Pontifícia Universidade Católica do Paraná
Curso de Pós-Graduação em Informática Aplicada

Dissertação
apresentada à PUC-PR como requisito para obtenção do título de

Mestre em Ciências

por

Alexandre Denes dos Santos

Desenvolvimento de um Sistema para a Geração de Núcleos Reativos a Partir de Bases de Conhecimento

Banca Examinadora

Orientador:
Prof. Dr. Celso Antônio Alves Kaestner — PUC-PR

Examinadores:
Prof. Dr. Bráulio Coelho Ávila — PUC-PR
Prof. Dr. Joni da Silva Fraga — UFSC
Profa. Dra. Maria Carolina Monard — USP-São Carlos

Curitiba, outubro de 1998.

Alexandre Denes dos Santos

Desenvolvimento de um sistema para a geração de núcleos reativos a partir de bases de conhecimento

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciências.
Programa de Pós-Graduação em Informática Aplicada, Setor de Ciências Exatas e de Tecnologia, Pontifícia Universidade Católica do Paraná.
Orientador: Prof. Dr. Celso Antônio Alves Kaestner

Curitiba
1998

PROGRAMA DE PÓS – GRADUAÇÃO EM INFORMÁTICA APLICADA DA PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ

"Desenvolvimento de um Sistema para a Geração de Núcleos Reativos a Partir de Bases de Conhecimento"

por

Alexandre Denes dos Santos

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Ciências.

Banca Examinadora:

Prof. Dr. Celso Antônio Alves Kaestner (orientador)

Profa. Dra. Maria Carolina Monard

Prof. Dr. Joni da Silva Fraga

Prof. Dr. Bráulio Coelho Ávila

Agradecimentos

Ao Prof. Dr. Celso Antônio Alves Kaestner, pela orientação desta dissertação, por sempre me guiar pela área da Inteligência Artificial e, principalmente por confiar a mim a realização deste projeto.

Ao Prof. Cláudio de Oliveira, pela grande amizade prestada desde os tempos de graduação.

Ao Prof. Dr. Carlos Alberto Maziero, pela grande ajuda tanto no projeto quanto na elaboração deste trabalho.

Aos amigos Prof. Altair Santin, Cláudio Carvilhe, Kristian Capeline e Joel Larocca pelo apoio prestado.

À Adriane, pelo sacrifício, compreensão e motivação.

Aos meus pais Jaime e Ana, e à minha irmã Danielle, pelo suporte.

À Siemens, pelo financiamento parcial deste trabalho através da lei 8248.

Finalmente, a todos aqueles que não foram mencionados e que contribuíram, de forma direta ou indireta, na realização deste trabalho.

A.D.S.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

A presente dissertação apresenta o sistema ARKS, destinado à geração de núcleos reativos a partir de bases de conhecimento. O ARKS incorpora uma linguagem para a descrição do conhecimento, um compilador para o tratamento do conhecimento descrito pelo usuário e um gerador de código para a criação do núcleo reativo.

O sistema é baseado em um formalismo bem definido, fundamentado no cálculo de modelos de uma lógica proposicional estendida.

A linguagem ARKS utiliza regras de produção para a descrição de heurísticas e permite a partição do conhecimento em módulos, permitindo a partição do problema em sub-problemas.

O compilador realiza o tratamento do conhecimento descrito pelo usuário, através do cálculo dos modelos da base de conhecimento e armazenamento desses modelos em uma estrutura compacta e de rápida inferência.

O gerador de código cria uma implementação de um núcleo reativo a partir do comportamento descrito, em linguagem C++ ANSI ou Java, incluindo todas as funções necessárias para execução.

Resumo

A presente dissertação apresenta o sistema ARKS, destinado à geração de núcleos reativos a partir de bases de conhecimento. O ARKS incorpora uma linguagem para a descrição do conhecimento, um compilador para o tratamento do conhecimento descrito pelo usuário e um gerador de código para a criação do núcleo reativo.

O sistema é baseado em um formalismo bem definido, fundamentado no cálculo de modelos de uma lógica proposicional estendida.

A linguagem ARKS utiliza regras de produção para a descrição de heurísticas e permite a partição do conhecimento em módulos, permitindo a partição do problema em sub-problemas.

O compilador realiza o tratamento do conhecimento descrito pelo usuário, através do cálculo dos modelos da base de conhecimento e armazenamento desses modelos em uma estrutura compacta e de rápida inferência.

O gerador de código cria uma implementação de um núcleo reativo a partir do comportamento descrito, em linguagem C++ ANSI ou Java, incluindo todas as funções necessárias para execução.

Abstract

This work presents the ARKS system, intended to generate reactive kernels from knowledge bases. The ARKS incorporates a language to the knowledge description, a compiler to handle the knowledge described by the user and a code-generator to create the reactive kernel.

The systems is based in a well-defined formalism, founded in the calculus of the models in a extended proposicional logic.

The ARKS language uses production rules to the heuristics description and allows the partition of the knowledge into modules, allowing the partition of the problem in sub-problems.

The compiler handles the knowledge described by the user through the calculus of the models in the knowledge base and stores these models in a compact structure with fast inference.

The code-generator creates a source code of a reactive kernel from the described behavior, in ANSI C++ or Java, including all the needed functions for execution.

Abstract

This work presents the ARKS system, intended to generate reactive kernels from knowledge bases. The ARKS incorporates a language to the knowledge description, a compiler to handle the knowledge described by the user and a code-generator to create the reactive kernel.

The systems is based in a well-defined formalism, founded in the calculus of the models in a extended proposicional logic.

The ARKS language uses production rules to the heuristics description and allows the partition of the knowledge into modules, allowing the partition of the problem in sub-problems.

The compiler handles the knowledge described by the user through the calculus of the models in the knowledge base and stores these models in a compact structure with fast inference.

The code-generator creates a source code of a reactive kernel from the described behavior, in ANSI C++ or Java, including all the needed functions for execution.

Capítulo 1

Introdução

Atualmente poucas são as ferramentas destinadas ao desenvolvimento de sistemas reativos que utilizam bases de conhecimento como forma de descrição de seu comportamento. Estes sistemas devem interagir de maneira contínua com o ambiente, e utilizar heurísticas para a obtenção das respostas, a serem enviadas ao ambiente, a partir dos estímulos recebidos do mesmo. De fato, quando do desenvolvimento de tais sistemas, o desenvolvedor geralmente tem que optar entre ferramentas destinadas ao desenvolvimento de sistemas reativos, ou ferramentas voltadas ao desenvolvimento de sistemas baseados em conhecimento.

As ferramentas destinadas ao desenvolvimento de sistemas reativos são, em sua maioria, compiladores para linguagens inadequadas para a representação de heurísticas.

As ferramentas voltadas ao desenvolvimento de sistemas baseados no conhecimento, por sua vez, são inadequadas para a representação da interação contínua requerida pelos sistemas reativos, uma vez que tais ferramentas são voltadas para o modo de representação do conhecimento e para o processo de inferência sobre esse conhecimento, que geralmente consome muitos recursos.

Uma alternativa proposta para a solução deste problema é apresentada em [Kae93]: o sistema RETIKS — *Real-Time Knowledge-based System* — constituído de um gerador de sistemas tempo-real que utilizam bases de conhecimento para a tomada de decisão. Para isto, o usuário define o comportamento esperado do sistema em uma base de conhecimento, que é pré-processada pelo sistema, utilizando um formalismo bem definido, e armazenada em uma estrutura compacta e de rápida inferência, adequada às restrições temporais exigidas por um sistema tempo-real.

A utilização do conhecimento descrito pelo usuário em uma estrutura compacta e de rápida inferência é possível através do armazenamento dos modelos que representam a

base de conhecimento do usuário, o que justifica a definição do núcleo reativo gerado como um núcleo que utiliza base de conhecimento para a tomada de decisão.

Esta dissertação tem como objetivo a apresentação do sistema ARKS (*Another Reactive Knowledge-based System*), fundamentado na proposta RETIKS e destinado à geração de núcleos reativos que utilizem bases de conhecimento para a tomada de decisão. De fato, o projeto ARKS é a implementação da proposta RETIKS sem os operadores temporais, e com a adição de algumas características para a descrição da base de conhecimento, além de um gerador de código para construir este núcleo reativo em linguagem C++ ANSI ou Java, permitindo uma aceleração no processo de desenvolvimento de tais sistemas.

As principais etapas cumpridas para a construção do sistema ARKS foram:

1. A especificação completa da linguagem utilizada pelo sistema: projetada para permitir uma fácil descrição do comportamento do núcleo reativo a ser gerado. Esta linguagem descreve o conhecimento na forma de regras de produção, utilizando o conceito de partição desse conhecimento em "contextos" (ou módulos), o que permite uma melhor legibilidade e facilita a manutenção da base de conhecimento;

2. O desenvolvimento de um compilador para o tratamento dessa linguagem, bem como os algoritmos para o tratamento do conhecimento descrito pelas regras seguindo as idéias expostas em [Kae93]: isto é realizado através da obtenção dos modelos do conjunto de fórmulas que representam o conhecimento descrito pelo usuário através do método das conexões [Gri90], e uma estrutura de dados compacta, os *conjuntos de hipercubos*;

3. O desenvolvimento de um gerador de código-fonte em linguagem C++ ANSI e/ou Java que implementa o núcleo reativo correspondente. Este núcleo deve refletir o comportamento descrito pelas regras da base de conhecimento e permitir uma fácil portabilidade deste código entre diversas plataformas computacionais;

4. A aplicação do sistema a casos típicos.

O Capítulo 2 faz uma breve descrição dos sistemas reativos, dos sistemas Tempo-Real e da hipótese de sincronismo, bem como de algumas ferramentas já existentes para a construção de sistemas reativos.

O Capítulo 3 apresenta o formalismo utilizado para o tratamento do conhecimento dentro do sistema ARKS, fundamentado em uma lógica proposicional estendida.

O Capítulo 4 descreve o sistema ARKS: sua linguagem de descrição de regras, suas estruturas de manipulação do conhecimento e geração de código, além de um exemplo de aplicação.

Finalmente, no Capítulo 5 são apresentadas as principais conclusões sobre o trabalho desenvolvido e algumas perspectivas para a continuidade do mesmo.

Capítulo 2

Sistemas Reativos, Sistemas Tempo-Real e a Hipótese de Sincronismo

Apresenta-se neste capítulo uma descrição geral sobre sistemas reativos — área onde são classificadas as aplicações-alvo do sistema ARKS — bem como a hipótese de sincronismo.

Três exemplos de ferramentas cujo objetivo é a geração automática de sistemas reativos também são descritos, permitindo uma visão geral do panorama de aplicação dos núcleos reativos gerados pelo sistema ARKS, bem como uma justificativa da adoção da hipótese de sincronismo como premissa para o projeto de uma base de conhecimento utilizando este sistema.

2.1 Sistemas reativos e sistemas tempo-real

Neste trabalho será seguida a definição de Maffeïs e Poigné [Poi96], que define sistemas reativos como processos que mantém uma interação contínua com o ambiente, alternando entre dois tipos de períodos: espera pelo estímulo e reação ao mesmo.

Esta alternância de estados pode ser visualizada na Figura 2.1, que mostra a carga de processamento de um sistema reativo sobre uma linha do tempo t. Pode-se notar na figura os seguintes períodos:

1. Período de processamento zero, referente ao momento em que o sistema reativo está aguardando os estímulos provindos do ambiente;

2. Período de grande carga computacional, referente ao processamento dos dados recebidos (estímulos) e envio dos resultados (respostas) ao ambiente.

Como indicado na Figura 2.1, este ciclo se repete continuamente.

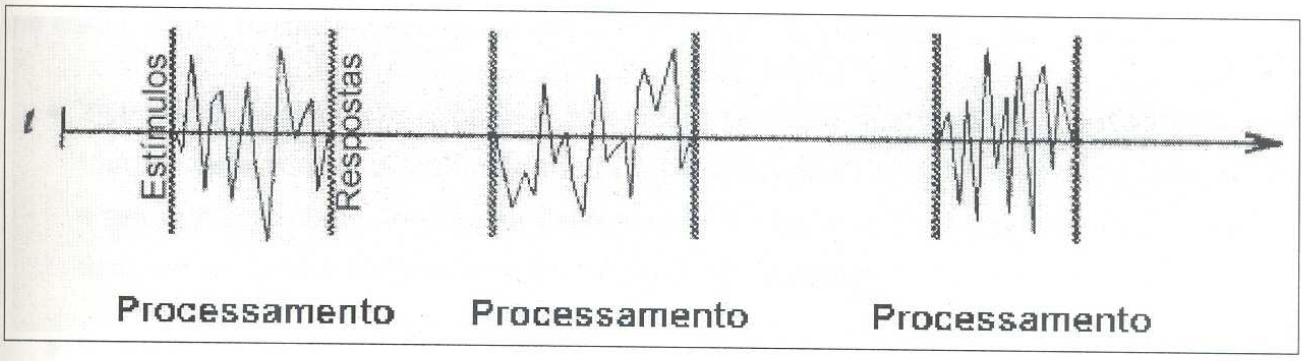

Figura 2.1: Representação gráfica do comportamento de um sistema reativo

Esta característica contínua, no sentido de um sistema reativo nunca cessar as reações aos estímulos vindos do ambiente, é geralmente usada para diferenciar estes sistemas dos sistemas de transformação, caracterizados pela disponibilidade dos dados de entrada no início do processamento e pelo fornecimento dos dados de saída ao final do mesmo (Figura 2.2).

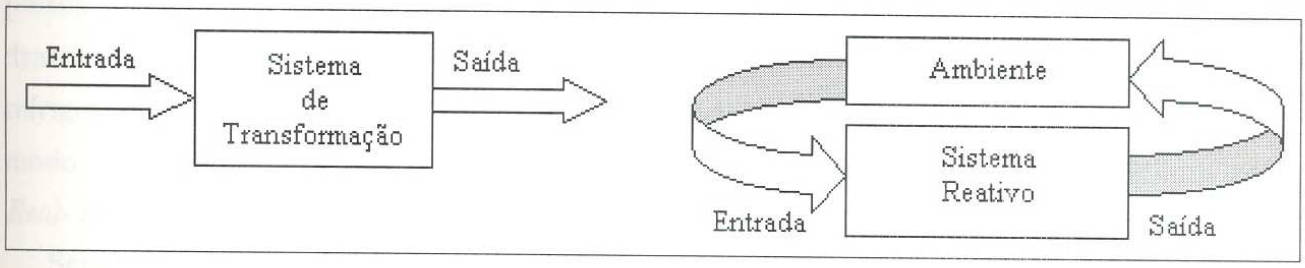

Figura 2.2: Sistemas de transformação e sistemas reativos

Esta definição de sistemas reativos também pode ser aplicada aos sistemas Tempo-Real. De fato, a diferença fundamental entre estes e os sistemas reativos está na importância do fator tempo para cada um destes sistemas.

Para os sistemas reativos o fator tempo não é crucial, ou seja, o ambiente ainda estará receptivo à resposta não importando o instante no qual esta será fornecida pelo sistema reativo.

Como exemplo de sistema reativo pode-se citar um robô que, uma vez colocado sobre um gramado, fica em estado de espera até que seus sensores indiquem a necessidade de corte na grama. A partir deste momento, o robô percorre o gramado cortando a grama, retornando ao estado de espera após a finalização da tarefa. Nota-se que a qualidade do

sistema não é afetada se o robô iniciar o processo de corte algum tempo depois de atingido o ponto de corte.

Um sistema Tempo-Real [Ben88], por sua vez, possui restrições temporais sendo que cada violação de uma restrição temporal ocasiona uma *falha*. A cada falha está associada um custo, o que permite a classificação dos sistemas Tempo-Real em duas categorias:

- Sistemas *soft Real-Time*, onde é *desejada* a resposta dentro de um certo limite de tempo, após o qual o ambiente ainda estará receptivo à esta resposta, porém com o grau de qualidade do sistema decrescendo conforme a distância entre o limite e o instante no qual o sistema envia a resposta ao ambiente;
- Sistemas *hard Real-Time*, onde é *necessária* a resposta dentro de um certo limite de tempo, após o qual o ambiente não estará mais receptivo à mesma, sendo que a qualidade do sistema cairá para zero após ultrapassado este limite.

Seja o caso de uma máquina automática de venda de refrigerantes: uma pessoa ao colocar uma certa quantia dentro da máquina e selecionar o refrigerante escolhido, espera que este seja liberado de imediato; se isto ocorre, então a qualidade do sistema é máxima. Caso a máquina demore um certo intervalo (por exemplo entre 30 e 60 segundos), a pessoa ainda estará esperando pelo refrigerante, porém a qualidade do sistema terá caído drasticamente. Caso a máquina demore mais do que um minuto para a liberação do refrigerante, provavelmente a pessoa terá desistido do refrigerante e ido embora; deste modo, a qualidade do sistema terá sido nula. Este é um exemplo de um sistema *soft Real-Time*.

Seja agora o caso de uma turbina de avião comercial. Tais turbinas necessitam que sinais de controle sejam enviados em intervalos de 20 a 50 milissegundos, sob risco de explosão da mesma. O sistema que envia os dados de controle necessita receber os dados do ambiente, processar tais dados e enviar a resposta à turbina sem nunca ultrapassar esse limite de tempo. Tal sistema pode ser classificado como um sistema *hard Real-Time*.

A relação entre qualidade e tempo nos sistemas reativos e nos sistemas Tempo-Real é ilustrada na Figura 2.3, que pretende mostrar que o tempo de resposta obtido por um sistema reativo não é tão crucial quanto os tempos de respostas desejados para um sistema *soft Real-Time* ou *hard Real-Time*.

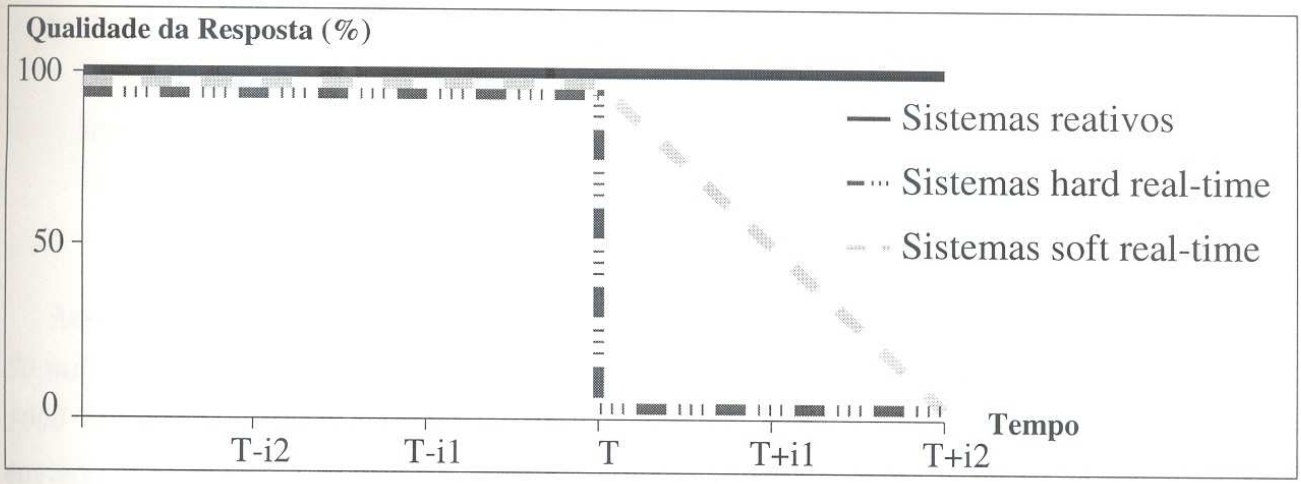

Figura 2.3: Relação entre instante de resposta e qualidade do sistema. T indica o instante esperado para o envio da resposta pelo sistema ao ambiente.

2.2 Hipótese de sincronismo

A hipótese de sincronismo foi estabelecida por Berry [Rig83] para os sistemas reativos. Segundo Gonthier [Gon89], quando do uso das técnicas até então existentes (como Automatos Determinísticos, Redes de Petri, Sistemas Concorrentes, etc.), o desenvolvedor de sistemas reativos era forçado a optar entre determinismo e concorrência, sendo que a opção pela concorrência acarretava no uso de *modelos assíncronos de implementação*, onde processos competem não-deterministicamente pelos recursos. Tal opção traz ao projeto do sistema problemas não inerentes ao sistema reativo em questão, como concorrência entre processos — novas entradas poderão chegar ao sistema antes do final de uma interação — problemas de sincronismo entre processos — dois processos que leiam dados de um mesmo sensor podem ler valores diferentes, uma vez que acessam o sensor em diferentes instantes — entre outros.

Tais problemas não acontecem quando da adoção da hipótese de sincronismo, uma vez que é assumido que cada reação é *instantânea*, e portanto atômica. A hipótese de sincronismo assume que o dispositivo computacional encarregado da execução do sistema não consome tempo algum para a execução dos processos, manipulação dos mesmos, comunicação entre processos e manipulação básica dos dados (adição, por exemplo). Isto significa que:

1. O processo é instantâneo para o *ambiente externo*, ou seja, o instante no qual o ambiente enviou sinais para o sistema é o mesmo no qual o ambiente recebeu a resposta deste;

2. Cada subprocesso é instantâneo em relação a todos os outros subprocessos, sendo que cada subprocesso reage instantaneamente com os outros. Em linguagens síncronas, todos os processos possuem a mesma visão do ambiente e de cada um dos outros, uma vez que os dados lidos de sensores e os dados disponibilizados por cada um dos subprocessos são recebidos por todos os subprocessos no mesmo instante.

Assim a hipótese de sincronismo permite assumir que uma instrução do tipo *aguarde 30 milissegundos* dura exatamente 30 milissegundos, e que uma instrução do tipo *a cada 1000 milissegundos envie um sinal de segundo* significa que um sinal de *segundo* será enviado exatamente a cada 1000 *milissegundos*. Ao passo que em um sistema concorrente, a exatidão do tempo *não pode* ser garantida, uma vez que os processos são dependentes do algoritmo de escalonamento do sistema operacional utilizado.

Berry também afirma que esta abordagem é mais natural do ponto de vista do usuário, através do seguinte exemplo: "O usuário de um relógio não se preocupa com os tempos de reações internas, enquanto perceber que seu relógio reage instantaneamente aos seus comandos" [Gon89] (*tradução livre*), e que o sincronismo também é natural para o programador, uma vez que permite a conciliação entre determinismo e concorrência, a escrita de programas de maneira mais simples (e o raciocínio sobre tais programas), e a separação entre a lógica do sistema e as características dependentes da implementação, como os tempos de reação.

Assim, simplesmente assumindo que o sistema não leva tempo algum no processamento, resolve-se o problema da concorrência entre as tarefas (uma vez que o processamento não leva tempo algum, não será possível a chegada de um sinal do ambiente entre o início e o término de uma iteração) e o problema de sincronismo entre processos (já que todos receberão informações do ambiente ao mesmo tempo), permitindo um raciocínio mais simples sobre sistemas que não têm nestas características seus pontos críticos.

Cabe ressaltar que o enfoque síncrono é fundamentado na abstração do tempo de processamento do sistema, separando as características de projeto das características dependentes de implementação. Obviamente, esta abordagem somente pode ser utilizada para o projeto de sistemas que possam se utilizar desta abstração; sistemas naturalmente assíncronos (e.g. controle de telefonia pública) não são passíveis de projeto utilizando esta abordagem uma vez que os fatores de concorrência e sincronismo entre processos necessitam ser levados em consideração dentro do projeto; da mesma forma os sistemas cujo tempo de processamento seja tão elevado a ponto de não ser possível sua abstração dentro do projeto.

2.3 Descrição de algumas linguagens síncronas

2.3.1 ESTEREL

A linguagem ESTEREL é uma linguagem imperativa voltada ao desenvolvimento de sistemas tempo-real síncronos, e foi desenvolvida observando-se um conjunto mínimo de primitivas gramaticais, estritamente formalizadas, o que garante uma análise precisa de programas escritos nesta linguagem.

Basicamente, o projeto ESTEREL, que resultou na linguagem ESTEREL, baseava-se nas seguintes hipóteses [Rig83]:

1. Uma maneira de garantir a temporização apropriada do sistema é torná-lo síncrono;
2. Não existem referências temporais particularizadas (ou universais). De fato, a noção de tempo é o *fluxo de eventos* que o programa recebe ou gera. Assim, a noção de tempo no projeto ESTEREL é *multiforme*, uma vez que podem existir tantos "relógios" quanto eventos repetivivos, gerados tanto externamente (*hardware*) como internamente (*software*), sendo que não existe distinção entre esses dois tipos de eventos;
3. O que é necessário para a implementação de instruções temporais é uma maneira simples de especificar um *escopo temporal* para um conjunto de ações. De fato, esta é a única instrução temporal da linguagem (excluindo-se os eventos mencionados anteriormente);
4. A linguagem deve possuir uma semântica formal, visto ser esta a única maneira de provar propriedades do programa, realizar análises sobre o mesmo ou raciocinar formalmente sobre a equivalência de programas. Assim, uma teoria adequada é necessária para a descrição da semântica de instruções temporais.

Baseando-se nestas hipóteses, foi criada a linguagem ESTEREL, que tem como características principais:

1. Natureza síncrona, que permite garantir a resposta correta a eventos simultâneos;
2. Noção de tempo baseada no fluxo de eventos do programa;
3. Não existência de distinção entre eventos de *hardware* (interrupções) e *software*;

4. Existência de uma única instrução temporal, determinando um *escopo temporal* e permitindo comunicação inter-processos;

5. Não existência de variáveis compartilhadas;

6. Existência de um conjunto mínimo de primitivas, estritamente formalizadas, que servem de base para construções de mais alto nível.

A compilação de um programa ESTEREL gera um código em linguagem C que implementa um autômato, processo que tem como vantagens:

- o programa paralelo inicial é transformado num código sequencial equivalente: não há processos a gerenciar durante a execução, e problemas de sincronização e comunicação são minimizados;

- o determinismo das aplicações é conservado;

- a hipótese do sincronismo pode ser facilmente verificada: o tempo de transição maximal do autômato é calculável e imperativamente limitado.

Um programa ESTEREL é composto por módulos; cada módulo contém uma parte declarativa, que descreve os dados utilizados e seus tipos, as constantes e funções empregadas, e uma parte procedural, composta pelas instruções propriamente ditas.

O compilador ESTEREL trabalha somente com o nome dos tipos de dados e constantes, sendo que a implementação desses tipos e a atribuição dos valores às constantes deverão ser feitas na linguagem hospedeira (C).

As construções básicas da linguagem são:

- *nothing*: não executa ação e termina instantâneamente, geralmente usada para facilitar a legibilidade do código como, por exemplo, indicar que o bloco *then* de uma expressão condicional não deve tomar ação alguma (Figura 2.4);

- *halt*: não executa ação e não termina; na prática, paralisa o programa, sendo utilizado principalmente na validação deste, quando o acontecimento de um estado anormal recomenda a paralisação do sistema (Figura 2.5);

- $X :=$ exp: uma sentença de atribuição em ESTEREL funciona de maneira análoga às linguagens procedurais, atualizando a memória e terminando instantâneamente;

- *call P* (variáveis) (expressões): Uma chamada de função atualiza a memória e termina instantaneamente; cabe lembrar que funções com grande carga computacional não devem ser implementadas por chamadas de procedimentos, mas através de envios de sinais com parâmetros, a dispositivos externos, que realizarão esse processamento e aguardo da resposta através de sinais enviados por esse dispositivo ao autômato;

- *emit* S(exp): Uma sentença de envio de sinal avalia a expressão *exp*, envia o sinal com o valor avaliado e temina;

- *trap* T *in* stat *end*: Esta construção define um ponto de saída para o bloco de comandos *stat*. Se o bloco de comandos terminar ou sair via a *trap*, então a construção *trap T* também termina; se um bloco de comandos sair via uma *trap* mais externa, então a(s) *trap(s)* mais internas também são terminadas (Figura 2.5);

- $stat_1$; $stat_2$: o operador ; indica sequência entre comandos, sendo o primeiro comando iniciado ao início da sequência; se o primeiro comando terminar via um ponto de saída (*trap*), então a sequência também é terminada e o segundo comando não é executado; se o primeiro comando terminar, o segundo comando inicia instantâneamente;

- *loop* stat *end*: O bloco de comandos (*stat*) é iniciado quando o loop inicia, sendo reiniciado quando o bloco de comandos termina; assim, um loop só termina quando o bloco de comandos é terminado via uma *trap*;

- *if* exp *then* $stat_1$ *else* $stat_2$ *end*: a construção condicional em ESTEREL se comporta de maneira análoga às linguagens procedurais;

- *present* S *then* $stat_1$ *else* $stat_2$ *end*: possui comportamento semelhante à condicional, sendo a expressão de avaliação a presença do sinal S na reação atual;

- *do* stat *watching* S: esta construção limita a execução do bloco de comandos *stat*, sendo este limite a próxima iteração onde o sinal S estiver presente; se o bloco de comandos terminar o processamento ou sair via uma *trap* antes da ocorrência de S, a expressão *watching* também é terminada, caso contrário, assim que houver a ocorrência de S, o bloco de comandos é terminado *sem ser executado* nessa iteração; ou seja, a expressão não executa a ação e não sai via uma *trap*.

- $stat_1$ // $stat_2$: Os dois ramos de uma expressão de paralelismo iniciam quando a expressão inicia; se um dos ramos termina via uma *trap*, então a expressão de

paralelismo termina e os ramos se tornam inativos. Caso contrário, uma expressão de paralelismo termina quando seus dois ramos terminam;

- *exit* T: o comando *exit* força uma saída de um bloco de comandos via a *trap T* (Figura 2.5);

```
⋮
if  expressão then
   nothing
else
    bloco de comandos
end
⋮
```

Figura 2.4: Exemplo de uso do comando *nothing*

```
⋮
trap T1 in
   trap T2 in
      X := 0
    ||
      Y := 0; exit T2
    ||
      Z := 0; exit T1
   end;
   halt
end
⋮
```

Figura 2.5: Exemplo de uso dos comandos *halt*, *trap* e *exit*.

Exemplo Ilustrativo do uso de ESTEREL

A seguir é apresentada a base de regras para a geração de um núcleo reativo que implementa o jogo da vida de Conway (*Conway's Game of Life*) a título de ilustração do uso da linguagem.

O jogo da vida de Conway não é um jogo no sentido convencional da palavra, mas uma simulação de um modelo matemático da evolução de uma colônia de organismos.

Regras da Simulação Esta implementação do jogo da vida de Conway simula a evolução de uma colônia de organismos que ocupam um mundo representado por uma matriz. A intervalos regulares, a colônia evolui instantaneamente de uma geração para outra, de acordo com as seguintes regras:

- Em uma célula vazia com precisamente 3 vizinhos[1], um novo organismo é criado;
- Em uma célula ocupada com 2 ou 3 vizinhos, o organismo sobrevive;
- Em uma célula ocupada com menos de 2 vizinhos, o organismo morre de isolamento;
- Em uma célula ocupada com mais de 3 vizinhos, o organismo morre devido à superpopulação.

Esta implementação em ESTEREL é descrita em [Kae91] e assume uma matriz de 3 linhas por 3 colunas para a representação do mundo. São considerados como vizinhos as células adjacentes acima, abaixo, a esquerda e a direita, não sendo consideradas as células adjacentes nas diagonais.

```
module POINT:

function ADD (integer, integer) : integer/

input P1 (boolean), P2 (boolean), P3 (boolean), P4 (boolean), CLOCK1;
output PX (boolean);

every CLOCK1 do
signal N (combine integer with ADD) in
      if(?P1 = true) then emit N(1) end
   || if(?P2 = true) then emit N(1) end
   || if(?P3 = true) then emit N(1) end
   || if(?P4 = true) then emit N(1) end
   || if((?N > 3) or (? N <= 1)) then
           emit PX(false)
      else
         if (?N = 3) then
```

[1]Um vizinho é qualquer uma das células adjacentes à célula em questão.

```
                emit PX (true)
            end;
        end;
end
end.

module JEUDELAVIE:

input CLOCK;

output SP11 (boolean), SP12 (boolean), SP13 (boolean),
       SP21 (boolean), SP22 (boolean), SP23 (boolean),
       SP31 (boolean), SP32 (boolean), SP33 (boolean);

var    zP11, zP12, zP13,
       zP21, zP22, zP23,
       zP31, zP32, zP33: boolean in

       zP11 := true;  zP12 := false; zP13 := true;
       zP21 := false; zP22 := true;  zP23 := false;
       zP31 := true;  zP32 := false; zP33 := true;

signal P11 (boolean), P12 (boolean), P13 (boolean),
       P21 (boolean), P22 (boolean), P23 (boolean),
       P31 (boolean), P32 (boolean), P33 (boolean),
       DUMMY1 (boolean), DUMMY2 (boolean) in

every CLOCK do
       emit DUMMY1 (false)
    || emit DUMMY2 (false)
    || copymodule POINT [signal P12/P1, P22/P2, P13/P3, DUMMY2/P4, SP12/Px,
                         CLOCK/CLOCK1]
    || copymodule POINT [signal P11/P1, P22/P2, P13/P3, DUMMY2/P4, SP12/PX,
                         CLOCK/CLOCK1]
    || copymodule POINT [signal P12/P1, P23/P2, DUMMY1/P3, DUMMY2/P4,
```

```
                    SP13/PX, CLOCK/CLOCK1]
  || copymodule POINT [signal P11/P1, P22/P2, P31/P3, DUMMY2/P4, SP21/PX,
                    CLOCK/CLOCK1]
  || copymodule POINT [signal P12/P1, P21/P2, P23/P3, P32/P4, SP22/PX,
                    CLOCK/CLOCK1]
  || copymodule POINT [signal P13/P1, P22/P2, P33/P3, DUMMY2/P4, SP23/PX,
                    CLOCK/CLOCK1]
  || copymodule POINT [signal P21/P1, P32/P2, DUMMY1/P4, SP31/PX,
                    CLOCK/CLOCK1]
  || copymodule POINT [signal P31/P1, P22/P2, P33/P3, DUMMY2/P4, SP32/PX,
                    CLOCK/CLOCK1
  || copymodule POINT [signal P23/P1, P32/P2, DUMMY1/P3, DUMMY2/P4,
                    SP33/PX, CLOCK/CLOCK1]
  || emit P11 (zP11);
     emit P12 (zP12);
     emit P13 (zP13);
     emit P21 (zP21);
     emit P22 (zP22);
     emit P23 (zP23);
     emit P31 (zP31);
     emit P32 (zP32);
     emit P33 (zP33);
     zP11 := ?SP11;
     zP12 := ?SP12;
     zP13 := ?SP13;
     zP21 := ?SP21;
     zP22 := ?SP22;
     zP23 := ?SP23;
     zP31 := ?SP31;
     zP32 := ?SP32;
     zP33 := ?SP33;
end;
end;
end.
```

Este código necessita de um bom conhecimento da linguagem ESTEREL para a sua compreensão, o que foge do escopo deste trabalho. Assim, ele é descrito a título ilustrativo, servindo para uma comparação entre as estruturas desta linguagem e das linguagens SIGNAL e ARKS, descritas adiante.

2.3.2 SIGNAL

SIGNAL [LeG91, LeG90] é uma linguagem *data-flow* que define um autômato a partir de um conjunto de equações, cujas variáveis são identificadores para elementos primitivos da linguagem chamados de sinais. Estes sinais são constituídos de sequências de valores ordenados temporalmente. Tais equações devem ser verificadas e mantidas consistentes durante toda a execução, imprimindo a esta linguagem os princípios empregados na Teoria de Controle de Sistemas Contínuos.

Um sinal na linguagem é definido como uma sequência de valores, sobre o qual está associado um relógio. Os valores de um sinal pertencem a um mesmo domínio pré-definido (booleanos, inteiros, reais, complexos), ou a um domínio definido dentro do programa (tabelas).

Um relógio H de um sinal S é determinado da seguinte forma: $H(S) = 1$ (TRUE) se e somente se S está presente na reação corrente e $H(S) = 0$ (FALSE) caso contrário. Assim, a noção de sincronismo entre sinais é imediata: dois sinais são síncronos se tem o mesmo relógio, i.e. $H(S_1) = H(S_2)$.

Uma comunicação associa, dentro de um instante lógico do programa, um valor a uma variável. Um evento é um conjunto de comunicações simultâneas formando uma transição no autômato. Dentro de um evento é possível que uma variável não receba um valor; neste caso se diz que o sinal é ausente, sendo neste instante o valor da variável igual a $\perp$. Um evento possui pelo menos uma comunicação.

Os principais operadores temporais são:

- fusão determinista (S_1 *default* S_2): o valor de um sinal produzido pela fusão determinista entre dois sinais S_1 e S_2 em um determinado instante é dado por:
  - S_1 se o sinal S_1 não estiver ausente neste instante;
  - S_2 se o sinal S_1 estiver ausente neste instante e o sinal S_2 não estiver ausente neste instante;
  - $\perp$ caso contrário.

  A semântica deste operador pode ser visualizada na Tabela 2.1

- amostragem (S_1 *when* B): o valor de um sinal produzido pela amostra de um sinal S_1 em relação a um sinal booleano B em um determinado instante é dado por:
  - S_1 caso o sinal S_1 não esteja ausente neste instante e o sinal B possuir valor **TRUE**;
  - $\perp$ caso contrário.

  A semântica deste operador pode ser visualizada na Tabela 2.2

- memorização (S_2 *cell* B): o valor de um sinal produzido pela memorização de um sinal S_1 em relação a um sinal booleano B em um determinado instante é dado por:
  - S_1 caso o sinal S_1 não esteja ausente neste instante;
  - $\perp$ caso o sinal S_1 esteja ausente neste instante e o valor do sinal B for diferente de **TRUE**;
  - valor do último sinal de S_1 caso o sinal S_1 esteja ausente neste instante e o valor do sinal B for igual a **TRUE**.

  A semântica deste operador pode ser visualizada na Tabela 2.3

- atraso (S_1 \$ x): o valor de um sinal produzido por um atraso x de um sinal S_1 em um instante t é determinado pelo valor do sinal S_1 no instante $t - x$. A semântica deste operador pode ser visualizada na Tabela 2.4.

S_1	1	$\perp$	3	$\perp$
S_2	3	4	$\perp$	$\perp$
S_1 *default* S_2	1	4	3	$\perp$

Tabela 2.1: Semântica do operador *default*.

S_1	1	$\perp$	3	$\perp$	5	$\perp$
B	T	T	F	F	$\perp$	$\perp$
S_1 *when* B	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Tabela 2.2: Semântica do operador *when*.

S_1	1	$\perp$	$\perp$	4	$\perp$	$\perp$
B	T	F	T	F	T	T
$S_1 cell\ B$	1	$\perp$	1	4	4	4

Tabela 2.3: Semântica do operador *cell*.

S_1	1	2	3	4	5	6
$S_1\$1$	$\perp$	1	2	3	4	5
$S_1\$2$	$\perp$	$\perp$	1	2	3	4

Tabela 2.4: Semântica do operador *$*.

Uma vez que cada sinal possui seu próprio relógio, o compilador SIGNAL os ordena segundo uma hierarquia de frequências, formando um reticulado, sendo que o valor maximal deste reticulado será o passo do autômato gerado. O compilador também analisa a sincronização entre os vários relógios, verificando a existência de ciclos de casualidade, reportando ao usuário os possíveis problemas existentes no código. O código gerado é totalmente seqüencial, sendo que as transições entre os estados do autômato é feita por uma série de condicionais sobre os valores dos estados.

Exemplo Ilustrativo do uso de SIGNAL

A seguir é apresentado um esboço de uma implementação simplificada do jogo da vida de Conway (*Conway's Game of Life*), conforme apresentado na seção anterior. Assim como no exemplo da linguagem ESTEREL, são consideradas somente as células adjacentes acima, abaixo, a esquerda e a direita, não sendo consideradas as células adjacentes nas diagonais. O mundo é representado por uma matriz formada por 3 linhas de 3 colunas [Kae91].

```
process jeudelavie=
(integer m,n)
{ ? ! [m n] logical tp }
(| array i to m of
        array j to n of ztp [i j] := tp [i j] $1
 | array i to m of
        array j to n of tp [i j] := one_point()
```

```
                                    {ztp [i-1 j], ztp[i+1, j]
                                     ztp [i j-1], ztp[i, j+1]}
 |) / ztp
where
 [m n] logical ztp init [{to m} [{to n}: 0]]

process one_point= ()
{ ? logical p1, p2, p3, p4
  ! logical p }
(| p := (false when (p1 and p2 and p3 and p4))
         default
        (false when (p1 and (not p2 and (not p3) and (not p4)))
         default
        (false when ((not p1) and p2 and (not p3) and (not p4)))
         default
        (false when ((not p1) and (not p2) and p3 and (not p4)))
         default
        (false when ((not p1) and (not p2) and (not p3) and p4)))
         default
         true
 | aux := (event p1) default (event p2) default
          (event p3) default (event p4)
 | synchro {p, aux}
 |) /aux
where
    event aux
end
end
```

2.3.3 RETIKS

A proposta RETIKS (*REal-TIme Knowledge-based System*) [Kae93] trata de um sistema para a geração automática de núcleos Tempo-Real que utilizem heurísticas para a tomada de decisão. Suas principais características são:

1. A geração de um núcleo Tempo-Real com o comportamento descrito pelo usuário

através de uma base de conhecimento;

2. A utilização de uma linguagem para a descrição do conhecimento baseada em regras de produção;

3. O uso do enfoque síncrono para o tratamento do problema Tempo-Real;

4. A fundamentação do sistema em um formalismo bem definido (lógica proposicional estendida e cálculo dos modelos);

5. O tratamento de problemas de inconsistência e incompletude da base de conhecimento;

6. A correspondência entre a proposta e uma classe restrita de Sistemas Dinâmicos dirigidos por eventos;

7. O emprego da modularidade e da abstração como forma de reduzir o problema de explosão combinatória gerado pelo cálculo dos modelos;

Em [Kae93] são definidos os algoritmos utilizados na proposta, os testes destes algoritmos sobre um protótipo em Lisp, e a proposição inicial da linguagem do usuário.

A versão protótipo gerado não é aplicável à muitas situações práticas devido ao ambiente de execução, o que dificulta a avaliação da proposta.

A discussão dos itens acima será feita ao decorrer do texto, principalmente no capítulo destinado à descrição do sistema ARKS, uma vez que este sistema implementa grande parte destas características.

Exemplo Ilustrativo do uso de RETIKS

A seguir é apresentado um esboço de uma implementação do jogo da vida de Conway (*Conway's Game of Life*), conforme apresentado nas seções anteriores. Neste exemplo, são consideradas como vizinho as oito células adjacentes a uma célula.

```
(MODULE Calc

  INPUTS N, S, L, O, NE, NO, SE, SO;
  OUTPUTS Vizinhos;

  (QUALITATIVE_EQUATION Vizinhos = N+S+L+O+NE+NO+SE+SO)
)
```

```
(MODULE Sobrevivencia

   INPUTS Estado, Vizinhos;
   OUTPUTS Vive;

   (RULE R1 (IF (Estado = 0.0 AND Vizinhos = 3.0) THEN Vive = 1.0))
   (RULE R2 (IF (Estado = 0.0 AND Vizinhos <> 3.0) THEN Vive = 0.0))
   (RULE R3 (IF (Estado = 1.0 AND (Vizinhos = 2.0 OR Vizinhos = 3.0)) THEN
          Vive = 1.0))
   (RULE R4 (IF (Estado = 1.0 AND (Vizinhos < 2.0 OR Vizinhos > 3.0)) THEN
          Vive = 0.0))
)
```

2.4 Conclusão

Apresentou-se neste capítulo uma descrição suscinta dos sistemas reativos, nos quais podem ser classificadas as aplicações-alvo do sistema ARKS (e da hipótese do sincronismo).

O enfoque síncrono, utilizado como metodologia de projeto dos núcleos reativos gerados pelo sistema ARKS, se mostrou uma possibilidade bastante interessante, devido a capacidade de abstração das características de implementação — por exemplo, concorrência entre processos e sincronismo dos dados — e ênfase sobre os problemas realmente inerentes ao projeto da aplicação propriamente dita.

As linguagens aqui apresentadas são exemplos claros de que esta abordagem é viável. De fato, estas linguagens são utilizadas em diversas aplicações [Kae93], [Far96], [Rut96], [Ber89].

Outro objetivo da descrição destas linguagens é o indicativo de como cada paradigma descreve o comportamento de um sistema. Isto possibilita que o leitor contraste estas descrições com a da proposta ARKS, que será vista adiante.

Capítulo 3

O Formalismo de Base do Sistema ARKS

Apresenta-se neste capítulo o formalismo matemático em que o sistema ARKS se baseia, com o objetivo de demonstrar formalmente o processo de manipulação da base de conhecimento.

O sistema ARKS tem sua fundamentação na proposta RETIKS (*Real-Time Knowledge-Based System*), apresentada em [Kae93]. De fato, o sistema ARKS é a implementação da proposta RETIKS, sem a inclusão dos operadores temporais. O formalismo desenvolvido para o cálculo proposicional estendido, os algoritmos e as estruturas de dados para o tratamento do conhecimento utilizados no sistema ARKS também são similares àqueles apresentados na proposta RETIKS, com algumas extensões, como por exemplo a avaliação de expressões matemáticas.

3.1 Formalização do cálculo proposicional adotado na proposta

O cálculo proposicional apresentado a seguir é uma formalização de uma lógica proposicional estendida [Kae93], que foi definida sob a forma equivalente de "dedução natural" utilizada na construção do sistema KHEOPS [Phi89]. O cálculo será denotado $P\mathcal{D}_A$.

Seja $A = (a_1, a_2, \ldots, a_n)$ um conjunto finito e ordenado de símbolos proposicionais, cujos elementos serão chamados de *atributos*. A cada $a_i \in A$ está associado um domínio finito e não vazio de valores $\mathcal{D}_{a_i}$.

Definição 3.1 (Aridade) *A* aridade *de um atributo* $a_i \in A$ *é denotada por* $\|\mathcal{D}_{a_i}\|$ *(car-*

dinalidade do conjunto $\mathcal{D}_{a_i}$). Os elementos de $\mathcal{D}_{a_i}$ podem então ser enumerados e, desta forma, são associados a números naturais $0, 1, \ldots n \in \mathbf{N}$.

3.1.1 Sintaxe

- O cálculo possui quatro tipos de símbolos primitivos, quais sejam:
  1. atributos $a_i \in A$;
  2. números naturais $0, 1, \ldots$ que designam os valores possíveis de cada domínio;
  3. conectivos lógicos: $\neg$ (negação), $\rightarrow$ (implicação), $\wedge$ (e), $\vee$ (ou), 1 (verdadeiro), 0 (falso);
  4. símbolos auxiliares: parênteses e chaves.
- E seu conjunto de fórmulas $\mathcal{F}$ é dado por:
  1. se $a_i \in A$ é um atributo e $m \in \mathcal{D}_{a_i}$, então $a\{m\}$ é uma fórmula atômica; 1 e 0 também são fórmulas atômicas;
  2. se F_1 e F_2 são fórmulas, então $(\neg F_1)$, $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$ são fórmulas;
- Serão utilizadas as seguintes simplificações de notação:
  - $(F_1 \rightarrow F_2) \equiv (\neg F_1 \vee F_2)$;
  - $(F_1 \leftrightarrow F_2) \equiv (F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$.
- Também serão utilizadas as convenções usuais de precedência de operadores e de eliminação de parênteses.

3.1.2 Semântica

Definição 3.2 (Interpretação) *Uma* interpretação I *é a associação de um elemento de $\mathcal{D}_{a_i}$ para cada atributo a_i. Este elemento é denominado* valor do atributo a_i sob I.

Definição 3.3 (Valoração) *A toda interpretação I está associada uma* valoração $v_I : \mathcal{F} \longrightarrow \{0, 1\}$, definida da seguinte forma:

1. $v_I(1) = 1$ e $v_I(0) = 0$;
2. se a_i é um atributo, então:

- $v_I(a_i\{m\}) = 1$ sse $I(a_i) = m$;
- $v_I(a_i\{m\}) = 0$ sse $I(a_i) = n$, com $m \neq n$;

3. se F_1 e F_2 são fórmulas, então:

- $v_I(\neg F_1) = 1$ sse $v_I(F_1) = 0$;
- $v_I(F_1 \wedge F_2) = 1$ sse $v_I(F_1) = 1$ e $v_I(F_2) = 1$ simultaneamente;
- $v_I(F_1 \vee F_2) = 1$ sse $v_I(F_1) = 1$ ou $v_I(F_2) = 1$ ou ambos;

Definição 3.4 (Teoria) *Um conjunto qualquer de fórmulas* Γ *é chamado uma* teoria *sobre o conjunto de atributos* A.

Definição 3.5 (Satisfação) *Uma interpretação* I satisfaz *uma fórmula* F, *denotando-se por* $I \models F$ *sse* $v_I(F) = 1$; *se* $v_I(F) = 0$ *diz-se que* I falsifica F.

Definição 3.6 (Modelo) *Uma interpretação* I *é um* modelo *para uma teoria* Γ *sse* $I \models F$, *para toda fórmula* $F \in \Gamma$; *o conjunto de modelos de uma teoria* Γ *será denotado* $M(\Gamma)$.

Definição 3.7 (Consequência Semântica) *Se* Γ *é uma teoria e* F *é uma fórmula, diz-se que* F *é uma* consequência semântica *de* Γ, *e escrever-se-á* $\Gamma \models F$, *sse para toda interpretação* I *na qual* $v_I(G) = 1$ *para toda* $G \in \Gamma$, *tem-se também* $v_I(F) = 1$.

Definição 3.8 (Tautologia) *Uma fórmula* F *é* válida, *i.e. uma tautologia, sse* $v_I(F) = 1$ *para qualquer* I; *neste caso escrever-se-á* $\models F$.

3.1.3 Axiomatização

A axiomatização de $P\mathcal{D}_A$ inclui:

- Os axiomas do cálculo proposicional clássico:

1. $F_1 \rightarrow (F_2 \rightarrow F_1)$;
2. $((F_1 \rightarrow (F_2 \rightarrow F_3)) \rightarrow ((F_1 \rightarrow F_2) \rightarrow (F_1 \rightarrow F_3)))$;
3. $((\neg F_1 \rightarrow \neg F_2) \rightarrow ((\neg F_1 \rightarrow F_2) \rightarrow F_1))$;

para fórmulas F_1, F_2, F_3; e o axioma que assegura a associação de um só valor do domínio para cada atributo em cada estado:

4. $(a\{0\} \wedge \neg a\{1 \ldots n\}) \vee \ldots \vee (a\{n\} \wedge \neg a\{0 \ldots n-1\})$, com $n+1 = \|D_a\|$ e para todo $a \in A$.

- Regra de inferência (*Modus Ponens*):

$$\frac{F_1 \rightarrow F_2,\ F_1}{F_2}$$

Definição 3.9 (Dedução) *Se Γ é um conjunto de fórmulas, uma* dedução a partir de Γ *é uma sequência finita $(F_1, F_2 \ldots F_n)$ tal que para todo i com $1 \leq i \leq n$ tem-se:*

1. *F_i é um axioma da lógica;*
2. *F_i é um elemento de Γ;*
3. *F_i pode ser obtida a partir de F_j e F_k, com $j < i, k < i$, pela regra de inferência. Neste caso escreve-se $\Gamma \vdash F_n$.*

Definição 3.10 (Prova) *Uma* prova *é uma dedução a partir de $\Gamma = \emptyset$ (dedução utilizando somente os axiomas e a regra de inferência do cálculo); neste caso F_n é dito um* teorema, *e escreve-se $\vdash F_n$.*

Para este cálculo proposicional, os seguintes resultados podem ser obtidos:

Proposição 3.1 *São válidas as seguintes fórmulas, para todo $a_i \in A$:*

1. $a_i \mathcal{D}_{a_i}$;
2. $\neg a_i \mathbf{D} \leftrightarrow a_i(\mathcal{D}_{a_i} - \mathbf{D})$;
3. $a_i \mathbf{D} \vee a_i \mathbf{E} \leftrightarrow a_i(\mathbf{D} \cup \mathbf{E})$.

Proposição 3.2 *Para a inferência, podem ser utilizados:*

1. *Modus Ponens* generalizado:

$$\frac{a\mathbf{D},\ a\mathbf{D}' \rightarrow b\mathbf{E},\ \mathbf{D} \subseteq \mathbf{D}'}{b\mathbf{E}}$$

2. restrição de domínio:

$$\frac{a\mathbf{D},\ a\mathbf{E}}{a(\mathbf{D} \cap \mathbf{E})}$$

3.2 Método das conexões

O método das conexões proposto por L. Wallen [Wal89], é uma extensão do método das conexões de Bibel [Bib82], que permite um tratamento computacional da lógica clássica e das lógicas não-clássicas [Tur96, Cat90].

3.2.1 Geração da árvore sintática

No método de Wallen [Gri90], o primeiro passo é a geração da *árvore sintática* de uma fórmula ou de um conjunto de fórmulas, sendo que no último caso, considera-se que as fórmulas do conjunto estão implicitamente conectadas por uma conjunção ($\wedge$)[1]. Esta árvore (*parsing tree*) enumera os *componentes sintáticos* da fórmula, criando, em certo sentido, uma ordem de precedência entre os nós, sendo os nodos de maior profundidade aqueles com maior precedência na fórmula. A Figura 3.1 mostra a árvore sintática da fórmula $P \wedge Q \rightarrow R \wedge S$.

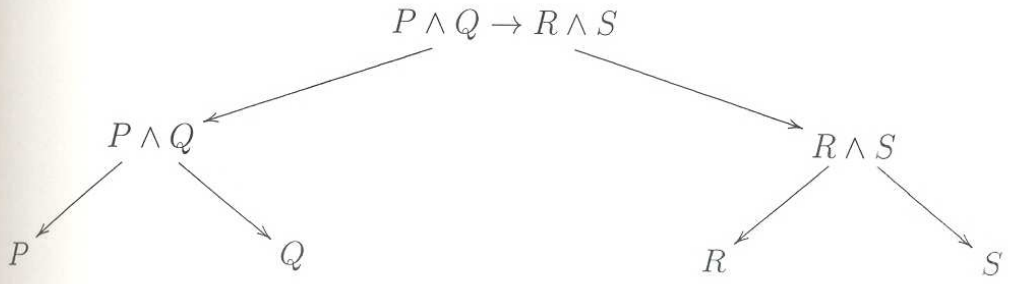

Figura 3.1: Árvore sintática

3.2.2 Definição da polaridade de uma fórmula

O próximo passo, após a criação da árvore sintática, é a definição das polaridades das sub-fórmulas, sendo que para tanto é necessária a definição de uma *ocorrência positiva* e uma *ocorrência negativa* de uma fórmula.

Uma sub-fórmula B de uma fórmula A é dita "aparecendo como uma ocorrência positiva dentro de A" (*polaridade* 1) se ela aparece no escopo de um número par de negações explícitas ou implícitas. Caso contrário, esta sub-fórmula B é dita aparecendo como uma ocorrência negativa (*polaridade* 0) [Wal89].

[1] Ver [Gri90], [Bib82], [Wal89]

É chamado de negação explícita o símbolo da negação $\neg$. Uma negação implícita é a premissa de uma implicação (de fato, p $\rightarrow$ q equivale a $\neg$ p $\vee$ q).

Intuitivamente, a polaridade associada a uma sub-fórmula indica se a mesma deve ou não ser satisfeita, seguindo a definição indicada na semântica da lógica.

A partir do conectivo que domina cada sub-fórmula e de sua polaridade é possível a aplicação das regras de eliminição de conectivos, como no caso do cálculo por seqüentes.

3.2.3 Regras de prolongamento e ramificação

Chamam-se regras do tipo α, as regras de *prolongamento* (para a satisfação da fórmula todos seus ramos devem ser satisfeitos), e seus descendentes serão do tipo α_1 e α_2.

Chamam-se regras do tipo β as regras de *ramificação* (para a satisfação da fórmula ao menos um dos ramos deve ser satisfeito), e seus descendentes serão do tipo β_1 e β_2.

As regras são apresentadas na Tabela 3.1.

Intuitivamente, o tipo α está associado ao conectivo "e" ($\wedge$) e o tipo β está associado ao conectivo "ou" ($\vee$).

α	α_1	α_2
$F_1 \wedge F_2, 1$	$F_1, 1$	$F_2, 1$
$F_1 \vee F_2, 0$	$F_1, 0$	$F_2, 0$
$F_1 \rightarrow F_2, 0$	$F_1, 1$	$F_2, 0$
$\neg F_1, 1$	$F_1, 0$	
$\neg F_1, 0$	$F_1, 1$	

β	β_1	β_2
$F_1 \wedge F_2, 0$	$F_1, 0$	$F_2, 0$
$F_1 \vee F_2, 1$	$F_1, 1$	$F_2, 1$
$F_1 \rightarrow F_2, 1$	$F_1, 0$	$F_2, 1$

Tabela 3.1: Regras de eliminação de conectivos

3.2.4 Tipos de fórmulas

Existe distinção entre as fórmulas que originam uma *ramificação* e as fórmulas que originam um *prolongamento*. Assim, cada fórmula (com exceção das fórmulas atômicas e da fórmula inicial) possui dois tipos: um tipo primário e um tipo secundário.

Uma fórmula F possui um tipo *primário* α se ela dá origem a um *prolongamento*, ou um tipo *primário* β se ela dá origem a uma *ramificação*. Uma fórmula possui um tipo *secundário* α_1 ou α_2 se o pai desta fórmula possuir um tipo primário α, ou um tipo *secundário* β_1 ou β_2 se o pai desta fórmula possuir um tipo primário β.

3.2.5 Noção de caminho

Wallen [Gri90] introduz em seguida a noção de caminho através da seguinte definição recursiva:

Base: se k_0 é a raiz, k_0 é um caminho.

Recursão: Se S é um caminho que contém o nó α^k, $(S \setminus \{\alpha^k\}) \cup \{\alpha_1^k \alpha_2^k\}$ é um caminho. Assim, dado um nó α^k do tipo α (prolongamento), a continuação do caminho a partir de α^k é a conjunção de seus descendentes, que são respectivamente do tipo secundário α_1 e α_2.

Se S é um caminho que contém o nó β^k, $(S \setminus \{\beta^k\}) \cup \{\beta_1^k\}$ e $(S \setminus \{\beta^k\}) \cup \{\beta_2^k\}$ são caminhos. Em outras palavras, dado um nó β^k do tipo β (ramificação), a continuação do caminho a partir de β^k cria uma bifurcação, uma união de cada um dos ramos formados pelos seus descendentes, que serão do tipo secundário β_1 e β_2.

3.2.6 Conexão

Uma *conexão* num caminho S é um sub-caminho S' em que duas posições indicam a mesma fórmula atômica (a mesma variável proposicional e domínio) mas com polaridades distintas. Um conjunto de conexões *recobre* uma fórmula F sse todo caminho atômico obtido a partir de F contém uma conexão.

3.2.7 Utilização do método para a demonstração de teoremas

Proposição 3.3 (Andrews,Bibel) *[Wal89] Uma fórmula proposicional F é válida sse existe um conjunto de conexões que recobre F_0.*

Esta proposição é uma conseqüência do seguinte resultado sobre tabelas semânticas (Smullyan):Uma fórmula proposicional é válida sse existe uma tabela semântica fechada para F_0.

3.2.8 Utilização do método para a obtenção dos modelos de uma teoria

Proposição 3.4 *Para uma teoria Γ, os modelos $M(\Gamma)$ são obtidos pelas conexões atômicas de $\Gamma, 1$ (axiomas da teoria ligados por conjunção) [Gri90].*

3.2.9 Análise do método

Análises do método das conexões podem ser encontradas em [Gri90] e [EI89]. Salientam-se aqui as principais conclusões:

- não há redundância pois nenhuma fórmula é repetida: de fato, somente os elementos a_k são repetidos (a_k não é uma fórmula mas uma *posição*); permitindo que se trabalhe com estruturas de dados eficientes;
- nenhuma pesquisa não-necessária é executada: as conexões formam o mínimo de interpretações exigidas para que se proceda a verificação: o método se preocupa somente com as conexões, ignorando os vários caminhos que resultam nos átomos;
- a complexidade é reduzida: não há necessidade de várias árvores de prova, o cálculo é realizado somente sobre a árvore sintática;
- o indeterminismo inerente ao cálculo dos seqüentes [Gri90], devido às suas várias árvores de prova possíveis, ou ao método das tabelas, é eliminado.

3.2.10 A obtenção dos modelos através do método

Nesta seção descreve-se o procedimento para a obtenção dos modelos de uma teoria Γ, utilizando o método das conexões e as estruturas de dados **hcubo** e **Hcuboset**.

A estrutura de dados **hcubo** é baseada na representação concisa de conjuntos de dados [Gia88], e na extensão das operações usuais sobre conjuntos para n-uplas destes conjuntos.

Definição 3.11 (hcubo) *Seja A um conjunto de atributos. Denomina-se hipercubo (***hcubo***) um vetor de dimensão $||A||$, onde cada coordenada (seção) i representa um subconjunto de $\mathcal{D}_{a_i}$. A representação de um* **hcubo** *será feita utilizando a notação*[2] *$a_1 D_{a_1} a_2 D_{a_2} \ldots a_n D_{a_n}$, onde a_i é um elemento (atributo) de A, e D_{a_i} é um subconjunto de $\mathcal{D}_{a_i}$.*

Definição 3.12 (conjunto de modelos associados a um hcubo) *O conjunto de modelos associados a um* **hcubo** *é formado pelo produto cartesiano entre cada seção do* **hcubo** *$D_{a_1} \times D_{a_2} \times \ldots \times D_{a_n}$.*

[2] *Elementos genéricos de A serão denotados por $a, b, \ldots$ ao invés de a_i.*

Exemplo 3.2.1 *Sejam* $A = (a, b, c)$, $\mathcal{D}_a = \mathcal{D}_b = \{0, 1, 2\}, \mathcal{D}_c = \{0, 1\}$. *O* **hcubo** $h = [\{0, 1, 2\}\{1, 2\}\{0\}]$ *representa o conjunto de modelos apresentados na Tabela 3.2.*

a	b	c
0	1	0
0	2	0
1	1	0
1	2	0
2	1	0
2	2	0

Tabela 3.2: Lista de modelos representados pelo **hcubo** h

Definição 3.13 (Hcuboset) *Define-se um* **Hcuboset** *como um conjunto de elementos* **hcubo**.

Os passos a serem realizados para a obtenção do conjunto de modelos são:

1. construção da árvore sintática da teoria Γ; a raiz da árvore é formada pela conjunção de todos os axiomas de Γ;

2. propagação da raiz para as folhas (*top-down*) da polaridade e dos tipos - primário e secundário - de todas as sub-fórmulas que compõem a teoria; utilizando para isto o algoritmo descrito na seção 3.2.5;

3. propagação das folhas à raiz (*bottom-up*) dos conjuntos de átomos ("pontos") que satisfazem cada sub-fórmula, utilizando as operações básicas definidas sobre os **hcubo** e **Hcuboset** descritas no apêndice B. Cabe lembrar que associado ao tipo primário α de uma conexão está a operação de **interseção** entre **Hcuboset**, assim como a operação de **união** entre **Hcuboset** está associada ao tipo primário β de uma conexão;

4. o **Hcuboset** associado à raiz da teoria é a representação compacta de $M(\Gamma)$.

Exemplo 3.2.2 *Sejam as seguintes fórmulas, onde* $(A = (a, b, c), D_a = D_b = \{0, 1, 2\}, D_c = \{0, 1\})$

Γ a{1,2} $\wedge$ c{0} $\rightarrow$ b{0,1}, $\neg$(a{0} $\wedge$ b{1}) $\vee$ c{0})

A Figura 3.2 apresenta a árvore sintática deste conjunto de fórmulas, bem como a propagação dos tipos primário e secundário.

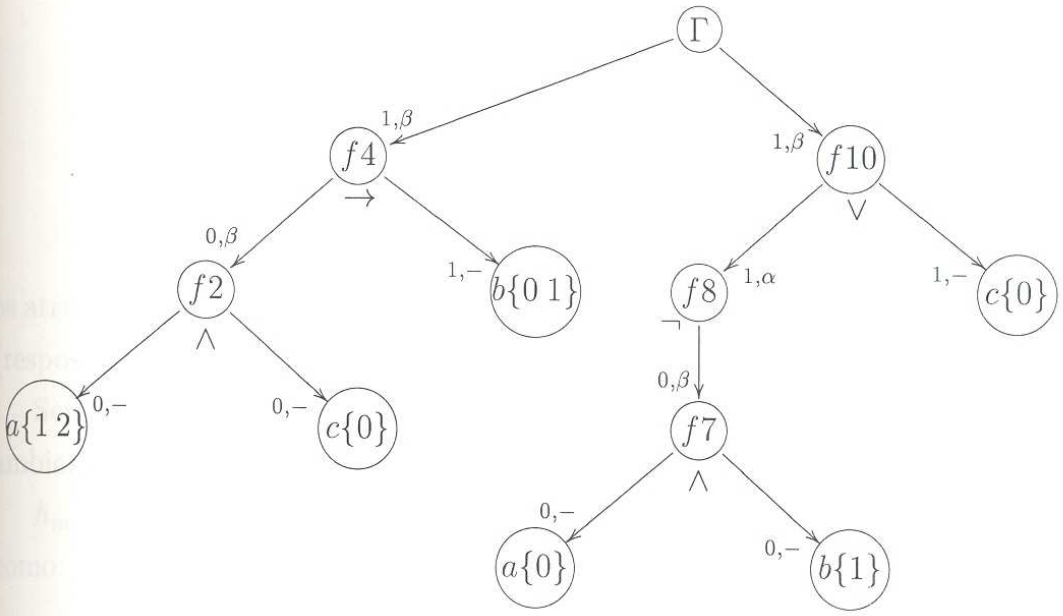

Figura 3.2: Árvore sintática, polaridades e tipos associados a Γ

Uma vez gerada a árvore sintática, os **Hcuboset** que representam os modelos deste conjunto são calculados. Para as fórmulas atômicas este cálculo é direto (representado como os rótulos dos nós-folha), de acordo com a semântica da lógica correspondente. Para os nós-folha, o cálculo é feito com base no seu tipo primário, conforme citado anteriormente. O conjunto de modelos (**Hcuboset**) gerados para este exemplo podem ser visualizados na Tabela 3.3.

3.2.11 Processo de inferência

Seja H_c um **Hcuboset** que representa os modelos de uma teoria Γ que possui um conjunto de atributos A.

Define-se como A_{in} um subconjunto de A que representam os atributos cujos valores serão fornecidos pelo ambiente (estímulos) e A_{out} um subconjunto de A que representam

fórm.	con.	pol.	t1	t2	**Hcuboset**
a{1 2}		0		β_1	{ [{0} {0 1 2} {0 1}] }
c{0}		0		β_2	{ [{0 1 2} {0 1 2} {1}] }
f2	$\wedge$	0	β	β_1	{ [{0} {0 1 2} {0 1}] [{1 2} {0 1 2} {1}] }
b{0 1}		1		β_2	{ [{0 1 2} {0 1} {0 1}] }
f4	$\rightarrow$	1	β	α_1	{ [{0} {0 1 2} {0 1}] [{1 2} {0 1 2} {1}] [{1 2} {0 1} {0}] }
a{0}		0		β_1	{ [{1 2} {0 1 2} {0 1}] }
b{1}		0		β_2	{ [{0 1 2} {0 2} {0 1}] }
f7	$\wedge$	0	β	α_1	{ [{1 2} {0 1 2} {0 1}] [{0} {0 2} {0 1}] }
f8	$\neg$	1	α	β_1	{ [{1 2} {0 1 2} {0 1}] [{0} {0 2} {0 1}] }
c{0}		1		β_2	{ [{0 1 2} {0 1 2} {0}] }
f10	$\vee$	1	β	α_2	{ [{1 2} {0 1 2} {0 1}] [{0} {0 2} {0 1}] [{0} {1} {0}] }
Γ	$\wedge$	1	α		{ [{1 2} {0 1 2} {0}] [{0} {0 2} {0 1}] [{0} {1} {0}] }

Tabela 3.3: Conjuntos de hipercubos associados às fórmulas de Γ

os atributos que devem ter seus valores enviados ao ambiente após o processo de inferência (respostas), de forma que $A_{in} \cup A_{out} \subseteq A$ e $A_{in} \cap A_{out} = \emptyset$.

Seja $\mathcal{E}_{a_i}$ com $a_i \in A_{in}$ um subconjunto de $\mathcal{D}_{a_i}$ formado pelos valores enviados pelo ambiente.

h_{in} é um **hcubo** representado na forma $a_1 D_{a_1}\ a_2 D_{a_2}\ \ldots\ a_n D_{a_n}$ sendo D_{a_i} calculado como:

- $\mathcal{E}_{a_i}$ se $a_i \in A_{in}$;
- $\mathcal{D}_{a_i}$ se $a_i \in A_{out}$;

O processo de inferência é definido como a projeção de $h_{in} \cap H_c$, em relação a A_{out}; e corresponde ao fato que o conjunto de valores dos atributos de entrada, locais e de saída em uma reação devem corresponder a modelos previamente calculados pelo sistema.

3.3 Conclusão

Apresentou-se neste capítulo o formalismo em que o sistema ARKS se baseia: a geração do conjunto de modelos obtidos a partir do conjunto de fórmulas proposicionais correspondentes às regras de produção escritas pelo usuário.

Este formalismo permite uma representação compacta do conhecimento, bem como a implementação de um algoritmo de inferência bastante eficiente, permitindo, desta forma, que os núcleos reativos gerados pelo sistema ARKS sejam compactos o suficiente para atenderem às restrições de tamanho inclusive de sistemas embarcados.

Quanto à complexidade do sistema, pode se afirmar que a compilação da base de conhecimento é um problema NP-completo (problema de satisfação em lógica proposicional). Contudo, o código gerado possui uma complexidade constante e igual ao tempo maximal de transição do autômato, independente da configuração dos estímulos do ambiente, devido ao processo de inferência adotado.

Capítulo 4

O Gerador de Núcleos Reativos ARKS

Neste capítulo são mostrados os detalhes de implementação do sistema ARKS — suas tabelas de símbolos e seus métodos de manipulação da base de conhecimento — bem como as características dos núcleos reativos gerados nas duas opções disponíveis de linguagens (C++ e Java).

O sistema ARKS, objetivo principal deste trabalho, é a implementação da proposta apresentada no Capítulo 3, com a inclusão de conceitos tais como expressões matemáticas envolvendo variáveis, e modo de dependência disjuntiva entre as regras descritas na base de conhecimento. Basicamente, o sistema ARKS é um compilador para a gramática apresentada no Apêndice A, cujo resultado do processo de compilação é um conjunto de arquivos que implementam um núcleo reativo em linguagem fonte C++ ANSI e/ou Java. Este núcleo gerado é implementado de tal forma que a única alteração necessária para a sua aplicação em um ambiente real é a adequação dos métodos de interface com o ambiente. Deste modo, cabe ao usuário somente a descrição do comportamento do sistema através das regras, e dos métodos de interface com o ambiente.

4.1 Visão geral

O desenvolvimento de um sistema reativo utilizando ARKS é feito através da execução dos seguintes passos:

1. Descrição do conhecimento necessário para a solução do problema utilizando a linguagem ARKS na forma de regras de produção;

   A Figura 4.1 é uma listagem escrita na linguagem ARKS que modela um sistema reativo para a venda automática de refrigerantes. Se o usuário inserir um ou mais

créditos na máquina e pressionar o botão, o sistema libera o refrigerante (*Slot = 1.0*) e devolve como troco a quantidade de créditos que o usuário inseriu na máquina menos 1.0 (o valor do refrigerante). Se o usuário inserir menos de um crédito, o sistema não libera o refrigerante (*Slot = 0.0*) e devolve o total de créditos inseridos na máquina.

```
MODULE Refrigerante
{
    DEPENDENCY CONJUNCTION

    INPUTS Valor,Botao;
    OUTPUTS Troco,Slot;

    RULE Libera_Refrigerante IF Valor >= 1.0 AND Botao = 1.0
        THEN Troco = Valor - 1.0 AND Slot = 1.0
        ELSE Troco = Valor AND Slot = 0.0;
}
```

Figura 4.1: Exemplo de uma listagem escrita na linguagem ARKS

2. Compilação desta base de conhecimento pelo sistema ARKS, que criará um conjunto de arquivos que implementam o núcleo reativo correspondente ao comportamento descrito pelas regras, em linguagem C++ ANSI ou Java, de acordo com a opção passada ao sistema (ver Apêndice E);

   A Figura 4.2 mostra a função principal do núcleo reativo gerado a partir da listagem mostrada em 4.1. Esta função lê os dados da entrada padrão (representação *default* do ambiente para leitura de dados no sistema ARKS), executa o processo de inferência, e envia os resultados para a saída padrão (representação *default* do ambiente para envio dos dados no sistema ARKS).

3. Alteração das rotinas de interface com o ambiente geradas pelo ARKS, de modo a implementar o correto acionamento dos dispositivos físicos necessários;

   A Figura 4.3 mostra uma função de entrada de dados a partir da entrada padrão, e uma função de saída de dados para a saída padrão, ambas em linguagem C++ ANSI. Estas funções são criadas em um arquivo apropriado para as alterações que se façam necessárias para atender às necessidades da aplicação (leitura de sensores, acionamento de dispositivos, etc.)

```
void main()
{
    CArks Arks;

    while(1)
    {
        Arks.Refrigerante_Input_Valor();
        Arks.Refrigerante_Input_Botao();
        Arks.Executa();
        Arks.Refrigerante_Output_Troco();
        Arks.Refrigerante_Output_Slot();
    }
}
```

Figura 4.2: Exemplo de código gerado pelo sistema ARKS

4. Compilação e linkedição do código do núcleo reativo e do código de interface para a geração do código executável final, usando um compilador C++ ou Java padrão (e.g. gcc, javac, entre outros).

O processo completo pode ser visualizado na Figura 4.4

4.2 Descrição da base de conhecimento: a linguagem ARKS

A linguagem ARKS, apresentada em detalhe no Apêndice A, descrita preliminarmente em [Kae93] e extendida neste trabalho, foi projetada para permitir a descrição do conhecimento na forma de regras de produção (*IF ... THEN ...*), visto ser esta a representação mais utilizada para a implementação de heurísticas [Ste95].

A base de conhecimento pode ser particionada em módulos, permitindo uma descrição mais simples de um problema, através da partição em subproblemas, sendo cada módulo a descrição do conhecimento necessário para a resolução de cada uma das partes do problema original. A comunicação é feita através das variáveis de entrada e saída, sendo que uma variável de saída pode enviar seus valores para tantos módulos quanto forem necessários.

As principais construções da linguagem ARKS são as seguintes:

//: indica que o texto entre este sinal e o caracter de final de linha é um comentário do usuário e deve ser ignorado pelo compilador.

```
//Funcao de entrada
void CArks::Refrigerante_Input_Valor()
{
    //Altere esta parte do codigo para ler os valores referentes a variavel
    //valor do local apropriado. Após a leitura, atribua este valor a variavel
    // Val

    float Val;
    cout << "Entre com o Valor da Variavel Valor no Modulo Refrigerante: ";
    cin >> Val;

    //Nao alterar esta funcao
    Refrigerante->Input_Valor(Val);
}

//Funcao de Saída
void CArks::Refrigerante_Output_Troco()
{
    //Nao alterar a linha abaixo
    LIST_VALUES::iterator it;

    cout << "Resultados para a Variavel Troco no modulo Refrigerante: n";

    //laco para percorrer todos os valores possiveis para a variavel valor.
    //Consulte manual de referência para implementar as devidas alterações na
    //interpretacao destes dados
    for(it = lRefrigerante_Troco->begin(); it != lRefrigerante_Troco->end();
    it++)
    {
         if((*it)->fValorIni == INFINITO && (*it)->fValorFin == INFINITO)
             cout << " t Qualquer Valor Possível n";
         else if((*it)->fValorIni == INFINITO)
             cout << " t Menor que " << (*it)->fValorFin << " n";
         else if((*it)->fValorFin == INFINITO)
             cout << " t Maior que " << (*it)->fValorIni << " n";
         else if((*it)->fValorIni == (*it)->fValorFin)
             cout << " t Igual a " << (*it)->fValorIni << " n";
         else
             cout << " t Entre " << (*it)->fValorIni << " e " <<
     (*it)->fValorFin << " excluindo os extremos n";
    }
}
```

Figura 4.3: Exemplo de uma função de entrada de dados e de uma função de saída de dados geradas pelo sistema ARKS, em linguagem C++ ANSI

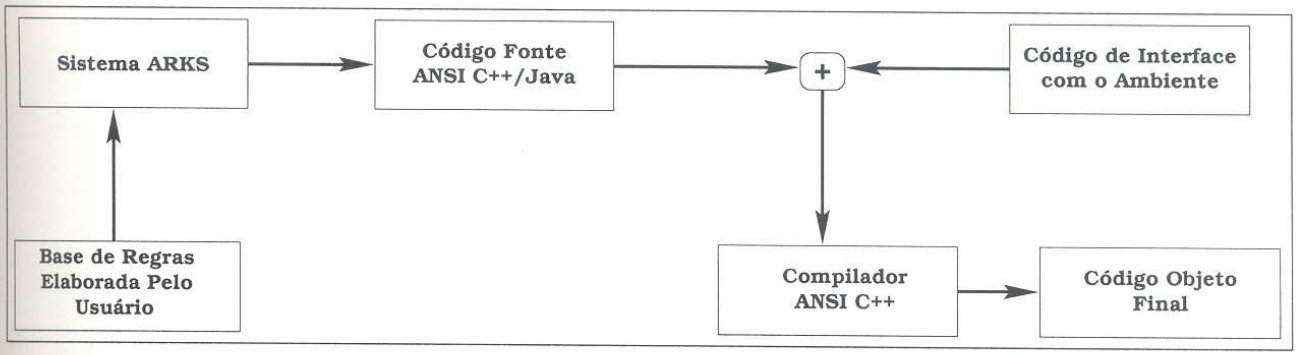

Figura 4.4: Visão geral do funcionamento do sistema ARKS

MODULE *nome-do-módulo*: A estrutura básica de um programa escrito em ARKS é chamado de **módulo** e é delimitado pela construção *MODULE* (Figura 4.5), que delimita uma partição do conhecimento (*contexto*) a ser descrito pelo usuário que trata de uma parte específica do problema. Isto permite uma melhor legibilidade do código escrito e uma manutenção mais simples. Basicamente, um módulo recebe entradas do ambiente ou de outros módulos, processa essas informações e envia os resultados de volta para o ambiente ou para outros módulos. O processo de recepção e envio de dados pelo módulo será descrito adiante.

```
MODULE foo
{
    ...
}
```

Figura 4.5: Declaração de módulo

DEPENDENCY: A construção *DEPENDENCY* (Figura 4.6) indica o tipo de dependência entre as regras do módulo. Esta dependência é relacionada ao conectivo associado à raiz da árvore sintática do conjunto de fórmulas dentro do método das conexões (Seção 3.2).

Se este conectivo for a conjunção, então os modelos associados à raiz da árvore sintática serão aqueles que satisfaçam a *todas* as fórmulas de Γ simultaneamente, ou seja, Γ é considerada uma teoria da qual devem derivar todos os modelos.

Entretanto, é notório que a semântica utilizada pelo usuário na descrição de regras de produção geralmente é a mesma adotada pela área de sistemas especialistas, na qual uma regra é disparada se sua premissa for verdadeira, sendo simplesmente ignorada

caso contrário. Esta abordagem não é passível de utilização direta no método das conexões, visto que este conjunto de fórmulas não constitui uma teoria. Para este caso, seria mais útil que o método das conexões realizasse uma disjunção entre os modelos de cada fórmula, resultando em um conjunto de modelos que satisfaz cada fórmula independentemente.

Portanto, se o conectivo associado à raiz for a disjunção, então o conjunto de modelos associados à raiz da árvore sintática será formado pela união dos conjuntos de modelos individuais de cada fórmula pertencente a Γ, ou seja, de cada regra presente na base de conhecimento do sistema.

A seleção do conectivo associado à raiz da árvore sintática de um módulo é feita pela declaração *DEPENDENCY DISJUNCTION*, para o conectivo de disjunção, e *DEPENDENCY CONJUNCTION* para o conectivo de conjunção.

Obviamente, a descrição do conhecimento dentro de um módulo utilizando a dependência disjuntiva entre as regras se torna mais simples; porém, a utilização da dependência conjuntiva (e da consequente obrigação da descrição do conjunto de regras na forma de uma teoria), produz como resultado um conjunto de modelos mais reduzido, e passível de utilização pelos algoritmos de verificação de propriedades.

```
MODULE foo
{
    DEPENDENCY CONJUNCTION;
    ...
}
```

Figura 4.6: Indicação de dependência entre as regras de um módulo

DIMENSION ***nome-do-atributo[índice]***: Esta construção cria um conjunto de variáveis locais com *índice* elementos que são referenciados através do nome do conjunto *nome-do-atributo* e de sua posição, enumerada de 0 a *índice-1*. A semântica e o funcionamento deste conjunto de variáveis é a mesma dada aos vetores nas linguagens procedurais (Figura 4.7);

INPUTS ***atributo1[,atributo2,...]***: Declara os atributos que receberão os dados para o módulo (Figura 4.8). Este dados podem vir tanto do ambiente quanto de outro módulo, sendo a distinção feita da seguinte forma:

```
MODULE foo
{
    DEPENDENCY CONJUNCTION;
    DIMENSION a[5];
    //cria os atributos a[0], a[1], a[2], a[3], a[4]
    ...
}
```

Figura 4.7: Declaração de conjunto de atributos (*vetor*)

Dado um módulo M, que possui um atributo de *entrada* A.

- Se existe algum módulo M_1 que possua um atributo de *saída* A, então o atributo A de M recebe seus valores do atributo A de M_1;
- Caso contrário, o atributo A do módulo M recebe seus valores do ambiente.

Os atributos podem ser declarados de três modos:

- *nome-do-atributo*: cria um atributo único com o nome *nome-do-atributo*;
- *nome-do-atributo [índice]*: cria um vetor de atributos com o nome *nome-do-atributo*, enumerados de 0 a *indice* − 1;
- *nome-do-atibuto [limite-inferior .. limite-superior]*: cria um vetor de atributos com o nome *nome-do-atributo*, enumerados de *limite-inferior* a *limite-superior*.

OUTPUTS *atributo1[,atributo2,...]*: Declara os atributos que enviarão os dados do módulo (Figura 4.8). Este dados podem ir tanto para o ambiente quanto para outro módulo, sendo a distinção feita da seguinte forma:

Dado um módulo M, que possui um atributo de *saída* A.

- Se existe algum módulo M_1 que possua um atributo de *entrada* A, então o atributo A de M envia seus valores para o atributo A de M_1;
- Caso contrário, o atributo A do módulo M envia seus valores para o ambiente.

Os atributos podem ser declarados de maneira análoga à descrita no ítem anterior.

RULE *nome-da-regra fórmula*: cria uma regra com o nome *nome-da-regra*, que representa a expressão lógica descrita em *fórmula* (Figura 4.9). A descrição das expressões lógicas possíveis para uma regra será mostrada adiante.

```
MODULE foo
{
    INPUTS a, b[3], c[2..4];
    OUTPUTS d, e[5], f[3..9];
    ...
}
```

Figura 4.8: Declaração dos atributos de entrada e saída

TEMPLATE ***nome-do-template*** **[** ***limite-inferior .. limite-superior*** *fórmula:*]
esta construção permite a criação de *limite-superior* − *limite-inferior* regras utilizando o modelo descrito em *fórmula* (Figura 4.9). Um modelo significa que os operadores lógicos serão mantidos e os atributos serão reescritos da seguinte maneira:

- se o atributo for único (não é um vetor), então o nome do atributo será mantido em todas as regras;
- se o atributo for o nome de um vetor, então o atributo será substituído a cada fórmula pelo nome do atributo e um índice, variando de *limite-inferior* a *limite-superior*.

```
MODULE foo
{
    INPUTS a, c[0..4];
    OUTPUTS d, f[2..9];

    TEMPLATE T [2..4] IF a = 1.0 AND c = 0.0 THEN d = 3.0 AND f = 5.0;
    // esta construção internamente gera as seguintes regras:
    // RULE T IF a = 1.0 AND c[2] = 0.0 THEN d = 3.0 AND f[2] = 5.0;
    // RULE T IF a = 1.0 AND c[3] = 0.0 THEN d = 3.0 AND f[3] = 5.0;
    // RULE T IF a = 1.0 AND c[4] = 0.0 THEN d = 3.0 AND f[4] = 5.0;
    ...
}
```

Figura 4.9: Visualização de uma construção *TEMPLATE* em função de suas regras (*RULES*) correspondentes

Expressões Lógicas O conhecimento dentro de um módulo é descrito pelas expressões lógicas das suas regras, sendo cada regra constituída de uma série de construções básicas, divididas em quatro grupos:

- Expressões relacionais: constituídas por um atributo, um operador relacional (<, <= ($\leq$), >, >= ($\geq$), =, != ($\neq$)) e um valor. A semântica das expressões relacionais é descrita da seguinte forma: se o atributo for um atributo de entrada, então a expressão relacional *compara* o valor com o atributo, seguindo a semântica usual do operador relacional correspondente; caso contrário, a expressão relacional *atribui* o valor com o atributo, de acordo com a semântica do operador relacional associado.

  O sistema ARKS possui um valor especial, chamado **ABSENT**, disponível somente para variáveis de entrada, e somente sobre a operação relacional de equivalência (=). Este valor indica ao núcleo reativo que não foi enviado um valor para esta variável na interação atual do núcleo reativo. A seguinte regra:

  $$RULE\ R1\ IF\ In = ABSENT\ THEN\ Out = 1.0\ ELSE\ Out = 2.0;$$

  indica que quando a variável *In* não receber valor algum em determinada interação, o valor 1.0 será atribuído à variável *Out*.

  Cabe lembrar que a *atribuição* $A \geq 3.0$ indica que o atributo A receberá o intervalo $]3.0, +\infty[$. A atribuição através dos outros operadores ocorre de maneira análoga.

- Operadores lógicos: responsáveis pela interação entre as expressões relacionais dentro da expressão lógica, sendo permitido o uso dos seguintes operadores:

  exp_1 **AND** exp_2**:** Conjunção entre duas expressões lógicas;

  exp_1 **OR** exp_2**:** Disjunção entre duas expressões lógicas;

  NOT exp**:** Negação de uma expressão lógica;

  exp_1 **IMPLIES** exp_2**:** Aplicação do operador condicional entre duas expressões lógicas ($exp_1 \rightarrow exp_2$);

  exp_1 **IFONLYIF** exp_2**:** Aplicação do operador bicondicional entre duas expressões lógicas ($exp_1 \leftrightarrow exp_2$);

  As expressões acima permitem a construção de todas as fórmulas bem formadas do cálculo proposicional utilizado, sendo portanto suficientes para a descrição de qualquer base de conhecimento passível de ser manipulada pelo sistema ARKS.

Entretanto, é notória a tendência do especialista humano para a descrição do conhecimento na forma de regras *Se ... Então ...*, na qual a conclusão somente é "executada" se a premissa for verdadeira. Tal construção não é intuitivamente representável em termos de operadores lógicos (ver seção 4.2). Para minimizar este problema, a linguagem ARKS implementa duas construções que permitem a descrição do conhecimento por parte do especialista na forma de regras de produção, quais sejam:

IF exp_1 **THEN** exp_2**:** Este operador apresenta duas semânticas distintas, dependendo da ordem de dependência entre as regras:

- $exp_1 \wedge exp_2$ caso a dependência entre as regras dentro do módulo for disjuntiva. A semântica intuitiva desta construção pode ser entendida como *"Se a premissa é verdadeira, então também o é a conclusão"*; isto significa que a conclusão da expressão só será "executada" se a premissa da regra for verdadeira;
- $(exp_1 \wedge exp_2) \vee (\neg exp_1 \wedge \neg exp_2)$ caso a dependência entre as regras dentro do módulo for conjuntiva. A semântica intuitiva desta construção pode ser entendida como *"Se a premissa é verdadeira, então também o é a conclusão; caso a premissa seja falsa, então a conclusão também será falsa"*; isto significa que exp_2 deve necessariamente ter o mesmo valor-verdade de exp_1. Uma discussão mais detalhada sobre a adoção desta semântica será feita a seguir.

IF exp_1 **THEN** exp_2 **ELSE** exp_3**:** Aplicação do operador condicional nos mesmos moldes das linguagens de programação procedurais. Logicamente, esta expressão é traduzida como $(exp_1 \wedge exp_2) \vee (\neg exp_1 \wedge exp_3)$.

O uso desta última construção por especialistas em sistemas baseados em conhecimento é discutível, visto sua proximidade com as construções condicionais das linguagens imperativas. Entretanto, optou-se por manter esta construção dentro da linguagem como uma simplificação sintática para a descrição de regras com premissas contraditórias, ficando seu uso a critério do usuário.

- Atributos: São as variáveis utilizadas dentro do módulo, sendo possível o uso, além das variáveis de entrada e das variáveis de saída descritas anteriormente, as seguintes variáveis:

variáveis locais: utilizadas para auxiliar a descrição do conhecimento dentro de um módulo. Estas variáveis não recebem valores, de modo que sua função é "particionar" o conjunto de modelos gerado conforme a necessidade do usuário.

variáveis de atraso: é possível para as variáveis de entrada ou para a variáveis de saída a definição de variáveis de atraso, que permitem a consulta aos valores atribuídos a esta variável em interações (reações) passadas. Estas variáveis são descritas na forma *nome-da-variável$n*, indicando que esta variável possui o valor que a variável *nome-da-variável* possuía há n interações. Por exemplo: $A\$1$ indica que esta variável sempre terá o valor atribuído à variável A na interação passada, $A\$2$ na interação retrasada, e assim sucessivamente.

- Valores: valores constantes, pertencentes ao domínio dos reais, declarados dentro da fórmula.

A linguagem utiliza as regras de precedência entre operadores e parentização usuais.

Semântica da construção *IF ... THEN ...*

Conforme mostrado anteriormente, exitem duas semânticas distintas para a construção *IF ... THEN ...* (IF), de acordo com o tipo de dependência entre as regras adotada dentro do módulo.

Intuitivamente, a semântica do operador IF deveria ser a mesma da lógica condicional ($P \rightarrow Q \Leftrightarrow \neg P \vee Q$). O problema da adoção desta semântica dentro da linguagem ARKS se refere ao mecanismo de tratamento do conhecimento utilizado (*modelos de uma fórmula lógica*).

Pela definição, um modelo pode ser visto como uma combinação de valores dados aos atributos de uma fórmula lógica que tornam esta fórmula verdadeira. Aplicando-se este conceito à semântica do operador condicional na lógica proposicional, é possível obter a seguinte interpretação:

"*Se P é verdadeiro, então Q é obrigatoriamente verdadeiro; se P for falso, a Q é permitido qualquer valor-verdade*".

Assim, transportando esta interpretação para as regras da base de conhecimento utilizada no sistema ARKS, pode-se notar que se uma premissa for verdadeira, então a conclusão é verdadeira; porém, se a premissa for falsa, então a conclusão pode ser verdadeira ou não. Na prática, observou-se que realmente, utilizando esta semântica para o operador IF, se a premissa do operador assumisse o valor falso, a conclusão assumia todos

os seus valores possíveis. Do ponto de vista da lógica, este seria o comportamento esperado do sistema; contudo, cabe lembrar que para o usuário que irá descrever o conhecimento utilizando a linguagem, o operador IF possui a mesma semântica que a construção condicional nas linguagens procedurais, ou seja, se a premissa for verdadeira, então a conclusão também é verdadeira (*executada*), caso a premissa for falsa, então a conclusão deve ser ignorada (*não executada*).

Partindo desta análise, assumiu-se como equivalência do operador IF para o sistema ARKS a expressão $IF\ P\ THEN\ Q \Leftrightarrow P \wedge Q$, utilizada quando o tipo de dependência entre as regras for disjuntivo; contudo, quando é definida para um módulo a dependência conjuntiva entre as regras, tal semântica acarreta um problema dentro do método das conexões (mais especificamente, quando da conjunção entre o conjunto de modelos de cada regra para a criação do conjunto de modelos global do módulo).

A equivalência mostrada acima permite que quando a premissa seja verdadeira, a conclusão assuma o valor verdadeiro; sendo que quando a premissa é falsa, a regra simplesmente é ignorada. Agora, seja o seguinte exemplo, mostrando duas regras dentro de um mesmo módulo que possua a dependência conjuntiva entre as regras (Figura 4.10).

```
⋮
RULE R1 IF Pressão = 1.0 THEN Alarme = 0.0;
RULE R2 IF Pressão = 20.0 THEN Alarme = 1.0;
⋮
```

Figura 4.10: Duas regras que tratam de sub-domínios diferentes do atributo *pressão*

Assumindo que não existam outras regras neste módulo, o conjunto de modelos gerados para cada uma das regras seria:

$$R1 = \{\ [Pressão\{1\}\ Alarme\{0\}]\ \}$$
$$R2 = \{\ [Pressão\{20\}\ Alarme\{1\}]\ \}$$

É importante salientar que, do ponto de vista do usuário, este módulo está perfeitamente consistente, indicando os fatores que levam ao acionamento, ou não, de um alarme. Porém, quando o método das conexões realizar a conjunção entre os conjuntos de modelos de R1 e R2, a interseção entre os dois conjuntos será vazia (dado que as regras tratam de sub-domínios diferentes dos atributos), e o sistema acusará que o modelo do módulo será contraditório.

Para amenizar este problema, a semântica do operador IF foi alterada (quando do modo conjuntivo de dependência entre regras) para:

$$IF\ P\ THEN\ Q \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$$

Ou seja, a conclusão assumirá o mesmo valor verdade que a premissa. Na prática, isto cria para cada regra uma espécie de "sombra" que, quando da realização da interseção entre os outros conjuntos de modelos das outras regras, possibilita a criação de um conjunto de modelos que atenda a todas as regras simultaneamente.

Cabe lembrar que, apesar das construções condicionais $IF \ldots THEN \ldots$ e $IF \ldots THEN \ldots ELSE \ldots$ possuírem uma semântica muito parecida com a das linguagens imperativas, estas construções nada mais são do que simplificações sintáticas de expressões lógicas mais complexas, estando portanto de acordo com o formalismo adotado.

4.3 Processo de manipulação da base de conhecimento

Este processo é responsável pela geração dos modelos de cada módulo descrito pelo usuário, através da representação em estruturas **hcubo** e **Hcuboset**.

Consiste basicamente de uma lista dos contextos descritos pelo usuário, sendo que cada contexto possui todas as informações necessárias para a geração dos modelos, conforme será descrito nesta seção.

As ilustrações das tabelas de símbolos utilizadas no sistema ARKS, serão feitas com base no código apresentado na Figura 4.11 (pag. 48).

4.3.1 Representação dos módulos do usuário

Representa cada módulo descrito pelo usuário, e possui os seguintes atributos:

- Uma tabela de símbolos para armazenamento das informações sobre as variáveis dimensionadas;
- Uma lista de variáveis para controle das variáveis utilizadas no módulo;
- Uma lista de regras para representação de cada regra descrita dentro do módulo;
- Uma lista de conjuntos de hipercubos para armazenamento dos conjuntos de hipercubos criados por cada regra;

- Um indicador do tipo de dependência entre as regras;
- Um indicador do nome do módulo;
- Um mapa de bits usado para detecção de ciclos de execução entre as regras;

4.3.2 Tabela de informações sobre dimensionamento de variáveis

Quando o usuário cria um *array* de variáveis, é criada uma entrada na tabela de informações de dimensionamento com informações sobre o nome do *array*, o limite inferior e o limite superior do dimensionamento, com o objetivo de validar referências feitas à este *array* como, por exemplo, verificar se o índice do *array* em uma dada regra é válido.

Para o sistema ARKS, um *array* é uma coleção de variáveis distintas, enumeradas a partir do limite inferior do *array* até o seu limite superior. Deste modo, quando da criação de uma entrada na tabela de dimensionamento, o sistema cria $limite_{inferior} - limite_{superior}$ variáveis, denominadas segundo a seguinte regra:

$$nome\text{-}do\text{-}array\_i, \text{ para todo } limite_{inferior} \leq i \leq limite_{superior}$$

Um exemplo da criação de um array pode ser visto na Tabela 4.1.

declaração da variável	variáveis criadas
A[1..3]	A_1 A_2 A_3
B[2]	B_0 B_1 B_2

Tabela 4.1: Relação entre a criação de um array de variáveis pelo usuário e o conjunto de variáveis criadas pelo ARKS

4.3.3 Tabela de variáveis

Cada módulo possui uma tabela própria de variáveis, com todas as informações necessárias sobre cada variável, sendo as mais significativas para este trabalho:

- Um indicador do nome da variável;

- Uma lista de valores declarados explicitamente para a variável dentro do módulo, chamada de *lista de intervalos*;
- Um indicador de atraso da variável: quando o usuário declara uma variável de atraso, por exemplo, $A\$1$, o sistema ARKS cria uma nova entrada na tabela de variáveis com o mesmo nome A para a variável de atraso, setando neste campo o atraso correspondente (1). Isto facilita o processo de geração de código, uma vez que a procura pelas variáveis de atraso de uma determinada variável se resume à procura de entradas com o mesmo nome de variável;
- Um indicador de atraso máximo para a variável: utilizado somente para as variáveis que possuem variáveis de atraso declaradas na base de conhecimento. Este valor indica o instante máximo de atraso declarado para aquela variável, facilitando o processo de geração de código, uma vez que a informação necessária para a construção do código de atualização de variáveis no núcleo reativo está disponível na própria variável, sem necessidade de procura por todas as suas variáveis de atraso;
- Um indicador de que a variável é uma variável de entrada, saída ou local.

A lista de variáveis de um módulo tem como função o armazenamento das variáveis de cada módulo e a verificação das ações semânticas sobre variáveis enviadas pelo compilador (verificar se uma variável que está sendo declarada já não existe, verificar se as variáveis estão sendo inicializadas, etc ...). Esta lista pode ser visualizada na Figura 4.12.

```
MODULE M
{
    INPUTS Pressão;
    OUTPUTS Nível_Alarme;

    RULE R1 IF Pressão < 20.0 THEN Nível_Alarme = 0.0;
    RULE R2 IF Pressão >= 20.0 AND Pressão <=80.0 THEN Nível_Alarme = 1.0;
    RULE R3 IF Pressão >80.0 AND Pressão < 100.0  THEN Nível_Alarme = 2.0;
}
```

Figura 4.11: Código utilizado para os exemplos sobre as tabelas de símbolos.

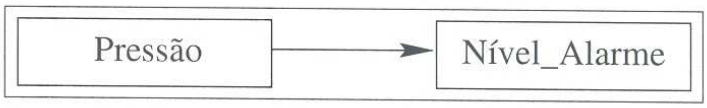

Figura 4.12: Visualização da lista de variáveis, de acordo com as regras apresentadas na Figura 4.11

4.3.4 Discretização dos intervalos

Cada variável dentro do módulo possui uma lista de intervalos que representam os valores declarados explicitamente dentro da base de conhecimento. Esta lista é ordenada em ordem crescente, sendo que cada novo valor inserido na lista é colocado na sua devida posição observando esta ordem, sem repetições (Figura 4.13).

Está lista será utilizada para a criação dos hipercubos deste módulo, bem como o preenchimento destes hipercubos de acordo com as regras declaradas na base de conhecimento para a geração dos modelos, conforme será discutido adiante.

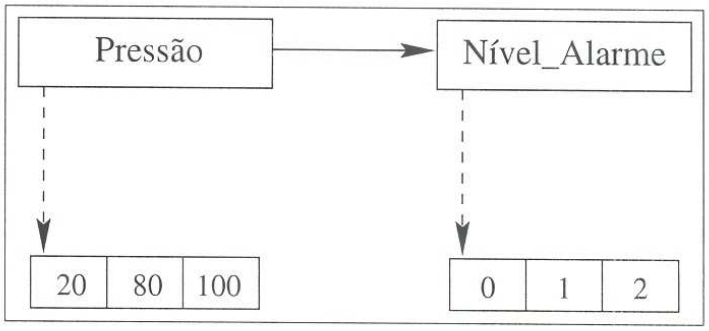

Figura 4.13: Visualização da lista de intervalos de cada variável, de acordo com as regras apresentadas na Figura 4.11

4.3.5 Criação do hcubo

Uma vez terminada a compilação do módulo, é possível dar início à etapa de manipulação, sendo o primeiro passo a definição dos hipercubos que serão utilizados para a geração dos modelos do módulo. Esta definição leva em conta o número de variáveis dentro do módulo, e o número de valores declarados explicitamente para cada variável, de forma a criar uma representação dos intervalos de cada variável dentro de um mapa de bits.

A representação dos intervalos dentro do hipercubo se faz da seguinte forma: cada hipercubo é dividido em *seções*, sendo cada seção referente a uma variável dentro do módulo, de acordo com a sua posição dentro da lista de variáveis, isto é, a primeira seção do hipercubo se refere à primeira variável da lista de variáveis do módulo, e assim por diante.

Dentro de cada seção, o primeiro bit representa todos os valores que sejam menores que o primeiro valor da lista de intervalos da respectiva variável (o menor valor declarado explicitamente para aquela variável dentro da base de conhecimento). A partir daí, até o antepenúltimo bit da seção, a seção é dividida em $n-1$ pares de bits, onde n é o número de valores declarados explicitamente para a variável. A cada par i, o primeiro bit se refere ao valor igual ao *i-ésimo* valor representado dentro da lista de valores da variável, e o segundo bit se refere aos valores compreendidos entre os valores das posições i e $i+1$ da lista de valores da variável. O penúltimo bit da seção representa o último valor da lista de valores da variável, e o último bit da seção representa os valores maiores que o último valor da lista de valores da variável (maior valor declarado explicitamente para aquela variável dentro da base de conhecimento). Caso a variável seja uma variável de entrada, é adicionado ao final da seção um bit que representa o valor **ABSENT**, para mapeamento do caso onde o valor para esta variável de entrada não foi enviado para uma determinada inferência do módulo. A representação gráfica do mapeamento das variáveis do módulo no hipercubo pode ser visualizada na Figura 4.14.

Cabe lembrar que é possível o mapeamento dos intervalos representados por cada um dos bits dentro de cada seção do hipercubo para o domínio dos números naturais (utilizando-se a posição do bit dentro da seção), o que torna esta representação coerente com o formalismo apresentado no Capítulo 3.

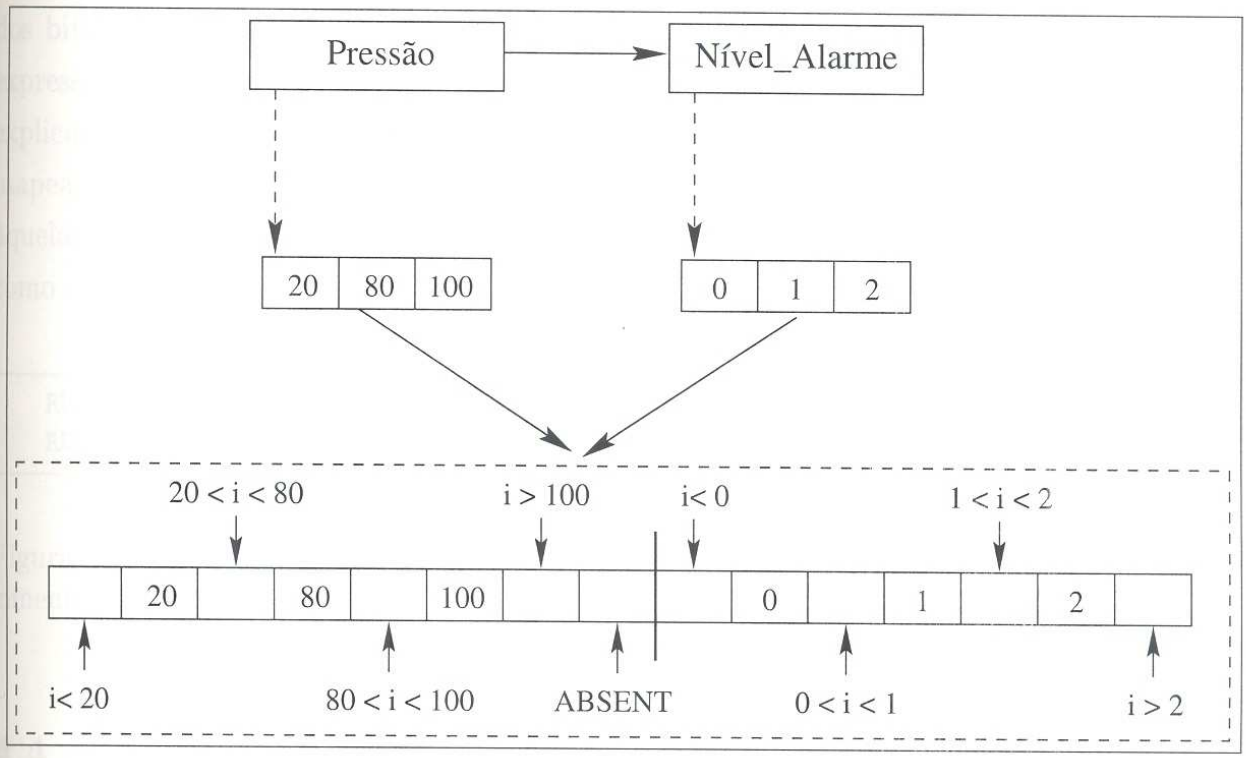

Figura 4.14: Visualização do hipercubo criado a partir da lista de intervalos das variáveis, de acordo com as regras apresentadas na Figura 4.11

Representação de Expressões Matemáticas

A linguagem ARKS permite que o valor a ser atribuído a uma variável seja obtido de uma expressão matemática que envolva variáveis de entrada do ambiente. Uma vez que não é possível a resolução de tal expressão em tempo de compilação, a representação desta expressão é feita através da alocação de um bit para cada expressão dentro do **hcubo** após o bit que representa os valores maiores que o maior valor declarado explicitamente para a variável, ou o valor **ABSENT**, caso seja uma variável de entrada. A partir deste ponto, uma expressão é tratada de modo idêntico ao anteriormente apresentado, sujeita aos mesmos algoritmos de tratamento de hipercubos. Uma exceção é a semântica dos bits posteriores ao bit referente à expressão, que passa a assumir a negação desta expressão ao invés do intervalo entre este valor e o próximo. Esta diferenciação será explicada adiante, na seção que trata do preenchimento de um **hcubo**. Para ilustrar o mapeamento de expressões matemáticas, considere o acréscimo das regras da Figura 4.15 àquelas apresentadas na Figura 4.11. Esta nova base de conhecimento criará um **hcubo** como o mostrado na Figura 4.16

```
RULE R4 IF Pressão = 100.0 THEN Nível_Alarme = Pressão*2.0;
RULE R5 IF Pressão > 100.0 THEN Nível_Alarme = Pressão*3.0;
```

Figura 4.15: Inclusão de regras que lidam com expressões matemáticas na base de conhecimento apresentada na Figura 4.11

4.4 Preenchimento dos hcubos

4.4.1 Para as expressões atômicas de uma fórmula

Dentro do algoritmo de geração dos modelos de uma fórmula, pode-se delimitar dois procedimentos distintos:

1. A geração de um conjunto de modelos (**Hcuboset**) que corresponde a um nó não-terminal (conectivos da fórmula), que é realizada de acordo com a lógica descrita no Capítulo 3;

2. A geração de um **hcubo** que corresponde a um nó folha da fórmula (expressões atômicas), que é feito através do preenchimento da estrutura **hcubo**, de acordo com o operador relacional envolvido.

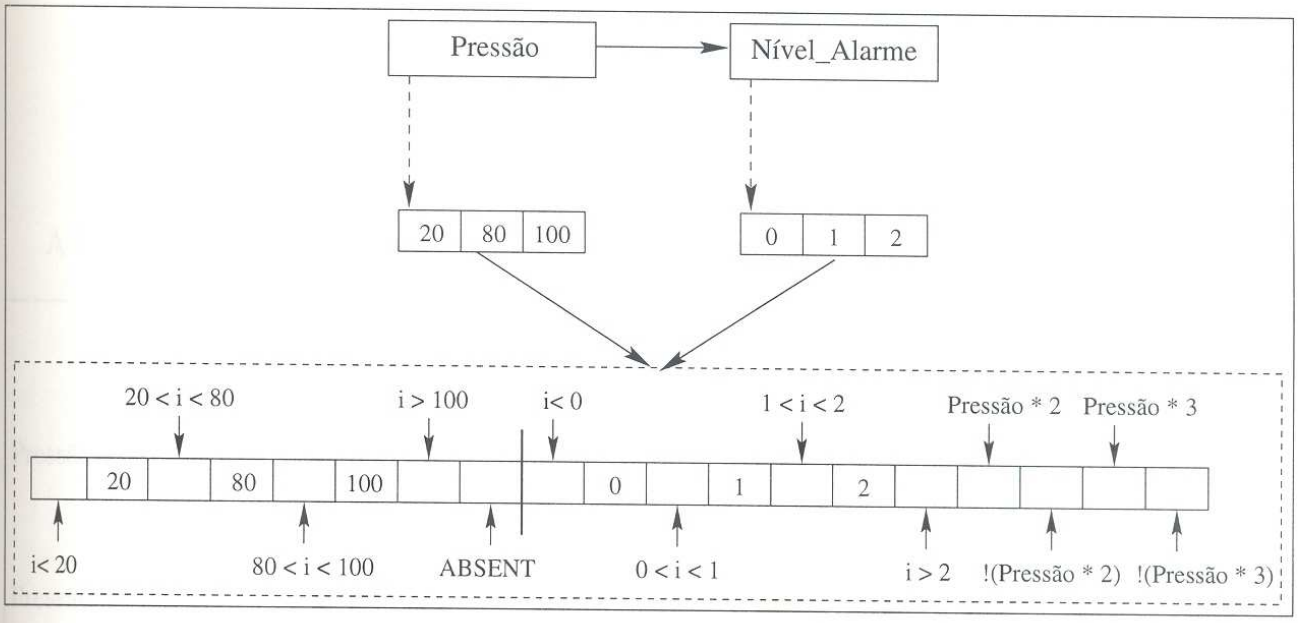

Figura 4.16: Visualização do hipercubo criado a partir das regras apresentadas nas Figuras 4.11 e 4.15

O algoritmo utilizado dentro do sistema ARKS para o preenchimento de uma estrutura **hcubo**, quando da geração dos modelos de um nó terminal de uma fórmula, segue os seguintes passos:

1. Criação de um **hcubo**;
2. Atribuição do valor 1 a todos os bits do **hcubo** que forem referentes a qualquer variável que não seja a variável objeto da expressão;
3. Atribuição do valor 1 a todos os bits referentes a expressões desta variável;
4. Atribuição de valores aos demais bits da seção segundo as regras abaixo:
   - Operador =: os bits são setados em 0, com exceção daquele referente ao valor utilizado na operação, que deve ser setado em 1;
   - Operador ! =: os bits são setados em 1, com exceção daquele referente ao valor utilizado na operação, que deve ser setado em 0;
   - Operador <: Todos os bits anteriores ao bit referente ao valor utilizado na operação devem ser setados em 1; os demais bits são setados em 0;
   - Operador <=: Todos os bits posteriores ao bit referente ao valor utilizado na operação devem ser setados em 0; os demais bits são setados em 1;

- Operador >: Todos os bits posteriores ao bit referente ao valor utilizado na operação devem ser setados em 1; os demais bits são setados em 0;
- Operador >=: Todos os bits anteriores ao bit referente ao valor utilizado na operação devem ser setados em 0; os demais bits são setados em 1;

A ilustração do preenchimento de um **hcubo** pode ser vista na Figura 4.17.

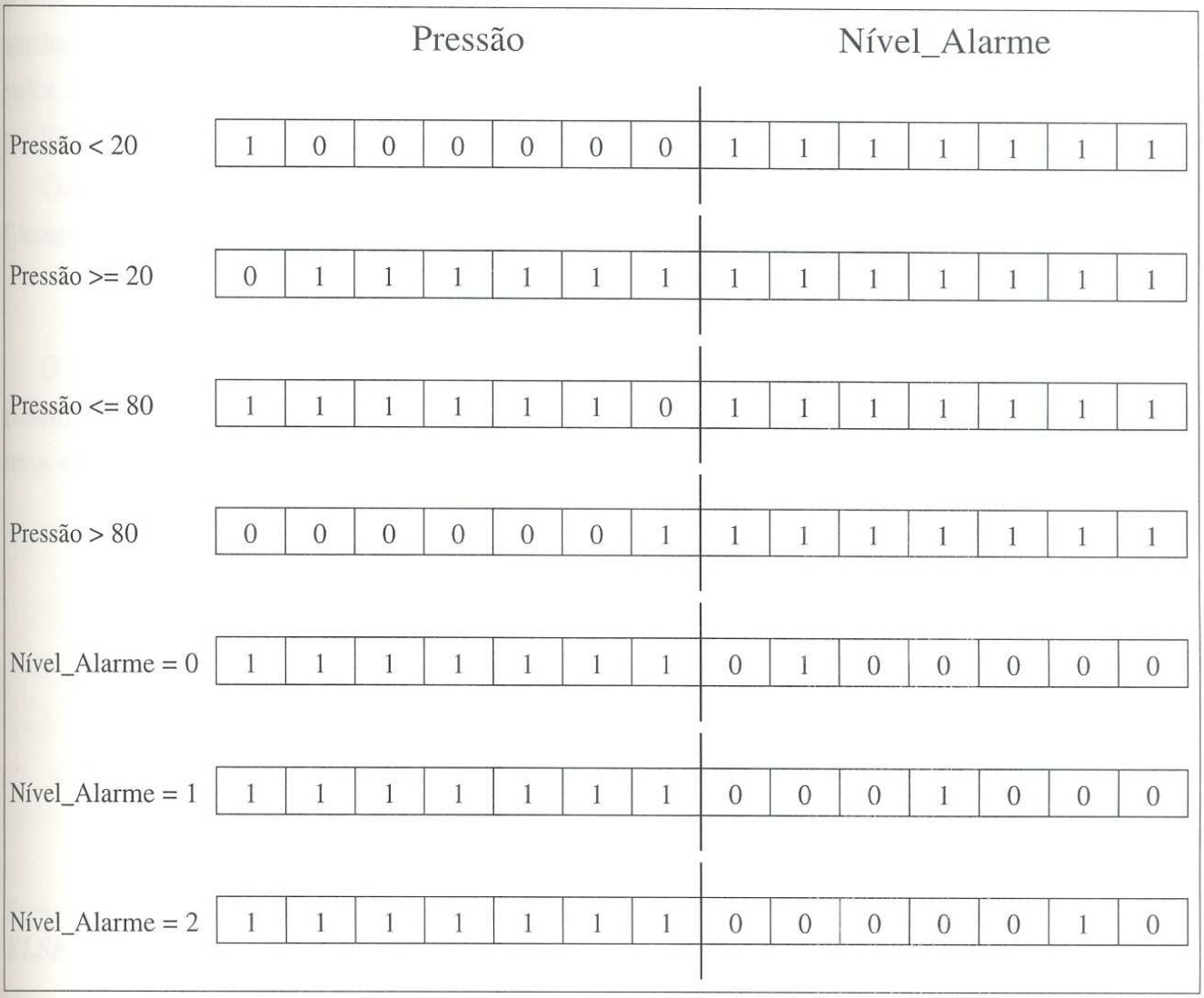

Figura 4.17: Visualização dos hipercubos gerados para os nós terminais das fórmulas apresentadas na Figura 4.11

Existe uma diferença quando do preenchimento de uma estrutura **hcubo** para as expressões matemáticas. Devido à impossibilidade da avaliação de uma expressão em tempo de compilação, para a geração dos modelos é considerado que estas expressões são verdadeiras, sendo que em tempo de execução, estas expressões são avaliadas da seguinte forma:

- Se a expressão for verdadeira, o bit referente à esta expressão é setado em 1, e o bit posterior a esta expressão é setado em 0;
- Se a expressão for falsa, o bit referente à esta expressão é setado em 0, e o bit posterior a esta expressão é setado em 1.

Este procedimento é adotado porque a negação de um valor dentro de um **hcubo** é o complemento deste **hcubo**, ou seja, a negação de uma expressão não é feita através da atribuição do valor 0 ao bit referente a esta expressão, mas sim através da atribuição do valor 1 aos demais bits.

Este processo pode ser melhor compreendido através do seguinte exemplo:

Considere um módulo com modo de dependência disjuntiva contendo a seguinte regra (Assume-se que X e Y são variáveis de entrada, e A é uma variável de saída):

$$IF\ X > Y\ THEN\ A = 0.0\ ELSE\ A = 1.0;$$

O **Hcuboset** obtido para este módulo é mostrado na Figura 4.18 (a variável Y não possui representação dentro do **hcubo**, uma vez que ela somente é utilizada dentro de uma expressão referente à variável X).

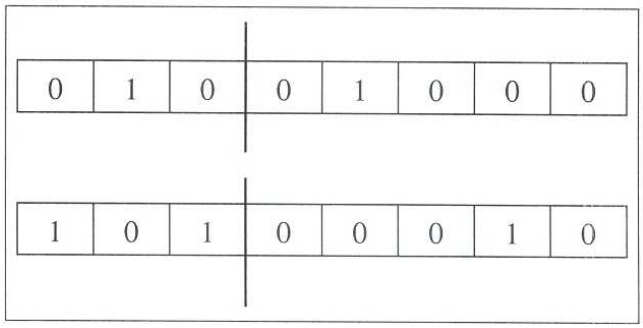

Figura 4.18: Visualização do **Hcuboset** obtido para a regra $IF\ X > Y\ THEN\ A = 0.0\ ELSE\ A = 1.0$ (A primeira seção se refere à variavel X, a segunda à variável A).

O primeiro **hcubo** representa a avaliação verdadeira da premissa, enquanto que o segundo representa a avaliação falsa da mesma. A Figura 4.19 mostra dois **hcubos** criados a partir dos dados de entrada enviados pelo ambiente, utilizando-se duas abordagens para a representação de uma expressão quando esta assume o valor falso:

1. Setar o bit correspondente à expressão em 0;
2. Setar o bit correspondente à expressão em 0 e o bit posterior à expressão em 1.

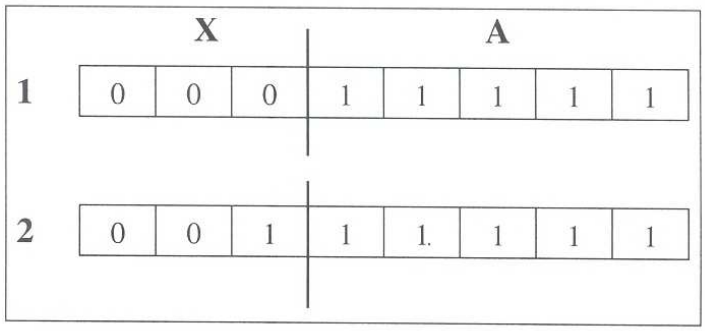

Figura 4.19: Visualização do **hcubo** obtido a partir dos dados de entrada do ambiente, utilizando-se as duas interpretações possíveis para a avaliação de uma expressão.

Uma vez que o processo de inferência é basicamente a conjunção do **hcubo** obtido pelos dados de entrada com o **Hcuboset** obtido pelos modelos do módulo, nota-se que o **hcubo** 1 não produziria resultados, uma vez que a sua interseção com o **Hcuboset** produziria um conjunto vazio. Porém, a interseção do **hcubo** 2 com o **Hcuboset** produz como resultado de saída exatamente a seção referente ao **hcubo** produzido para a representação da negação da expressão (o processo de inferência é descrito mais detalhadamente na seção 4.5).

4.4.2 Para as fórmulas

Cada módulo também possui uma lista com todas as regras nele descritas, para a posterior geração dos modelos do módulo.

Para o formato de armazenamento de cada fórmula, foi selecionada a forma pós-fixada, principalmente por dois motivos:

1. Por natureza, o processo de compilação de uma expressão lógica resulta na forma pós-fixada da mesma, de modo que a alteração desta característica envolveria um aumento na complexidade do sistema;

2. Uma vez que a expressão já está na forma pós-fixada, utiliza-se um algoritmo específico para a sua resolução [Sor85].

Uma visualização da estrutura da lista de fórmulas pode ser vista na Figura 4.20.

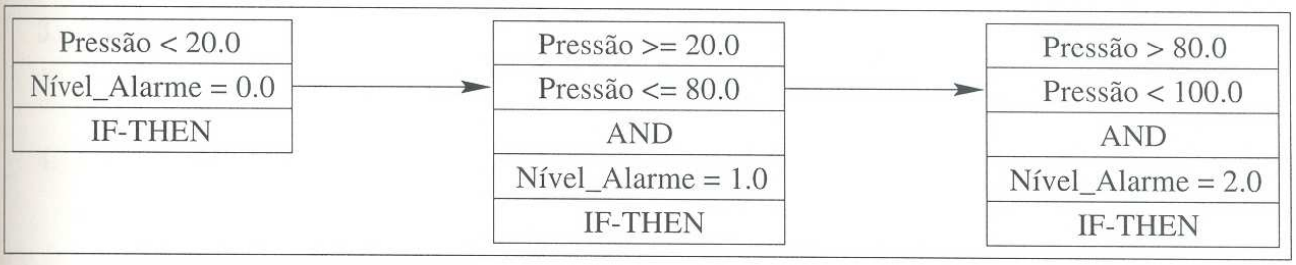

Figura 4.20: Visualização da lista de fórmulas, de acordo com a base de regras apresentada na Figura 4.11

4.5 O Processo de inferência

O processo de inferência dentro de um módulo é realizado através dos seguintes passos:

- Criar um **hcubo** vazio;
- Para cada valor enviado para um variável de entrada, setar o(s) bit(s) correspondente(s) dentro da sua seção no **hcubo**;
- Atribuir o valor 1 a todos os bits das seções referentes à variáveis de saída;
- Efetuar a conjunção entre este **hcubo** e o **Hcuboset** que representa os modelos do módulo;
- Eliminar do **Hcuboset** resultante todos os **hcubos** vazios, isto é, todos os **hcubos** que possuem pelo menos uma seção com todos os bits com valor 0;
- Interpretar as seções referentes às variáveis de saída e enviar esses dados para o ambiente.

Geração dos modelos das fórmulas Conforme descrito anteriormente, o sistema ARKS utiliza o algoritmo de resolução de expressões pós-fixadas, aproveitando a característica inerente ao compilador de gerar expressões pós-fixadas das expressões infixadas descritas na base de conhecimento.

Para a descrição do algoritmo, serão utilizadas as seguintes definições:

1. A pilha de expressões pós-fixadas que representa a fórmula descrita pelo usuário será denotada *pExpressões*;
2. É necessária a criação de uma pilha de conjuntos de hipercubos **Hcuboset** vazia, que será denotada *pHcuboset*;

3. Ambas as pilhas possuem um método chamado *pop*, que retira o elemento do topo da pilha e produz como resultado uma referência a este elemento retirado.

4. A pilha *pHcuboset* possui um método *push(parâmetro)*, que aloca *parâmetro* no topo de *pHcuboset*;

5. A pilha *pExpressões* possui um método *top*, que produz como resultado uma referência ao elemento alocado no topo de *pExpressões*;

6. Também é definido em *pExpressões* o método *vazio*, que produz como resultado o valor *verdadeiro* caso *pExpressões* não possua elementos, e o valor *falso* caso contrário;

7. O operador *pilha*$\longrightarrow$*método* indica a execução de *método* sobre *pilha*;

8. Pressupõe-se a existência de variáveis do tipo *Hcuboset*, nominadas $tmp_1, \ldots, tmp_n$, utilizadas para armazenamento temporário das operações;

9. O procedimento *cria-hipercubo* cria um **hcubo** a partir da expressão relacional alocada no topo de *pExpressões*, conforme descrito na seção anterior;

10. Serão utilizadas as construções usuais de laço para a descrição de algoritmos;

O algoritmo é descrito na Figura 4.21 e os procedimentos adicionais nele utilizados são apresentados detalhadamente no Apêndice C.

Quando do término da execução do algoritmo, existirá apenas um elemento em pHipercubos, que representa o conjunto de modelos desta fórmula. Este elemento é armazenado em uma lista auxiliar, de modo que, uma vez executado o algoritmo para todas as fórmulas, esta lista possua o conjunto de modelos de cada fórmula descrita no módulo. Executando a operação de interseção de conjuntos entre todos os elementos desta lista, obtém-se o conjunto de modelos do módulo, que é armazenado em uma estrutura **Hcuboset** para posterior utilização pelo gerador de código. Este procedimento é operacionalmente similar ao método das conexões, fundamento formal do sistema.

```
enquanto(pExpressões⟶vazio = falso) faça
  escolha(pExpressoes⟶top)
    caso AND:
      tmp1 = pHipercubos⟶pop
      tmp2 = pHipercubos⟶pop
      tmp3 = procedimento-interseção(tmp2 ,tmp1 )
      pHipercubos⟶push(tmp3 )
    caso OR:
      tmp1 = pHipercubos⟶pop
      tmp2 = pHipercubos⟶pop
      tmp3 = procedimento-união(tmp2 ,tmp1 )
      pHipercubos⟶push(tmp3 )
    caso NOT:
      tmp1 = pHipercubos⟶pop
      tmp2 = procedimento-complemento(tmp1 )
      pHipercubos⟶push(tmp2 )
    caso IMPLIES:
  tmp1 = pHipercubos⟶pop
      tmp2 = pHipercubos⟶pop
      tmp3 = procedimento-implicacao(tmp2 ,tmp1 )
      pHipercubos⟶push(tmp3 )
    caso IF ... THEN
      Se modo de dependência entre regras for conjuntiva
        tmp1 = pHipercubos⟶pop
        tmp2 = pHipercubos⟶pop
        tmp3 = procedimento-condicional-conjuntiva(tmp2 ,tmp1 )
      Senão
        tmp1 = pHipercubos⟶pop
        tmp2 = pHipercubos⟶pop
        tmp3 = procedimento-condicional-disjuntiva(tmp2 ,tmp1 )
      Fim-se
      pHipercubos⟶push(tmp3 )
    caso IF ... THEN ... ELSE
      tmp1 = pHipercubos⟶pop
      tmp2 = pHipercubos⟶pop
      tmp3 = pHipercubos⟶pop
      tmp4 = procedimento-interseção(tmp3 ,tmp2 )
      tmp5 = procedimento-interseção(tmp3 ,tmp1 )
      tmp6 = procedimento-união(tmp4 ,tmp5 )
      pHipercubos⟶push(tmp6 )
    caso IF ... ONLYIF
      tmp1 = pHipercubos⟶pop
      tmp2 = pHipercubos⟶pop
      tmp3 = procedimento-bicondicional(tmp2 ,tmp1 )
      pHipercubos⟶push(tmp3 )
    caso-contrário
      tmp1 = cria-hipercubo
      pHipercubos⟶push(tmp1 )
  fim-escolha
  pExpressoes⟶pop
fim-enquanto
```

Figura 4.21: Decrição do algoritmo de geração dos modelos de uma fórmula no sistema ARKS.

4.6 Detecção de ciclos entre os módulos

No sistema ARKS, o fluxo de dados entre os módulos é feito através dos nomes dados às variáveis de entrada e às variáveis de saída dentro de cada módulo, obedecendo às seguintes regras:

- Se um determinado módulo A possui uma variável de saída V_1, e um outro módulo B possui uma variável de saída com o mesmo nome V_1, então o valor da variável V_1 no módulo B será fornecido pela variável V_1 no módulo A, obrigando, deste modo, que a execução do módulo A seja feita antes do módulo B.

- Se um determinado módulo A possui uma variável de entrada V_1, e nenhum outro módulo descrito na base de conhecimento possua uma variável de saída com o nome V_1, então o valor desta variável V_1 será enviado pelo ambiente; se todas as variáveis deste módulo tiverem seus valores enviados pelo ambiente, então este módulo será o primeiro a ser executado;

- Caso mais de um módulo possua todas as suas entradas vindas do ambiente, a ordem de execução entre estes módulos será a ordem de sua descrição na base de conhecimento.

A Figura 4.22 apresenta a listagem de uma base de conhecimento composta de vários módulos; o fluxo de execução gerado pelo sistema ARKS para este exemplo é ilustrado pela Figura 4.23.

Uma vez que a ordem de execução de processos entre os módulos é definida exclusivamente sobre os fluxos de dados entre eles, fica claro o problema da definição, por parte do usuário, de ciclos de execução entre os módulos, como no código apresentado na Figura 4.24.

Um exemplo de ciclo é apresentado nas Figuras 4.24 e 4.25, onde é possível verificar que uma das entradas necessárias para a execução do módulo M2 é a variável V7, cujo valor será fornecido pelo módulo M4. Porém, o módulo M4 necessita como entrada a variável V5 enviada pelo módulo M3, que por sua vez necessita que o módulo M2 tenha sido executado para possibilitar o envio do valor de V4, fechando assim o ciclo. Em resumo, o módulo M2 necessita que M4 tenha sido executado para poder realizar a sua inferência, e o módulo M4 não pode ser executado enquanto M2 não o for.

```
MODULE M1
{
   INPUTS V1;
   OUTPUTS V2;
}
MODULE M2
{
   INPUTS V2,V3;
   OUTPUTS V4,V5;
}
MODULE M3
{
   INPUTS V2,V4;
   OUTPUTS V6,V7;
}
MODULE M4
{
   INPUTS V5,V6,V8;
   OUTPUTS V9;
}
```

Figura 4.22: Código exemplo da criação da ordem de execução entre os módulos.

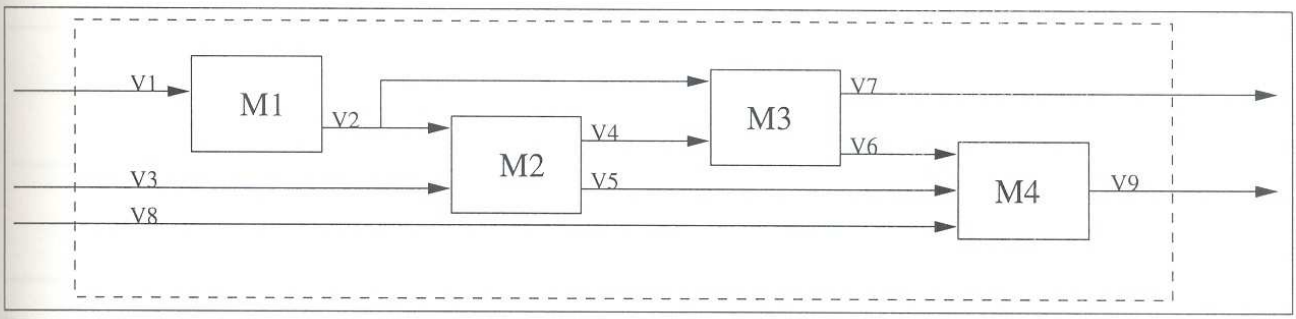

Figura 4.23: Ordem de execução dos módulos apresentados na Figura 4.22

```
MODULE M1
{
   INPUTS V1;
   OUTPUTS V2;
}
MODULE M2
{
   INPUTS V2,V3,V7;
   OUTPUTS V4;
}
MODULE M3
{
   INPUTS V2,V4;
   OUTPUTS V5,V6;
}
MODULE M4
{
   INPUTS V5;
   OUTPUTS V7;
}
```

Figura 4.24: Código exemplo da descrição de uma base de conhecimento com ciclo de execução entre os módulos M2 e M4.

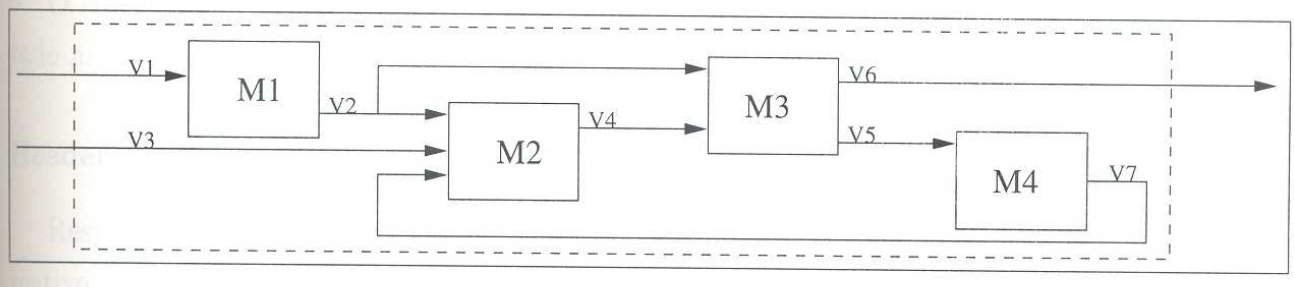

Figura 4.25: Ordem de execução dos módulos apresentados na Figura 4.24, mostrando o ciclo entre M2 e M4.

Para evitar este problema, o sistema ARKS verifica a existência de ciclos entre os módulos, reportando um erro no processo de compilação da base de conhecimento caso esse ciclo seja encontrado. O algoritmo para a verificação da ordem de execução entre os módulos de uma base de conhecimento é apresentado no Apêndice D.2.

4.7 Geração do código-fonte a partir da base de conhecimento

Uma vez completada a etapa de geração dos modelos da base de conhecimento descrita pelo usuário, o sistema ARKS passa para a etapa de geração de código, responsável por criar os arquivos com o código-fonte, que implementa um núcleo reativo com o comportamento descrito pelos modelos.

4.7.1 Em linguagem C++ ANSI

Uma opção é a geração em linguagem C++ ANSI. Nesta modalidade, são criados três arquivos para a implementação do núcleo reativo:

- *Header file*, cujo nome é formado pelo nome do arquivo que contém a base de conhecimento descrita pelo usuário, seguido da extensão *.hpp*;
- *Interface file*, cujo nome é formado pela palavra *main_*, seguido do nome do arquivo que contém a base de conhecimento, seguido da extensão *.cpp*.
- *Source file*, cujo nome é formado pelo nome do arquivo que contém a base de conhecimento, seguido da extensão *.cpp*;

O Apêndice G apresenta a listagem dos três arquivos gerados para o exemplo apresentado na Figura 4.26.

Header File

Responsável pela descrição dos atributos e protótipos da classe principal do núcleo reativo, bem como das classes que implementam cada módulo descrito pelo usuário.

A definição da classe principal (*CArks*) é apresentada na Figura 4.27.

```
MODULE Refrigerante
{
    DEPENDENCY DISJUNCTION;

    INPUTS Valor, Botao;
    OUTPUTS Troco, Slot;

    RULE T1 IF Valor >= 1.0 AND Botao = 1.0 THEN Troco = Valor - 1.0 AND
    Slot = 1.0
            ELSE Troco = Valor AND Slot = 0.0;
}
```

Figura 4.26: Base de conhecimento exemplo.

```
class CArks {
private:
    ponteiros para instâncias das classes que representam os módulos da
     base de conhecimento
public:
    definição das estruturas que receberão os valores a serem enviados
     para o ambiente após cada interação

   CArks();
   ~CArks();
   void Executa();
    protótipos dos métodos de entrada
    protótipos dos métodos de saída
};
```

Figura 4.27: Modelo de definição da classe *CArks*.

Toda a comunicação entre o ambiente e os módulos do núcleo reativo se faz através da classe CArks, sendo esta a única estrutura a possuir as referências às instâncias que representam cada um dos módulos. Também é função da classe CArks disponibilizar para o ambiente os valores obtidos para as variáveis de saída do núcleo reativo, sendo para isto utilizada para cada variável de saída uma estrutura que mantém todos os valores obtidos para aquela variável em cada interação do núcleo reativo com o ambiente.

Cabe, portanto, a cada um dos módulos, o gerenciamento de seu conjunto de modelos, de suas variáveis utilizadas em expressões e de suas variáveis de atraso, bem como do processo de inferência propriamente dito. O modelo da definição de uma classe que representa um módulo descrito pelo usuário é mostrado na Figura 4.28.

```
class Cnome-do-módulo : public CModulo {
private:
   declaração das variáveis utilizadas em expressões
   declaração das variáveis de atraso
   declaração dos ponteiros para as estruturas de dados que armazenarão
    os valores obtidos para as variáveis de saída
   void Atualiza_Variaveis_Expressao_Atraso();
public:
  Cnome-do-módulo(CArks*);
  ~Cnome-do-módulo();
   void Executa();
    protótipos dos métodos de entrada de dados
    protótipos dos métodos de saída de dados
};
```

Figura 4.28: Modelo de definição das classes que representam os módulos descritos pelo usuário.

Quando uma variável de entrada é utilizada como operando em uma expressão matemática, é criada uma variável auxiliar para o armazenamento do valor enviado pelo ambiente para esta variável, tornando-o disponível no momento da avaliação desta expressão.

Variáveis auxiliares também são criadas para o armazenamento dos valores atribuídos às variáveis em reações passadas, permitindo a consulta desses valores pelas variáveis de atraso declaradas dentro do módulo.

O mecanismo de funcionamento das variáveis de atraso e das variáveis que são utilizadas em expressões é descrito na seção 4.7.1, bem como a forma de implementação dos métodos para a entrada e a saída de dados.

Interface File

Este arquivo é responsável pela interação entre o ambiente e o núcleo reativo. Por *default*, esta interação é feita via a entrada e saída padrão do sistema. Porém, os métodos de entrada e saída de dados do núcleo reativo são implementados de forma a permitir uma fácil alteração para quaisquer dispositivos que se façam necessários para a aplicação.

Também está no arquivo de interface o ponto de entrada do núcleo reativo, implementado como na Figura 4.29.

```
void main()
{
   CArks Arks;

   while(1)
   {
       chamada aos métodos de entrada de dados
      Arks.Executa();
       chamada aos métodos de saída de dados
   }
}
```

Figura 4.29: Modelo do método principal do sistema ARKS

Métodos para a entrada de dados Cada variável de entrada declarada dentro de um módulo que receba seus dados do ambiente possui, neste arquivo, um método para a leitura e envio de dados ao núcleo reativo, conforme Figura 4.30.

Este método permite total alteração por parte do usuário para a adequação às necessidades da aplicação, desde que seja observada a última linha de código do método, que envia o valor lido para o núcleo reativo.

Métodos para o acionamento do núcleo reativo Este método invoca o processo de inferência do núcleo reativo, através da execução dos métodos de inferência de cada módulo, obedecendo a ordem de execução descrita na seção 4.6.

Métodos de saída de dados para o ambiente O método de execução não invoca automaticamente os métodos de saída de dados, permitindo assim que o usuário execute alguns comandos entre o processo de inferência e o processo de leitura dos valores enviados

```
void CArks::Nome-do-módulo_Input_Nome-da-variável()

   float Val;
   //código customizável pelo usuário, atribuir em Val o valor a ser passado
   //ao núcleo reativo
   cout << "Entre com o Valor da Variavel Nome-da-variável no Modulo
       Nome-do-módulo: ";
   cin >> Val;
   //fim do código customizável
    Nome-do-módulo->Input_Nome-da-Variável(Val);
```

Figura 4.30: Modelo do método de entrada de dados do sistema ARKS

pelo núcleo reativo, conforme a necessidade da aplicação. Este processo também permite ao usuário consultar os valores enviados pelo núcleo reativo quantas vezes for desejado, até que outro processo de inferência seja acionado.

Cada variável de saída declarada dentro dos módulos possui um método neste arquivo que recebe os valores inferidos pelo núcleo reativo, sendo este método classificado em um dentre dois tipos possíveis:

- Saída para o ambiente: o método recebe os dados do núcleo reativo, interpreta e envia os mesmos para o ambiente;
- Envio de dados para outro módulo: é possível que os dados de saída de um módulo sejam direcionados para outro, conforme visto anteriormente. Porém, o sistema ARKS também cria o método que realiza este envio no arquivo de interface, permitindo que o usuário o utilize como sinal de saída para o ambiente.

Ambos os métodos trabalham com uma estrutura chamada **LIST_VALUES**, que implementa uma lista de pares que representam todos os valores obtidos pelo processo de inferência do núcleo reativo para aquela variável.

Tal procedimento foi adotado devido à característica intervalar utilizada para os valores das variáveis de saída, sendo possível para o usuário descrever uma base de conhecimento cujas variáveis de saída não representem um ponto, mas sim um ou mais segmentos na reta dos reais. Assim, cada elemento de LIST_VALUES é um par que representa o limite inferior (*fValorIni*) e o limite superior (*fValorFin*) deste segmento; obviamente, se os dois componentes do par possuírem o mesmo valor, a representação da variável é pontual.

Para ilustrar algumas possibilidades para os valores de saída, considere os exemplos apresentados na Tabela 4.2, onde a primeira coluna mostra uma regra que pode ser descrita dentro de um contexto, e a segunda apresenta os intervalos associados à variável de saída *Out* quando a variável de entrada *In* recebe o valor 1.0 (assume-se que o contexto onde está descrita cada regra possui modo de dependência disjuntiva entre regras). A representação destas estruturas pode ser visualizada na Figura 4.31.

Num.	Regra	Intervalo
1	RULE R1 IF In = 1.0 THEN Out = 2.0;	[2.0 , 2.0]
2	RULE R1 IF In = 1.0 THEN Out > 2.0;	]2.0 , $+\infty$[
3	RULE R1 IF In = 1.0 THEN Out >= 2.0;	[2.0 , $+\infty$[
4	RULE R1 IF In = 1.0 THEN Out < 2.0;	] $-\infty$, 2.0[
5	RULE R1 IF In = 1.0 THEN Out <= 2.0;	] $-\infty$, 2.0]
6	RULE R1 IF In = 1.0 THEN Out > 0.0 AND Out < 2.0;	]0.0 , 2.0[
7	RULE R1 IF In = 1.0 THEN Out >= 0.0 AND Out <= 2.0;	[0.0 , 2.0]
8	RULE R1 IF In = 1.0 THEN Out >= 0.0 AND Out <= 2.0 AND Out != 1.0;	[0.0 , 1.0[ ]1.0 , 2.0]

Tabela 4.2: Exemplos de intervalos atribuídos à variável *Out*.

A Figura 4.31 mostra a estrutura LIST_VALUES criada para representar o intervalo da variável *Out* para cada um dos exemplos da Tabela 4.2

O modelo de saída de dados para o ambiente é apresentado na Figura 4.32, e o modelo de envio de dados de um módulo a outro é apresentado na Figura 4.33.

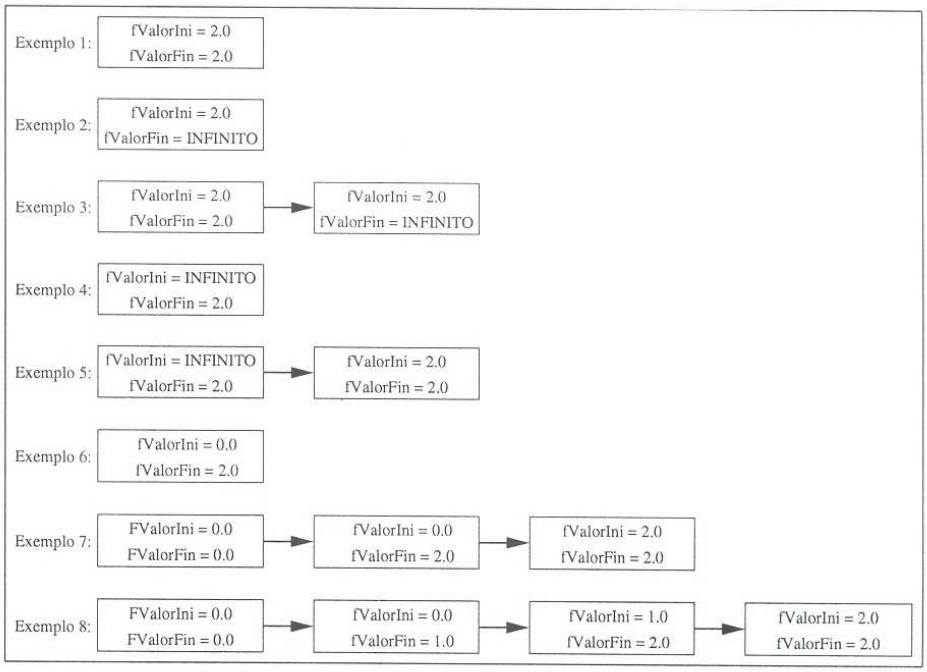

Figura 4.31: Representação das estruturas *LIST_VALUES* criadas para cada um dos exemplos apresentados na Tabela 4.2

Source File

Este arquivo é responsável pela implementação do núcleo reativo propriamente dito, ou seja, a implementação do processo de inferência, a ordem de execução dos módulos e as atualizações das variáveis de atraso. Tomando-se cada módulo em particular, o fluxo de execução é o seguinte:

1. Receber as mensagens do ambiente com os valores para as variáveis de entrada. Cada valor recebido é armazenado em um **hcubo**, chamado de *hipercubo de entrada*;
2. Receber a mensagem para execução do núcleo, iniciando assim o processo de inferência;
3. Interpretar os valores obtidos para as variáveis de saída, armazenando-os nas estruturas LIST_VALUES apropriadas;
4. Criar um novo hipercubo de entrada;
5. Voltar ao passo 1.

Este processo pode ser visualizado na Figura 4.34.

```
void CArks:: Nome-do-módulo_Output_Nome-da-variável()
{
   LIST_VALUES::iterator it,end;
   cout << "Resultados para a Variavel  Nome-da-variável no modulo
    Nome-do-módulo:\n";
   end = lNome-do-módulo_Nome-da-variável->end();
   for(it = lNome-do-módulo_Nome-da-variável->begin(); it != end; it++)
   {
      if((*it)->fValorIni == INFINITO && (*it)->fValorFin == INFINITO)
         cout << "\tqualquer valor possivel\n";
      else if((*it)->fValorIni == INFINITO)
         cout << "\tmenor que " << (*it)->fValorFin << "\n";
      else if((*it)->fValorFin == INFINITO)
         cout << "\tmaior que " << (*it)->fValorIni << "\n";
      else if((*it)->fValorIni == (*it)->fValorFin)
         cout << "\tigual a " << (*it)->fValorIni << "\n";
      else
         cout << "entre " << (*it)->fValorIni << " e " << (*it)->fValorFin
         << " excluindo os extremos\n";
   }
}
```

Figura 4.32: Modelo do método de saída de dados para o ambiente no sistema ARKS

```
void CArks::Nome-do-módulo_Output_Nome-da-variável(LIST_VALUES *Val)
{
   Nome-do-módulo-destino->Input_Nome-da-variável(Val);
}
```

Figura 4.33: Modelo do método de envio de dados entre módulos no sistema ARKS.

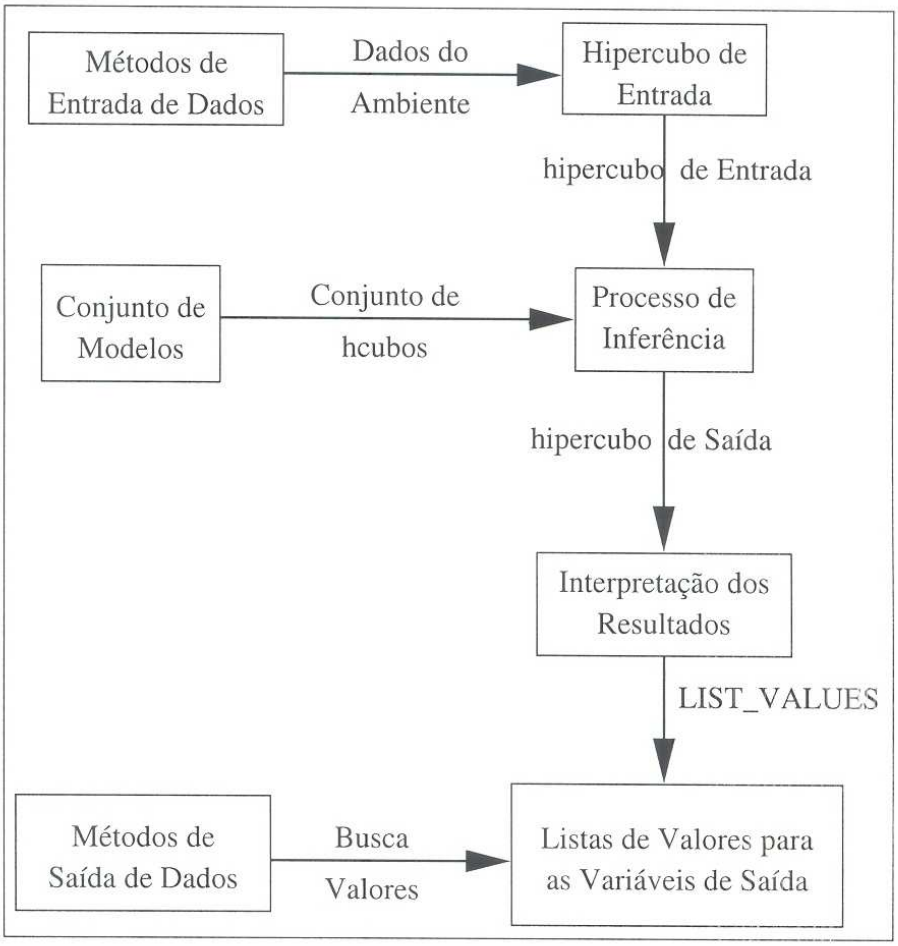

Figura 4.34: Visualização do fluxo de execução de um módulo.

Recepção das mensagens do ambiente O formato do método para interpretação dos valores recebidos do ambiente e a sua representação dentro do hipercubo de entrada é mostrado na Figura 4.35 para as variáveis que recebem seus dados do ambiente, e na Figura 4.36 para as variáveis que recebem seus dados a partir de outros módulos. Ambos os métodos procedem da mesma maneira: ao receber um valor do ambiente, setam o bit referente ao valor *ABSENT* como 0 e os bits correspondentes ao valor recebido como 1 dentro do hipercubo de entrada.

```
void CNome-do-módulo::Input_Nome-da-variável(float Val)
{
    seta o bit referente ao valor ABSENT como 0
   float Intervalos[] = {  lista de valores da variável}
   long x=0;
    avalia o valor recebido de acordo com os valores de Intervalos,
     setando os bits correspondentes no hipercubo de entrada
    atribui o valor recebido na variável auxiliar, se necessário
}
```

Figura 4.35: Modelo do método de entrada de dados a partir do ambiente

O processo de execução (inferência) de um módulo O procedimento de execução de um módulo é descrito na Figura 4.37. Basicamente, é a realização da operação de interseção entre o hipercubo de entrada e o conjunto de modelos do módulo, e a interpretação do resultado obtido para a geração das listas de valores para as variáveis de saída.

Avaliação de expressões e variáveis de atraso. Uma vez que não é possível garantir que, quando do envio de um valor para uma variável de entrada, os valores referentes às outras variáveis já estejam disponíveis, o processo de avaliação de expressões é feito no início do processo de execução. Caso neste instante o valor de uma variável utilizada em uma expressão ainda não estiver disponível (a variável está com o valor ABSENT), então a expressão é avaliada como falsa.

A avaliação das variáveis de atraso também é feita neste ponto por motivos de desempenho: qualquer operação entre uma variável e uma variável de atraso é considerada uma expressão e analisada neste método.

O modelo deste método pode ser visto na Figura 4.38.

```
void CNome-do-módulo::Input_Nome-da-variável(LIST_VALUES *Val)
{
   LIST_VALUES::iterator it,end;
   long x;

   end = Val->end();
   float Intervalos[] = { lista de valores da variável};

    Se Val não é vazio então
       seta o bit referente ao valor ABSENT como 0
    fim-se
    avalia o valores de Val de acordo com os valores de Intervalos,
     setando os bits correspondentes no hipercubo de entrada
    remove todos os elementos da lista auxiliar que armazena os valores
     recebidos através de Val, se esta variável possuir variáveis de atraso
    copia todos os elementos de Val para a lista auxiliar, se esta
     variável possuir variáveis de atraso
}
```

Figura 4.36: Modelo do método de entrada de dados a partir de outro módulo.

```
CNome-do-módulo::Executa()
{
    avalia expressões e variáveis de atraso
    realiza a interseção entre o hipercubo de entrada e o conjunto de
     hipercubos
    interpreta o resultado da interseção acima, criando as listas de
     valores para as variáveis de saída
    atualiza as variáveis de atraso
    reinicializa o hipercubo de entrada
}
```

Figura 4.37: Algoritmo para inferência dos dados de entrada dentro de um módulo.

```
void CNome-do-módulo::Atualiza_Variaveis_Expressao_Atraso()
{
   LIST_VALUES::iterator it,end;
   long x,y;
    para cada em variável de atraso faça
       seta o bit referente ao valor ABSENT como 0
      float Intervalos[] = { lista de valores da variável de atraso}
      long x=0;
       avalia o valor da variável de acordo com os valores de Intervalos,
        setando os bits correspondentes no hipercubo de entrada
    fim-para

    para cada expressão faça
       Se avaliação da expressão for verdadeira então
          sete o bit referente a esta expressão no hipercubo de entrada
   como 1
       senão
          sete o bit referente a esta expressão no hipercubo de entrada
   como 0
       fim-se
    fim-para
}
```

Figura 4.38: Modelo do método de avaliação de expressões e variáveis de atraso.

Interpretação dos resultados. Depois de realizada a operação de interseção entre o hipercubo de entrada e o conjunto de modelos do módulo, o resultado é interpretado pelo núcleo reativo para a geração das listas com os valores obtidos para as variáveis de saída. Este processo é realizado, a grosso modo, da seguinte maneira: para cada bit com o valor 1 dentro da variável de saída, é criado um elemento na estrutura LIST_VALUE com o valor referente a este bit (ver seção 4.7.1). Esta estrutura é então compactada e disponibilizada para consulta pelo ambiente. O modelo do método de interpretação dos resultados obtidos pela execução do módulo é apresentado na Figura 4.39; o modelo do método de compactação de uma estrutura LIST_VALUE pode ser visto na Figura 4.40.

```
void CNome-do-módulo::Output_Nome-da-variável()
{
    apaga os valores existentes na lista desta variável
   float Intervalos[] = {  lista de valores desta variável };
    para cada  bit da seção desta variável com o valor 1  faça
       adicione um elemento na lista desta variável com os valores
setados de acordo com o bit
    fim-para
    minimize a lista desta variável
    Se  esta variável envia seus valores para outros módulos
    então
       executa o método de entrada de dados para a variável nos módulos
      correspondentes
    fim-se
}
```

Figura 4.39: Modelo do método de interpretação do resultado para uma variável de saída.

```
void CModulo::Minimiza(LIST_VALUES *List)
{
    enquanto houverem agrupamentos a serem feitos faça
        para cada i ∈ List faça
            para cada j ∈ List faça
                Se i ≠ j então
                    Se i.fValorIni ≥ j.fValorIni e i.fValorFin ≤ j.fValorFin
                    então
                        remova i
                        retorne ao laço enquanto
                    fim-se
                    Se i.fValorIni < j.fValorIni e i.fValorFin ≤ j.fValorFin
                    então
                        j.fValorIni ← i.fValorIni
                        remova i
                        retorne ao laço enquanto
                    fim-se
                    Se i.fValorIni ≥ j.fValorIni e i.fValorFin > j.fValorFin
                    então
                        j.fValorFin ← i.fValorFin
                        remova i
                        retorne ao laço enquanto
                    fim-se
                fim-se
            fim-para
        fim-para
    fim-enquanto
}
```

Figura 4.40: Modelo do método de compactação da lista de valores de uma variável de saída.

Atualização das variáveis de atraso. Após a operação de interseção e a interpretação dos resultados, pode-se considerar que a interação dentro do módulo foi terminada; sendo necessário então a atualização dos valores de atraso.

Para tanto, o núcleo reativo procede da seguinte maneira: são definidas tantas variáveis de atraso quanto o instante máximo referenciado dentro da base de conhecimento. Por exemplo, se existe uma regra do tipo $V\$3 = 0.0$, então este módulo possuirá as variáveis V_1, V_2, e V_3 para o armazenamento dos valores atribuídos à variável V nos três instantes anteriores.

A cada atualização, faz-se $V_n \leftarrow V_{n-1}$ para todos os índices variando do instante máximo referenciado dentro da base de conhecimento até 2. Finalmente, é atribuído à variável V_1 o valor da variável V.

Assumindo-se que em um determinado módulo, o maior instante referenciado a uma variável X fosse 4, o código gerado para a atualização dos valores das variáveis de atraso para esta variável seria como o apresentado na Figura 4.41.

```
⋮
X$4 = X$3;
X$3 = X$2;
X$2 = X$1;
X$1 = X;
⋮
```

Figura 4.41: Exemplo do código gerado para atualização dos valores de uma variável de atraso.

Reinicialização do hipercubo de entrada. Ao final de uma interação dentro do módulo, é necessário reinicializar o hipercubo de entrada para permitir o recebimento dos valores do ambiente e de outros módulos na próxima interação. Isto é feito através dos seguintes passos:

- Setar todos os bits das seções do hipercubo de entrada referentes às variáveis de saída em 1;
- Setar todos os bits das seções do hipercubo de entrada referentes às variáveis de entrada em 0, com exceção do bit referente ao valor ABSENT;

- Setar o bit referente ao valor ABSENT de cada seção correspondente às variáveis de entrada dentro do hipercubo de entrada como 1.

4.7.2 Em Linguagem JAVA

Outra modalidade de geração de código disponível no ARKS é o uso da linguagem Java, incorporada como opção de geração de código no projeto devido ao uso crescente da linguagem.

Nesta etapa, são criados dois arquivos para a implementação do núcleo reativo:

- *Kernel file*, cujo nome é formado pelo nome do arquivo que contém a base de conhecimento descrita pelo usuário, seguido da extensão *.java*;
- *Interface file*, cujo nome é formado pela letra *C*, seguido do nome do arquivo que contém a base de conhecimento, seguido da palavra *Interface*, e com a extensão *.java*.

O apêndice H apresenta a listagem dos três arquivos gerados para o exemplo apresentado na Figura 4.26.

Kernel File

Responsável pela implementacão da classe principal do núcleo reativo, bem como das classes que implementam cada módulo descrito pelo usuário.

A definição da classe principal (*CArks*) é apresentada na Figura 4.42.

Toda interação entre o ambiente e os módulos do núcleo reativo é feita por uma instância de uma classe de interface, derivada da classe CArks e definida no *Interface File*, sendo, portanto, a classe CArks uma abstração responsável pelo encapsulamento dos métodos próprios para a execução do núcleo reativo, deixando para a classe de interface os métodos de comunicação com o ambiente, que podem ser modificados pelo usuário de acordo com a aplicação.

Assim como no código gerado em linguagem C++ ANSI, cada módulo é responsável pelo gerenciamento de seu conjunto de modelos, de suas variáveis utilizadas em expressões e de suas variáveis de atraso, bem como do processo de inferência propriamente dito. O modelo da implementação de uma classe que representa um módulo descrito pelo usuário é mostrado na Figura 4.43.

```
class CArks {
    referências para instâncias das classes que representam os módulos da
     base de conhecimento

    referências para as estruturas que receberão os valores a serem enviados
     para o ambiente após cada interação

   public CArks()
   {
       declaracao do construtor da classe, responsavel pela instanciação
dos módulos e das estruturas de valores
   }
   public void Executa()
   {
       invocação dos métodos de execução de cada módulo
   }
};
```

Figura 4.42: Modelo de definição da classe *CArks* em Java.

```
class Cnome-do-módulo extends CModulo {
    declaração das variáveis a utilizadas em expressões
    declaração das variáveis de atraso
    declaração das referências para as estruturas de dados que armazenarão
    os valores obtidos para as variáveis de saída
   public void Executa()
   {
        declaração do método de inferência do módulo
   }
    métodos de entrada de dados
    métodos de saída de dados
};
```

Figura 4.43: Modelo de implementação em Java das classes que representam os módulos descritos pelo usuário.

Tais módulos também necessitam de variáveis auxiliares para armazenar os valores enviados pelo ambiente e que são utilizados em expressões matemáticas, de forma a permitir a recuperação desse valor quando da necessidade de avaliação da expressão. Da mesma forma, é necessária a criação de variáveis auxiliares para armazenar os valores que foram enviados para determinadas variáveis em interações passadas, conforme a necessidade das variáveis de atraso descritas na base de conhecimento. O mecanismo de funcionamento das variáveis de atraso e das variáveis que são utilizadas em expressões é o mesmo descrito na seção 4.7.1, que trata da geração de código em linguagem C++ ANSI, obviamente observadas as diferenças entre as duas linguagens.

Interface File

Este arquivo é responsável pela interação entre o ambiente e o núcleo reativo. Por *default*, esta interação é feita via a entrada e a saída padrão do sistema. Porém, os métodos de entrada e saída de dados do núcleo reativo é implementada de forma a permitir uma fácil alteração para quaisquer dispositivos necessários à aplicação.

Métodos para a entrada de dados Cada variável de entrada declarada dentro de um módulo que receba seus dados do ambiente possui, neste arquivo, um método para a leitura e envio de dados ao núcleo reativo, conforme Figura 4.44.

Da mesma forma que os métodos de entrada de dados implementados em linguagem C++, este método pode ser alterado para permitir a leitura de valores de qualquer fonte de dados desejada, mantendo sempre a última linha de código declarada dentro do método, que envia esse valor lido para o núcleo reativo.

Métodos de saída de dados para o ambiente Os métodos de saída de dados para o ambiente em Java são criados da mesma forma que os métodos de saída de dados em C++ ANSI descritos na seção 4.7.1, guardadas as diferenças sintáticas entre as duas linguagens.

4.8 Exemplo de uma aplicação em ARKS

A seguir é apresentada a listagem da implementação pelo sistema ARKS do jogo da vida de Conway descrito na Seção 2.3.1.

Esta implementação utiliza como representação uma matriz de dimensões definidas pelas constantes *LARGURA* e *ALTURA*, sendo assumido que o mundo possui um formato

```
public void Nome-do-módulo_Input_Nome-da-variável()
{
    StreamTokenizer in = new StreamTokenizer(System.in);

    do
    {
        System.out.print("Entre com o Valor da Variavel  Nome-da-variável
   no Modulo  Nome-do-módulo: ");
        try
        {
           in.nextToken();
        }
        catch(Exception e)
        {
        }
    }while(in.ttype != StreamTokenizer.TT_NUMBER);
     Nome-do-módulo.Input_ Nome-da-variável(in.nval);
}
```

Figura 4.44: Modelo do método de entrada de dados em Java do sistema ARKS

toróide, ou seja, considera-se que a última e a primeira linha da matriz são adjacentes, assim como a primeira e a última coluna.

A listagem da base de conhecimentos para este exemplo na linguagem do sistema ARKS é mostrada na Figura 4.45

O controle da matriz é feito no código de interface, que a cada geração envia os dados de cada célula para o núcleo reativo, e armazena a resposta em uma matriz auxiliar. Após o final da geração, a matriz auxiliar é copiada para a matriz principal, indicando uma nova geração de organismos. O código do método que realiza este tratamento é listado na Figura 4.46.

As modificações necessárias para a integração do núcleo reativo com o código de interface do ambiente foram realizadas somente nos métodos de entrada e saída de dados, conforme discutido anteriormente. O código do núcleo reativo propriamente dito não necessitou de alteração, uma vez que foi projetado para ser independente do ambiente com o qual está interagindo.

```
MODULE Calc
{
  INPUTS N, S, L, O, NE, NO, SE, SO;
  OUTPUTS Vizinhos;

  RULE Calcula_Vizinhos Vizinhos = N+S+L+O+NE+NO+SE+SO;
};

MODULE Sobrevivencia
{
   DEPENDENCY CONJUNCTION;

   INPUTS Estado, Vizinhos;
   OUTPUTS Vive;

   RULE R1 Estado = 0.0 AND Vizinhos = 3.0 IMPLIES Vive = 1.0;
   RULE R2 Estado = 0.0 AND Vizinhos <> 3.0 IMPLIES Vive = 0.0;
   RULE R3 Estado = 1.0 AND (Vizinhos = 2.0 OR Vizinhos = 3.0) IMPLIES
           Vive = 1.0;
   RULE R4 Estado = 1.0 AND (Vizinhos < 2.0 OR Vizinhos > 3.0) IMPLIES
           Vive = 0.0;
}
```

Figura 4.45: Listagem ARKS para o Conway's Game of Life

```
int aux[LARGURA][ALTURA];

for(int x=0; x<LARGURA; x++)
  for(int y=0; y<ALTURA; y++)
  {
     pDoc->pArks->Calc_Input_N(pDoc->matriz[x][(ALTURA-1+y)%ALTURA]);
     pDoc->pArks->Calc_Input_S(pDoc->matriz[x][(y+1)%ALTURA]);
     pDoc->pArks->Calc_Input_L(pDoc->matriz[(x+1)%LARGURA][y]);
     pDoc->pArks->Calc_Input_O(pDoc->matriz[(LARGURA-1+x)%LARGURA][y]);
     pDoc->pArks->Calc_Input_NE(
                  pDoc->matriz[(x+1)%LARGURA][(ALTURA-1+y)%ALTURA]);
     pDoc->pArks->Calc_Input_NO(
                  pDoc->matriz[(LARGURA-1+x)%LARGURA][(ALTURA-1+y)%ALTURA]);
     pDoc->pArks->Calc_Input_SE(
                  pDoc->matriz[(x+1)%LARGURA][(y+1)%ALTURA]);
     pDoc->pArks->Calc_Input_SO(
            pDoc->matriz[(LARGURA-1+x)%LARGURA][(y+1)%ALTURA]);
     pDoc->pArks->Live_Input_Estado(pDoc->matriz[x][y]);
     pDoc->pArks->Executa();
     aux[x][y]=pDoc->pArks->Live_Output_Vive();
  }
for(x=0; x<LARGURA; x++)
  for(int y=0; y<ALTURA; y++)
    pDoc->matriz[x][y]=aux[x][y];
```

Figura 4.46: Código de interface para interação com o núcleo reativo gerado.

4.9 Conclusão

Neste capítulo foram descritas as características de funcionamento e implementação do sistema ARKS.

O paradigma utilizado para o desenvolvimento de sistemas reativos, através da descrição do comportamento do núcleo pela base de conhecimento e linkedição do código gerado pelo sistema ARKS com o código de interface com o ambiente permite uma separação clara das características de projeto do sistema (seu comportamento) das características de implementação (no caso, as rotinas de interface com o ambiente, já que o núcleo reativo com o comportamento descrito será gerado automaticamente).

A implementação de geradores de código em várias linguagens permite uma conversão mais simples de uma linguagem para outra, o que possibilita o teste do núcleo reativo gerado em uma linguagem mais adequada à prototipação, e a implementação final em uma linguagem mais adequada às necessidades do projeto.

Para as linguagens disponíveis atualmente (C++ e Java), ambas produzem um núcleo reativo com o código padrão (ANSI com STL no caso do C++ e JDK 1.0.2 no caso do Java), o que permite uma migração mais fácil entre plataformas.

A geração de código em um arquivo separado permite a alteração na base de conhecimento sem perda das alterações efetuadas nos métodos de interface com o ambiente.

De acordo com o formalismo adotado, pode-se afirmar que, apesar do processo de compilação da base de conhecimento possuir uma complexidade exponencial, o código gerado possui uma complexidade constante, permitindo uma fácil verificação de atendimento à restrições temporais e, consequentemente, habilitando o sistema ARKS a gerar núcleos que possam ser utilizados em aplicações tempo-real.

Capítulo 5

Considerações Finais

O sistema ARKS se propõe a ser uma ferramenta eficaz para a construção de núcleos reativos a partir de bases de conhecimentos, conciliando dois aspectos conflitantes neste tipo de sistema: o não-determinismo que caracteriza os sistemas de Inteligência Artificial, devido à natureza dos algoritmos que tratam com a base de conhecimento, e o determinismo necessário às aplicações reativas.

Sobre as extensões realizadas a partir do sistema RETIKS, é possível afirmar que as principais contribuições deste trabalho são:

- A criação do modo de dependência disjuntivo entre as regras facilitou a descrição do conhecimento por parte do usuário, uma vez que este não precisa mais definí-lo na forma de uma teoria;
- A alteração da semântica do operador $IF \ldots THEN \ldots$ tornou mais intuitiva a descrição do conhecimento;
- A incorporação de expressões matemáticas e do valor $ABSENT$ dentro das regras aumentou o poder de descrição do conhecimento das mesmas;

Quanto ao código gerado pelo sistema, como são representados os modelos obtidos da base de conhecimento e não as regras propriamente ditas, pode-se supor que:

1. Uma vez que esses modelos são representados em mapas de bits, o tamanho do código executável final deverá ser bem menor do que o obtido com implementação específica;
2. Como o processo de inferência possui uma complexidade constante, se resumindo a união e interseção de conjuntos representados por mapas de bits; e como as operações

sobre bits são as de mais alta velocidade computacional, é possível assumir que o código gerado pelo sistema ARKS é tão veloz quanto um código obtido com implementação específica.

Uma possibilidade permitida pelo formalismo utilizado no sistema é a verificação da consistência da base de conhecimento, através da verificação do conjunto de modelos de cada um dos módulos, bem como a verificação da existência de ciclos no fluxo de execução dos mesmos, o que garante, de certa forma, o bom funcionamento do núcleo reativo gerado.

5.1 Características do sistema

A linguagem ARKS fornece ao especialista — responsável pela descrição do comportamento do núcleo reativo — uma forma mais simples de descrever o comportamento do seu sistema do que seria possível com a linguagem C++, Java ou a maioria das outras linguagens procedurais, uma vez que a linguagem ARKS abstrai o conhecimento necessário para a implementação dos mecanismos de inferência e exige do desenvolvedor apenas o conhecimento sobre a aplicação propriamente dita, que deve ser descrito em uma linguagem muito mais simples e projetada exclusivamente para este fim.

Obviamente, algum conhecimento da linguagem C++ ou Java é necessário para a adequação do núcleo reativo gerado ao ambiente desejado (adequação dos métodos de interface entre o ambiente e o sistema reativo); porém, tal conhecimento é muito mais restrito do que o necessário para a implementação do sistema como um todo.

Outro ponto importante é que o sistema ARKS foi totalmente descrito na linguagem C++ ANSI, com utilização da biblioteca STL (*Standard Template Library*), o que permite a sua portabilidade para qualquer plataforma, através da recompilação de seu código-fonte, sem nenhuma alteração; de fato, o sistema já foi compilado e está executando em plataforma Intel, sobre os sistemas operacionais Windows NT e Linux, onde apresenta desempenhos semelhantes quando se compara o tempo gasto na compilação de uma base de conhecimentos e na qualidade do código gerado.

5.2 Características do núcleo gerado

O núcleo reativo gerado pelo sistema ARKS também se mostrou portável a qualquer plataforma que possua um compilador C++ ANSI com suporte à STL ou uma JVM (*Java Virtual Machine*), sem que seja necessária alguma alteração em seu código.

A geração dos métodos de interface em arquivos separados do código do núcleo reativo facilita a adequação destes de acordo com as necessidades da aplicação, sem necessidade de alteração do núcleo reativo. Além disso, ainda é possível a recompilação da base de conhecimento, sem necessariamente recriar o arquivo de interface, preservando assim as alterações que o usuário já tenha feito.

Um ponto interessante do formalismo adotado é que cada regra representa uma restrição ao conjunto de modelos de uma teoria. Portanto, quanto mais regras forem declaradas dentro da base de conhecimento (utilizando o modo de dependência conjuntiva entre regras), menor será o conjunto de modelos resultante e, consequentemente, menor e mais rápido será o código gerado para o núcleo reativo, ao contrário do que seria esperado de um núcleo gerado com implementação específica.

5.3 Perspectivas

Considera-se que as perspectivas sobre este trabalho possam ser divididas em duas modalidades: adoção e pesquisa de outras técnicas para a manipulação e verificação do conhecimento descrito pelo usuário; e adoção e pesquisa de outras técnicas para a geração do núcleo reativo.

5.3.1 Sugestões para a manipulação e verificação do conhecimento descrito pelo usuário

- Incorporação de uma ferramenta para a verificação de propriedades da base de conhecimento;
- Criação de uma interface gráfica para a programação icônica da base de conhecimento, com o objetivo de facilitar a descrição do mesmo pelo usuário;
- Incorporação de ferramentas para a verificação de restrições temporais, de forma a permitir a geração de núcleos para aplicações tempo-real;
- Inclusão de uma lógica anotada, de forma a capacitar o sistema a lidar com informações inconsistentes [Kae93];
- Adequação do sistema para a geração de sistemas reativos multi-agentes (*proposta MARKS — Multi-agent Reactive Knowledge-based System*).

5.3.2 Sugestões para a geração do núcleo reativo

- Geração do núcleo reativo em outras linguagens de programação (Java/RTR, SpringC, etc.);
- Criação de uma ferramenta para facilitar a adequação do núcleo reativo ao ambiente (automação do processo de alteração dos métodos de interface do núcleo reativo);
- Verificação do tipo de sinal enviado por uma variável de saída: atualmente, toda variável de saída possui uma estrutura de lista de pares (LIST_VALUES) para armazenar seus resultados, já que para uma variável de saída é possível resultados na forma de segmentos sobre a reta dos reais (intervalos); porém, se a base de conhecimentos for descrita de forma que uma determinada variável de saída só possua resultados pontuais, não faz sentido a utilização desta estrutura para o armazenamento desse resultado, uma vez que ele poderia ser armazenado em uma variável simples, cujo acesso é muito mais rápido. Por isso, é interessante a elaboração de um processo de inferência sobre a base de conhecimento descrita pelo usuário, com o objetivo de descobrir se uma determinada variável de saída pode ou não ter somente resultados pontuais, sendo gerado um determinado tipo de método de interface para um e para outro caso;
- Adequação do código gerado para a implementação de núcleos Tempo-Real, com a adição das primitivas necessárias.

Estas são apenas algumas considerações finais sobre este trabalho de pesquisa e desenvolvimento, os atributos apresentados serão efetivamente validados a partir do momento em que forem efetuados testes em campo, sob condições reais de aplicação dos núcleos reativos gerados pelo sistema. Nestas circunstâncias, novas perspectivas práticas e conceituais irão surgir, além das já mencionadas, abrindo espaço para novas implementações e pesquisas.

Apêndice A

Descrição sintática preliminar da linguagem ARKS

A linguagem utilizada no sistema ARKS foi descrita inicialmente em [Kae93], tendo sofrido algumas alterações para adequação a este projeto. Esta linguagem já modificada é apresentada a seguir num formato BNF.

Os símbolos não terminais serão prefixados por < e sufixados por >. São utilizados os meta-símbolos ::⇒ (é definido por), * (repetição zero ou mais vezes da expressão quantificada), + (repetição uma ou mais vezes da expressão quantificada), | (alternância "ou"), e { e } (que cercam elementos opcionais e servem para o agrupamento de ítens). Alguns outros elementos são considerados como terminais, tais como **numero**, **numero-inteiro**, **cadeia-de-caracteres**, etc. Literais são escritos em negrito.

A linguagem permite comentários, sendo considerado como comentário, e portanto ignorado pelo compilador, qualquer combinação de símbolos entre "//" e o final da linha.

⟨programa⟩::⇒ ⟨modulo⟩+.

⟨modulo⟩::⇒ "MODULO" ⟨nome-do-modulo⟩ "{" ⟨declaracoes⟩ ⟨entradas⟩ ⟨saidas⟩ {⟨inicializacoes⟩} ⟨corpo-do-modulo⟩ "}".

⟨nome-do-modulo⟩::⇒ "cadeia-de-caracteres".

⟨declaracoes⟩::⇒ {dependencia} {dimensionamento}.

⟨dependencia⟩::⇒ "DEPENDENCY" {"DISJUNCTION" | "CONJUNCTION"} ";".

⟨dimensionamento⟩::⇒ "DIMENSION" ⟨nome-do-atributo⟩ "[" ⟨indice⟩ "]".

⟨nome-do-atributo⟩::⇒ "cadeia-de-caracteres".

⟨indice⟩::⇒ "numero-inteiro".

⟨entradas⟩::⇒ "INPUTS" ⟨atributos-de-entrada⟩ ";".

⟨atributos-de-entrada⟩::⇒ ⟨declaracao-atributos⟩ {"," ⟨declaracao-atributos⟩}*.

⟨saidas⟩::⇒ "OUTPUTS" ⟨atributos-de-saida⟩ ";".

⟨atributos-de-saida⟩::⇒ ⟨declaracao-atributos⟩ {"," ⟨declaracao-atributos⟩}*.

⟨declaracao-atributos⟩::⇒ ⟨nome-do-atributo⟩ |
⟨nome-do-atributo⟩ "[" ⟨limite-inferior⟩ ".." ⟨limite-superior⟩ "]" |
⟨nome-do-atributo⟩ "[" ⟨tamanho⟩ "]".

⟨limite-inferior⟩::⇒ "numero-inteiro".

⟨limite-superior⟩::⇒ "numero-inteiro".

⟨tamanho⟩::⇒ "numero-inteiro".

⟨inicializacoes⟩::⇒ "INITIAL" ⟨inicializacoes-de-atributo⟩ ";".

⟨inicializacao-atributo⟩::⇒ ⟨atributo⟩ "=" "numero" {"," ⟨atributo⟩ "=" "numero"}*.

⟨atributo⟩::⇒ ⟨nome-do-atributo⟩ "$" "numero-inteiro" |
⟨nome-do-atributo⟩ "[" "numero-inteiro" "]" "$" "numero-inteiro" |
⟨nome-do-atributo⟩ "[" "numero-inteiro" "]" |
⟨nome-do-atributo⟩.

⟨corpo-do-modulo⟩::⇒ {⟨regra⟩}+.

⟨regra⟩::⇒ "RULE" ⟨nome-da-regra⟩ ⟨formula⟩ ";" |
"TEMPLATE" ⟨nome-do-template⟩ "[" ⟨limite-inferior⟩ ".." ⟨limite-superior⟩
"]" ⟨formula⟩ ";".

⟨nome-da-regra⟩::⇒ "cadeia-de-caracteres".

⟨nome-do-template⟩::⇒ "cadeia-de-caracteres".

⟨formula⟩::⇒ "(" ⟨formula⟩ ")" |
"IF" ⟨formula⟩ "THEN" ⟨formula⟩ |
"IF" ⟨formula⟩ "THEN" ⟨formula⟩ "ELSE" ⟨formula⟩ |
"IF" ⟨formula⟩ "ONLYIF" ⟨formula⟩ |
⟨formula⟩ "IMPLIES" ⟨formula⟩ |
⟨formula⟩ "AND" ⟨formula⟩ |
⟨formula⟩ "OR" ⟨formula⟩ |
"NOT" ⟨formula⟩ |
⟨formula-atomica⟩.

⟨formula-atomica⟩::⇒ ⟨atributo⟩ ⟨operador-relacional⟩ ⟨expressao⟩ |
⟨atributo⟩ "=" "ABSENT".

⟨operador-relacional⟩::⇒ "="|
"<>"|
"<"|
">"|

"<="|
">=".

⟨expressao⟩::⇒ "(" ⟨expressao⟩ ")" |
⟨expressao⟩ ⟨operador-aritmetico⟩ ⟨expressao⟩ |
⟨atributo⟩ |
"numero".

⟨operador-aritmetico⟩::⇒ "+"|
"-"|
"*"|
"/".

Apêndice B

Algoritmos para operar sobre hipercubos e conjuntos de hipercubos

Os algoritmos para a operação sobre hipercubos são baseados naqueles apresentados em [Kae93]. Contudo, alguns algoritmos foram modificados para atender às particularidades desta implementação. Estes novos algoritmos estão listados nesta seção.

Utilizar-se-á a notação $h_1, h_2, \ldots$ para os **hcubo** (hipercubos) e $H_1, H_2, \ldots$ para os **Hcuboset** (conjuntos de hipercubos). Cada variável declarada dentro do módulo possui sua discretização representada em cada hipercubo, sendo esta discretização aqui chamada de *seção*. Todas as seções estão mapeadas sobre um mesmo conjunto de índices J. A i-ésima seção do **hcubo** h será denotada $h[i]$ e os índices serão denotados por $i, j, \ldots$. Os operadores **&** e **|** são usados para representar as operações de E-binário e OU-binário entre seções, respectivamente.

As operações definidas e os algoritmos correspondentes são:

- $h_1 \cap h_2$

hcubo Interseção(**hcubo** h_1, **hcubo** h_2)
início
  variáveis: hcubo h_3;
  $h_3 := \emptyset$;
  para todo $i \in I$ **faça**
    $h_3[i] := h_1[i] \& h_2[i]$;
  fim-para
  se $(\forall i \in I | h_3[i] \neq \emptyset)$ **então**
    retornar h_3;

senão
retornar $\emptyset$;
fim-se
fim

- $h_1 \cap H_2$

Hcuboset Interseção(hcubo h_1, **Hcuboset** H_2)
início
variáveis: Hcuboset H;
hcubo h;
$H := \emptyset$;
para todo $h_2 \in H_2$ **faça**
$h := h_1 \cap h_2$;
se $h \neq \emptyset$ **então**
$H := H \cup h$;
fim-se
fim-para
retornar H **fim**

- $h_1 - h_2$

Hcuboset Diferença(hcubo h_1, **hcubo** h_2)
início
variáveis: Hcuboset H_1, H_2;
$H_1 := \bar{h_2}$;
$H_2 := h_1 \cap H_1$;
retornar H_2;
fim

- $\bar{h_1}$

Hcuboset Complemento(hcubo h)
início

variáveis: **Hcuboset** H;
hcubo h_1;
para todo $i \in I$ **faça**
para todo $j \in I$**faça**
se $i = j$ **entao**
$h_1[j] := h[j]$;
senão
$h_1[j] := \emptyset$;
fim-se
fim-para
$H := H \cup h_1$;
fim-para
retornar H;
fim

- $H \cup h$

  A união de um **Hcuboset** H e um **hcubo** h é definida como a inclusão de h em H.

- $h_1 - H_2$

Hcuboset Diferença(hcubo h_1, **Hcuboset** H_2)
início
variáveis: **Hcuboset** H_3;
$H_3 := \emptyset$;
para todo $h_2 \in H_2$ **faça**
se $H_3 = \emptyset$ **então**
$H_3 := h_1 - h_2$;
senão
$H_3 := H_3 - h_2$;
fim-se
fim-para
retornar H_3;
fim

- $H_1 - h_2$

Hcuboset Diferença(Hcuboset H_1**, hcubo** h_2**)**
início
  variáveis: Hcuboset H;
  $H := \emptyset$;
  para todo $h_1 \in H_1$ **faça**
    $H := H \cup (h_1 - h_2)$;
  fim-para
  retornar H;
fim

- $H_1 \cap H_2$

Hcuboset Interseção(Hcuboset H_1**, Hcuboset** H_2**)**
início
  variáveis: Hcuboset H;
  $H := \emptyset$;
  para todo $h_1 \in H_1$ **faça**
    para todo $h_2 \in H_2$ **faça**
      $H := H \cup (h_1 \cap h_2)$;
    fim-para
  fim-para
  retornar H;
fim

- $H_1 \cup H_2$

Hcuboset União(Hcuboset H_1**, Hcuboset** H_2**)**
início
  variáveis: Hcuboset H;
  $H := \emptyset$;
  para todo $h_1 \in H_1$ **faça**
    $H := H \cup h_1$;
  fim-para

para todo $h_2 \in H_2$ **faça**
$H := H \cup h_2$;
fim-para
retornar H;
fim

- $H_1 - H_2$

Hcuboset Diferença(HcubosetH_1, **Hcuboset**H_2)
início
variáveis: Hcuboset H_3, H_4;
$H_3 := \emptyset$;
para todo $h_1 \in H_1$ **faça**
$H_4 := h_1 - H_2$;
$H_3 := H_3 \cup H_4$;
fim-para
retornar H_3;
fim

- $\bar{H_1}$

Hcuboset Complemento(Hcuboset H)
início
variáveis: Hcuboset H_1, H_2, H_3;
$H_1 := \emptyset$
para todo $h \in H$ **faça**
$H_2 := \bar{h}$;
$H_3 := H_2 - H$;
$H_1 := H_1 \cup H_3$;
fim-para
retornar H_1;
fim

Apêndice C

Algoritmos para Obtenção dos Modelos de uma Fórmula

Aqui estão apresentados os algoritmos utilizados para a geração do conjunto de modelos de uma fórmula. Todos os algoritmos aqui descritos tem como valor de retorno uma referência a um **Hcuboset**

Utilizar-se-á a notação **retorne**(*variável*) para indicar qual variável possui a referência que será utilizada como valor de retorno do procedimento; também será utilizada a notação **nome-do-procedimento**($parametro_1$, $parametro_2$, ...) para indicar os nomes das variáveis **Hcuboset** que são passadas ao procedimento via parâmetros; por último, é assumida a existência das variáveis $tmp_1, tmp_2, \ldots, tmp_n$ que serão utilizadas para armazenar os conjuntos de hipercubos intermediários produzidos pelos procedimentos.

- **procedimento-interseção**(P, Q)

  início
  $tmp_1 := P \cap Q$;
  $tmp_2 :=$ procedimento-minimização(tmp_1);
  retorne tmp_2;
  fim

- **procedimento-união**(P, Q)

  início
  $tmp_1 := P \cup Q$;

$tmp_2 :=$ procedimento-minimização(tmp_2);
retorne tmp_2;
fim

- **procedimento-complemento**(P)

início
$tmp_1 := \bar{P}$;
retorne tmp_1;
fim

- **procedimento-implicação**(P, Q)

início
$tmp_1 := \bar{P}$;
$tmp_2 := tmp_1 \cup Q$;
$tmp_3 :=$ procedimento-minimização(tmp_2);
retorne tmp_3;
fim

- **procedimento-condicional-conjuntiva**(P, Q)

início
$tmp_1 := P \cap Q$;
$tmp_2 := \bar{P}$;
$tmp_3 := \bar{Q}$;
$tmp_4 := tmp_2 \cap tmp_3$;
$tmp_5 := tmp_1 \cap tmp_4$;
$tmp_6 :=$ procedimento-minimização(tmp_5);
retorne tmp_6;
fim

- **procedimento-condicional-disjuntiva**(P, Q)

  início
  $tmp_1 := P \cap Q$;
  $tmp_2 :=$ procedimento-minimização(tmp_1);
  retorne tmp_2;
  fim

- **procedimento-bicondicional**(P, Q)

  início
  $tmp_1 :=$ procedimento-implicação(P,Q);
  $tmp_2 :=$ procedimento-implicação(Q,P);
  $tmp_3 := tmp_1 \cap tmp_2$;
  $tmp_4 :=$ procedimento-minimização(tmp_3);
  retorne tmp_4; **fim**

Apêndice D

Descrição dos demais algoritmos utilizados no sistema ARKS

D.1 Algoritmo de minimização de Hcuboset

Este algoritmo é utilizado para otimizar o número de **hcubo** dentro do **Hcuboset** que representa os modelos do módulo descrito pelo usuário, com o objetivo de diminuir a quantidade de memória necessária para a representação dos modelos dentro do código executável final.

Será utilizada a notação h_n para indicar um **Hcubo**; i para indicar os índices dentro de um **hcubo**, e $h_n[i]$ para indicar uma seção (porção do **hcubo** referente a uma variável).

É assumida a existência dos operadores & (E-binário) e | (OU-binário) entre dois hcubo.

procedimento-minimização(H)
início
  faça
    laço = **falso**
    para todo h_1 pertencente a H **faça**
      para todo h_2 pertencente a H **faça**
        Se $h_1 \neq h_2$ **então**
          Se exite apenas um i tal que $h_1[i] \neq h_2[i]$ **então**
            $h = h_1 \mid h_2$
            retire h_1 de H
            retire h_2 de H

```
            inclua h em H
            laço = verdadeiro
          fim-se
        fim-se
      fim-para
    fim-para
  enquanto laço = verdadeiro
  para todo h1 pertencente a H faça
    para todo h2 pertencente a H faça
      Se h1 ≠ h2 então
        Se h1&h2 = h1 então
          remove h1 de H
        fim-se
      fim-se
    fim-para
  fim-para
```

O primeiro passo deste algoritmo é o agrupamento de **hcubo**: sejam h_1 e h_2 estruturas do tipo **hcubo**, o agrupamento de h_1 e h_2 gera uma estrutura h do tipo **hcubo** que contém todos os modelos obtidos pela união de h_1 e h_2. Logo, é possível a substituição de h_1 e h_2 por h, diminuindo assim em uma unidade o número de **hcubo** dentro do **Hcuboset**.

Um agrupamento entre h_1 e h_2 se, e somente se, todas as seções (representação das variáveis do módulo dentro do **hcubo**) de h_1 e h_2 forem iguais, com exceção de uma. O agrupamento é realizado através da operação de OU-binário entre os bits de h_1 e h_2, originando h. Por exemplo, sejam os seguintes **hcubo** h_1 e h_2:

$$h_1 = a\{2\}\ b\{2\ 3\}\ c\{1\ 2\}$$
$$h_2 = a\{3\}\ b\{2\ 3\}\ c\{1\ 2\}$$

O hipercubo h resultante do agrupamento de h_1 e h_2 é dado por:

$$h = a\{2\ 3\}\ b\{2\ 3\}\ c\{1\ 2\}$$

Esta operação pode ser visualizada na Figura D.1.

Uma vez realizado todos os agrupamentos possíveis, o algoritmo entra na etapa de eliminação de redundâncias. Um **hcubo** é redundante se a diferença entre ele e os demais

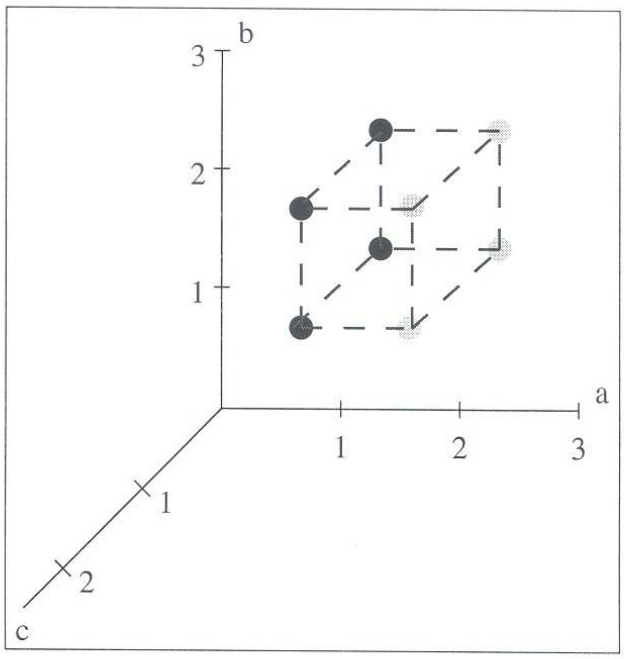

Figura D.1: Visualização do agrupamento de variáveis. h_1 é representado pelos pontos escuros; h_2 pelos pontos claros. h é representado por todos os pontos, agrupados pelas linhas tracejadas.

hcubo pertencentes ao **Hcuboset** for redundante. Na prática, observou-se que a implementação desta definição para a eliminação de um **hcubo** cria um número muito grande de **Hcuboset** intermediários para uma base de regras pequena, o que tornaria o algoritmo inviável devido ao grande custo computacional e à grande quantidade de memória exigida.

Para resolver este problema foi implementada a seguinte definição de **hcubo** redundante: Um **hcubo** h é redundante se a interseção entre h e algum outro **hcubo** pertencente ao **Hcuboset** for igual a h. Apesar de não ser mais possível a garantia da solução ótima para o problema da minimização de um **Hcuboset**, este algoritmo produz uma solução satisfatória para o problema.

D.2 Algoritmo de verificação de ciclos entre os módulos

O algoritmo para verificação de ciclos utilizado no sistema ARKS é baseado no algoritmo *Depth-first search* apresentado em [Riv90], pags 477-479.

Para a descrição do algoritmo, é assumido a existência das seguintes estruturas:

- *Lista_Inicial*, que contém uma referência a todos os módulos que recebem seus dados de entrada somente do ambiente;

- *Lista*, pertencente a cada módulo, que contém uma referência a todos os módulos que possuam pelo menos uma variável de entrada que receba seus valores de uma variável de saída deste;

- *Flags*, que possui n elementos booleanos, onde n é o número de módulos existentes na base de conhecimento, utilizados para a detecção do ciclo. Cada módulo possui uma variável *pos*, que possui como valor um índice único dentro de *Flags*.

procedimento verifica-ciclo
  para cada elemento $M \in Lista\_Inicial$ **faça**
    para cada elemento $i \in Flags$ **faça**
      $Flags[i] \leftarrow falso$
    fim-para
    ciclo(M)
  fim-para
fim

procedimento ciclo(M)
  Se $Flags[M.pos] = verdadeiro$ **então**
    acusa ciclo
  Senão
    $Flags[M.pos] \leftarrow verdadeiro$
    para cada elemento $N \in M.Lista$ **faça**
      ciclo(N)
    fim-para
  fim-se
fim

Apêndice E

Execução do compilador Arks

A sintaxe para execução do compilador Arks é dada por:
arks *nome-do-arquivo opções*, onde:

nome-do-arquivo : nome do arquivo, em formato ASCII, que contém a listagem do núcleo reativo em linguagem Arks;

opções : opções de execução do compilador, sendo possível qualquer combinações entre as seguintes:

-p : imprime informações sobre cada módulo do núcleo reativo, mostrando informações sobre as variáveis, as fórmulas descritas e os conjuntos de hipercubos gerados;

-nointerface : não gera o arquivo de interface que contém os métodos de interação do núcleo reativo com o ambiente. Esta opção permite que o núcleo reativo seja regerado, sem perda das alterações feitas nos métodos de interface do núcleo reativo;

-java : gera o código do núcleo reativo em linguagem Java. Por *default*, o sistema Arks gera o núcleo reativo na linguagem C++ ANSI.

Apêndice F

Descrição dos Erros Tratados pelo Compilador Arks

Basicamente, existem dois tipos de erros possíveis dentro de uma base de conhecimento escrita em Arks: os erros fatais e os erros não-fatais.

F.1 Erros fatais

Erros fatais são erros que não permitem a continuação do processo de compilação da base de conhecimentos, quando então o mesmo é abortado, sendo enviada uma mensagem para a saída padrão indicando o erro e a linha onde este ocorreu. A seguir é apresentada uma descrição de cada um dos erros fatais possíveis dentro do sistema Arks.

- **Variável *nome-da-variável* necessita de índice**: uma variável criada com o comando *DIMENSION* foi utilizada dentro de uma regra sem um índice. Este erro ocorre geralmente quando se modifica um construção *TEMPLATE*, cujas variáveis dimensionadas não necessitam de índices, para uma construção *RULE*.

```
MODULE foo
{
    DIMENSION A[3];

    :
    RULE R1 IF A = 1.0 THEN B = 1.0;
Erro: Variável A necessita de índice
    RULE R2 IF A[0] = 1.0 THEN B = 1.0;
Ok
}
```

- **Variável *nome-da-variável* já foi definida**: Houve a tentativa de criação de duas variáveis do mesmo nome dentro de um módulo.

```
MODULE foo
{
     INPUTS A;
     OUTPUTS A;
Erro: Variável A já foi definida como variável de entrada
}
```

- **Variável de atraso não pode fazer referência a instante menor do que 1**: Por definição, uma variável de instante 0 é a própria variável, de modo que não é permitido a criação de variáveis de atraso com instantes menores que 1.

```
MODULE foo
{
    INPUTS A;
    :
    RULE R1 IF A$0 = 1.0 THEN B = 1.0;
Erro
    RULE R2 IF A = 1.0 THEN B = 1.0;
Ok
}
```

- **Não Existe a Variável *nome-da-variável* para o instante 0**: Houve a tentativa de referência a uma variável de atraso, cuja variável do instante atual ainda não foi criada. O sistema Arks permite variáveis de atraso somente para variáveis de entrada e saída, de forma que as mesmas devem ser declaradas no início do módulo.

```
MODULE foo
{
    INPUTS A;
    :
    RULE R1 IF B$1 = 1.0 THEN C = 1.0;
Erro: ainda não foi criada a
    variável B
    RULE R2 IF A$1 = 1.0 THEN C  =1.0;
Ok
  }
```

- **Variável *nome-da-variável* não é dimensionada**: Houve referência a uma variável dimensionada que não foi declarada dentro da construção *DIMENSION*.

```
MODULE foo
{
   DIMENSION A[3];
   :
   RULE R1 IF B[1] = 1.0 THEN C = 1.0;
Erro variável B não foi declarada
   como uma variável dimensionada
   RULE R2 IF A[1] = 1.0 THEN C = 1.0;
Ok
```

- **Indice *valor-do-índice* não é válido para a variável *nome-da-variável***: Houve tentativa de referência a uma posição inválida de um dimensionamento.

```
MODULE foo
{
   DIMENSION A[2..4];
   :
   RULE R1 IF A[1] = 1.0 THEN B = 1.0;
Erro: dimensionamento A possui
   posições enumeradas de 2 a 4;
   RULE R2 IF A[4] = 1.0 THEN B = 1.0;
Ok
  }
```

- **O módulo *nome-do-módulo* é contraditório**: A conjunção dos modelos das fórmulas do módulo é vazia, o que implica que algumas regras dentro do módulo são contraditórias.

```
MODULE foo
{
   ⋮
   IF A = 1.0 THEN B = 1.0;
   IF A = 1.0 THEN B = 2.0;
}
```

- **Variável *nome-da-variável* não foi definida**: Houve tentativa de comparação entre uma variável e uma expressão para uma variável não definida. O sistema Arks permite comparações com expressões matemáticas somente para variáveis de entrada, ou para variáveis de atraso de variáveis de entrada, que devem ser definidas na construção *INPUTS*.

```
MODULE foo
{
   INPUTS A,B;
   ⋮
   RULE R1 IF A = B*3.0;
Ok
   RULE R2 IF C = B*3.0;
Erro: variável C não foi definida como variável
   de entrada
}
```

- **Existem contradições na fórmula da linha *número-da-linha***: o conjunto de modelos obtido para a fórmula é vazio.

```
MODULE foo
{
    ⋮
    RULE R1 IF A = 1.0 THEN A = 2.0;
Erro
}
```

- **Variável *nome-da-variável* não possui dimensionamento compatível com o template**: Os limites do template não estão contidos nos limites do dimensionamento da variável.

```
  MODULE foo
  {
     DIMENSION A[1..4];

     TEMPLATE T1[0..2] IF A = 1.0 THEN B = 1.0;
Erro: Variável A não
     possui a posição 0
     TEMPLATE T2[2..3] IF A = 1.0 THEN B = 1.0;
Ok
  }
```

- **Não existem variáveis dimensionadas no template**: Um template foi criado sem referências a alguma variável dimensionada, de forma que esta construção deveria ser do tipo *RULE*.

- **Variável *nome-da-variável* não é de entrada**: Houve tentativa de comparação entre uma variável e uma expressão para uma variável ou variável de atraso que não são de entrada. Devido à natureza contínua das variáveis de saída, o sistema Arks só permite comparações com expressões matemáticas para as variáveis de entrada ou para as variáveis de atraso destas.

- **Variável de template não pode ser dimensionada**: A construção *TEMPLATE* permite a construção de várias regras a partir de um modelo, onde são variados os índices de todas as variáveis dimensionadas referenciadas no modelo (ver seção 4.2). O sistema Arks não permite a referência a uma posição de um dimensionamento dentro da declaração de um template.

- **Já existe um módulo com o nome *nome-do-módulo***: Existem dois módulos com o mesmo nome dentro da base de conhecimentos. Isto não é permitido porque o nome das classes geradas para representar cada módulo é formado pelo nome do módulo, o que geraria ambiguidade no código do núcleo reativo gerado.

- **Ligação entre *nome-do-módulo* e *nome-do-módulo* forma ciclo**: Houve detecção de ciclo de execução entre os dois módulos listados, causados pela interação entre as suas variáveis de entrada e suas variáveis de saída (ver seção 4.6).

- **Não existem módulo iniciais**: Para o início do processo de execução do núcleo reativo, é necessário que pelo menos um módulo receba todos os seus dados de entrada do ambiente, ou seja, pelo menos um módulo deve poder iniciar o seu processamento sem dependência dos resultados obtidos pela execução de um outro módulo.

- **Variável *nome-da-variável* no módulo *nome-do-módulo* possui links com os módulos *nome-do-módulo* e *nome-do-módulo***: Uma variável de entrada do módulo recebe seus dados de dois módulos diferentes, o que não é permitido porque o sistema Arks não possui mecanismos para o tratamento das possíveis inconsistências entre os resultados obtidos pelos dois módulos.

- **Variável *nome-da-variável* foi definida como entrada e saída no módulo *nome-do-módulo***.

- **Variável *nome-da-variável* não é de entrada e não pode ter referência ao valor *ABSENT***: O valor *ABSENT* indica que o valor de uma variável não está presente na interação atual, o que só é possível para as variáveis de entrada do módulo.

- **Variável *nome-da-variável* não é de entrada do ambiente e faz parte de uma expressão no módulo *nome-do-módulo***: Uma variável de entrada de um módulo que receba seus valores de um outro módulo é representada na forma de uma lista de intervalos, uma vez que a variável de saída que forneceu estes dados nessa forma. O sistema Arks não possui mecanismos para avaliações de expressões matemáticas com este tipo de variáveis.

F.2 Erros não-fatais

Erros não fatais são avisos emitidos pelo compilador sobre alguma característica presente na base de conhecimento que poderia ser aprimorada para melhor performance do núcleo reativo. Estes erros são simples avisos, que não impedem a continuação do processo de compilação da base de conhecimento. A seguir é apresentada uma descrição de cada um dos erros não-fatais possíveis dentro do sistema Arks.

- **Variável *nome-da-variável* no módulo *nome-do-módulo* não é de entrada do ambiente e não pode ser inicializada. Inicialização ignorada**: Uma

variável que não é de entrada foi inicializada através da contrução *INITIAL*, sendo que, por definição, somente variáveis de entrada podem ter seus valores setados antes do processo de inferência do módulo. Neste caso, a tentativa de inicialização é simplesmente ignorada.

- **Variável *nome-da-variável* não é utilizada e foi removida do módulo *nome-do-módulo***: A variável não possui representação dentro do conjunto de hipercubos que representa o conjunto de modelos do módulo, e também não é utilizada em alguma expressão matemática, de forma que não é necessária dentro do módulo e eliminada. Isto geralmente ocorre quando é declarada uma variável de entrada, de saída ou de dimensionamento que não é referenciada em alguma regra dentro do módulo.

Apêndice G

Listagem do Código Exemplo em Linguagem C++

Aqui estão apresentadas as listagens dos três arquivos gerados pelo sistema ARKS, em linguagem C++, para a compilação do exemplo apresentado na Figura 4.26, descrito em um arquivo chamado *refri.rks*.

Tal código-fonte utiliza, para a construção das estruturas de dados, a STL (*Standard Template Library*), o que possibilita sua compilação em um compilador C++ genérico, desde que o mesmo possua as bibliotecas STL.

O código para a implementação dos **hcubo** é baseado na classe para implementação de mapas de bits apresentado em [Wei93] pags. 286-290.

Para este exemplo, foi utilizado o compilador Microsoft Visual C++ 5.0, que gerou uma aplicação de 60.928 bytes em modo Release com opção de otimização para tamanho do executável.

G.1 Arquivo *main_refri.cpp* — interface do núcleo reativo com o ambiente

```
#include "refri.hpp"

void main()
{
   CArk Arks;
```

```
   while(1)
   {
      Arks.Refrigerante_Input_Valor();
      Arks.Refrigerante_Input_Botao();
      Arks.Executa();
      Arks.Refrigerante_Output_Troco();
      Arks.Refrigerante_Output_Slot();
   }
}

void CArks::Refrigerante_Input_Valor()
{
   float Val;
   cout << "Entre com o Valor da Variavel Valor no Modulo Refrigerante: ";
   cin >> Val;
   Refrigerante->Input_Valor(Val);
}

void CArks::Refrigerante_Input_Botao()
{
   float Val;
   cout << "Entre com o Valor da Variavel Botao no Modulo Refrigerante: ";
   cin >> Val;
   Refrigerante->Input_Botao(Val);
}

void CArks::Refrigerante_Output_Troco()
{
   LIST_VALUES::iterator it,end;
   cout << "Resultados para a Variavel Troco no modulo Refrigerante:\ n";
   end = lRefrigerante_Troco->end();
   for(it = lRefrigerante_Troco->begin(); it != end; it++)
   {
      if((*it)->fValorIni == INFINITO && (*it)->fValorFin == INFINITO)
         cout << "\ tqualquer valor possivel\ n";
```

```
      else if((*it)->fValorIni == INFINITO)
         cout << "\ tmenor que " << (*it)->fValorFin << "\ n";
      else if((*it)->fValorFin == INFINITO)
         cout << "\ tmaior que " << (*it)->fValorIni << "\ n";
      else if((*it)->fValorIni == (*it)->fValorFin)
         cout << "\ tigual a " << (*it)->fValorIni << "\ n";
      else
         cout << "entre " << (*it)->fValorIni << " e " << (*it)->fValorFin
         << " excluindo os extremos\ n";
   }
}

void CArks::Refrigerante_Output_Slot()
{
   LIST_VALUES::iterator it,end;
   cout << "Resultados para a Variavel Slot no modulo Refrigerante:\ n";
   end = lRefrigerante_Slot->end();
   for(it = lRefrigerante_Slot->begin(); it != end; it++)
   {
      if((*it)->fValorIni == INFINITO && (*it)->fValorFin == INFINITO)
         cout << "\ tqualquer valor possivel\ n";
      else if((*it)->fValorIni == INFINITO)
         cout << "\ tmenor que " << (*it)->fValorFin << "\ n";
      else if((*it)->fValorFin == INFINITO)
         cout << "\ tmaior que " << (*it)->fValorIni << "\ n";
      else if((*it)->fValorIni == (*it)->fValorFin)
         cout << "\ tigual a " << (*it)->fValorIni << "\ n";
      else
         cout << "entre " << (*it)->fValorIni << " e " << (*it)->fValorFin
         << " excluindo os extremos\ n";
   }
}
```

G.2 Arquivo *refri.cpp* — implementação do núcleo reativo

```
#include <memory.h>
#include "refri.hpp"

//Funcoes do Hipercubo
CHipercubo::CHipercubo(const CHipercubo& x)
{
   numbytes = x.numbytes;
   numbits = x.numbits;
   vect = new unsigned char[numbytes];
   memcpy(vect, x.vect, numbytes);
}

CHipercubo::CHipercubo(long NumElms, int val): numbits(NumElms)
{
   numbytes = (numbits+7) >> 3;
   vect = new unsigned char[numbytes];
   memset(vect, (val) ? 0xff : 0, numbytes);
}

CHipercubo::CHipercubo(long NumElms): numbits(NumElms)
{
   numbytes = (numbits+7) >> 3;
   vect = new unsigned char[numbytes];
}

CHipercubo::~CHipercubo()
{
   delete vect;
}

int CHipercubo::operator[] (long inx)
{
```

```
   return(vect[inx>>3] & (1 << (inx & 7))) != 0;
}

CHipercubo& CHipercubo::operator &= (CHipercubo &x)
{
   unsigned char *src = x.vect;
   unsigned char *dst = vect;
   for(int cnt = numbytes; cnt--; src++, dst++)
      *dst &= *src;
   return *this;
}

int CHipercubo::GetBit(long inx)
{
   return (*this)[inx];
}

void CHipercubo::SetBit(long inx, int on)
{
   if(on)
      vect[inx>>3] |= (1 << (inx & 7));
   else
      vect[inx>>3] &= ~(1 << (inx &7));
}

void CHipercubo::SetRange(long min, long max, int on)
{
   long truemin,truemax;
   truemin = (min < max) ? min : max;
   truemax = (max > min) ? max : min;
   truemin = (truemin < 0) ? 0 : truemin;

   for(long i = truemin; i <= truemax; i++)
      SetBit(i,on);
}
```

```
int CHipercubo::E_Vazio(long Inicio, long Fim)
{
   for(long x=Inicio; x<=Fim; x++)
      if((vect[x>>3] & (1 << (x &7))))
         return 0;
   return 1;
}

CONJUNTO_HIPERCUBOS* CModulo::Operacao_Intersecao(CHipercubo *Hipercubo,
                                                  CONJUNTO_HIPERCUBOS *Conjunto)
{
   CONJUNTO_HIPERCUBOS *H;
   H = new CONJUNTO_HIPERCUBOS;

   CONJUNTO_HIPERCUBOS::iterator it,end;
   CHipercubo *hAux;

   end = Conjunto->end();
   for(it = Conjunto->begin(); it != end; it++)
   {
      hAux = Operacao_Intersecao(Hipercubo,(*it));
      if(hAux != NULL)
         H->push_back(hAux);
   };
   return H;
}

CHipercubo* CModulo::Operacao_Intersecao(CHipercubo *h1, CHipercubo *h2)
{
   CHipercubo *hAux;

   hAux = new CHipercubo(*h1);
   (*hAux) &= *h2;
```

```
   for(int x=0; x<nVariaveis; x++)
   {
      if(hAux->E_Vazio(Posicoes[0][x],Posicoes[1][x]))
      {
         delete hAux;
         hAux = NULL;
         break;
      };
   };
   return hAux;
}

void CModulo::Minimiza(LIST_VALUES *List)
{
   int Loop;
   LIST_VALUES::iterator it,it1,end;
   do
   {
      Loop = 0;
      end = List->end();
      for(it = List->begin(); it != end; it++)
         for(it1 = List->begin(); it1 != end; it1++)
         {
            if(it != it1)
            {
               if(((*it1)->fValorIni >= (*it)->fValorIni) &&
     ((*it1)->fValorFin <= (*it)->fValorFin))
               {
                   delete *it1;
                   it1 = List->erase(it1);
               }
               else if(((*it1)->fValorIni < (*it)->fValorIni) &&
     ((*it1)->fValorFin < (*it)->fValorFin) &&
     ((*it1)->fValorFin >= (*it)->fValorIni))
               {
```

```
                    (*it)->fValorIni = (*it1)->fValorIni;
                    delete *it1;
                    it1 = List->erase(it1);
                    Loop = 1;
                }
                else if(((*it1)->fValorIni > (*it)->fValorIni) &&
        ((*it1)->fValorIni < (*it)->fValorFin) &&
        ((*it1)->fValorFin > (*it)->fValorFin))
                {
                    (*it)->fValorFin = (*it1)->fValorFin;
                    delete *it1;
                    it1 = List->erase(it1);
                    Loop = 1;
                }
                end = List->end();
            }
        }
    }while(Loop);
}

CRefrigerante::CRefrigerante(CArks* R)
{
    CHipercubo *Hipercubo;
    Arks = R;
    Saida = new CONJUNTO_HIPERCUBOS;
    int Refrigerante[3][20] = {
{0 , 1 , 1 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0
  , 0 , 1 , 0},
{1 , 0 , 0 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 0 , 0 , 0 , 1 , 0 , 0 , 1
  , 0 , 0 , 0},
{1 , 1 , 1 , 1 , 1 , 1 , 0 , 1 , 1 , 1 , 0 , 0 , 0 , 1 , 0 , 0 , 1
  , 0 , 0 , 0}
    };
    for(int x=0; x<3; x++)
    {
```

```
      Hipercubo = new CHipercubo(20);
      for(int y=0; y<20; y++)
         Hipercubo->SetBit(y,Refrigerante[x][y]);
      lHipercubos.push_back(Hipercubo);
   };

   Entrada = new CHipercubo(20,0);
   fValor = 240282339694467130000000000000000000000.000000;
   Entrada->SetBit(3,1);
   Entrada->SetBit(8,1);
   Entrada->SetRange(10,14,1);
   Entrada->SetRange(15,19,1);

   Posicoes[0] = new long[4];
   Posicoes[1] = new long[4];
   nVariaveis = 4;
   Posicoes[0][0] = 0;
   Posicoes[1][0] = 4;
   Posicoes[0][1] = 5;
   Posicoes[1][1] = 9;
   Posicoes[0][2] = 10;
   Posicoes[1][2] = 14;
   Posicoes[0][3] = 15;
   Posicoes[1][3] = 19;

   lTroco = new LIST_VALUES;
}

CRefrigerante::~CRefrigerante()
{
   while(!lHipercubos.empty())
   {
      delete lHipercubos.front();
      lHipercubos.pop_front();
   }
```

```
   delete Entrada;
}

void CRefrigerante::Atualiza_Variaveis_Expressao_Atraso()
{
   LIST_VALUES::iterator it,end;
   long x,y;
}

void CRefrigerante::Input_Valor(float Val)
{
   Entrada->SetBit(3,0);
   float Intervalos[] = { 1.000000 ,
     240282339694467130000000000000000000000.000000 };
   long x=0;

   for(long y=0 ; y<2; y+=2, x++)
   {
      if(x>0)
      {
         if(Val < Intervalos[x] && Val > Intervalos[x-1])
            Entrada->SetBit(y,1);
      }
      else
      {
         if(Val < Intervalos[x])
            Entrada->SetBit(y,1);
      }
      if(Val == Intervalos[x])
         Entrada->SetBit(y+1,1);
   };
   if(Val > Intervalos[x-1])
      Entrada->SetBit(2,1);
```

```
   fValor = Val;
}

void CRefrigerante::Input_Botao(float Val)
{
   Entrada->SetBit(8,0);
   float Intervalos[] = { 1.000000 ,
     240282339694467130000000000000000000000.000000 };
   long x=0;

   for(long y=5 ; y<7; y+=2, x++)
   {
      if(x>0)
      {
         if(Val < Intervalos[x] && Val > Intervalos[x-1])
            Entrada->SetBit(y,1);
      }
      else
      {
         if(Val < Intervalos[x])
            Entrada->SetBit(y,1);
      }
      if(Val == Intervalos[x])
         Entrada->SetBit(y+1,1);
   };
   if(Val > Intervalos[x-1])
      Entrada->SetBit(7,1);
}

void CRefrigerante::Output_Troco()
{
   tag_LIST_VALUES *Aux;
   CONJUNTO_HIPERCUBOS::iterator it,end;

   while(!Arks->lRefrigerante_Troco->empty())
```

```
    {
        delete Arks->lRefrigerante_Troco->front();
        Arks->lRefrigerante_Troco->pop_front();
    }
    end = Saida->end();
    for(it = Saida->begin(); it != end; it++)
    {

        if((*it)->GetBit(11))
        {
            Aux = new tag_LIST_VALUES;
            Aux->fValorIni = Aux->fValorFin = fValor-1.000000;
                Arks->lRefrigerante_Troco->push_back(Aux);
        };

        if((*it)->GetBit(13))
        {
            Aux = new tag_LIST_VALUES;
            Aux->fValorIni = Aux->fValorFin = fValor;
                Arks->lRefrigerante_Troco->push_back(Aux);
        };
    };
    Minimiza(Arks->lRefrigerante_Troco);
}

void CRefrigerante::Output_Slot()
{
    tag_LIST_VALUES *Aux;
    CONJUNTO_HIPERCUBOS::iterator it,end;

    while(!Arks->lRefrigerante_Slot->empty())
    {
        delete Arks->lRefrigerante_Slot->front();
        Arks->lRefrigerante_Slot->pop_front();
    }
```

```
float Intervalos[] = { 0.000000 , 1.000000 };
end = Saida->end();
for(it = Saida->begin(); it != end; it++)
{
   long x=0;

   for(long y=15 ; y<19; y+=2, x++)
   {
      if((*it)->GetBit(y))
      {
         Aux = new tag_LIST_VALUES;
         Aux->fValorFin = Intervalos[x];
         if(x>0)
            Aux->fValorIni = Intervalos[x-1];
         else
            Aux->fValorIni = INFINITO;
         Arks->lRefrigerante_Slot->push_back(Aux);
      };
      if((*it)->GetBit(y+1))
      {
         Aux = new tag_LIST_VALUES;
         Aux->fValorFin = Intervalos[x];
         Aux->fValorIni = Intervalos[x];
         Arks->lRefrigerante_Slot->push_back(Aux);
      };
   };
   if((*it)->GetBit(19))
   {
      Aux = new tag_LIST_VALUES;
      Aux->fValorFin = INFINITO;
      Aux->fValorIni = Intervalos[x-1];
         Arks->lRefrigerante_Slot->push_back(Aux);
   };
};
Minimiza(Arks->lRefrigerante_Slot);
```

```
}

void CRefrigerante::Executa()
{
   while(!Saida->empty())
   {
      delete Saida->front();
      Saida->pop_front();
   };
   delete Saida;

   Atualiza_Variaveis_Expressao_Atraso();

   Saida = Operacao_Intersecao(Entrada,&lHipercubos);
   Output_Troco();
   Output_Slot();
   Entrada->SetRange(0,19,0);
   fValor = 240282339694467130000000000000000000000.000000;
   Entrada->SetBit(3,1);
   Entrada->SetBit(8,1);
   Entrada->SetRange(10,14,1);
   Entrada->SetRange(15,19,1);
}

CArks::CArks()
{
   Refrigerante = new CRefrigerante(this);
   lRefrigerante_Troco = new LIST_VALUES;
   lRefrigerante_Slot = new LIST_VALUES;
}

CArks::~CArks()
{
   delete Refrigerante;
   while(!lRefrigerante_Troco->empty())
```

```
    {
        delete lRefrigerante_Troco->front();
        lRefrigerante_Troco->pop_front();
    }
    delete lRefrigerante_Troco;
    while(!lRefrigerante_Slot->empty())
    {
        delete lRefrigerante_Slot->front();
        lRefrigerante_Slot->pop_front();
    }
    delete lRefrigerante_Slot;
}

void CArks::Executa()
{
    Refrigerante->Executa();
}
```

G.3 Arquivo *refri.hpp* — definições de classes

```
#include <iostream.h>

#include <list>
#include <string>

using namespace std;

#define INFINITO 340282346638528860000000000000000000000.000000

class CHipercubo {
private:
    unsigned char *vect;
    int numbytes;
    long numbits;
```

```
public:
   inline CHipercubo(const CHipercubo&);
   inline CHipercubo(long, int);
   inline CHipercubo(long);
   inline ~CHipercubo();
   inline int GetBit(long);
   inline int operator[] (long);
   inline void SetBit(long, int);
   inline CHipercubo& operator &= (CHipercubo& x);
   void SetRange(long, long, int);
   inline long GetSize(void) { return numbits; };
   int E_Vazio(long,long);
};

struct tag_LIST_VALUES {
   float fValorIni;
   float fValorFin;
};

typedef list<tag_LIST_VALUES*>LIST_VALUES;
typedef list<CHipercubo*> CONJUNTO_HIPERCUBOS;

class CArks;

class CModulo {
protected:
   CONJUNTO_HIPERCUBOS lHipercubos, *Saida;
   CHipercubo *Entrada;
   CArks *Arks;
   long *Posicoes[2], nVariaveis;

   CONJUNTO_HIPERCUBOS *Operacao_Intersecao(CHipercubo*,CONJUNTO_HIPERCUBOS*);
   CHipercubo *Operacao_Intersecao(CHipercubo*,CHipercubo*);
   void Minimiza(LIST_VALUES*);
};
```

```
class CRefrigerante : public CModulo {
private:
   float fValor;
   LIST_VALUES *lTroco;

   void Atualiza_Variaveis_Expressao_Atraso();


public:
  CRefrigerante(CArks*);
  ~CRefrigerante();
   void Executa();
   void Input_Valor(float Val);
   void Input_Botao(float Val);
   void Output_Troco();
   void Output_Slot();
};

class CArks {
private:
   //ponteiros para os objetos que representarao cada modulo
   CRefrigerante *Refrigerante;

public:
   LIST_VALUES *lRefrigerante_Troco;
   LIST_VALUES *lRefrigerante_Slot;

   CArks();
   ~CArks();
   void Executa();
   void Refrigerante_Input_Valor();
   void Refrigerante_Input_Botao();
   void Refrigerante_Output_Troco();
   void Refrigerante_Output_Slot();
```

```
};
```

Apêndice H

Listagem do Código Exemplo em Linguagem Java

Aqui estão apresentadas as listagens dos dois arquivos gerados pelo sistema ARKS, em linguagem Java, a partir do exemplo apresentado na Figura 4.26, descrito em um arquivo chamado *refri.rks*.

Tal código-fonte é implementado utilizando a JDK 1.0.2, o que possibilita sua execução em qualquer plataforma que possua uma JVM compatível.

Para este exemplo, foi utilizado o compilador javac da Sun Microsystems, constante da JDK 1.1.6, sobre o sistema operacional Windows NT Workstation 4.0, que gerou um conjunto de arquivos com os *byte-codes* de cada classe, obtendo um tamanho total da aplicação de 9.856 bytes.

H.1 Arquivo *CrefriInterface.java* — interface do núcleo reativo com o ambiente

```
import java.io.StreamTokenizer;
import java.util.Vector;

public class CrefriInterface extends CArks
{
    public void Refrigerante_Input_Valor()
    {
        StreamTokenizer in = new StreamTokenizer(System.in);
```

```
    do
    {
        System.out.print("Entre com o Valor da Variavel Valor no Modulo
                          Refrigerante: ");
        try
        {
            in.nextToken();
        }
        catch(Exception e)
        {
            ;
        }
    }while(in.ttype != StreamTokenizer.TT_NUMBER);
    Refrigerante.Input_Valor(in.nval);
}

public void Refrigerante_Input_Botao()
{
    StreamTokenizer in = new StreamTokenizer(System.in);

    do
    {
        System.out.print("Entre com o Valor da Variavel Botao no Modulo
                          Refrigerante: ");
        try
        {
            in.nextToken();
        }
        catch(Exception e)
        {
            ;
        }
    }while(in.ttype != StreamTokenizer.TT_NUMBER);
    Refrigerante.Input_Botao(in.nval);
```

```
}

public void Refrigerante_Output_Troco()
{
   System.out.println("Resultados para a Variavel Troco no modulo
                       Refrigerante:");
   for(int x=0; x<lRefrigerante_Troco.size();x++)
   {
       if(((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).dValorIni
          == Double.NEGATIVE_INFINITY
          && ((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).dValorFin
              == Double.POSITIVE_INFINITY)
           System.out.println("Qualquer Valor possivel");
       else if(((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).dValorIni
              == Double.NEGATIVE_INFINITY)
           System.out.println("menor que "+((LIST_VALUES)lRefrigerante_Troco.
              elementAt(x)).dValorFin);
       else if(((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).dValorFin
              == Double.POSITIVE_INFINITY)
           System.out.println("maior que "+((LIST_VALUES)lRefrigerante_Troco.
              elementAt(x)).dValorIni);
       else if(((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).dValorIni
              == ((LIST_VALUES)lRefrigerante_Troco.elementAt(x)).
              dValorFin)
           System.out.println("igual a "+((LIST_VALUES)lRefrigerante_Troco.
              elementAt(x)).dValorIni);
       else
           System.out.println("entre "+((LIST_VALUES)lRefrigerante_Troco.
            elementAt(x)).dValorIni+" e "+((LIST_VALUES)lRefrigerante_Troco.
            elementAt(x)).dValorFin+" excluindo os extremos");
   }
}

public void Refrigerante_Output_Slot()
{
```

```
    System.out.println("Resultados para a Variavel Slot no modulo
                        Refrigerante:");
    for(int x=0; x<lRefrigerante_Slot.size();x++)
    {
        if(((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorIni
           == Double.NEGATIVE_INFINITY
           && ((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorFin
           == Double.POSITIVE_INFINITY)
            System.out.println("Qualquer Valor possivel");
        else if(((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorIni
           == Double.NEGATIVE_INFINITY)
            System.out.println("menor que "+((LIST_VALUES)lRefrigerante_Slot.
            elementAt(x)).dValorFin);
        else if(((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorFin
           == Double.POSITIVE_INFINITY)
            System.out.println("maior que "+((LIST_VALUES)lRefrigerante_Slot.
           elementAt(x)).dValorIni);
        else if(((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorIni
           == ((LIST_VALUES)lRefrigerante_Slot.elementAt(x)).dValorFin)
            System.out.println("igual a "+((LIST_VALUES)lRefrigerante_Slot.
            elementAt(x)).dValorIni);
        else
            System.out.println("entre "+((LIST_VALUES)lRefrigerante_Slot.
            elementAt(x)).dValorIni+" e "+((LIST_VALUES)lRefrigerante_Slot.
            elementAt(x)).dValorFin+" excluindo os extremos");
    }
}

public static void main(String[]args)
{
    CrefriInterface refriInterface = new CrefriInterface();

    while(true)
    {
      refriInterface.Refrigerante_Input_Valor();
```

```
            refriInterface.Refrigerante_Input_Botao();
            refriInterface.Executa();
            refriInterface.Refrigerante_Output_Troco();
            refriInterface.Refrigerante_Output_Slot();
        }
    }
}
```

H.2 Arquivo *refri.java* — implementação do núcleo reativo

```
import java.util.Vector;
import java.util.BitSet;

class LIST_VALUES
{
    public double dValorIni;
    public double dValorFin;

    public boolean Contem(LIST_VALUES L1)
    {
        return L1.dValorIni >= dValorIni && L1.dValorFin <= dValorFin;
    }

    public boolean Agrupa(LIST_VALUES L1)
    {
        boolean Result;

        Result = false;
        if(L1.dValorIni < dValorIni && L1.dValorFin < dValorFin &&
           L1.dValorFin >= dValorIni)
        {
            dValorIni = L1.dValorIni;
            Result = true;
```

```
        }
        if(L1.dValorIni > dValorIni && L1.dValorIni < dValorFin &&
           L1.dValorFin > dValorFin)
        {
            dValorFin = L1.dValorFin;
            Result = true;
        }
        return Result;
    }
}

class CHipercubo
{
    private BitSet BitVect;

    public CHipercubo()
    {
        super();
    }

    public CHipercubo(int nBits)
    {
        BitVect = new BitSet(nBits);
    }

    public void and(CHipercubo h)
    {
        BitVect.and(h.getBitSet());
    }

    public BitSet getBitSet()
    {
        return BitVect;
    }
```

```
public static CHipercubo clone(CHipercubo h)
{
    CHipercubo Result = new CHipercubo();
    Result.BitVect = (BitSet)(h.getBitSet().clone());
    return Result;
}

public boolean getBit(int index)
{
    return BitVect.get(index);
}

public void setBit(int index, boolean on)
{
    if(on)
        BitVect.set(index);
    else
        BitVect.clear(index);
}

public void setRange(int Inicio, int Fim, boolean on)
{
    for(int x=Inicio; x<=Fim; x++)
        if(on)
           BitVect.set(x);
        else
           BitVect.clear(x);
 }

public boolean vazio(int Inicio, int Fim)
{
    for(int x=Inicio; x<=Fim; x++)
       if(BitVect.get(x))
          return false;
    return true;
```

```
    }
}

class CModulo
{
    protected Vector lHipercubos;
    protected Vector Saida;
    protected CArks Arks;
    protected CHipercubo Entrada;
    protected int [][]Posicoes;
    protected int nVariaveis;

    public Vector Operacao_Intersecao(CHipercubo Hipercubo, Vector Conjunto)
    {
        Vector H = new Vector();
        CHipercubo Aux;

        for(int x=0; x<Conjunto.size(); x++)
        {
            Aux = Operacao_Intersecao(Hipercubo,(CHipercubo)Conjunto.elementAt(x)
            if(Aux != null)
               H.addElement(Aux);
        }
        return H;
    }

    private CHipercubo Operacao_Intersecao(CHipercubo h1, CHipercubo h2)
    {
        CHipercubo Aux;

        Aux = CHipercubo.clone(h1);
        Aux.and(h2);
        for(int x=0; x<nVariaveis; x++)
           if(Aux.vazio(Posicoes[0][x],Posicoes[1][x]))
           {
```

```
                Aux = null;
                break;
            }
        return Aux;
    }

    protected void Minimiza(Vector List)
    {
        boolean Loop;

        do
        {
            Loop = false;
            for(int x=0; x<List.size(); x++)
               for(int y=0; y<List.size(); y++)
                  if(x != y)
                  {
                      if(((LIST_VALUES)List.elementAt(x)).Contem(
                         (LIST_VALUES)List.elementAt(y)))
                         List.removeElementAt(y);
                      else if(((LIST_VALUES)List.elementAt(x)).Agrupa(
                              (LIST_VALUES)List.elementAt(y)))
                      {
                         List.removeElementAt(y);
                         Loop = true;
                      }
                  }
        }while(Loop);
    }
}

class CRefrigerante extends CModulo
{
   double dValor;
```

```
public CRefrigerante(CArks Arks)
{
    lHipercubos = new Vector(3);
    CHipercubo Hipercubo;
    this.Arks = Arks;
    Saida = new Vector();
    int Refrigerante[][] = {
  {0,1,1,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0},
  {1,0,0,1,1,1,1,1,1,1,1,0,0,0,1,0,0,1,0,0,0},
  {1,1,1,1,1,1,0,1,1,1,0,0,0,1,0,0,1,0,0,0}
    };
    for(int x=0; x<3; x++)
    {
       Hipercubo = new CHipercubo(20);
       for(int y=0; y<20; y++)
          if(Refrigerante[x][y]==1)
             Hipercubo.setBit(y,true);
       lHipercubos.addElement(Hipercubo);
     };

    Entrada = new CHipercubo(20);

    dValor = Double.MAX_VALUE;
    Entrada.setBit(3,true);
    Entrada.setBit(8,true);
    Entrada.setRange(10,14,true);
    Entrada.setRange(15,19,true);

    Posicoes = new int[2][4];
    nVariaveis = 4;
    Posicoes[0][0] = 0;
    Posicoes[1][0] = 4;
    Posicoes[0][1] = 5;
    Posicoes[1][1] = 9;
    Posicoes[0][2] = 10;
```

```
    Posicoes[1][2] = 14;
    Posicoes[0][3] = 15;
    Posicoes[1][3] = 19;
}

public void Input_Valor(double Val)
{
    Entrada.setBit(3,false);
    double Intervalos[] = { 1.000000 };
    int x=0;

    for(int y=0 ; y<2; y+=2, x++)
    {
        if(x>0)
        {
            if(Val < Intervalos[x] && Val > Intervalos[x-1])
                Entrada.setBit(y,true);
        }
        else
        {
            if(Val < Intervalos[x])
                Entrada.setBit(y,true);
        }
        if(Val == Intervalos[x])
            Entrada.setBit(y+1,true);
    };
    if(Val > Intervalos[x-1])
        Entrada.setBit(2,true);

    dValor = Val;
}

public void Input_Botao(double Val)
{
    Entrada.setBit(8,false);
```

```
    double Intervalos[] = { 1.000000 };
    int x=0;

    for(int y=5 ; y<7; y+=2, x++)
    {
       if(x>0)
       {
          if(Val < Intervalos[x] && Val > Intervalos[x-1])
             Entrada.setBit(y,true);
       }
       else
       {
          if(Val < Intervalos[x])
             Entrada.setBit(y,true);
       }
       if(Val == Intervalos[x])
          Entrada.setBit(y+1,true);
    };
    if(Val > Intervalos[x-1])
       Entrada.setBit(7,true);
}

public void Output_Troco()
{
    LIST_VALUES Aux;
    int it;

    Arks.lRefrigerante_Troco.removeAllElements();
    for(it = 0; it < Saida.size(); it++)
    {

       if(((CHipercubo)Saida.elementAt(it)).getBit(11))
       {
          Aux = new LIST_VALUES();
          Aux.dValorIni = Aux.dValorFin = dValor-1.000000;
```

```
                Arks.lRefrigerante_Troco.addElement(Aux);
        };

        if(((CHipercubo)Saida.elementAt(it)).getBit(13))
        {
            Aux = new LIST_VALUES();
            Aux.dValorIni = Aux.dValorFin = dValor;
                Arks.lRefrigerante_Troco.addElement(Aux);
        };
    };
    Minimiza(Arks.lRefrigerante_Troco);
}

public void Output_Slot()
{
    LIST_VALUES Aux;
    int it;

    Arks.lRefrigerante_Slot.removeAllElements();
    double Intervalos[] = { 0.000000 , 1.000000 };
    for(it = 0; it < Saida.size(); it++)
    {
        int x=0;

        for(int y=15 ; y<19; y+=2, x++)
        {
            if(((CHipercubo)Saida.elementAt(it)).getBit(y))
            {
                Aux = new LIST_VALUES();
                Aux.dValorFin = Intervalos[x];
                if(x>0)
                    Aux.dValorIni = Intervalos[x-1];
                else
                    Aux.dValorIni = Double.NEGATIVE_INFINITY;
                Arks.lRefrigerante_Slot.addElement(Aux);
```

```
            };
            if(((CHipercubo)Saida.elementAt(it)).getBit(y+1))
            {
               Aux = new LIST_VALUES();
               Aux.dValorFin = Intervalos[x];
               Aux.dValorIni = Intervalos[x];
               Arks.lRefrigerante_Slot.addElement(Aux);
            };
         };
         if(((CHipercubo)Saida.elementAt(it)).getBit(19))
         {
            Aux = new LIST_VALUES();
            Aux.dValorFin = Double.POSITIVE_INFINITY;
            Aux.dValorIni = Intervalos[x-1];
               Arks.lRefrigerante_Slot.addElement(Aux);
         };
      };
      Minimiza(Arks.lRefrigerante_Slot);
   }

   public void Executa()
   {
      Saida = null;

      Saida = Operacao_Intersecao(Entrada,lHipercubos);
      Output_Troco();
      Output_Slot();
      Entrada.setRange(0,19,false);

      dValor = Double.MAX_VALUE;
      Entrada.setBit(3,true);
      Entrada.setBit(8,true);
      Entrada.setRange(10,14,true);
      Entrada.setRange(15,19,true);
   }
```

```
};


class CArks
{
   CRefrigerante Refrigerante;
   Vector lRefrigerante_Troco;
   Vector lRefrigerante_Slot;

    public CArks()
    {
       Refrigerante = new CRefrigerante(this);
       lRefrigerante_Troco = new Vector();
       lRefrigerante_Slot = new Vector();
    }


    public void Executa()
    {
       Refrigerante.Executa();
    }
};
```

Referências Bibliográficas

[Ben88] S. Bennet, *Real-time computer control*, Prentice-Hall, 1988.

[Ber89] Gérard Berry, *Programming a digital watch in esterel v3*, Technical report, INRIA, 1989.

[Bib82] W. Bibel, *Automated theorem proving*, Vieweg, Braunschweig, 1982.

[Cat90] L. Catach, *Logiques non classiques. présentation et applications à l'intelligence artificielle*, Modéles logiques et systèmes d'intelligence artificielle (A. Duchaussoy L.Iturrioz, ed.), Hermes, Paris, 1990.

[Dzi90] Daniel Dzierzgowski, *Quatre exemples de languages ou environnements pour le développement de programmes où le temps intervient*, Technique et Science Informatiques (1990), 289–312.

[EI89] G. Escalada-Imaz, *Optimisation d'algorithmes d'inférence monotone en logique des propositions et du premier ordre*, Ph.D. thesis, Universidade Paul Sabatier de Toulouse, julho 1989.

[Far96] Celso A. A. Kaestner; Jean-Marie Farines, *From the synchronous approach to hybrid systems*, 21st IFAC/IFIP Workshop on REAL TIME PROGRAMMING, 1996, pp. 77–82.

[Gia88] E. R. Dougherty; C. R. Giardina, *Mathematical methods for artificial intelligence and autonomous systems*, Prentice-Hall, Cambridge, 1988.

[Gon89] G. Berry; G. Gonthier, *The esterel synchronous programming language: Desing, semantics, implementation*, Technical report, INRIA, 1989.

[Gri90] Paul Gochet; Pascal Gribomont, *Logique - mèthodes pour l'informatique fondamentale*, Hermes, Paris, 1990.

[Kae91] Celso A. A. Kaestner, *Comparativo entre as linguagens síncronas esterel, signal, lustre e kheops*, Relatório interno LAAS, Toulouse, 1991.

[Kae93] Celso A. A. Kaestner, *Uma proposta de sistema baseado no conhecimento para aplicações tempo-real utilizando o enfoque síncrono*, Ph.D. thesis, Universidade Federal de Santa Catarina, 1993.

[LeG90] Albert Benveniste; Paul LeGuernic, *Hybrid dynamical systems theory and the signal language*, IEEE Transactions on Automatic Control **35** (1990), no. 5, 535–546.

[LeG91] Patricia Bournai; Bruno Cheron; Bernard Houssais; Paul LeGuernic, *Manual signal*, Publicação Interna 575, IRISA, Fevereiro 1991.

[Oua89] N. Halbwachs; D. Pilaud; F. Ouabdesselam, *Specifying, programming and verifying real-time systems using a synchronous declarative language*, Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science, vol. 407, Springer-Verlag, junho 1989, pp. 213–231.

[Phi88] M. Ghallab; H. Philippe, *A compiler for real-time knowledge-base systems*, International Workshop on Artificial Intelligence for Industrial Applications, 1988.

[Phi89] H. Phillipe, *Algorithmes pour la compilation de bases de connaissances en logique propositionnelle et du premier ordre: les systèmes kheops et clops*, Ph.D. thesis, Universidade Paul Sabatier de Toulouse, maio 1989.

[Poi96] O. Maffeïs; A Poigné, *Synchronous automata for reactive, real-time or embedded systems*, Tech. report, GMD, 1996.

[Ree90] Steve Reeves, *Logic for computer science*, Addison Wesley, 1990.

[Rig83] G. Berry; S. Moisan; J.P. Rigault, *Esterel: Towards a synchronous and semantically sound high-level language for real-time applications*, IEEE Real-Time Systems Simposium, 1983, pp. 30–37.

[Riv90] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest, *Introduction to algorithms*, The MIT Engineering and Computer Science Series, MIT Press, 1990.

[Rut96] P. Le Guernic; É. Rutten, *Experiments with the synchronous methodology illustrating its support of predictability*, 21st IFAC/IFIP Workshop on REAL TIME PROGRAMMING, 1996, pp. 71–76.

[San98] Alexandre D. Santos, *Geração de sistemas baseados no conhecimento para aplicações reativas - a proposta rks*, I Workshop de Sistemas Tempo-Real (Rio de Janeiro), 1998.

[Sor85] Jean-Paul Tremblay; Paul G. Sornson, *The theory and practice of compiler writing*, McGraw-Hill, Singapure, 1985.

[Ste95] Mark Stefik, *Introduction to knowledge systems*, Morgan Kaufmann, 1995.

[Tur96] R. Turner, *Logiques pour l'intelligence artificielle*, Masson, Paris, 1996.

[Ull86] Alfred V. Aho; Ravi Sethi; Jeffrey D. Ullman, *Compiladores. princípios, técnicas e ferramentas*, Guanabara Koogan, 1986.

[Var95] Ronald Fagin; Joseph Y. Halpern; Yoram Moses; Moshe Y. Vardi, *Reasoning about knowledge*, MIT Press, 1995.

[Wal89] L. Wallen, *Automated proof search in non-classical logics: Efficient matrix proof methods for modal and intuitionistic logics*, MIT Press, 1989.

[Wat86] D. A. Waterman, *A guide to expert systems*, Addison-Wesley, 1986.

[Wei93] Jeff Duntemann; Keith Weiskamp, *C/c++ ferramentas poderosas*, Berkeley Brasil Editora, 1993.

[You82] S. J. Young, *Real-time languages*, Ellis Horwood Publishers, 1982.