,O2),
           map(O,O2),
           map(L1,O2),
           map(L2,E)
   #
S: Regra incluida.
U: inclui regra map_deslocadores.
   objetivo: Mapear   operacoes  de   multiplicacao  ou
             divisao com constantes 2 para operacoes de
             deslocamento de 1 bit para a esquerda (<-)
             ou para a direita (->).;
   condicoes: reg([R,1],const(2)),
              sai(R,L1,O),
              oper([O,1],OP1),
              mult_div(OP1,OP2),
              entr(O,L1,I1),
              entr(O,L2,I2),
              I1=\=I2,
              sai(O,L,O);
   criar: oper([O2,2],OP2),
          lin([E,2]);
   marcar: entr(O2,E,O),
           sai(O2,L,O),
           map(R,O2),
           map(O,O2),
           map(L1,O2),
           map(L2,E)
   #
S: Regra incluida.
U: inclui regra map_comparadores_0.
   objetivo: Mapear   operacoes   de   comparacao   com
             com  constantes  0  para  os  comparadores
             '=0?', '>0?', '<0?', '>=0?' ou '=<0?'.;
   condicoes: reg([R,1],const(0)),
              sai(R,L1,O),
              oper([O,1],OP1),
              relacional(OP1,OP2),
              entr(O,L1,I1),
              entr(O,L2,I2),
              I1=\=I2,
              sai(O,L,O);
   criar: oper(O2,E,O),
          lin([E,2]);
   marcar: entr(O2,E,O),
           sai(O2,L,O),
           map(R,O2),
           map(O,O2),
           map(L1,O2),
           map(L2,O2)
   #
```

```
S: Regra incluida.
U: inclui regra map_reset_simples.
   objetivo: Mapear entrada  unica de registrador  para
             sinal de reset.;
   condicoes: reg([C,1],const(0)),
              sai(C,E,0),
              entr(R,E,0),
              entr(R,K,contr),
              sai(R,S,0);
   criar: reg([R2,2]),
          lin([K2,2]);
   marcar: entr(R2,K2,reset),
           sai(R2,S,0),
           map(R,R2),
           map(K,K2),
           map(E,K2),
           map(C,K2)
   #
S: Regra incluida.
U: inclui regra map_reset_multiplo1.
   objetivo: Criar sinais de reset.;
   condicoes: oper([O,1],sel),
              reset_inicial(S),
              entr(S,N,sel),
              entr(S,E,I),
              sai(RC,E,0),
              reg([Rc,1],const(0)),
              port_sai(T,N,const(I)),
              port_entr(T,C,0),
              incr_reset(S);
   criar: lin([K2,2]);
   marcar: map(E,K2),
           map(RC,K2),
           map(T,K2),
           map(C,K2)
   #
S: Regra incluida.
U: inclui regra map_reset_multiplo2.
   objetivo: Criar seletor e registrador de nivel 2.;
   condicoes: valor_reset(S,V),
              V>0,
              sai(S,L,0),
              entr(R,L,0),
              entr(S,N,sel),
              entr(R,K,contr),
              sai(R,Ls,0);
   criar: oper([S2,2],sel),
          reg([R2,2]),
          lin([N2,2]),
          lin([L2,2]),
          lin([K2,2]);
   marcar: sai(S2,L2,0),
           entr(R2,L2,0),
           entr(R2,K2,0),
           entr(S2,N2,sel),
```

```
          sai(R2,Ls,O),
          map(S,S2),
          map(R,R2),
          map(N,N2),
          map(K,K2)
   #
S: Regra incluida.
U: inclui regra map_reset_multiplo3.
   objetivo: Ligar  entradas  diferentes  de  reset  ao
             seletor.;
   condicoes: valor_reset(S,V),
              V>0,
              map(S,S2),
              entr(S,N,sel),
              map(N,N2),
              sai(S2,L,O),
              entr(R,L,O),
              ordem_inicial(O),
              entr(S,E,I),
              \+ map(E,KR),
              port_sai(T,N,const(I)),
              port_entr(T,P,O),
              ordem(O);
   criar: transf([T2,2],contr);
   marcar: entr(S2,E,O),
           port_sai(T2,N2,const(O)),
           port_sai(T2,K,O),
           port_entr(T2,P,O),
           map(T,T2)
   #
S: Regra incluida.
U: inclui regra map_reset_multiplo4.
   objetivo: Ligar resets ao registrador.;
   condicoes: valor_reset(S,V),
              V>0,
              map(S,S2),
              entr(S,N,sel),
              sai(S2,L,O),
              entr(R,L,O),
              entr(S,E,I),
              map(E,K);
   marcar: entr(R,K,reset)
   #
S: Regra incluida.
U: inclui regra map_reset_multiplo5.
   objetivo: Eliminar seletor quando este  possuir  uma
             unica entrada.;
   condicoes: valor_reset(SO,V),
              V>0,
              map(SO,S),
              \+ entr(S,Ek,2),
              entr(S,N,sel),
              entr(S,E,I),
              sai(S,L,O),
              entr(R,K,contr),
```

```
                port_sai(T,N,const(I)),
                port_entr(T,C,O),
                sai(R,Ls,O);
   criar: reg([R3,3]),
          lin([E3,3]),
          lin([K3,3]);
   marcar: entr(R3,E3,O),
           sai(R3,Ls,O),
           entr(R3,K3,contr),
           map(R,R3),
           map(S,E3),
           map(E,E3),
           map(L,E3),
           map(N,K3),
           map(K,K3),
           map(T,K3),
           map(C,K3)
   #
S: Regra incluida.
U: inclui estilo estendido.
   objetivo: Sequencia de regras que estendem o  estilo
             basico ao nivel de  deslocadores de  bits,
             incrementadores,  decrementadores,  sinais
             de reset e comparadores com O.;
   regras: map_permanentes,
           map_constantes,
           map_entradas,
           map_ligadores_entrada,
           map_extratores,
           map_operadores,
           map_modif_simples,
           map_modif_multiplo1,
           map_modif_multiplo2,
           map_saidas,
           map_incr_decr,
           map_deslocadores,
           map_comparadores_0,
           map_reset_simples,
           map_reset_multiplo1,
           map_reset-multiplo2,
           map_reset_multiplo3,
           map_reset_multiplo4,
           map_reset_multiplo5
   #
S: Estilo incluido.
U: liste base 'PROTO/TXT2/BC'.
S: Texto gerado --> PROTO/TXT2/BC'
U: fim
```

Além das modificações explícitas nessa interação, a base de conhecimento PROTO/BC também sofre os seguintes ajustes:

```
        . . . . . .

estilos(registrador,
        [basico,
         estendido]).
origem(registrador,
       [map_permanentes,
        map_reset_simples,
        map_reset_multiplo2,
        map_reset_multiplo5]).

        . . . . . .

estilos(registrador_constante,
        [basico,
         estendido]).

        . . . . . .

resulta(registrador_constante,
        [map_incr_decr,
         map_deslocadores,
         map_comparadores_0,
         map_reset_multiplo1]).

        . . . . . .

estilos(operador,
        [basico,
         estendido]).
origem(operador,
        [map_operadores,
         map_modif_multiplo1,
         map_incr_decr,
         map_deslocadores,
         map_comparadores_0,
         map_reset_multiplo2]).
resulta(operador,
        [map_incr_decr,
         map_deslocadores,
         map_comparadores_0,
         map_reset_multiplo2]).

        . . . . . .
```

O texto PROTO/TXT2/BC explicita apenas as informações definidas pelo utilizador de tal forma que seja este compilável pelo construtor de bases de conhecimento. Isto significa que o conhecimento pré-definido, bem como as referências de estilos,

origem e resultantes explicitados na base pelo sistema, não constam desse texto.

7 CONCLUSÃO

Podemos considerar que foram atingidos os objetivos do trabalho, que eram:

(a) implementar um protótipo de sistema especialista que utlizasse o método de solução de problemas por transformação de redes; e

(b) aplicar este método no problema de projeto lógico de blocos operacionais de circuitos digitais no nível RT.

No decorrer do trabalho foi dada ênfase a dois fatores: a independência do solucionador de problemas (interpretador das regras de transformação de redes) em relação ao problema do projeto lógico de blocos operacionais de circuitos; e a portabilidade do protótipo. O primeiro foi obtido através da separação entre interpretador de regras e o conhecimento sobre o domínio do problema. A portabilidade foi alcançada pela escolha de uma versão da linguagem Prolog muito próxima do que tende a ser o padrão para essa linguagem. De fato, o protótipo apresenta apenas detalhes mínimos não portáveis, facilmente identificáveis, mencionados nos capítulos 5 e 6.

No desenvolvimento do protótipo, as maiores dificuldades se concentraram em dois pontos: a definição da representação e organização do conhecimento, especialmente quais informações deveriam ser explicitadas para manter a coerência da base; e a definição da abrangência do subsistema de aquisição de conhecimento, sendo que se optou por uma forma simplificada, com

validações apenas ao nível sintático, deixando a responsabilidade pela correção das informações da base de conhecimento a cargo do engenheiro de conhecimento.

No âmbito do problema de projeto de circuitos digitais, este trabalho fornece um ambiente inicial para o desenvolvimento de um sistema de projeto automatizado de circuitos digitais a partir do nível RT. Para tornar o ambiente efetivo faz-se necessário:

(a) avaliar e ampliar a linguagem de descrição do circuito;

(b) definir um conjunto mais abrangente de regras de transformação de redes; e

(c) estender o protótipo para o projeto do bloco de controle.

Quanto ao mecanismo de solução de problemas, a continuação desse trabalho está na exploração de sua aplicação em outras tarefas que possam ser expressas como problemas de transformação de redes.

É nosso projeto prosseguir o trabalho neste protótipo, especialmente no que se refere à representação do conhecimento, sua aquisição e verificação de sua consistência e completude.

ANEXO: Listagem do interpretador de regras

Apresenta-se todos os procedimentos que compõem o interpretador de regras, inclusive os procedimentos definidos no bloco de controle. No interpretador, os únicos procedimentos diretamente relacionados ao problema do projeto lógico de blocos operacionais são os que fornecem o bloco operacional do circuito.

```
/*                                                              */
/*  Objetivo: Gerar a rede estrutural do bloco operacional de   */
/*  --------  um circuito logico a partir da rede comportamen-  */
/*            tal do mesmo, de acordo com especificacao  do u-  */
/*            tilizador. A  rede estrutural e' gerada  atraves  */
/*            de um processo de  transformacao de subredes, u-  */
/*            sando regras de transformacao  contidas na  base  */
/*            de conhecimento. Estas regras atuam sobre subre-  */
/*            des de nivel 0 (nivel comportamental) e/ou nivel  */
/*            maior que 0 (niveis estruturais intermediarios),  */
/*            conforme o estilo escolhido.                      */
/*                                                              */
/*  Procedimentos: 1. Obtencao da rede de entrada;              */
/*  --------------  2. Obtencao do tipo da rede de saida: final */
/*                     ou parcial;                              */
/*                  3. Obtencao do estilo de  geracao  da  rede */
/*                     estrutural;                              */
/*                  4. Verificacao do desejo de  monitorar a e- */
/*                     xecucao das regras;                      */
/*                  5. Aplicacao das regras de transformacao de */
/*                     redes; e                                 */
/*                  6. Fornecimento da rede de saida.           */
/*                                                              */
gera_estrutural(Arq):-obtem_rede(Arq),
                      obtem_base,
                      obtem_tipo_rede,
                      verifica_monitoracao,
                      interpreta_regras,
                      fornece_rede.
/*                                                              */
/*  Procedimento que obtem, se necessario, o nome do arquivo    */
/*  que contem a rede de entrada, consultando seu conteudo.     */
/*                                                              */
obtem_rede(Arq):-var(Arq),
                 write('Forneca a rede de entrada.'),nl,
                 ( read(Arq),reconsult(Arq) ;
                   write('Rede inexistente.'),
                   obtem_rede(Arq2) ).
obtem_rede(Arq):-reconsult(Arq).
```

```
/*                                                              */
/*  Procedimento que obtem o nome do arquivo que contem a base */
/*  de conhecimento, consultando o seu conteudo.                */
/*                                                              */
obtem_base:-write('Forneca a base de conhecimento.'),nl,
            ( read(Arq),reconsult(Arq) ;
              write('Base inexistente.'),nl,
              obtem_base ).
/*                                                              */
/*  Procedimento que obtem o tipo da rede a ser gerada: final  */
/*  ou parcial.                                                 */
/*                                                              */
obtem_tipo_rede:-write('Indique o tipo da rede de saida:'),nl,
                 write('(a) Rede final.'),nl,
                 write('(b) Rede parcial.'),nl,
                 read(Opcao),
                 ( Opcao=a,
                   asserta(rede(final)) ;
                   Opcao=b,
                   asserta(rede(parcial)) ;
                   write('Opcao invalida.'),nl,
                   obtem_tipo_rede ).
/*                                                              */
/*  Procedimento que obtem o estilo de geracao da rede estru-  */
/*  ral.                                                        */
/*                                                              */
obtem_estilo(E):-
     write('Forneca o estilo de geracao da rede estrutural.'),nl,
     read(X),
     ( X='L',
       printestilos,
       obtem_estilo(E) ;
       estilo(X,_,_),
       E=X ;
       write('Estilo inexistente.'),nl,
       obtem_estilo(E) ).
/*                                                              */
/*  Procedimento  que  lista  os  estilos de geracao  da rede  */
/*  estrutural existentes na base de conehcimento. Fornece  o  */
/*  nome do estilo e seu objetivo.                              */
/*                                                              */
printestilos:-estilo(E,O,_),
              write('estilo: '),
              write(E),nl,
              tab(9),
              printlist(O,9,9),
              fail.
printestilos.
/*                                                              */
/*  Procedimento que lista o conteudo de uma lista de palavras */
/*  iniciando numa coluna inicial e  trocando de linha  sempre */
/*  que a coluna ultrapassar 80.                                */
/*                                                              */
printlist([],_,_):-nl,!.
printlist([H:T],I,C):-name(H,L),
```

```
                    lenght(L,N),
                    TL is C+N,
                    ( TL > 80,
                      X is 80-C,
                      repetir(X,' '),
                      nl,
                      tab(I),
                      escr(L),
                      escr([32]),
                      P is I+1 ;
                      TL=80,
                      escr(L),
                      nl,
                      tab(I),
                      P=I ;
                      escr(L),
                      escr([32]),
                      P is TL+1 ),!,
                    printlist(T,I,P).
escr([]).
escr([H:T]):-put(H),escr(T).
repetir(0,_).
repetir(N,C).-put(C),M is N-1,repetir(M,C).
/*                                                                    */
/*  Procedimentos que verificam e efetuam a monitoracao da e-         */
/*  xecucao das regras.                                               */
/*                                                                    */
veifica_monitoracao:-
        write('Deseja monitorar a execucao das regras?'),nl,
        write('Responda sim ou nao.'),nl,
        read(R),
        ( R=sim,
          asserta(monitor(true)) ;
          R=nao,
          asserta(monitor(false)) ;
          write('Resposta invalida.'),nl,
          verifica_monitoracao ).
monitora(R):-monitor(true),
             write('Regra: '),
             write(R),nl,!,.
monitora(_):-monitor(false),!.
monitora_cond(C):-monitor(true),
                  write('Condicoes: '),
                  converte(lista,C,LC),
                  escreve_monitor(LC,12),!.
monitora_cond(C):-monitor(false),!.
monitora_acao(A):-monitor(true),
                  write('Acoes: '),
                  converte(lista,A,LA),
                  escreve_monitor(LA,8),!.
monitora_acao(A):-monitor(false),!.
monitora_instancias(C):-monitor(true),
                        write('Instancia de subrede: '),
                        converte(lista,C,LC),
                        escreve_monitor(LC,23),!.
```

```
monitora_instancias(_):-monitor(false),!.
monitora_valores(A):-monitor(true),
                     write('Procedimentos: '),
                     converte(lista,A,LA),
                     escreve_monitor(LA,16),!.
monitora_valores(_):-monitor(false),!.
escreve_monitor([],_).
escreve_monitor([H:T],C):-tab(C),
                          write(H),
                          escreve_monitor(T,C).
/*                                                              */
/*  Procedimento que obtem a sequencia de regras de transfor-   */
/*  macao de redes a ser aplicada.                              */
/*                                                              */
interpreta_regras:-estilo(E,_,Seq_regras),
                   extrai_regra(Seq_regras).
/*                                                              */
/*  Procedimento que extrai uma regra da sequancia de regras,   */
/*  invocando sua execucao.                                     */
/*                                                              */
extrai_regra([]).
extrai_regra([H:T]):-monitora(H),
                     exec(H),
                     extrai_regra(T).
/*                                                              */
/*  Procedimento que invoca as condicoes e acoes que  compoem   */
/*  um regra, unificando seus parametros (variaveis).           */
/*  Forma geral: 1. Condicoes -> condicao(<nome da regra>,      */
/*                                  <lista de variaveis>,*/
/*                                  <condicoes>).               */
/*               2. Acoes -> acao(<nome da regra>,              */
/*                               <lista de variaveis>,          */
/*                               <acoes>).                      */
/*                                                              */
exec(R):-condicao(R,P,C),
         monitora_cond(C),
         acao(R,P,A),
         monitora_acao(A),
         C,
         monitora_intancias(C),
         A,
         monitora_valores(A),
         fail.
exec(_).
/*                                                              */
/*  Duas acoes podem ser feitas pelo sistema:                   */
/*  1. criar(Pred) --> Esta acao gera um novo elemento na rede */
/*                     estrutural.                              */
/*  2. marcar(Pred) -> Esta acao estabelece um relacionamento   */
/*                     entre elementos presentes na(s) rede(s) */
/*                                                              */
marcar(Pred):-assertz(Pred).
criar(Pred):-Pred=..[_:T],
             monta_arg(T),
             assertz(Pred).
```

```
monta_arg([[P,N]]:-obtem_id(P).
monta_arg([[P,N],T]):-obtem_id(P).
obtem_id(ID):-retract(valor_id(ID_ANT)),
              ID is ID_ANT+1,
              asserta(valor_id(ID)).
/*                                                                  */
/*  Procedimento  que gera um arquivo com a rede resultante da */
/*  aplicacao das regras: bloco operacional estrutural e bloco */
/*  de controle comportamental.                                     */
/*                                                                  */
fornece_rede:-
        write('Forneca um nome para a rede de saida.'),nl,
        read(N),
        name('PROTO/ESTRUTURAL/',L1),
        name(N,L2),
        concatenar(L1,L2,L),
        name(Arq,L),
        tell(Arq),
        escreve_operacional,
        escreve_controle,
        ( rede(parcial),
          retract(valor_id(V)),
          write((valor_id(V)).) ;
          true ),
        told,
        write('Rede final gerada --> '),
        write(Arq),nl.
escreve_operacional:-write('bloco_operacioanl(estrutural).'),
                     escreve_elementos,
                     escreve_marcas,
                     escreve_relacoes.
escreve_elementos:-nivel_elem(Id,1),
                   padrao_elem(Id,P,_),
                   P,
                   busca_elementos(P),
                   fail.
escreve_elementos.
busca_elementos(P):-( P=..[NP,[E,N]],
                      \+ map(E,_),
                      Q=..[NP,E] ;
                      P=..[NP,[E,N],T],
                      \+ map(E,_),
                      Q=..[NP,E,T] ),
                    ( rede(parcial),
                      write((P).) ;
                      write((Q).) ),
                    !.
escreve_marcas:-marca(Id,_,e),
                padrao_marca(Id,P),
                arg_marca(Id,PA,LT,_),
                P,
                busca_marca(P,PA,LT),
                fail.
escreve_marcas.
busca_marca(P,PA,LT):-P=..[NP,E],
```

```
                          mapeamentos(E,En),
                          marca_compativel(En,PA,LT),
                          Q=..[NP,En],
                          write((Q).),
                          !.
marca_compativel(En,PA,LT):-var(PA),
                             ( LT=[],
                               nivel_elem(Id,N),
                               N>0,
                               padrao_elem(Id,P,[]),
                               P=..[NP,[E,_]],
                               Pn=..[NP,[En,_]],
                               Pn ;
                               nivel_elem(Id,N),
                               N>0,
                               padrao_elem(Id,P,_),
                               P=..[NP,[E,_],T],
                               Pn=..[NP,[En,_],T],
                               Pn,!,
                               membro(T,LT) ).
marca_compativel(En,PA,[]):-PA=..[NA,[A,N]],
                             P=..[NA,[En,N]],
                             P.
marca_compativel(En,PA,LT):-PA=..[NA,[A,N],T],
                             P=..[NA,[En,N],T],
                             P.
escreve_relacoes:-relacao(Id,_,e),
                  padrao_rel(Id,P),
                  P,
                  busca_relacoes(P),
                  fail.
escreve_relacoes.
busca_relacoes:-P=..[NP,E1,E2,I],
                \+ map(E1,_),
                mapeamentos(E2,En),
                Q=..[Np,E1,En,I],
                write((Q).),
                !.
mapeamentos(P,E):-map(P,I),!,
                  mapeamentos(I,E).
mapeamentos(E,E):-!.
escreve_controle:-write('bloco_controle(comportamental).'),
                  lista_portadores,
                  lista_transformadores,
                  lista_marcas_controle,
                  lista_relacoes_controle.
lista_portadores:-port([P,N],contr),
                  \+ map(P,_),
                  lista_controle(port,P,N,contr),
                  fail.
lista_portadores.
lista_transformadores:-transf([T,N],Tipo),
                       \+ map(T,_),
                       ( Tipo=contr ;
                         Tipo=se    ;
```

```
                          Tipo=enq    ),
                         lista_controle(transf,T,N,Tipo),
                         fail.
lista_transformadores.
lista_controle(NP,E,N,T):-( rede(parcial),
                             P=..[NP,[E,N],T] ;
                             P=..[NP,E,T] ),
                           write((P).),!.
lista_marcas_controle:-marca_controle(inicio_bloco),
                         marca_controle(fim_bloco),
                         marca_controle(entao),
                         marca_controle(senao),
                         marca_controle(fazer),
                         marca_controle(fim_enq).
marca_controle(NM):-M=..[NM,A],
                      M,
                      mapeamentos(A,E),
                      M2=..[NM,E],
                      write((M2).),
                      fail.
marca_controle(_).
lista_relacoes_controle:-lista_relacoes(port_entr),
                           lista_relacoes(port_sai).
lista_relacoes(NR):-( transf([T,N],contr) ;
                        transf([T,N],se)    ;
                        transf([T,N],enq)    ),
                      \+ map(T,_),
                      R=..[NR,T,P,I],
                      R,
                      escr_relacao(NR,T,P,I),
                      fail.
lista_relacoes(_).
escr_relacao(NR,T,P,I):-R=..[NR,T,I],
                          mapeamentos(P,E),
                          R2=..[NR,T,E,I],
                          write((R2).),
                          !.
/*                                                              */
/*          Fim do modulo interpretador de regras               */
/*                                                              */
```

A seguir são mostrados os operadores e procedimentos definidos no módulo de controle do subsistema de projeto lógico que são referenciados pelo interpretador de regras.

```
projeto_logico:-op(850,xf,'.'),
                pl_controle.

    . . . . . . . . . . . .

membro(H,[H:_]).
```

```
membro(I,[_:T]).-membro(I,T).
concatenar([],L,L).
concatenar([H:T1],L,[H:T2]):-concatenar(T1,L,T2).
converte(lista,(X,Y),[X:Z]):-!,converte(lista,Y,Z).
converte(lista,X,[X]).

    . . . . . . . . . . .
```

BIBLIOGRAFIA

/AIK 83/ AIKINS, Janice S. Prototypical knowledge for expert system. Artificial Intelligence, Amsterdam, 20(2):163-210, 1983.

/BAR 75/ BARBACCI, Mario R. A Comparision of register transfer languages for describing computers and digital systems. IEEE Transctions on Computers, New York, 24(2):137-50, 1985.

/BOA 86/ BOAS, Ghica van Emde & BOAS, Peter van Emde. Storing and evaluating horn-clause rules in a relational database. IBM Journal of Research and Development, New york, 30(1):80-92, 1996.

/BOD 77/ BODEN, Margaret A. Artificial intelligence and natural man. New York, Basic Books, 1977. Apud WYER, Jo-Anne. New bird on the branch: artificial intelligence and computer-assisted instruction. Bedford, Digital Equipment Corporation, 1983. p.1.

/BUC 78/ BUCHANAN, B. & FEIGENBAUM, E. DENDRAL and META-DENDRAL: their applications dimension. Artificial Intelligence, Amsterdam, 11(1):5-24, 1978.

/BOW 81/ BOWEN, D. L. DECsystem-10 Prolog user's manual. Edinburg, University of Edinburg, Department of Artificial Intelligence, 1981.

/CLA 83/ CLANCEY, William J. The Epistemology of a rule-based expert system - a framework for explanation. Artificial intelligence, Amsterdam, 20(3):215-51, 1983.

/CLO 81/ CLOCKSIN, W. F. & MELLISH, C. S. Programming in Prolog. New York, Springer-Verlag, 1981.

/COE 80/ COELHO, H. Elementos para uma engenharia da linguagem. Lisboa, LNEC, 1980.

/COH 85/ COHEN, William & GEUS, Aaart J. de. A Rule-based system for optimizing combinational logic. IEEE Desig & Test of Computers, New York, 2(4):22-32, 1985.

/COL 73/ COLMERAUER, A.; KANQUI, H.; PASERO, R. & ROUSSEL, P. Um Systeme de comunication homme-machine en français. d'AIX-MARSEILLE, G.I.A., 1973.

/COS 80/ COSTA, A. C. R. Utilização de redes de petri na descrição de sistemas de processos. Porto Alegre, PGCC da UFRGS, 1980.

/COS 82/ COSTA, A. C. R. Redes-TR: um formalismo para modelagem funcional de circuitos lógicos. Porto Alegre, PGCC da UFRGS, 1983.

/COS 83/ COSTA, A. C. R. Modelos de rede para alguns componentes de circuitos lógicos. Porto Alegre, PGCC da UFRGS, 1983.

/COS 83a/ COSTA, A. C. R. Modelos em rede para representação de conhecimentos em compiladores heurísticos de circuitos lógicos. In: SIMPOSIO BRASILEIRO DE CONCEPÇAO DE CIRCUITOS INTEGRADOS, 1, Porto Alegre, nov. 7-11, 1983. Anais, Porto Alegre, Soc. Bras. de Computação, Novembro, 1983. p.53-73.

/COS 84/ COSTA, A. C. R. Caracterização dos conhecimentos e da arquitetura de um sistema especialista em projeto lógico de circuitos digitais. Porto Alegre, PGCC da UFRGS, 1984.

/COS 85/ COSTA, A. C. R. Especificação das tarefas do sistema especialista em projeto lógico de circuitos digitais. Porto Alegre, PGCC da UFRGS, 1985.

/DAV 77/ DAVIS, Randall & BUCHANAN, Bruce. Production rules as a representation for a knowledge-based consultation program. Artificial Intelligence, Amsterdam, 8(1):15-45, 1977.

/DAV 79/ DAVIS, Randall. Interactive transfer of expertise: acquisition of new inference rules. Artificial Intelligence, Amsterdam, 12(2):121-57, 1979.

/DUD 83/ DUDA, Richard O. & SHORTLIFFE, Edward H. Expert Systems Research. Science, New York, 220(4594): 121-57, 1983.

/FEI 83/ FEIGENBAUM, Edward & McCORDUCK, Pamela. The Fifth generation. Reading, Addison-Wesley, 1983.

/FIK 85/ FIKES, R. & KEHLER, T. The Role of frame-based representation in reasoning. Communications of the ACM, New York, 28(9):904-20, 1985.

/FOR 81/ FORGY, C. L. OPS5 user´s manual. Pitsburg, Carnegie-Mellon University, Department of Computer Science, 1981.

/GIL 78/ GILOI, W. K.; BALACI, R. & BEHR, P. APL*DS - a powerful portable description and simulation, microprogram specification and the simulation of parallel processing concepts. Berlin, Technische Universität Berlin, 1978.

/GRE 69/ GREEN, C. Theorem-proving by resolution as a basis for question-answering system. Machine intelligence, Edinburg, Univ. Press, 1969. n.4.

/GUE 86/ GUENTHNER, Franz; LEHMANN, Hubert & SHONFELD, Wolfgang A Theory for the representation o f knowledge. IBM Journal of Research and Development, New York, 30(1):39-56, 1986.

/HEI 82/ HEINES, Jesse M. Basic concepts in knowledge based systems. Massachusets, Digital Equipment Corporation, 1982.

/HIR 86/ HIRSH, P.; KATKE, W.; SNYDER, S. & STILLMAN, R. Interfaces for knowledge-base bulders´ control knowledge and application-specific procedures. IBM Journal of Research and Development, New york, 30(1):29-38, 1986.

/KIT 83/ KITAKAMI, H.; FURUKAWA, K.; KUNIFUJI, S.; & MIYACHI, K A Methodology for implementation of a knowledge acquisition system. Tokyo, Institute of New Generation Computer Technology, 1983.

/KOW 79/ KOWALSKI, R. Logic for problem solving. Edinburg, University of Edinburg, 1979.

/KOW 85/ KOWALSKI, R. & SERGOT, M. A Logic-based calculus of events. London, Imperial College, Department of Computing, 1985.

/LEV 84/ LEVESQUE, Hector J. Fundations of a functional approach to knowledge representation. Artificial Intelligence, Amsterdam, 23(2):155-212, 1984.

/LOW 85/ LÖW, Roberto Porto. Implementação de um interpretador para a linguagem PROLOG. Porto Alegre, CPGCC da UFRGS, 1985.

/McD 81/ McDERMOTT, J. R1: a rule-based configurer of computer systems. Artificial Intelligence, Amsterdam, 19(1):1200-11, 1981.

/MIC 83/ MICHALSKY, R. S.; CARBONELL, J. G. & MICHELL, T. M., Eds. Machine learning. Palo Alto, Tioga Publing Co., 1983.

/MUE 85/ MUELLER, Robert A. & VARGHESE, Joseph. Knowledge-based code selection methods in retargetable microcode synthesis. IEEE Design & Test of Computers, New York, 2(4):44-55, 1985.

/NEW 63/ NEWEL, A.; SHAW, J. & SIMON, H. A. Empirical explorations with the logic theory machine: a case study in heuristics. In: FEINGENBAUM, E. & FELDMAN, J., Eds. Computers and thought, New York, Mcgraw Hill, 1963. p.109-33.

/NEW 63a/ NEWEL, A. & SIMON, H. A. GPS, a program that simulates human thought. In: FEINGENBAUM, E. & FELFMAN, J., Eds. Computers and thought, New York, McGraw Hill, 1963. p.279-93.

/NEW 82/ NEWEL, A. The knowledge level. Artificial Intelligence, Amsterdam, 18(1):87-127, 1982.

/REB 81/ REBOH, René. Knowledge engineering techniques and tools in the Prospector environment. Sweden, SRI International, 1981.

/RIC 83/ RICH, Elaine. Artificial Intelligence, Austin, Copyrigth, 1983.

/ROT 83/ ROTH, Frederick et alli. Building expert systems. London, Addison-Wesley, 1983.

/RYC 81/ RYCHENER, Michael D. Approaches to knowledge acquisition: the instructable production system. Pitsburg, Carnegie-Mellon University, Department of Computer Science, 1981.

/STA 77/ STALLMAN, Richard M. & SUSSMAN, Gerald J. Forward reasoning and dependency-directed circuit analyses. Artificial Intelligence, Amsterdam, 9(2):135-96, 1977.

/STE 82/ STEFIK, M.; AIKINS, J.; BALZER, R.; BENOIT, J.; ROTH, F.; BIRNBAUM, L. & SACERDOT, E. The Organization of espert systems, a tutorial. Artificial Intelligence, Amsterdam, 18(2):135-73, 1982.

/THO 81/ THOMAS, D. The Automatic sinthesis of digital systems. Proc. of the IEEE, New York, 69(10):1200-11, 1981.

/WAL 82/ WALTZ, David L. Artificial Intelligence. Scientific American, New York, 247(4):118-33, 1982.

/WAL 84/ WALKER, A. Data bases expert systems and Prolog. In: REITMAN, W., Ed. Artificial intelligence applications for business, Norwood, Ablex Publishing Co., 1984.