Uma Ferramenta para Especificação e Geração de Modelos Markovianos Baseada numa Metodologia Orientada a Objeto

Morganna Carmem Diniz

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUACÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENCÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTACÃO.

Aprovada por:

Prof. Edmundo de Souza e Silva, Ph.D.
(presidente)

Profa. Sueli Bandeira Teixeira Mendes, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Daniel Alberto Menascé, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 1990

DINIZ, MORGANNA CARMEM

Uma Ferramenta para Especificação e Geração de Modelos Markovianos Baseada numa Metodologia Orientada a Objeto [Rio de Janeiro] 1990

XII, 127 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTACÃO, 1990)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Ferramenta para Análise de Desempenho

I. COPPE/UFRJ II. Título(Série).

A meus pais

Agradecimentos

Ao Prof. Edmundo, pela orientação e paciência no acompanhamento e desenvolvimento deste trabalho.

Ao Prof. Elian Machado, pelo incentivo e apoio.

À equipe de laboratório que esteve presente em todas as etapas deste trabalho.

À equipe da secretaria que de todas as formas colaborou durante o meu período de estudante.

Ao CNPQ, pelo apoio financeiro a este trabalho.

Aos meus pais e ao meu irmão, que mesmo de longe, sempre me apoiaram.

À minha amiga Darlene, pela sincera amizade que sempre me dedicou.

Ao Belchior, pela amizade que resiste a todas as distâncias.

A Paula Marisa, Cristiana Bentes e Graça Marietto que tiveram presenças marcantes em todos os momentos difíceis e de alegria no decorrer deste trabalho.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtencão do grau de Mestre em Ciências (M. Sc.)

Uma Ferramenta para Especificação e Geração de Modelos Markovianos Baseada numa Metodologia Orientada a Objeto

Morganna Carmem Diniz

Maio de 1990

Orientador: Edmundo de Souza e Silva
Programa: Engenharia de Sistemas e Computacão

Várias ferramentas têm sido desenvolvidas para a análise de sistemas de computação / comunicação. Em geral, é oferecida aos usuários destas ferramentas, uma interface de alto nível para a especificação de modelos de sistemas; uma representação matemática é então gerada para a análise. Na sua maioria, estas interfaces são específicas de uma determinada área de aplicação. Neste trabalho estamos interessados no desenvolvimento de uma interface que permita a definição de modelos de sistemas de diferentes aplicações a partir de uma biblioteca de tipos de objetos, como também, a definição, pelo usuário final, de novos tipos de objetos para a biblioteca. Faz ainda parte deste trabalho, um estudo sobre a aplicabilidade desta interface em diversas áreas como confiabilidade, redes de filas e protocolos de comunicação.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A Tool for the Specification and Generation of Markovian Models Using an Object Oriented Paradigm

Morganna Carmem Diniz

May, 1990

Thesis Supervisor: Edmundo de Souza e Silva

Department: Programa de Engenharia de Sistemas e Computacão

Many tools have been developed for the analysis of computer and communication systems. In general, the tools provide to the user a high level interface for specification of system models; then a representation tailored to the mathematical solver is produced for analysis. However these tools are often specific to a narrow application area. In this work we are interested in a general modeling interface development tool. Systems will be described in terms of previously defined objects that will be in an object type library. Users will also be able to define new object types. Finally, we will show applications in areas such as reliability, queueing and communications.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo I

Introdução

Várias ferramentas têm sido desenvolvidas para a análise de sistemas de computação / comunicação. Em geral, é oferecida aos usuários destas ferramentas, uma interface de alto nível para a especificação de modelos de sistemas; uma representação numérica é então gerada para a análise. A interface pode ou não ser próxima da representação *natural* do sistema, enquanto a representação numérica é uma representação de baixo nível que serve de entrada para o software de solução da ferramenta. Por exemplo, para um modelo de confiabilidade, o usuário forneceria o número de componentes do sistema, as taxas de falha e de reparo para cada componente e a política de reparo para quando existir mais de um componente falho; para um modelo de filas representando uma rede de comunicação de dados, o usuário descreveria as filas que fariam parte do sistema, as conexões entre as filas, as taxas de chegada de mensagens na rede, as rotas a serem seguidas pelas mensagens e as taxas de erro de transmissão dos canais de comunicação. A representação numérica para estes dois modelos poderia ser a matriz de transição da cadeia de Markov correspondente.

Na sua maioria, as interfaces oferecidas pelas ferramentas existentes são específicas de uma determinada área de aplicação: apenas determinados tipos de modelos de sistemas podem ser definidos e somente alguns parâmetros do modelo são analisados. Um exemplo é a ferramenta SAVE [1] utilizada para a análise de modelos de confiabilidade: uma cadeia de Markov é gerada e parâmetros, como

disponibilidade e confiabilidade do sistema, são fornecidos ao usuário.

O ideal seria a existência de uma interface que pudesse ser facilmente adaptada de acordo com as necessidades do usuário, que permitisse a especificação de modelos, para diferentes áreas de aplicação, em uma linguagem bem próxima da representação *natural* do sistema, e que fornecesse ao usuário a tradução, desta definição, para a representação numérica correspondente. Em [2], Berson *et al* propõem uma metodologia com estes objetivos.

Na metodologia proposta em [2], modelos de sistemas são descritos em termos de *objetos* que interagem entre si através de troca de mensagens. Aplicações específicas são desenvolvidas a partir dos tipos de objetos usados nos modelos de aplicação. Novos domínios de aplicação são facilmente incorporados pela utilização de novos tipos de objetos que podem ser armazenados em uma biblioteca de tipos de objetos.

Neste trabalho estamos interessados no desenvolvimento de uma ferramenta para geração de cadeias de Markov que permita a definição de modelos de sistemas a partir de uma biblioteca de tipos de objetos, e que também facilite a definição, pelo usuário final, de novos tipos de objetos para a biblioteca. Faz ainda parte deste trabalho, um estudo sobre a aplicabilidade da metodologia de [2] em diversas áreas como confiabilidade, redes de filas e protocolos de comunicação.

No capítulo II é apresentado um estudo sobre algumas ferramentas de análise existentes na literatura. No capítulo III fazemos uma revisão da metodologia de modelagem de [2] e apresentamos exemplos na área de confiabilidade, redes de filas e protocolos de comunicação. No capítulo IV apresentamos a ferramenta desenvolvida. No capítulo V fazemos a análise de alguns exemplos mostrados no capítulo III, e no capítulo VI apresentamos nossas conclusões.

Capítulo II

Algumas Ferramentas de Análise

II.1 Introdução

Este capítulo destina-se a dar uma visão geral das ferramentas existentes para análise de confiabilidade e desempenho de sistemas de computação / comunicação. Alguns conceitos utilizados neste capítulo na área de confiabilidade são definidos a seguir [3]:

- *Confiabilidade* é a probabilidade de que o sistema esteja operacional durante o intervalo de tempo $[t_0,t]$, dado que o sistema está operacional no instante t_0.
- *Disponibilidade instantânea* é a probabilidade de o sistema estar operacional em um dado instante.
- *Disponibilidade estável* é a fração de tempo em que o sistema ficou operacional durante o intervalo $[t_0,t]$.
- *Utilidade* diz respeito a facilidade com que as falhas no sistema são detectadas e corrigidas.
- *Sistemas tolerantes a falhas* são sistemas capazes de continuar executando corretamente suas funções apesar de terem sido detectadas falhas em alguns componentes.

- Um sistema *falha com segurança* quando a falha é detectada antes que cause algum dano às operações executadas pelo sistema.
- *Reconfiguração* é a substituição de um componente com falha por um componente de reserva.
- Falhas podem ser *permanentes*, *intermitentes* ou *transientes*. *Falhas permanentes* incorrem na perda de componentes e conseqüente degradação do sistema. *Falhas transientes* podem desaparecer antes de serem detectadas pelo sistema, mas se tiverem uma duração longa podem ser interpretadas como *permanentes*. E *falhas intermitentes* intercalam-se entre ativas e benignas em qualquer instante; quando ativas levam o sistema a operar incorretamente, e quando benignas não afetam o sistema.
- Uma falha é *catastrófica* quando o sistema pára de operar após a sua ocorrência.
- Cobertura (*Coverage*) é a probabilidade condicional de recuperação de um componente dado que uma falha ocorreu.
- *Tempo de missão* é o tempo de computação necessário para a execução de um trabalho.

Na seção II.2 são mostradas algumas técnicas para modelagem e solução de modelos de confiabilidade, e na seção II.3 é feita a avaliação de algumas ferramentas de análise existentes.

II.2 Exemplos de Técnicas para Modelagem e Solução de Modelos de Confiabilidade

II.2.1 Árvore de Falhas

Esta técnica [3] tem por objetivo modelar as condições que resultam na não operabilidade de um sistema($\overline{C}_{sistema}$). Isto requer a enumeração das falhas e/ou eventos normais que levam um sistema a falhar.

Considere um sistema composto de 2 processadores do tipo A e um processador do tipo B (fig. II.1). O sistema é dito operacional se pelo menos um processador do tipo A e o processador do tipo B estão operacionais. A confiabilidade do sistema é dada pela probabilidade de o mesmo estar em algum estado operacional. A árvore de de falhas para este sistema é mostrada na figura II.2, onde $\overline{C}_i$ é a probabilidade de que o componente i tenha falhado.

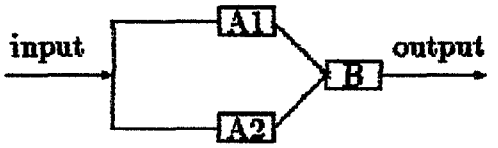

Figura II.1: Sistema de processadores

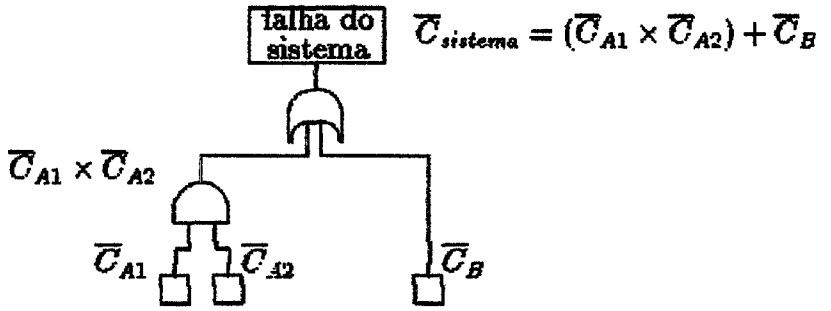

Figura II.2: Árvore de falhas

Uma vantagem desta modelagem é a facilidade de descrever sistemas grandes, e uma desvantagem é a impossibilidade de lidar com questões como dependência de falhas e de reparos [3].

II.2.2 Cadeia de Markov

Um conjunto de variáveis aleatórias $X(t)$ define um processo estocástico, onde $X(t)$ representa o estado do sistema no instante t. Um processo estocástico é markoviano ([4], [5]) se a probabilidade de que o próximo estado seja $X(t_{n+1}) = j$ depende

somente do estado presente $X(t_n)$ e não dos valores anteriores, ou seja,

$$P\{X(t_{n+1}) = j \mid X(t_0) = K_0, X(t_1) = K_1, ..., X(t_n) = i\} = P\{X(t_{n+1}) = j \mid X(t_n) = i\},$$

$$t_1 < t_2 < ... < t_{n+1}$$

As probabilidades condicionais $P\{X(t_{n+1}) = j \mid X(t_n) = i\}$ são chamadas de probabilidades de transição. Se para cada i e j, temos

$$P\{X_{t+s} = j \mid X_t = i\}, \quad \textit{independente de } s$$

então, as probabilidades de transição são ditas estacionárias (não mudam com o tempo) e a cadeia de Markov é chamada de homogênea.

Em caso de cadeias de Markov discretas, as transições entre estados ocorrem em instantes definidos no tempo; e em caso de cadeias de Markov contínuas, as transições podem ocorrer em qualquer instante de tempo. O tempo entre transições de estados de uma cadeia de Markov de tempo contínuo é dado por uma distribuição exponencial.

A imposição de que transições de estado obedeçam a uma determinada distribuição restringe os tipos de processos que podem ser modelados. Se desejarmos relaxar esta restrição, devemos fazer uso de cadeias semi-markovianas. Nelas, os tempos entre transições são arbitrários: a técnica de análise consiste em construir uma cadeia de Markov cujos estados representam o estado do processo nos instantes das transições e é por isso denominada de *cadeia de Markov embutida*.

Para cadeias de Markov com grande número de estados, algumas técnicas [3] foram desenvolvidas objetivando reduzir a complexidade de análise das mesmas. Uma técnica é a AGREGAÇÃO, onde estados semelhantes são agrupados (*estados agregados*). Uma segunda técnica é a TRUNCAGEM, onde estados com pouca probabilidade de ocorrerem são eliminados. E uma terceira técnica é a DECOMPOSIÇÃO, onde o sistema é dividido em um conjunto de subsistemas de acordo com o comportamento ou função dos seus componentes, e cada subsistema é resolvido isoladamente com uma representação simplificada do complemento.

II.2.3 Redes de Petri

Uma rede de Petri [6] corresponde a um grafo formado por lugares, transições e arcos. Um lugar é representado por um círculo e uma transição é representada por uma barra. Um lugar L é uma entrada de uma transição T, se existe um arco direcionado de L para T. Similarmente, um lugar L é saída de uma transição T, se existe um arco direcionado de T para L.

Os estados e as mudanças de estados são representados pela posição e movimento de fichas (*tokens* - pontos que residem nos círculos). O número de fichas contido em cada lugar da rede é usualmente especificado por uma n-upla chamada de *marcação*. Em análise de confiabilidade, por exemplo, estas fichas podem representar tanto falhas como componentes operacionais. Em geral, exite um número máximo de fichas presentes no sistema, ou seja, existe um número máximo de falhas permitidas ou um número máximo de componentes operacionais em um sistema. Isto limita o número de estados possíveis da rede.

Uma transição somente é permitida quando todas as suas entradas associadas possuem pelo menos uma ficha, e é representada pela retirada de uma ficha de cada entrada e colocação de uma ficha em cada saída. Mais de uma transição pode ser possível na rede, mas apenas uma delas pode ocorrer em um dado instante de tempo.

Se o tempo entre a retirada das fichas dos lugares de entrada e a colocação das fichas nos lugares de saída não for instantâneo, mas possuir uma distribuição exponencial, a rede é denominada de REDE DE PETRI ESTOCÁSTICA e pode ser convertida para uma cadeia de Markov contínua [7], e assim ser resolvida analiticamente.

Diversas extensões de rede de Petri foram desenvolvidas com o objetivo de incrementar seu poder de modelagem e facilitar a solução de problemas de análise [3]. Uma extensão é a incorporação de arcos inibidores dentro da rede; um ARCO INIBIDOR (figura II.3a) permite a uma transição disparar somente se o lugar associado não possui nenhuma ficha. Outra extensão é o ARCO PROBABILÍSTICO

(figura II.3b): quando a transição é permitida, a ficha removida é colocada em um dos lugares de saída apontados pelos arcos, onde cada lugar possui uma determinada probabilidade de ganhar a ficha e a soma destas probabilidades é 1. Outra extensão é o ARCO CONTADOR K (figura II.3c), que só vibializa a transição se o lugar associado possui pelo menos um número K de fichas; quando a transição ocorre, K fichas são removidas.

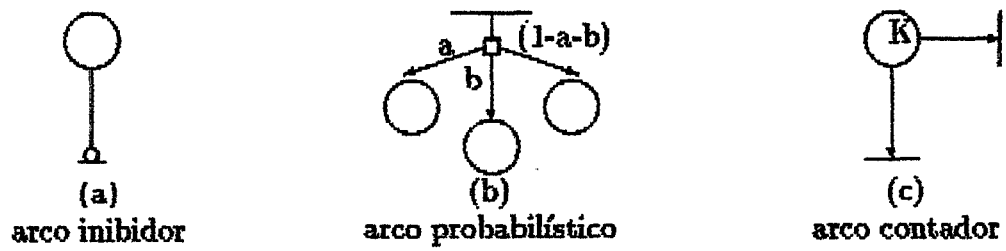

Figura II.3: Extensões da Rede de Petri

II.2.4 Simulação

Seja $X_1, X_2, \ldots$ uma sequência de variáveis aleatórias independentes e identicamente distribuidas, onde cada variável possui uma média finita $\mu = E[X_i]$. Pela lei forte dos grandes números com probabilidade 1 temos

$$\frac{X_1 + X_2 + \ldots + X_n}{n} \rightarrow \mu \text{ qdo } n \rightarrow \infty$$

Seja E um evento em um determinado experimento e P(E) a probabilidade que E ocorra em uma tentativa qualquer. Assim,

$$X_i = \begin{cases} 1 & \text{se E ocorreu na i-ésima tentativa} \\ 0 & \text{se E não ocorreu na i-ésima tentativa} \end{cases}$$

Então:

$$\sum_{i=1}^{n} \frac{X_i}{n} = \frac{\text{número de sucessos}}{\text{número de tentativas}} \rightarrow E(X_i) = P(\text{sucesso em n tentativas})$$

Ou seja, podemos utilizar a proporção de sucessos obtidos como uma estimativa da probabilidade de sucesso quando o número de tentativas é muito grande. Este método de determinar probabilidades por meio de experimentos é conhecido por SIMULAÇÃO ([3], [5]).

Tipicamente, simulação reproduz com maior fidelidade o comportamento de um sistema do que modelos analíticos. Entretanto, um maior custo computacional é exigido em decorrência do grande número de experimentos produzidos para se obter um intervalo de confiança razoável dos resultados obtidos [3]. Por exemplo, em modelos de confiabilidade as taxas de falhas dos componentes dos sistemas são, em geral, muito pequenas, o que torna necessário a geração de um número muito grande de experimentos para se obter uma representação real do comportamento do sistema. Para evitar isto, pode-se alterar, ou pelo menos distorcer, o problema original de forma que a incerteza nas respostas é reduzida. Tais procedimentos são conhecidos por Técnicas de Redução de Variância e são discutidos em [8] e [9].

II.3 Exemplos de Ferramentas Existentes

II.3.1 ARIES(Automated Reliability Interactive Estimation System)

ARIES [10] é utilizada para análise de confiabilidade e ciclo de vida de sistemas tolerantes a falhas. Podem ser modelados, nesta ferramenta, sistemas reparáveis, sistemas não reparáveis, sistemas com recuperação de falhas transientes e sistemas não reparáveis que periodicamente são reinicializados. A partir da descrição do modelo de sistema uma cadeia de Markov homogênea é gerada para a análise.

O sistema a ser modelado em ARIES é visto como um conjunto de subsistemas homogêneos. Cada subsistema é formado por um conjunto de módulos idênticos que podem estar ativos ou inativos (de reservas). Quando ativo, um módulo está participando das tarefas realizadas pelo sistema. Quando inativo, um módulo não participa da computação mas pode substituir um outro módulo, do mesmo sub-

sistema, que falhou. O sistema é dito *degradado* quando está operando com um número de módulos ativos menor que o número inicial de módulos ativos, ou seja, não existem mais módulos de reserva para substituir os módulos ativos que falharam.

São alguns dos parâmetros de entrada de ARIES:

1. físicos:
   - taxa de falha para cada módulo ativo;
   - taxa de falha para cada módulo de reserva;
   - taxa de chegada de falhas transientes para cada módulo ativo;
   - duração média de uma falha transiente;
2. estruturais:
   - número inicial de módulos ativos;
   - número inicial de módulos de reserva;
   - número D de degradações permitidas na configuração ativa;
   - vetor Y de degradações permitidas na configuração ativa ($Y = (Y[1], Y[2], \ldots, Y[D])$, onde Y[i] especifica o número de módulos ativos após a i-ésima degradação);
3. de reparo:
   - número de *técnicos de reparo*;
   - taxa de reparo para cada *técnico de reparo*;
4. de detecção e recuperação de falhas permanentes:
   - cobertura para recuperação de uma falha permanente em um módulo ativo;
   - cobertura para recuperação de uma falha permanente em um módulo de reserva;

- vetor CY para configurações ativas degradadas (CY = (CY[1], CY[2], ... , CY[D]), onde CY[i] é a *cobertura* associada com a transição para a configuração degradada descrita por Y[i]);

5. de detecção e recuperação de falhas transientes:

   - número n de fases de recuperação;
   - recuperabilidade - probabilidade condicional que a falha não seja catastrófica dado que uma falha ocorreu;
   - taxa de falha de todo o *hardware* engajado na execução dos processos de recuperação de falhas transientes;
   - vetor T de duração da recuperação (T = (T[1],T[2], ... , T[n]), onde T[i] é a duração da i-ésima fase da recuperação);
   - vetor E de eficácia da recuperação (E = (E[1],E[2], ... , E[n]), onde E[i] é a probabilidade condicional que a i-ésima fase da recuperação tenha sucesso dado que houve uma falha transiente não catastrófica).

São alguns dos parâmetros calculados por ARIES:

1. medidas do tempo de missão para o sistema:

   - confiabilidade;
   - tempo médio para a primeira falha;
   - taxa de falha;
   - influência da não confiabilidade de cada subsistema no sistema;
   - influência da redundância (módulos ativos que executam a mesma tarefa) na confiabilidade;
   - diferença do tempo de missão entre dois projetos distintos do sistema;

2. medidas do ciclo de vida para o sistema:

   - probabilidade que o sistema esteja operacional sem aparente degradação;

- probabilidade que o sistema esteja operacional em qualquer dos estados degradados;
- probabilidade que o sistema esteja operacional em um dos estados não seguros ou em estados com componentes de reserva com falhas e que não sejam recuperáveis;
- probabilidade que o sistema tenha falhado com segurança;
- probabilidade de uma falha catastrófica;
- disponibilidade instatânea - probabilidade que o sistema esteja operacional no instante t;
- segurança - probabilidade que o sistema esteja operacional ou que tenha falhado de modo seguro;
- tempo médio de computação perdido devido a não operabilidade do sistema;
- número médio de falhas dos módulos.

ARIES pode ser utilizado para modelar sistemas reparáveis ou não, com falhas permanentes e/ou transientes. Entretanto, é restrito a sistemas com taxas de transição constantes, além de oferecer uma solução de $O(n^5)$ para a matriz de transição, enquanto que ferramentas que apenas permitem definições de modelos de sistemas reparáveis, conseguem solução de $O(n^4)$ ([3], [11]).

II.3.2 HARP(Hybrid Automated Reliability Predictor)

HARP [12] é utilizada para avaliação da confiabilidade e disponibilidade instântanea de sistemas reparáveis e de sistemas não reparáveis. Esta ferramenta é dita híbrida por utilizar dois modelos: um para ocorrências e reparos das falhas e outro para o manuseamento das falhas. No primeiro modelo é utilizada uma cadeia de Markov não homogênea e no segundo, em geral, é usada uma rede de Petri estocástica.

São algumas das entradas de HARP:

1. árvore de falhas na forma gráfica;
2. cadeia de Markov na forma gráfica;
3. taxas de falhas;
4. taxas de reparo;
5. condições iniciais do sistema;
6. falhas encadeadas;
7. para rede de Petri estocástica:
    - transição de falha intermitente ativa para falha intermitente benigna;
    - transição de falha intermitente benigna para falha intermitente ativa;
    - tempo de vida de uma falha transiente;
    - probabilidade de detectar falhas;
    - probabilidade de isolar uma falha detectada;
    - número de tentativas para recuperar um componente que falhou;
    - probabilidade de sucesso na reconfiguração;
    - fração das falhas que são transientes;
8. descrição dos processos de recuperação em termos de uma rede de Petri;

São saídas de HARP :

1. confiabilidade e disponibilidade instantânea;
2. probabilidades de falhas encadeadas, de falhas atribuidas a não existência de componentes de reserva e de falhas que não afetam outros componentes.

HARP é capaz de modelar sistemas grandes e complexos, mas não permite adotar uma política de manutenção, e nem permite a definição de diferentes taxas de falha para componentes ativos e de reservas [3].

II.3.3 METFAC(Modeling and Evaluation of Complex Fault-Tolerant Computing Systems)

METFAC [13] é utilizada para modelar e avaliar complexos sistemas de computação tolerantes a falhas. O sistema a ser modelado em METFAC é visto como um conjunto de processos que executam determinadas ações. Cada processo é descrito por regras que especificam quando o processo está ativo (pode executar as ações associadas ao processo) e o que ocorre quando o processo está ativo (as ações que deve executar).

Esta ferramenta é organizada em tarefas implementadas nos seguintes níveis:

1. especificação do modelo: o usuário fornece um conjunto de regras do comportamento do modelo;
2. geração do diagrama de transição: um modelo markoviano é gerado a partir das regras fornecidas;
3. preparo da avaliação: inclui todo o processo de preparação dos algoritmos a serem executados no próximo nível;
4. avaliação.

A especificação das regras inclui os seguintes itens:

1. PE - conjunto de variáveis inteiras chamadas de Parâmetros da Estrutura. Ex: número total de componentes de um determinado tipo no sistema;
2. PF - conjunto de variáveis reais chamadas de Parâmetros Funcionais. Ex: taxas e probabilidades;
3. VE - conjunto de variáveis inteiras chamadas de Variáveis de Estado. São utilizadas para identificar o estado do sistema;
4. R - conjunto de regras r_k;

5. A - conjunto de funções λ^k(PE,PF,VE) reais positivas chamadas Taxas de Ação. Ex: taxa de falha transiente de um estado;

6. C - conjunto de funções c_i^k reais positivas chamadas de Probabilidades das Respostas. Ex: probabilidade de uma falha ser catastrófica;

7. I(PE,PF,VE) - uma função real positiva chamada Índice que associa pesos aos estados para avaliação de desempenho e de custo.

Seja a_k a condição para um determinado processo esteja ativo, e a_j^k as condições para que determinadas ações sejam executadas. Assim, podemos definir uma regra r_k como:

if a_k(PE,VE) then
  if $a_1^k(PE, VE)$ then $VE \leftarrow s_1^k(PE, VE)$
  .....
  if $a_{q^k}^k(PE, VE)$ then $VE \leftarrow s_{q^k}^k(PE, VE)$
end

Cada regra r_k tem uma Taxa de Ação λ_k e condições de respostas a_j^k associadas (j = 1,...q_k). Cada resposta j tem uma função c_j^k avaliando a probabilidade que a resposta ocorra, e uma função s_j^k do novo estado.

São parâmetros calculados pelo nível de avaliação:

1. para sistemas com capacidade total ou sistemas não operacionais:

   - constantes:
     - disponibilidade do estado estacionário;
     - tempo médio do sistema funcionando com capacidade total;
     - tempo médio de duração de uma falha;
     - tempo médio entre ocorrências de falhas;
     - tempo médio para a primeira falha;

  - tempo médio para funcionamento quando o sistema não está operacional;

- variáveis com o tempo:
  - disponibilidade;
  - confiabilidade (para sistema não reparáveis);
  - manutenabilidade - probabilidade de recuperar com sucesso, dentro de determinado período de tempo, um componente que falhou;
  - confiabilidade do ciclo de vida;
  - manutenabilidade do ciclo de vida;

2. desempenho do sistema (para cada estado operacional i da cadeia de Markov é associada uma taxa de desempenho $\eta(i)$):
   - constantes:
     - desempenho esperado para o estado estacionário;
     - serviço médio (quantidade de trabalho executada pelo sistema) até a primeira falha;
     - serviço médio durante operação;
   - variáveis com o tempo:
     - desempenho esperado;
     - utilidade (para sistema não reparáveis);
     - utilidade do ciclo de vida;

3. custos (para cada estado i da cadeia de Markov, operacional ou falho, é associada uma taxa de custo $c(i)$):
   - constante:
     - taxa de custo esperada para o estado estacionário;
   - variável com o tempo:
     - taxa de custo esperada.

II.3.4 RESQ(Research Queueing Package)

RESQ [14] é uma ferramenta para construção e resolução de modelos de redes de filas. O sistema é visto como um conjunto de recursos e de tarefas. Uma tarefa corresponde a uma ou mais unidades de trabalho a ser executada, onde cada unidade pode exigir o uso de um ou mais recursos. As tarefas circulam entre os recursos e disputam entre si o uso destes recursos.

Cada recurso possui uma fila associada que pode ser ativa ou passiva. Uma fila ativa (fig. II.4) proporciona serviço às tarefas e é composta de 3 elementos: 1) uma ou mais áreas de espera chamadas de CLASSES, onde cada classe possui uma distribuição de tempo de serviço, 2) um ou mais servidores e 3) uma política de alocação para decidir qual a próxima tarefa a fazer uso do recurso. Uma fila passiva (fig. II.5) consiste de um conjunto de fichas e um conjunto de elementos para manipulação das fichas. As fichas representam unidades distintas de um recurso e são alocadas às tarefas. Os elementos para manipulação das fichas incluem *nó de alocação*, *nó de liberação*, *nó de criação* e *nó de destruição* . Uma tarefa espera em um *nó de alocação* até conseguir o número de fichas desejadas. Em um *nó de liberação* a tarefa abandona todas as fichas que possui. Novas fichas são adicionadas às fichas já existentes em um *nó de criação*. E em um *nó de destruição* a tarefa elimina todas as fichas que possui.

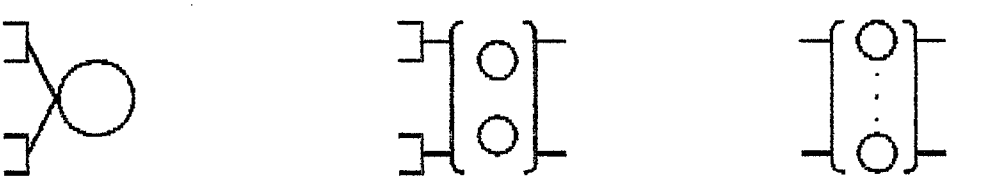

Figura II.4: Exemplos de filas ativas

Tipicamente uma tarefa não interage com outros elementos enquanto está em uma fila ativa, mas pode adquirir fichas em uma fila passiva e mantê-las enquanto

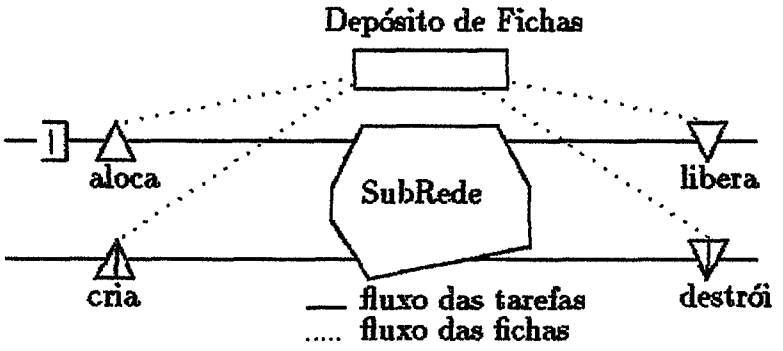

Figura II.5: Uma fila passiva

visita outras filas ou elementos do sistema. Assim, o uso de filas passivas permite que uma tarefa aloque mais de um recurso por vez.

Exemplos do uso das primitivas de modelagem acima podem ser encontrados em [14].

Outros elementos, conhecidos por nós auxiliares (Figura II.6), podem ser definidos no sistema: SOURCE para chegadas externas; SINK para partidas da rede; SET para atribuir valores às variáveis de tarefas anteriormente definidas e/ou variáveis comuns a todas as tarefas da rede; SPLIT para uma tarefa produzir outras tarefas independentes; FISSION para produzir tarefas dependentes; FUSION para eliminar tarefas dependentes; e DUMMY para simplificar descrições de caminhos.

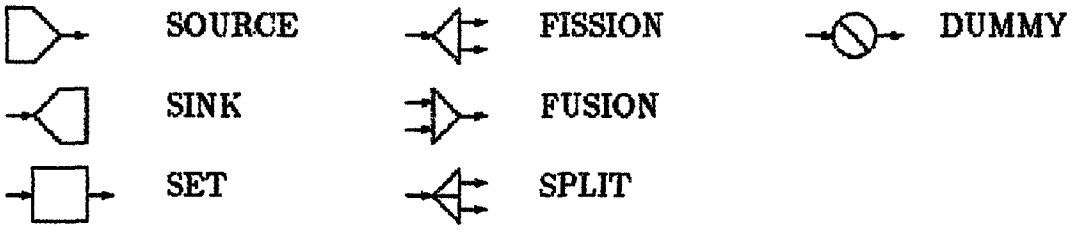

Figura II.6: nós auxiliares

Os caminhos percorridos pelas tarefas são chamados de *cadeias*. O número de tarefas independentes em uma cadeia pode mudar devido ao uso dos elementos *source*, *sink* e *split*. Cadeias com um número fixo de tarefas independentes são chamadas de *fechadas* e cadeias com um número flutuante de tarefas são chamadas de *abertas*.

São alguns parâmetros de entrada de RESQ:

1. nome do modelo:
   MODEL: <nome do modelo>

2. método a ser utilizado:
   METHOD: <NUMERICAL | SIMULATION>
   Filas passivas somente podem ser definidas no método de simulação. O método numérico ignora o uso simultâneo de recursos, pois considera que o sistema possui solução em forma de produto [14]. Este método, por consequinte, resolve o sistema utilizando-se de algoritmos de solução de redes de filas (redes com solução em forma de produto, exatos ou de aproximação).

3. parâmetros globais que receberão valores numéricos quando o modelo for resolvido:
   NUMERIC PARAMETERS: < parâmetro1 parâmetro2 .... >

4. constantes globais numéricas:

   NUMERIC IDENTIFIERS: < nome1 nome2 .....>
   nome1: < valor >
   nome2: < valor >
   .....

5. definição de filas:

   - nome da fila:
     QUEUE: < nome da fila >

- tipo da fila:

  TYPE: < tipo >

- para filas ativas:

  CLASS LIST: < classe1 classe2 .....>

  SERVICE TIMES: < tempo1 tempo2 ....>

  PRIORITIES: < prioridade1 prioridade2 ....>

- para filas passivas:

  TOKENS: < quantidade >

  DSPL: < disciplina >

  ALLOCATE NODE LIST: < nome >

  NUMBERS OF TOKENS TO ALLOCATE: < quantidade >

  .....

  ALLOCATE NODE LIST: < nome >

  NUMBERS OF TOKENS TO ALLOCATE: < quantidade >

  RELEASE NODE LIST: < nome >

  .....

  RELEASE NODE LIST: < nome >

  DESTROY NODE LIST: < nome >

  .....

  DESTROY NODE LIST: < nome >

  CREATE NODE LIST: < nome >

  NUMBERS OF TOKENS TO CREATE: < quantidade >

  .....

  CREATE NODE LIST: < nome >

  NUMBERS OF TOKENS TO CREATE: < quantidade >

6. definição da cadeia:

   - nome da cadeia:

     CHAIN: < nome >

- tipo da cadeia:

  TYPE: < tipo >

- chegadas externas:

  SOURCE: < nome source >

  ARRIVAL TIMES: < tempo >

  < nome source > → < nome1 nome2 ....>

  < prob1 prob2 ....>

7. definição de submodelos:

   - nome do submodelo:

     SUBMODEL: < nome >

   - parâmetros que receberão valores numéricos quando o modelo for resolvido:

     NUMERIC PARAMETERS: < parâmetro1 parâmetro2 .... >

   - constantes numéricas:

     NUMERIC IDENTIFIERS: < nome1 nome2 .....>

     nome1: < valor >

     nome2: < valor >

     .....

   - cadeia a ser utilizada:

     CHAIN PARAMETERS: < nome da cadeia >

8. para simulação:

   - intervalo de confiança:

     CONFIDENCE INTERVAL METHOD: < método >

     CONFIDENCE LEVEL: < valor >

     RESQ proporciona 3 métodos para estimação do intervalo de confiança: *duplicações independentes, regenerativo* e *spectral*. O primeiro método é

preferido para estimação de características transientes e os outros dois para estimar características em equilíbrio, sendo o segundo método para modelos com características regenerativas e o terceiro método para modelos sem características regenerativas.

- inicialização do sistema:

INITIAL STATE DEFINITION -

CHAIN: < nome da cadeia >

NODE LIST: < nome >

INIT POP < quantidade inicial >

.....

NODE LIST: < nome >

INIT POP < quantidade inicial >

- intervalo de confiança das filas:

CONFIDENCE INTERVAL QUEUES: < fila1 fila2 ....>

MEASURES: < parâmetros a serem verificados >

ALLOWED WIDTHS: < valor >

.....

MEASURES: < parâmetros a serem verificados >

ALLOWED WIDTHS: < valor >

9. parâmetros de saída desejados:

WHAT: < ALL | parâmetro1 parâmetro2 ....>

São saídas fornecidas pelo RESQ:

1. utilização (servidores ou fichas);
2. throughput (vazão);
3. tamanho médio da fila;

4. desvio padrão do tamanho da fila;
5. distribuição do tamanho da fila;
6. tempo médio na fila;
7. desvio padrão do tempo de fila;
8. distribuição do tempo de fila;
9. número médio de fichas em uso;
10. distribuição das fichas em uso;
11. número médio total de fichas que não estão em uso;
12. distribuição das fichas que não estão em uso;
13. o maior tamanho médio de fila;
14. o maior tempo médio na fila;
15. população da cadeia aberta;
16. tempo de resposta da cadeia aberta.

II.3.5 SAVE(System Availability Estimator)

SAVE é utilizada para construção e resolução de modelos probabilísticos de confiabilidade e disponibilidade. Modelos podem se resolvidos analiticamente [1] ou utilizando simulação de Monte Carlo Direta ou Analógica [8]. No método direto, o tempo para falha e o tempo para reparo de cada componente são gerados aleatoriamente a cada mudança de estado do sistema. No método analógico, as transições de estado são geradas randomicamente de acordo com as probabilidades de transição da cadeia de Markov correspondente. Para sistemas com grande número de componentes, o método analógico é mais eficiente por exigir a geração de um menor número de números aleatórios por transição de estado [8].

O sistema a ser modelado usando SAVE é considerado como uma coleção de componentes de vários tipos e interconectados entre si. Cada componente é sujeito a falhas e pode estar em um dos 4 estados abaixo:

1. operacional;
2. dormant: quando não existe no sistema o número mínimo de componentes necessário para o componente em questão funcionar.
3. falho;
4. de reserva.

Taxas de falhas distintas podem ser definidas para os diferentes estados dos componentes (operacional, *dormant*, reserva). Cada componente pode falhar em diferentes modos, onde cada modo de falha possue uma taxa específica de reparo. Além do mais, um componente ao falhar em um determinado modo pode provocar falhas em outros componentes, e estes componentes afetados também irão precisar de reparo.

SAVE possui dois programas: SVREAD e SVRUN. O primeiro é responsável pela análise dos dados de entrada e geração da cadeia de Markov do sistema e o segundo pela resolução numérica e análise da confiabilidade e disponibilidade da mesma.

São entradas a serem fornecidas ao SVREAD:

1. nome do modelo:
   MODEL: <nome do modelo>
2. o método a ser utilizado para construção da cadeia de Markov:
   METHOD:<NUMERICAL|SIMULATION|MARKOV|COMBINATORIAL>
   No método NUMERICAL, a cadeia de Markov é gerada a partir das especificações fornecidas pelo usuário. O segundo método simula as transições entre estados da cadeia de Markov. No método MARKOV, os estados e as taxas de

transição são fornecidos pelo usuário ao invés da descrição em alto nível do sistema. E no método COMBINATORIAL, a definição do sistema é dada em termos de uma árvore de falha.

3. componentes são especificados pelo tipo. Para cada tipo é assinalado um nome e uma quantidade que inclui componentes operacionais e de reserva.
   COMPONENT: <nome><quantidade>

4. o número de componentes de reserva e a taxa de falha dos mesmos para cada tipo especificado:
   SPARES: <quantidade>
   SPARES FAILURE RATE:<expressão>

5. o número mínimo de componentes operacionais necessário para cada tipo de componente poder ser considerado operacional:

   OPERATION DEPEND UPON: <nome><quantidade>,
   <nome><quantidade>,...

6. o número mínimo de componentes operacionais necessário para cada tipo de componente poder ser reparado:

   REPAIR DEPEND UPON: <nome><quantidade>,
   <nome><quantidade>,...

7. o estado do componente quando o sistema não está operacional:
   DORMANT WHEN SYSTEM DOWN: <YES | NO>

8. a taxa de falha dos componentes dormantes:
   DORMANT FAILURE RATE: <expressão>

9. a taxa de falha dos componentes operacionais:
   FAILURE RATE: <expressão>

10. as probabilidades de falha dos componentes em diferentes modos:

FAILURE MODE PROBABILITIES: <probabilidade>,
<probabilidade>,...

11. a taxa de reparo para cada tipo de componente:
    REPAIR RATE: <expressão>

12. os componentes afetados por cada modo de falha:

    COMPONENTS AFFECTED:
    <NONE | lista de nomes | nome(<quant>) >,...
    <lista | nome>: <probabilidade de afetar>,
    <lista | nome>: <probabilidade de afetar>,...

13. as condições para o sistema ser considerado operacional:
    EVALUATION CRITERIA: <condições>

14. a estratégia adotada para escolha do componente a ser reparado:
    REPAIR STRATEGY: <PRIORITY | ROS>
    A política PRIORITY especifica prioridades diferentes de reparo para os vários tipos de componentes, enquanto a política ROS (Random Order Service) dá igual prioridade para todos os componentes.

15. o número de servidores para reparo:
    REPAIRMEN: <quantidade>

O SVRUN lê a matriz de transição gerada pelo SVREAD e calcula os seguintes parâmetros:

1. disponibilidade do estado estacionário - fração do tempo em que o sistema está operacional sobre um intervalo infinitamente longo;

2. disponibilidade de um determinado intervalo de tempo - fração média de tempo em que o sistema é operacional sobre um dado intervalo de tempo;

3. distribuição do tempo operacional em um intervalo de tempo;

4. tempo médio para falha do sistema - tempo médio para a cadeia de Markov alcançar um estado não operacional;

5. confiabilidade de um intervalo de tempo - probabilidade que o sistema não falhe num dado intervalo de tempo;

6. sensibilidade da disponibilidade do estado estacionário em relação a um parâmetro p - é obtido pela multiplicação da derivada da disponibilidade (em relação ao parâmetro p) pela razão entre o parâmetro p e a disponibilidade naquele ponto. Um resultado negativo implica que um pequeno aumento no parâmetro irá diminuir o valor da disponibilidade anteriormente calculada, enquanto um valor positivo irá implicar em um aumento do valor da disponibilidade.

Utilizando técnicas de armazenamento de matrizes esparsas e métodos iterativos para resoluções de equações, SAVE é capaz de resolver modelos com milhares de estados, mas a sua interface é específica para a análise de modelos de confiabilidade.

II.3.6 SPNP (Stochastic Petri Net Package)

SPNP [15] permite modelar sistemas complexos de computação utilizando redes de Petri estocásticas. A linguagem de entrada de SPNP é o CSPL (C-based SPNP language). Um arquivo CSPL é compilado usando o compilador C e é então *linkado* com os outros arquivos C que compõe o SPNP.

A característica talvez mais importante do CSPL é a possibilidade de definir parâmetros dependentes das marcações. Parâmetros como taxas de transição e cardinalidade dos arcos de entrada podem ser especificados como uma função do número de fichas (marcação) de alguns lugares.

```
double rate1 { return (mark("p1") * MU1 + mark ("p2") * MU2); }
trans ("t1"), ratefun("t1",rate1);
```

O exemplo acima inicialmente define *rate1* como o número de fichas do lugar $p1$ vezes $MU1$ (uma constante) somado ao número de fichas de $p2$ vezes $MU2$ (uma outra constante). Depois a transição $t1$ é definida, e para a transição $t1$ é especificada a taxa *rate1*.

São extensões utilizadas em SPNP:

1. cardinalidade variável dos arcos de entrada - nas definições clássicas de rede de Petri e na maioria das definições de redes de Petri estocásticas, a cardinalidade de um arco é um valor inteiro constante: se um arco de entrada de um lugar p para um lugar q possui cardinalidade K, então o lugar p deve possuir pelo menos K fichas para a transição ocorrer, e quando ocorre, K fichas são removidas. Em SPNP, é possível definir de um lugar p para um lugar q uma transição t que é habilitada sempre que p possui pelo menos uma ficha, e que quando esta ocorre, todas as fichas são removidas de p para q. Na figura II.7 a transição t move apenas a primeira ficha de p para q e deposita uma ficha de controle em p_{flush}; a transição t_{flush} então move o restante das fichas, uma de cada vez, até p ficar vazio; e no final a transição t_{stop} remove a ficha de controle de p_{flush}.

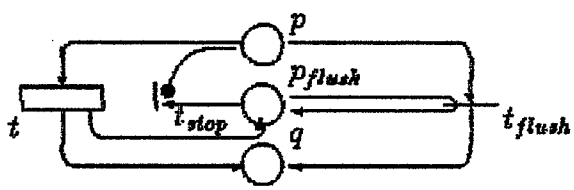

Figura II.7: Esvaziando um lugar

2. funções para habilitação das transições - para uma transição t estar habilitada é necessário que : 1) nenhuma transição com prioridade maior que t esteja habilitada; 2) o número de fichas em cada um dos lugares de entrada da transição t deve ser maior ou igual a cardinalidade do correspondente arco de entrada; e 3) o número de fichas em cada um dos arcos inibidores da transição t deve ser menor que a cardinalidade do correspondente arco inibidor. Cada

transição t possui uma função booleana e associada. A função é avaliada na marcação M quando existe a possibilidade de t estar habilitada. A transição t é dita habilitada em M se, e somente se, $e(M)$ = verdade.

3. condições - para facilitar a depuração das especificações fornecidas pelo usuário, uma função a pode ser definida com as condições para que uma marcação possa existir. Logo que uma marcação M é gerada, $a(M)$ é avaliada. M é considerada ilegal se $a(M)$ = falso; a execução é então abortada e o usuário é informado do erro encontrado durante a geração dos estados da rede de Petri.

4. matrizes de lugares e transições - SPNP permite a definição de matrizes de lugares e transições:

$$place\_1("cpu", 7)$$

define 7 lugares com nomes *cpu*.0 até *cpu*.6.

$$trans\_2("move", N, 2 * N)$$

para N = 3, define dezoito transições: *move*.0.0 até *move*.2.5.

5. subredes - por ser um arquivo C, o CSPL pode utilizar CALL para se conectar com outros arquivos. Assim, se uma determinada função contém a definição de uma subrede, para cada CALL desta função encontrada em um arquivo, será gerada uma instância da subrede correspondente.

A partir da especificação do modelo fornecida pelo usuário em CSPL, a cadeia de Markov correspondente é gerada, e parâmetros são analisados utilizando técnicas de solução [15] como ELIMINAÇÃO (estados transientes são ignorados) e PRESERVAÇÃO (apenas estados transientes ou que possuem distribuição exponencial do tempo de permanência são considerados).

São alguns dos parâmetros de saida do SPNP:

- características estacionárias:

  - probabilidades dos estados estacionários;
  - número médio de fichas em cada lugar;
  - throughput das transições;

- características transitórias:
  - disponibilidade instantânea;
  - intervalo da disponibilidade;
  - confiabilidade para sistemas tolerantes a falhas;

SPNP permite a definição de modelos de sistemas de diferentes áreas de aplicação, além de possibilitar a especificação de parâmetros de saida diferentes dos parâmetros calculados por SPNP. Entretanto, é exigido do usuário conhecimento da linguagem C e das primitivas de modelagem de SPNP.

II.3.7 Outras Ferramentas

Muitas ferramentas não foram discutidas neste capítulo por não ser objetivo deste trabalho fazer uma descrição completa da maioria das ferramentas de análise já desenvolvidas. Um estudo mais amplo pode ser encontrado em [3] para modelos de confiabilidade.

Capítulo III

Ferramenta Proposta

III.1 Introdução

Neste capítulo é descrito o modelo proposto em [2], além de fornecer exemplos na área de confiabilidade, redes de filas e protocolos de comunicação.

III.2 O Modelo

III.2.1 Descrição

Na metodologia proposta em [2] um modelo de um sistema é composto por um conjunto de componentes chamados de objetos. As interações entre os objetos são feitas por meio de troca de mensagens. Cada objeto é uma entidade com um estado interno que pode ser alterado com o tempo. O estado de um objeto pode mudar como conseqüência de um evento gerado pelo próprio objeto ou de uma mensagem recebida de um outro objeto.

O estado de um objeto determina os tipos de eventos que podem ocorrer e as taxas de ocorrência destes eventos. Um evento em um objeto pode simplesmente mudar o estado interno deste objeto sem afetar nenhum outro, mas, em geral, também provocará alguma reação em outros objetos. Isto é modelado pela geração e envio

de mensagens que poderão provocar alguma mudança de estado do objeto receptor. Portanto, a especificação de um objeto inclui a definição dos eventos que ele pode gerar, as ações tomadas quando da ocorrência de um evento (e.g., envio de mensagens) e a descrição de como o objeto reage ao recebimento de mensagens.

O estado do sistema é dado pelo conjunto de estados dos objetos e pelo conjunto de mensagens ainda não entregues no sistema. As mensagens são uma abstração introduzida no modelo. Elas são entregues em tempo zero, portanto é nulo o tempo de reação do objeto a uma mensagem recebida. Desta forma alguns estados são transitórios e denominados de *evanescentes* (*vanishing*). Estados que possuem tempo de permanência positiva são denominados de *tangíveis* (*tangible*). Os estados *evanescentes* são aqueles com uma ou mais mensagens não entregues e estados *tangíveis* são aqueles com nenhuma mensagem em trânsito.

Para propósito de análise, apenas os estados *tangíveis* são considerados. Uma seqüência *evanescente* é uma seqüência de transições de estados que começa e termina em um estado *tangível* onde todos os estados intermediários são *evanescentes*. As ações que podem ocorrer em resposta a um evento ou a uma mensagem determinam conjuntos de seqüências *evanescentes*. Uma vez que todos os conjuntos de seqüências *evanescentes* são determinados, as taxas de transição para cada par de estados *tangíveis* podem ser facilmente achadas. Estas taxas formam a matriz de transição de estados.

Formalmente, um objeto θ é definido por uma 9-upla:

$$\theta \equiv (I, S, s_0, \varepsilon, M^r, M^s, R, \delta^t, \delta)$$

onde:

- I = nome do objeto;
- S = conjunto de possíveis estados do objeto;
- $s_0 \in S$ = estado inicial do objeto;

- ϵ = conjunto de eventos que podem ser gerados pelo objeto;
- M^r = conjunto de mensagens que podem ser recebidas pelo objeto;
- M^s = conjunto de mensagens que podem ser enviadas pelo objeto;
- R = a função taxa do objeto

$$R : S \times \epsilon \rightarrow \Re^+$$

dado um estado $s \in S$ e um evento $e \in \epsilon$, esta função fornece a taxa na qual o evento e ocorre no estado s;

- δ' = a função evento do objeto

$$\delta' : S \times \epsilon \rightarrow \{(0,1] \times S \times [M^s]\}$$

dado um estado s e um evento e, esta função retorna um conjunto de possíveis respostas, onde cada resposta é composta de um novo estado, uma lista ordenada de mensagens a serem entregues e a probabilidade desta resposta ocorrer;

- δ = função mensagem do objeto

$$\delta : S \times M^r \rightarrow \{(0,1] \times S \times [M^s]\}$$

dado um estado s e uma mensagem m, esta função retorna o conjunto de possíveis respostas, onde cada resposta consiste de um novo estado, uma lista ordenada de mensagens a serem entregues e a probabilidade desta resposta ocorrer.

Um exemplo do uso desta metodologia pode ser visto na figura III.1, onde os objetos são representados por retângulos e as trocas de mensagens são representadas por setas. Neste modelo de confiabilidade temos dois tipos de objetos: Componente e Unidade de Reparo. A falha de um componente é modelada por um evento *falha* do objeto Componente. Quando uma falha ocorre, Componente envia uma mensagem *falha* (mens(falha)) para Unidade de Reparo. O reparo de um componente é modelado por um evento *reparo* do objeto Unidade de Reparo. Quando um componente é reparado pela Unidade de Reparo, uma mensagem *reparado* (mens(reparado)) é enviada para Componente.

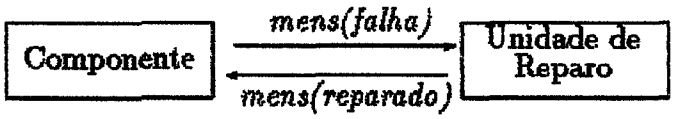

Figura III.1: Modelo orientado a objeto

III.2.2 Implementação

A ferramenta de software desenvolvida permite a especificação de modelos e geração da cadeia de Markov correspondente. Basicamente, a matriz de transição de estados é obtida pela geração dos estados alcançáveis a partir de um estado inicial e das regras que descrevem o comportamento do sistema. As regras fornecem as condições para que determinadas ações sejam tomadas a partir de certos estados de um objeto.

A ferramenta é organizada em quatro níveis como mostra a figura III.2.

1. o núcleo aceita a descrição do sistema em termos de objetos, eventos e mensagens. A partir desta descrição e de um estado inicial, o núcleo gera a matriz de transição de estado do sistema.

2. Objetos podem ter o mesmo comportamento *básico*, diferindo apenas no valor dos parâmetros especificados para cada objeto. No nível de definição do tipo de objeto, diferentes tipos de objetos são especificados. Um determinado modelo pode conter mais de uma ocorrência do mesmo tipo de objeto. Objetos diferentes mas do mesmo tipo terão parâmetros diferentes como por exemplo, taxas de eventos, valores de variáveis de condições, etc. Neste nível pode se criar uma biblioteca de tipos de objetos, a ser utilizada pelos níveis superiores para definir um modelo.

3. No nível de aplicação o usuário define um modelo criando objetos utilizando-se da biblioteca de tipos de objetos no nível abaixo e especificando os parâmetros necessários a cada objeto.

4. O nível de interface com o usuário permite que o usuário se abstraia dos

níveis anteriores. Uma linguagem de alto nível pode ser definida para se fazer uso dos objetos existentes talhados para um determinado domínio de aplicação. No capítulo IV descrevemos sucintamente a interface com o usuário da ferramenta desenvolvida.

Figura III.2: Níveis do Sistema

A linguagem escolhida para a implementação da ferramenta foi Prolog. São três os motivos principais da escolha de Prolog [2]: primeiro, Prolog não exige a especificação dos tipos de dados utilizados, permitindo inteira liberdade na descrição dos estados dos objetos; segundo, Prolog fornece *unificação*, uma forma poderosa de inicialização de variáveis e casamento de padrões usada na verificação das pré-condições; e terceiro, Prolog possui *backtrack*: quando mais de uma regra tem suas pré-condições satisfeitas ao mesmo tempo, todas elas são testadas para achar todos os estados alcançáveis.

Vejamos agora a descrição mais detalhada de cada um dos níveis da ferramenta acima mencionados.

Para gerar a cadeia de Markov, o núcleo executa basicamente oito passos:

1. Lista_Estados = (0, Estado_Global_Inicial);
2. mapeamento(0, Estado_Global_Inicial, Hash_Table);
3. Prox_Num_Estado = 1;
4. Lista_Estados = (Num_Estado_Inicial, Estado_Global).Tail;

5. para um objeto do sistema:
   (a) gera_ evento(Estado_Global, Novo_Estado_Global, Taxa);
   (b) mapeamento(Num_Estado_Final, Novo_Estado_Global, Hash_Table);
   (c) se Num_Estado_Final = Prox_Num_Estado então:
      - adiciona(Num_Estado_Final, Novo_Estado_Global, Lista_Estados);
      - Prox_Num_Estado := Prox_Num_Estado + 1;
   (d) grava_transicao(Num_Estado_Inicial, Num_Estado_Final, Taxa);
6. Lista_Estados = Tail;
7. se Lista_Estados $\neq$ [ ] vá para 4;
8. fim;

No primeiro passo o núcleo cria uma lista para guardar a descrição simbólica e a numeração dos estados da cadeia de Markov. Esta lista é criada apenas com o estado global inicial que possui numeração zero. O estado global inicial é composto pelo estado inicial de cada objeto e é fornecido pelo nível de aplicação. No segundo passo o núcleo armazena o estado global inicial em uma *hash table*. *Hash Table* é uma matriz com uma função de mapeamento associada: os estados globais são mapeados para uma posição da *hash table* e são então armazenados. Cada posição da *hash table* é composta por uma lista de estados, assim, mais de um estado pode ser armazenado em uma mesma posição. No passo três o núcleo especifica que o próximo estado global a ser gerado (Prox_Num_Estado) terá numeração um na cadeia de Markov. No passo quatro a lista de estados é instanciada com (Num_Estado_Inicial, Estado_Global).Tail, onde (Num_Estado_Inicial, Estado_Global) corresponde ao primeiro elemento da lista, e Tail corresponede ao resto da lista. Se a lista possuir apenas um elemento, Tail será igual a [ ] (vazia). No quinto passo o núcleo escolhe um objeto do sistema para gerar um evento. Este evento é gerado a partir do primeiro estado global da lista de estados. O evento gerado, as ações executadas pelo objeto e a taxa de ocorrência deste evento são especificados pelo nível de definição de tipos de objetos. O novo estado global gerado é então mapeado para uma posição da matriz de estados. A variável

Num_Final_Estado voltará com o número deste estado na cadeia de Markov. Se o estado não tinha sido anteriormente gravado na *hash table* (Num_Estado_Final = Prox_Num_Estado), o estado é adicionado a lista de estados e o número do próximo estado global a ser gerado é incrementado. A transicão é então gravada. Uma das utilidades do uso do Prolog pode ser verificada neste passo. Para o estado global especificado: (a) o objeto escolhido pode gerar mais de um evento; (b) mais de um objeto pode gerar eventos. Assim, na reavaliação das cláusulas (*backtrack*), todos os eventos de um objeto são gerados e todos os objetos do sistema são testados. No passo seis é retirado o primeiro elemento da lista de estados (todas as transições a partir deste estado já foram geradas pelo passo cinco). Se a nova lista de estados não for vazia o núcleo volta para executar o passo quatro.

A listagem do programa núcleo com alguns comentários se encontra no apêndice D.

Para o modelo de confiabilidade discutido na seção anterior teríamos, para os outros três níveis da ferramenta, a seguinte modelagem:

1. o nível de definição de tipos de objetos teria a descrição dos eventos *falha* e *reparo*, e das mensagens *falha* e *reparado*;
2. o nível de aplicação especificaria os parâmetros a serem lidos pelo nível de aplicação como taxa de falha e taxa de reparo;
3. o nível de interface com usuário seria o responsável pela geração do nível de aplicação do modelo.

Exemplos mais detalhados do uso desses três níveis da ferramenta são mostrados a seguir.

III.3 Exemplos

III.3.1 Modelos de Confiabilidade

No capítulo anterior foram mostradas várias ferramentas para análise de modelos de confiabilidade, como por exemplo, a ferramenta SAVE [1]. Vejamos,

nesta seção, como poderíamos implementar a definição de modelos de sistema usada em SAVE, utilizando a metodologia proposta por Berson *et al* [2].

Em SAVE, o sistema a ser modelado é visto como uma coleção de componentes interconectados de vários tipos. Cada componente pode ser classificado como operacional, de reserva, falho ou *dormant*. Um componente é operacional quando está participando das operações do sistema. Se o componente não participa destas operações, é dito: (a) de reserva, se pode substituir um componente do mesmo tipo quando houver uma falha; (b) falho, quando está a espera de um reparo feito por um *técnico de reparo*; (c) *dormant*, quando não está operacional porque um outro componente, de quem depende para operar, está falho, ou porque o sistema como um todo está falho. O sistema é dito *falho* quando não possue o número mínimo de componentes necessários para executar as suas operações. Taxas de falhas distintas podem ser definidas para os diferentes estados dos componentes (operacional, reserva, *dormant*). Cada componente pode falhar em diferentes modos, onde cada modo de falha possui uma taxa específica de reparo. Um componente ao falhar pode provocar falhas em outros componentes, e estes componentes afetados também irão precisar de reparo.

O componente da ferramenta SAVE pode ser modelado por um objeto que gera um evento, *falha*, e que pode receber duas mensagens, *falha* e *reparado*. Na primeira mensagem, o objeto é avisado da falha de um outro objeto, e pode verificar se é afetado por esta falha. Na segunda mensagem, o objeto é informado pela unidade de reparo que um dos seus componentes falhos já foi consertado. Assim, poderíamos ter a seguinte definição:

```
TIPO : componente_SAVE;
NOME : Nome_Objeto;
ESTADO : (Modo,Num_Falhas);
EVENTO falha: (Modo,Num_Falhas1) → (Modo,Num_Falhas2);
        CONDIÇÃO : comp(Nome_Objeto,Num_Comp),
                   calcula_total_falhas(Nome_Objeto,Total_Falhas),
                   Total_Falhas < Num_Comp,
                   calcula_taxa(Nome_Objeto,Modo,T),
                   T > 0;
        AÇÃO :     Num_Falhas2 := Num_Falhas1 + 1,
                   reparo(Nome_Objeto, Modo, Componente_Reparo),
                   envia_mensagem(mens("falha",Nome_Objeto,Modo,1), Componente_Reparo),
                   afeta_componentes(Nome_Objeto,Modo,Lista_de_Objetos_Afetados),
                   avisa_objetos("falha",Lista_de_Objetos_Afetados);
```

```
        TAXA :      T;
MENSAGEM ("reparado", Modo, Num): (Modo,Num_Falhas1) → (Modo,Num_Falhas2);
        AÇÃO : Num_Falhas2 := Num_Falhas1 - Num;
MENSAGEM ("falha",Modo,Num_Afetado,Prob_Afetar,Prob_Modo) :
        (Modo,Num_Falhas1) → (Modo,Num_Falhas2);
        CONDIÇÃO : comp(Nome_Objeto,Num_Comp),
                   total_falhas(Nome_Objeto,Total_Falhas),
                   Total_Falhas < Num_Comp;
        AÇÃO :     Num_Falhas2 = Num_Falhas1;
        PROBABILIDADE : 1 - (Prob_Afetar * Prob_Modo);
        CONDIÇÃO : comp(Nome_Objeto,Num_Comp),
                   total_falhas(Nome_Objeto,Total_Falhas),
                   Num_Afetado ≤ Num_Comp - Total_Falhas ;
        AÇÃO :     Num_Falhas2 := Num_Falhas1 + Num_Afetado,
                   reparo(Nome_Objeto, Modo, Componente_Reparo),
                   envia_mensagem(mens("falha",Nome_Objeto,Modo,Num_Afetado),
                      Componente_Reparo);
        PROBABILIDADE : Prob_Afetar * Prob_Modo;
        CONDIÇÃO : comp(Nome_Objeto,Num_Comp),
                   total_falhas(Nome_Objeto,Total_Falhas),
                   Num_Afetado > Num_Comp - Total_Falhas ;
        AÇÃO :     Num_Falhas2 = Num_Comp,
                   reparo(Nome_Objeto, Modo, Componente_Reparo),
                   envia_mensagem(mens("falha",Nome_Objeto,Modo,
                      Num_Comp - Total_Falhas), Componente_Reparo);
        PROBABILIDADE : Prob_Afetar * Prob_Modo;
```

O exemplo começa pela definição do tipo de objeto como sendo componente_SAVE. A cláusula NOME fornece o nome da variável a ser substituida pelo nome do objeto quando for instanciado. A cláusula ESTADO define os componentes que formam o estado do objeto. Neste exemplo, o estado é definido pelo modo de falha do objeto e pelo número de componentes falhos do modo correspondente (todos os modos de falha de um objeto são detectados quando da reavaliação das cláusulas). A cláusula CONDIÇÃO estabelece as condições sobre as quais o evento pode ocorrer e a cláusula AÇÃO descreve as ações do objeto quando o evento ocorre. COMP(Nome_Objeto,Num_Comp) é a leitura de um parâmetro do nível de aplicação que retorna em Num_Comp o número total de componentes do objeto Nome_Objeto. Calcula_total_falhas(Nome_Objeto,Total_Falhas) é uma função que soma as quantidades de componentes falhos nos diferentes modos do objeto Nome_Objeto e devolve o valor calculado em Total_Falhas. A função calcula_taxa(Nome_Objeto,Modo,T) calcula a taxa de falha T para um modo de falha do objeto:

```
T = (((número de componentes operacionais) x (taxa de falha quando operacional)) +
    ((número de componentes dormant)       x (taxa de falha quando dormant))    +
    ((número de componentes reservas)      x (taxa de falha quando reserva))) x
    (probabilidade de falha neste modo)
```

Para o evento *falha* ocorrer é necessário que exista pelo menos um componente sem falhas (Total_Falhas < Num_Comp) e que a taxa de falha calculada seja maior que zero (T > 0).

Quando o evento *falha* é gerado, o número de componentes falhos do modo correspondente é incrementado (Num_Falhas2 := Num_Falhas1 + 1) e uma mensagem é enviada para o objeto responsável pelo reparo e para os objetos que podem ser afetados por esta falha. A cláusula reparo(Nome_Objeto, Modo, Componente_Reparo) do nível de aplicação fornece o nome do objeto responsável pelo reparo de um determinado modo de falha do objeto. A cláusula afeta_componentes do nível de aplicação fornece a lista dos componentes que podem ser afetados pelo objeto quando há uma falha. Esta lista não só fornece o nome dos objetos que podem ser afetados, como também, o modo de falha do componente afetado, o número de componentes afetados, a probabilidade do objeto ser afetado pela falha e a probabilidade de falhar no modo especificado. Mais de uma cláusula afeta_componentes pode ser definida para um mesmo modo de falha de um objeto (estas cláusulas serão excludentes e serão detectadas quando for feita a reavaliação da cláusula afeta_componentes). A função avisa_objetos envia uma mensagem para cada componente da lista de objetos fornecida pela cláusula afeta_componentes. Podemos definir esta função como:

```
avisa_objetos("falha",[ ]).
avisa_objetos("falha",(Nome,Modo,Num_Afetado,Prob_Afetar,Prob_Modo).Tail) <-
    envia_mensagem(mens("falha",Modo,Num_Afetado,Prob_Afetar,Prob_Modo), Nome),
    avisa_objetos("falha",Tail).
```

Se a lista de componentes afetados for igual a [ ] (vazia), a função é instanciada com a primeira cláusula de avisa_objetos; senão, é instanciada com a segunda cláusula, onde (Nome, Modo, Num_Afetado, Prob_Afetar, Prob_Modo) é o primeiro elemento da lista, e Tail representa o resto da lista (Tail é igual a [ ] quando a lista possuir apenas um elemento). Uma mensagem é então enviada para o objeto que pode ser afetado e, recursivamente, a função avisa_objetos é chamada para avaliar o resto da lista.

Na primeira mensagem (*reparado*), o objeto é informado que componentes que estavam com defeitos já foram consertados, o número de componentes falhos

do modo de falha especificado na mensagem (Modo) é então decrementado (Num_Falha2 := Num_Falha1 - Num).

Na segunda mensagem (*falha*), o objeto é notificado da falha de componentes de um outro objeto. A mensagem também informa o modo de falha do componente afetado, o número de componentes que poderão ser afetados, a probabilidade do objeto ser afetado pela falha e a probabilidade do objeto falhar no modo especificado. Se o objeto possuir pelo menos um componente sem falha (Total_Falhas < Num_Comp), com probabilidade 1 - (Prob_Afetar * Prob_Modo) ele não é afetado, e com probabilidade Prob_Afetar * Prob_Modo ele perde componentes.

É interessante observar que foi definido, semelhante ao SAVE, que um objeto ao falhar, poderia afetar outros componentes, mas os componentes afetados não provocam falhas em outros objetos. Isto pode ser facilmente alterado pela inclusão, no final da definição da mensagem *falha*, das cláusulas afeta_componentes e avisa_objetos.

Em SAVE existem duas políticas para reparo de componentes: prioridade e ROS (Random Order Service). Na primeira estratégia, são especificadas prioridades distintas de reparo para diferentes tipos de objetos; e na segunda estratégia, é definida igual prioridade de reparo para todos os tipos de componentes. Estas duas políticas de reparo podem facilmente ser modeladas por um único objeto. Este objeto pode gerar um evento, *reparo*, e receber um tipo de mensagem, *falha*. O estado do objeto é representado por uma lista com os nomes dos objetos com falhas e seus respectivos modos de falhas e números de componentes falhos. O estado do objeto é [ ] quando nenhum componente está esperando reparo.

```
TIPO : servidor_prior;
NOME : Nome_Objeto;
ESTADO : Fila;
EVENTO reparo: Fila1 → Fila2;
        CONDIÇÃO : Fila1 ≠ [ ];
        AÇÃO :     compute(set, Prior, prior(Objeto,Prior), Lista_Prior),
                   seleciona_comp(Lista_Prior,Fila1,Num_Comp,Lista_Comp),
                   taxa_reparo(Componente,Modo,T),
                   pertence((Componente,Modo),Lista_Comp),
                   envia_mensagem(mens("reparado",Modo,1), Componente),
                   subtrai((Componente,Modo,1),Fila1,Fila2);
```

```
        TAXA :      (1 / Num_Comp) * T;
MENSAGEM ("falha", Comp_Falho, Modo_Falho, Num_Falho) : Fila1 — Fila2;
            AÇÃO : adiciona((Comp_Falho,Modo_Falho,Num_Falho),Fila1,Fila2);
            PROBABILIDADE : 1;
```

O evento *reparo* só ocorre quando há pelo menos um objeto com falha (Fila1 ≠ [ ]). Uma lista com as prioridades dos objetos do sistema (Lista_Prior) é então formatada pela função compute do VM Prolog: Lista_Prior possui todos os valores de Prior que satisfazem a cláusula prior(Objeto, Prior) do nível de aplicação, onde Objeto é o nome de um objeto e Prior é a prioridade deste objeto; o termo SET na função compute especifica que os valores da lista devem ser sem repetições e em ordem crescente. A função seleciona_comp(Prior_Lista, Fila1, Num_Comp, Lista_Comp) seleciona componentes em Fila1 para serem reparados, colocando-os em Lista_Comp e o número de diferentes objetos aptos para reparo em Num_Comp. Inicialmente são selecionados, entre os componentes com falha, os objetos que possuem maior prioridade. A prioridade de reparo para cada objeto é fornecida pelo nível de aplicação, onde a maior prioridade corresponde ao menor valor numérico. É então verificado se cada componente pode ser reparado, pois um outro componente de quem depende para ser consertado pode estar também esperando reparo (cada objeto possue, no nível de aplicação, uma cláusula que especifica os nomes dos objetos e as correspondentes quantidades necessárias para que o objeto em questão possa ser reparado). A verificação é feita por troca de mensagens entre a unidade de reparo e os objetos que precisam estar operacionais. Se nenhum objeto com uma determinada prioridade atende a estas especificações, uma prioridade menor é verificada. A cláusula taxa_reparo(Componente, Modo, T) fornece a taxa de reparo T para um objeto. A função pertence(Componente, Modo, Lista_Comp) verifica se (Componente, Modo) está em Lista_Comp, em caso afirmativo, com probabilidade 1 / Num_Comp, Componente é consertado e uma mensagem é enviada para este objeto. A função subtrai((Componente, Modo, 1), Lista_Comp) retira de Fila1 um componente do objeto consertado, e a fila resultante é instanciada com Fila2. Pela reavaliação da cláusula TAXA_REPARO todos os estado alcançáveis a partir de Fila1, num total de *Num_Comp* estados, são

gerados.

As funções seleciona_comp, pertence e subtrai acima comentadas podem ser facilmente implementadas como indicado abaixo.

Podemos definir a função seleciona_comp como:

```
seleciona_comp(Prior.Tail, Fila1, Num_Comp, Lista_Comp) <-
   prioridade(Prior, Fila1, Num_Comp, Lista_Comp),
   Num_Comp > 0.
seleciona_comp(Prior.Tail, Fila1, Num_Comp, Lista_Comp) <-
   seleciona_comp(Tail, Fila1, Num_Comp, Lista_Comp).

prioridade(*,[ ],0,[ ]).
prioridade(Prior, (Componente,Modo,Num).Tail, Num_Comp, (Componente,Modo).Novo_Tail) <-
   prior(Componente, Prior),
   reparo_depende(Componente, Lista_Reparo_Depende),
   verifica_reparo(Lista_Reparo_Depende),
   prioridade(Prior, Tail, Num_Comp_Ant, Novo_Tail),
   Num_Comp := Num_Comp_Ant + 1.
prioridade(Prior, (Componente,Modo,Num).Tail, Num_Comp, Lista_Comp) <-
   prioridade(Prior, Tail, Num_Comp, Lista_Comp),

verifica_reparo([ ]).
verifica_reparo((Componente,Num).Tail) <-
   num_comp_sem_falhas(Componente, Num_Comp),
   Num ≤ Num_Comp,
   verifica_reparo(Tail).
```

Na primeira cláusula, a função seleciona_comp chama a função prioridade para selecionar em Fila1, dada uma determinada prioridade de reparo (Prior), os objetos que podem ser reparados. A função prioridade fornece o número de diferentes objetos que podem ser reparados para a prioridade especificada (Num_Comp) e a lista com o nome e o modo de falha destes objetos (Lista_Comp). Se o número de objeto aptos para reparo for zero, a primeira cláusula de seleciona_comp *falha*, e a segunda cláusula avalia a próxima prioridade. A função prioridade verifica, para cada objeto da lista de objetos falhos, se o objeto em questão possui a prioridade especificada pela função seleciona_comp e se o objeto pode ser reparado. As cláusulas prior e reparo_depende da função prioridade são leituras do nível de aplicação. A primeira cláusula só sucede se o primeiro elemento da lista de objetos falhos tiver a prioridade especificada pela função seleciona_comp (Prior); e a segunda cláusula fornece uma lista com os objetos que devem estar operacionais para Componente poder ser reparado. Esta lista é analisada pela função

verifica_reparo: a função num_comp_sem falhas pergunta a cada objeto da lista, o número de componentes sem falhas que o objeto em questão possui.

A função pertence pode ser definida da seguinte maneira:

```
pertence({Componente,Modo}, {Componente,Modo,Num}.Tail).
pertence({Componente,Modo}, {Componente2,Modo2,Num2}.Tail} <-
    Componente ≠ Componente2 ou Modo ≠ Modo2,
    pertence({Componente,Modo}, Tail).
```

Se o componente e o modo especificado for igual ao primeiro elemento da lista de componentes afetados, a função é instanciada com a primeira cláusula da função pertence; senão, é instanciada com a segunda cláusula, e recursivamente, a função pertence é chamada para avaliar o resto da lista. Tail nas duas cláusulas representa o resto da lista dos objetos falhos (Tail é igual a [ ] quando a lista possui apenas um elemento). Se Tail for iqual a [ ] na segunda cláusula, implica que o elemento (Componente, Modo) não está na lista, e a função pertence *falha*.

A função subtrai pode ser definida como:

```
subtrai({Componente,Modo,Num}, {Componente,Modo,Num}.Tail, Tail).
subtrai({Componente,Modo,Num}, {Componente,Modo,Num2}.Tail, {Componente,Modo,Num2 - Num}.
        Tail) <-
    Num2 > Num.
subtrai({Componente,Modo,Num}, {Componente2,Modo2,Num2}.Tail, {Componente2,Modo2,Num2}.
        Novo_Tail} <-
    Componente ≠ Componente2 ou Modo ≠ Modo2,
    subtrai({Componente,Modo,Num}, Tail, Novo_Tail).
```

Na primeira cláusula, temos a retirada total de um componente da lista de componentes falhos. Na segunda cláusula, temos a diminuição do número de componentes falhos de um objeto. E na terceira cláusula, a função subtrai é chamada para avaliar o resto da lista.

Ao receber uma mensagem *falha*, o objeto do tipo servidor_prior adiciona à Fila1, o nome do objeto, o modo de falha e o número de componentes que falhou; a fila resultante é instanciada com Fila2. A função adiciona pode ter a seguinte definição:

```
adiciona({Componente,Modo,Num}, [ ], {Componente,Modo,Num}).
adiciona({Componente,Modo,Num}, {Componente,Modo,Num2}.Tail, {Componente,Modo,Num + Num2}.
        Tail).
adiciona({Componente,Modo,Num}, {Componente2,Modo2,Num2}.Tail, {Componente2,Modo2,Num2}.
        Novo_Tail) <-
   Componente ≠ Componente2 ou Modo ≠ Modo2,
   adiciona({Componente,Modo,Num}, Tail, Novo_Tail).
```

Como exemplo do uso dos dois tipos de objetos acima especificados, considere o modelo de confiabilidade descrito em [1] (fig. III.3):

> Um sistema consiste de três tipos de componentes: dois processadores, um *front end* e uma base de dados (fig. III.3). Os processadores podem falhar em dois modos distintos com probabilidade p e 1-p , respectivamente, onde taxas de reparo são diferentes nos diferentes modos. As falhas de um processador pode afetar o sistema em diferentes maneiras. No primeiro modo de falha, nenhum componente é afetado. No segundo modo de falha, a base de dados é afetada com probabilidade 1-c e um reparo torna-se necessário. A base de dados tem sua própria taxa de falha para falhas não provocadas por um processador. Para que a base de dados possa funcionar, ou possa ser reparada, pelo menos um processador deverá estar operacional. O sistema é considerado operacional quando pelo menos um componente de cada tipo está funcionando corretamente. Nenhum componente pode falhar se o sistema não estiver operacional. Componentes são reparados levando em consideração a seguinte prioridade: primeiro o *front end*, em segundo a base de dados e por último os processadores.

O sistema pode ser modelado com 4 objetos: PROCESSADOR, FRONT_END, BASE_DADOS e UNIDADE_DE_REPARO. Os três primeiros objetos são do tipo *componente_SAVE* e o quarto objeto é do tipo *servidor_prior*. Podemos, assim, fazer uso dos tipos de objetos discutidos anteriormente. O nível de aplicação para o exemplo acima descrito poderia ser:

```
TIPO(processador, componente_SAVE).
TIPO(front_end, componente_SAVE).
```

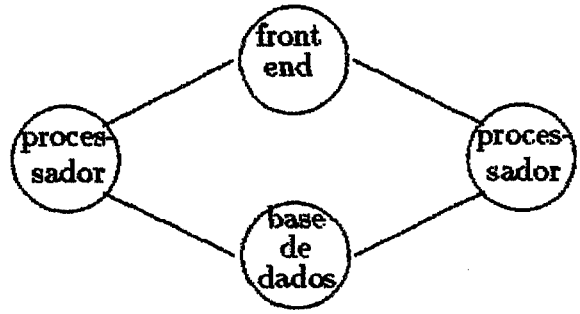

Figura III.3: Sistema de base de dados

```
TIPO(base_dados, componente_SAVE).
TIPO(unidade_de_reparo, servidor_prior).

INICIAL(processador,(1,0)).
INICIAL(processador,(2,0)).
INICIAL(front_end,(1,0)).
INICIAL(base_dados,(1,0)).
INICIAL(serviço_reparo,0).

COMP(processador,2).
COMP(front_end,1).
COMP(base_dados,1).

RESERVA(processador,0).
RESERVA(front_end,0).
RESERVA(base_dados,0).

DORMANT_QDO_SISTEMA_FALHO(processador,sim).
DORMANT_QDO_SISTEMA_FALHO(front_end,sim).
DORMANT_QDO_SISTEMA_FALHO(base_dados,sim).

TAXA_FALHA_QDO_DORMANT(processador,0).
TAXA_FALHA_QDO_DORMANT(front_end,0).
TAXA_FALHA_QDO_DORMANT(base_dados,0).

TAXA_FALHA_QDO_RESERVA(processador,0).
TAXA_FALHA_QDO_RESERVA(front_end,0).
TAXA_FALHA_QDO_RESERVA(base_dados,0).

TAXA_FALHA_QDO_OPERACIONAL(processador,pf).
TAXA_FALHA_QDO_OPERACIONAL(front_end,faf).
TAXA_FALHA_QDO_OPERACIONAL(base_dados,dbf).

TAXA_REPARO(processador,1,pr1).
TAXA_REPARO(processador,2,pr2).
TAXA_REPARO(front_end,1,far).
TAXA_REPARO(base_dados,1,dbr).

PROB_MODO(processador,1,p).
PROB_MODO(processador,2,1-p).
PROB_MODO(front_end,1,1).
PROB_MODO(base_dados,1,1).

AFETA_COMPONENTES(processador,1,[ ]).
AFETA_COMPONENTES(processador,2,[base_dados,1,1,1-c,1]).
AFETA_COMPONENTES(front_end,1,[ ]).
AFETA_COMPONENTES(base_dados,1,[ ]).

OPERAÇÃO_DEPENDE(processador,[ ]).
OPERAÇÃO_DEPENDE(front_end,[ ]).
OPERAÇÃO_DEPENDE(base_dados,[(processador,1)]).

REPARO_DEPENDE(processador,[ ]).
```

```
REPARO_DEPENDE(front_end,[ ]).
REPARO_DEPENDE(base_dados,[(processador,1)]).

PRIORIDADE(processador,3).
PRIORIDADE(front_end,1).
PRIORIDADE(base_dados,2).

REPARO(processador,1,serviço_reparo).
REPARO(processador,2,serviço_reparo).
REPARO(front_end,1,serviço_reparo).
REPARO(base_dados,1,serviço_reparo).

CONDIÇÃO_DE_OPERAÇÃO((processador,1),(front_end,1),(base_dados,1)).
```

A cláusula TIPO especifica o nome do objeto e o seu tipo correspondente. A cláusula INICIAL define o estado inicial de cada objeto (modos de falhas e número de componentes falhos para cada modo). A cláusula COMP determina o número total de componentes de cada objeto e a cláusula RESERVA especifica quantos desses componentes são de reservas. A cláusula DORMANT_QDO_SISTEMA_FALHO determina se o objeto deve ser considerado *dormant* quando o sistema estar falho, ou seja, se o objeto continua operacional, apesar do sistema não estar funcionando (isto é importante pois pode ter sido definido taxas de falhas distintas para quando o objeto estar operacional e para quando o objeto estar *dormant*). As cláusulas TAXA_FALHA_QDO_DORMANT, TAXA_FALHA_QDO_OPERACIONAL e TAXA_FALHA_QDO_RESERVA fornecem as taxas de falha para cada estado de funcionamento dos objetos. A taxa de reparo para cada modo de falha e a probabilidade de falha em cada modo de falha são dadas respectivamente por TAXA_REPARO e PROB_MODO. A cláusula AFETA_COMPONENTES especifica o componente que falhou, o modo de falha, e uma lista dos objetos que podem ser afetados por esta falha; para cada objeto desta lista é fornecido o nome do objeto, o modo de falha afetado, o número de componentes afetados, a probabilidade do componente ser afetado e a probabilidade da falha ser neste modo. As cláusulas OPERAÇÃO_DEPENDE e REPARO_DEPENDE definem as listas dos componentes de quem um objeto depende para ser considerado operacional ou pronto para reparo (quando vazias, estas listas são representadas por [ ]). A cláusula PRIORIDADE fornece a prioridade para reparo de cada objeto, onde a maior prioridade corresponde ao menor valor

numérico. A cláusula REPARO define o nome do objeto responsável pelo reparo de cada modo de falha. A cláusula CONDIÇÃO_DE_OPERAÇÃO fornece a lista dos objetos com os respectivos números mínimos de componentes para que o sistema seja considerado operacional. Se um modelo de confiabilidade possuir mais de uma condição de operabilidade, deve-se definir uma cláusula CONDIÇÃO_DE_OPERAÇÃO para cada uma dessas condições. Assim, similar ao SAVE (reliability block diagram) [16], expressões booleanas podem ser facilmente especificadas: uma cláusula CONDIÇÃO_DE_OPERAÇÃO é composta de várias expressões conectadas por AND, e as cláusulas CONDIÇÃO_DE_OPERAÇÃO são conectadas entre si por OR.

Vejamos agora a definição de um modelo de confiabilidade de uma rede de computadores Vaxes descrito em [17]. É importante observar que este modelo não poderia ser definido somente com a interface da ferramenta SAVE. Mostraremos então que o modelo pode ser descrito utilizando-se do objeto componente_SAVE definido acima e de outros objetos a serem definidos abaixo.

> O sistema do VAX é definido como um grupo (*cluster*) de N processadores. Cada processador pode estar *operacional* ou *falho*. Existem duas classes de falhas dos processadores: *permanente* e *intermitente*. Uma falha *permanente* requer um reparo físico do componente falho, e uma falha *intermitente* pode ser corrigida pela *reinicialização* do processador. Cada uma dessas classes de falhas dos processadores resulta em uma falha do sistema. Uma falha do sistema pode ser *coberta* ou *não coberta*. Uma falha *não coberta* torna o sistema não operacional até que ele seja *reinicializado*, e uma falha *coberta* torna o sistema não operacional por apenas um curto período de tempo até que ele seja *reconfigurado*. Falhas *permanentes* e *intermitentes* são exponencialmente distribuídas com médias $1/\lambda_P$ horas e $1/\lambda_I$ horas respectivamente. A *cobertura* para falhas *permanentes* é c e para falhas *intermitentes* é k. Quando uma falha *coberta* ocorre, todas as atividades dos processadores são suspensas e uma reconfiguração do sistema é inicializada se existe um número mínimo

suficiente de processadores para o sistema funcionar. A duração de uma reconfiguração do sistema para excluir ou reintegrar processadores é exponencialmente distribuida com média $1/\mu_R$ horas, o tempo para reparo de um processador com falha *permanente* é exponencialmente distribuido com média $1/\mu_P$, o tempo para reinicializar um processador com falha *intermitente* e *coberta* é exponencialmente distribuido com média $1/\mu_{PB}$ horas e o tempo para reinicializar o sistema quando uma falha *não coberta* ocorreu é exponencialmente distribuido com média $1/\mu_{CB}$. Processadores com falhas *intermitentes*, quando estão sendo reinicializados, podem sofrer falhas *permanentes*. Falhas que ocorrem quando o sistema não possue o número mínimo de processadores para funcionar, ou quando o sistema está sendo reconfigurado ou reinicializado, são permanentes e *não cobertas*. Processadores que sofreram falhas intermitentes são reinicializados em paralelo, enquanto que apenas um processador com falha permanente pode ser reparado por vez.

O sistema pode ser modelado por quatro objetos: PROCESSADOR, CONTROLADOR, FILA_DE_FALHAS_PERMANENTES e FILA_DE_FALHAS_INTERMITENTES (fig. III.4). PROCESSADOR e FILA_DE_FALHAS_PERMANENTES são módulos de *hardware*, enquanto os objetos CONTROLADOR e FILA_DE_FALHAS_INTERMITENTES são módulos de *software*. O objeto PROCESSADOR pode gerar um evento *falha* com taxa λ. Quando da ocorrência deste evento, uma mensagem *falha* é enviada ao CONTROLADOR. Com probabilidade p, a falha será considerada *permanente* pelo CONTROLADOR, e com probabilidade $1-p$, será considerada *intermitente*. Se a falha for *permanente*, com probabilidade c, o sistema será reconfigurado, e com probabilidade $1-c$, o sistema será reinicializado. Se a falha for *intermitente*, o sistema será reconfigurado com probabilidade k, e com probabilidade $1-k$, o sistema será reinicializado. As falhas *permanentes* (*intermitentes*) só são enviadas para FILA_DE_FALHAS_PERMANENTES (FILA_DE_FALHAS_INTERMITENTES), após o CONTROLADOR ter terminado a re-

configuração ou reinicialização do sistema, ou quando a ocorrência de uma falha tornar o sistema não operacional (número de processadores sem falhas não é suficiente para a execução das tarefas do sistema). Se falhas ocorrem quando o sistema está sendo reconfigurado ou reinicializado, o CONTROLADOR considera que a falha é permanente e *não coberta*, e o sistema é reinicializado. FILA_DE_FALHAS_PERMANENTES (FILA_DE_FALHAS_INTERMITENTES) possui taxa de reparo (reinicialização) igual a μ_P (μ_{PB}) para cada processador sendo reparado (reinicializado); após um processador ter sido reparado (reinicializado), uma mensagem *processador ok* é enviada ao CONTROLADOR. Ao receber uma mensagem *processador ok* de FILA_DE_FALHAS_PERMANENTES ou de FILA_DE_FALHAS_INTERMITENTES, se o sistema estiver operacional, o CONTROLADOR reconfigura o sistema, e depois envia uma mensagem *reparado* para PROCESSADOR; se o sistema estiver sendo reconfigurado ou reinicializado, a mensagem só é enviada após o sistema ter sido reinicializado ou reconfigurado; e se o sistema não possuir o número mínimo de processadores para operar, uma mensagem *reparado* é enviada para PROCESSADOR.

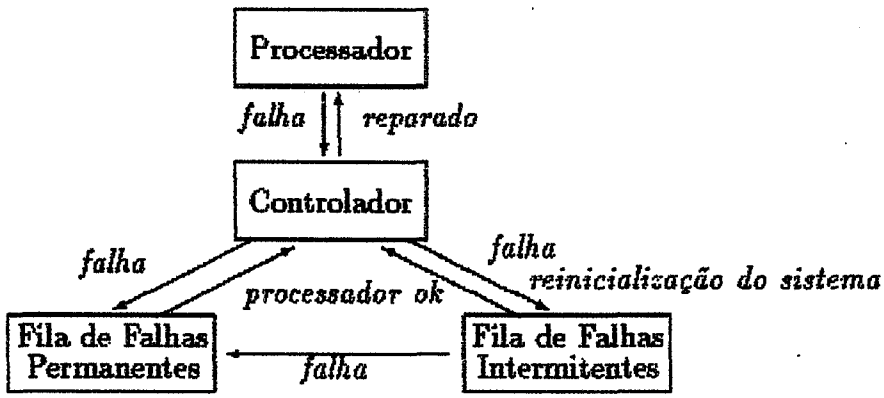

Figura III.4: Sistema VAX de processadores

O objeto PROCESSADOR pode ser definido como do tipo componente_SAVE. Este objeto terá apenas um modo de falha, e enviará a mensagem *falha* para o CONTROLADOR.

O objeto CONTROLADOR gera dois eventos, *reconfiguração* e *reinicialização*,

e pode receber duas mensagens, *falha* e *processador ok*. O estado deste objeto é representado pelo número de processadores a serem enviados para FILA_DE_FALHAS_PERMANENTES, pelo número de processadores a serem enviados para FILA_DE_FALHAS_INTERMITENTES, pelo número de processadores esperando pela reconfiguração do sistema para poderem voltar a operar, e pelas variáveis *reconf* e *reinic*. Estas duas variáveis possuem valores zero ou um. A variável *reconf* é um, quando o sistema está sendo reconfigurado, e é zero nos outros casos. A variável *reinic* é um, quando o sistema está sendo reinicializado, e é zero nos demais estados.

```
TIPO : controlador_sistema;
NOME : Nome_Objeto;
ESTADO : Num_Falhas_Perm, Num_Falhas_Interm, Num_Proc_OK, Reconf, Reinic;
EVENTO reconfiguração: (Num_Falhas_Perm1,Num_Falhas_Interm1,Num_Proc_OK1,Reconf1,Reinic1) →
                       (Num_Falhas_Perm2,Num_Falhas_Interm2,Num_Proc_OK2,Reconf2,Reinic2);
        CONDIÇÃO : Reconf1 = 1;
        AÇÃO :     taxa_reconfiguração_sistema(T),
                   Num_Falhas_Perm2 = 0,
                   Num_Falhas_Interm2 = 0,
                   Num_Proc_OK2 = 0,
                   Reconf2 = 0,
                   Reinic2 = 0,
                   tipo(Componente, "componente_SAVE"),
                   envia_mensagem(mens("reparado",1,Num_Proc_OK1), Componente),
                   tipo(Fila1, "fila_de_reparo"),
                   envia_mensagem(mens("falha",Num_Falhas_Perm1), Fila1),
                   tipo(Fila2, "fila_de_reinicialização"),
                   envia_mensagem(mens("falha",Num_Falhas_Interm1), Fila2);
        TAXA :     T;
EVENTO reinicialização: (Num_Falhas_Perm1,Num_Falhas_Interm1,Num_Proc_OK1,Reconf1,Reinic1) →
                        (Num_Falhas_Perm2,Num_Falhas_Interm2,Num_Proc_OK2,Reconf2,Reinic2);
        CONDIÇÃO : Reinic1 = 1;
        AÇÃO :     taxa_reinicialização_sistema(T),
                   Num_Falhas_Perm2 = 0,
                   Num_Falhas_Interm2 = 0,
                   Num_Proc_OK2 = 0,
                   Reconf2 = 0,
                   Reinic2 = 0,
                   tipo(Fila1, "fila_de_reinicialização"),
                   envia_mensagem(mens("reinicialização do sistema"), Fila1),
                   tipo(Componente, "componente_SAVE"),
                   envia_mensagem(mens("reparado",1,Num_Proc_OK1 + Num_Falhas_Interm1),
                      Componente),
                   tipo(Fila2, "fila_de_reparo"),
                   envia_mensagem(mens("falha",Num_Falhas_Perm1), Fila2);
        TAXA :     T;
MENSAGEM "falha" : (Num_Falhas_Perm1,Num_Falhas_Interm1,Num_Proc_OK1,Reconf1,Reinic1) →
                   (Num_Falhas_Perm2,Num_Falhas_Interm2,Num_Proc_OK2,Reconf2,Reinic2);
            CONDIÇÃO : estado_processador(Num_Processadores_Sem_Falhas),
                       Num_Min_Processadores(N_Min),
                       Num_Processadores_Sem_Falhas < N_min;
            AÇÃO :     Num_Falhas_Perm2 = 0,
                       Num_Falhas_Interm2 = 0,
                       Num_Proc_OK2 = 0,
                       Reconf2 = 0,
                       Reinic2 = 0,
                       tipo(Componente, "componente_SAVE"),
                       envia_mensagem(mens("reparado",1,Num_Proc_OK1), Componente),
                       tipo(Fila1, "fila_de_reparo"),
```

```
                    envia_mensagem(mens("falha",Num_Falhas_Perm1 + 1), Fila1),
                    tipo(Fila2, "fila_de_reinicialização"),
                    envia_mensagem(mens("falha",Num_Falhas_Interm1), Fila2),
                    /();
          PROBABILIDADE : 1;
          CONDIÇÃO : Reconf = 1 ou Reinic = 1;
          AÇÃO :     Num_Falhas_Perm2 = Num_Falhas_Perm1 + 1,
                     Num_Falhas_Interm2 = Num_Falhas_Interm1,
                     Num_Proc_OK2 = Num_Proc_OK1,
                     Reconf2 = 0,
                     Reinic2 = 1;
          PROBABILIDADE : 1;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0;
          AÇÃO :     prob_falha("permanente",Prob_Perm,Prob_Cobertura),
                     Num_Falhas_Perm2 = Num_Falhas_Perm1 + 1,
                     Num_Falhas_Interm2 = Num_Falhas_Interm1,
                     Num_Proc_OK2 = Num_Proc_OK1,
                     Reconf2 = 1,
                     Reinic2 = 0;
          PROBABILIDADE : Prob_Cobertura * Prob_Perm;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0;
          AÇÃO :     prob_falha("permanente",Prob_Perm,Prob_Cobertura),
                     Num_Falhas_Perm2 = Num_Falhas_Perm1 + 1,
                     Num_Falhas_Interm2 = Num_Falhas_Interm1,
                     Num_Proc_OK2 = Num_Proc_OK1,
                     Reconf2 = 0,
                     Reinic2 = 1;
          PROBABILIDADE : (1 - Prob_Cobertura) * Prob_Perm;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0;
          AÇÃO :     prob_falha("intermitente",Prob_Interm,Prob_Cobertura),
                     Num_Falhas_Perm2 = Num_Falhas_Perm1,
                     Num_Falhas_Interm2 = Num_Falhas_Interm1 + 1,
                     Num_Proc_OK2 = Num_Proc_OK1,
                     Reconf2 = 1,
                     Reinic2 = 0;
          PROBABILIDADE : Prob_Cobertura * Prob_Interm;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0;
          AÇÃO :     prob_falha("intermitente",Prob_Interm,Prob_Cobertura),
                     Num_Falhas_Perm2 = Num_Falhas_Perm1,
                     Num_Falhas_Interm2 = Num_Falhas_Interm1 + 1,
                     Num_Proc_OK2 = Num_Proc_OK1,
                     Reconf2 = 0,
                     Reinic2 = 1;
          PROBABILIDADE : (1 - Prob_Cobertura) * Prob_Interm;
MENSAGEM ("processadorok", Num) :
                     (Num_Falhas_Perm,Num_Falhas_Interm,Num_Proc_OK1,Reconf1,Reinic1) →
                     (Num_Falhas_Perm,Num_Falhas_Interm,Num_Proc_OK2,Reconf2,Reinic2);
          CONDIÇÃO Reconf = 1 ou Reinic = 1,
          AÇÃO :     Num_Proc_OK2 := Num_Proc_OK1 + Num,
                     Reconf2 = Reconf1,
                     Reinic2 = Reinic1;
          PROBABILIDADE : 1;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0,
                     estado_processador(Num_Processadores_Sem_Falhas),
                     Num_Min_Processadores(N_Min),
                     Num_Processadores_Sem_Falhas ≥ N_min;
          AÇÃO :     Num_Proc_OK2 = Num_Proc_OK1 + Num,
                     Reconf2 = 1,
                     Reinic2 = 0;
          PROBABILIDADE : 1;
          CONDIÇÃO : Reconf1 = 0,
                     Reinic1 = 0,
                     estado_processador(Num_Processadores_Sem_Falhas),
                     Num_Min_Processadores(N_Min),
```

```
                Num_Processadores_Sem_Falhas < N_min;
AÇÃO :          Num_Proc_OK2 = 0,
                Reconf2 = 0,
                Reinic2 = 0;
                tipo(Componente, "componente_SAVE");
                envia_mensagem(mens("reparado",1,Num), Componente);
PROBABILIDADE : 1;
```

Quando o evento *reconfiguração* (Reconf = 1) ou *reinicialização* (Reinic = 1) ocorre, três mensagens são enviadas: a primeira mensagem para o objeto PROCESSADOR, informando o número de processadores que foram reinicializados ou reparados; a segunda mensagem para a FILA_DE_FALHAS_PERMANENTES, informando o número de processadores que precisam de reparo; e a terceira mensagem para a FILA_DE_FALHAS_INTERMITENTES informando o número de processadores que precisam ser reinicializados.

Ao receber uma mensagem *falha*: (a) se não houver um número mínimo de processadores para o sistema funcionar (Num_Processadores_Sem_Falhas < N_min), o CONTROLADOR considera que a falha é permanente e *não coberta* e envia mensagens *falha* para as filas de reparo e reinicialização de processadores e mensagem *reparado* para o objeto PROCESSADOR; (b) existindo um número mínimo de processadores para o sistema funcionar, (b.1) se o sistema estiver sendo reinicializado ou reconfigurado (Reconf = 1 ou Reinic = 1), o CONTROLADOR adiciona esta falha à fila de falhas permanentes; (b.2) se o sistema não estiver nem sendo reconfigurado e nem sendo reinicializado (Reconf = 0 e Reinic = 0), com probabilidade Prob_Cobertura * Prob_Perm a falha será considerada permanente e *coberta*, ou com probabilidade (1 - Prob_Cobertura) * Prob_Perm a falha será permanente e *não coberta*, ou com probabilidade Prob_Cobertura * Prob_Interm a falha será intermitente e *coberta*, ou ainda com probabilidade (1 - Prob_Cobertura) * Prob_Interm a falha será intermitente e *não coberta*.

A função estado_processador verifica o número de processadores que estão operacionais. Esta função pode ser representada pela troca de mensagens entre CONTROLADOR e PROCESSADOR. A cláusula /() impede que sejam reavaliadas as cláusulas do mesmo predicado anteriores a /(). Assim, se

não houver um número mínimo de processadores para o sistema funcionar (Num_Processadores_Sem_Falhas < N_min), a primeira condição *sucede* e as outras condições não são avaliadas.

Ao receber uma mensagem *processador ok*: (a) se o sistema estiver sendo reconfigurado ou reinicializado (Reconf = 1 ou Reinic = 1), o CONTROLADOR adiciona a mensagem à fila de processadores esperando pela reconfiguração do sistema (Num_Proc_OK2 := Num_Proc_OK1 + 1); (b) se o sistema não estiver nem sendo reconfigurado e nem sendo reinicializado (Reconf = 0 e Reinic = 0) e o sistema possuir um número mínimo de processadores para funcionar, o CONTROLADOR inicia a reconfiguração do sistema para admitir este processador que foi reparado ou reinicializado; (c) se o sistema não estiver nem sendo reconfigurado e nem sendo reinicializado (Reconf = 0 e Reinic = 0) e não houver um número mínimo de processadores para o sistema funcionar (Num_Processadores_Sem_Falhas < N_min), o CONTROLADOR envia uma mensagem *reparado* para PROCESSADOR.

Note que, embora a descrição do objeto CONTROLADOR tenha sido feita em duas páginas, a descrição é modular em termos de condições e ações e pode ser facilmente verificada, com um conhecimento mínimo de Prolog.

FILA_DE_FALHAS_PERMANENTES gera um evento, *reparo*, e pode receber uma mensagem, *falha*. Processadores são reparados por ordem de ocorrência das falhas. O estado do objeto é representado pela fila dos processadores esperando reparo. Este objeto será definido na próxima seção, quando for discutido filas do tipo FCFS (First Come, First Serve).

FILA_DE_FALHAS_INTERMITENTES gera dois eventos, *reinicialização* e *falha*, e pode receber duas mensagens, *falha* e *reinicialização do sistema*. O estado deste objeto é representado pela fila dos processadores esperando serem reinicializados. Este objeto é idêntico a uma fila do tipo *servidor infinito*.

```
TIPO : fila_de_reinicialização;
NOME : Nome_Objeto;
ESTADO : Num_Comp_Falhas;
EVENTO reinicialização: Num_Comp_Falhos1 → Num_Comp_Falhos2;
        CONDIÇÃO : Num_Comp_Falhos1 > 0;
        AÇÃO :     taxa_reinicialização_processador(T),
```

```
                    Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1,
                    tipo(Objeto, "controlador_sistema"),
                    envia_mensagem(mens("processador ok",1),Objeto);
        TAXA :      Num_Comp_Falhos1 * T;
EVENTO falha: Num_Comp_Falhos1 → Num_Comp_Falhos2;
        CONDIÇÃO : Num_Comp_Falhos1 > 0;
        AÇÃO :     Taxa_falha_intermitente_para_falha_permanente(T),
                    Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1,
                    tipo(Objeto, "fila_de_reparo"),
                    envia_mensagem(mens("falha",1),Objeto);
        TAXA :      Num_Comp_Falhos1 * T;
MENSAGEM ("falha",Num) : Num_Comp_Falhos1 → Num_Comp_Falhos2;
          AÇÃO : Num_Comp_Falhos2 := Num_Comp_Falhos1 + Num;
          PROBABILIDADE : 1;
MENSAGEM ("reinicialização do sistema") : Num_Comp_Falhos1 → Num_Comp_Falhos2;
          AÇÃO : Num_Comp_Falhos2 = 0,
                 tipo(Objeto, "controlador_sistema"),
                 envia_mensagem(mens("processador ok", Num_Comp_Falhos1),Objeto);
          PROBABILIDADE : 1;
```

Quando o evento *reinicialização* ocorre, um processador é decrementado de FILA_DE_FALHAS_INTERMITENTES, e uma mensagem "processador ok" é enviada para o CONTROLADOR.

Quando o evento *falha* ocorre, um processador com falha intermitente é mandado para FILA_DE_FALHAS_PERMANENTES, isto se deve ao fato de que um processador que está sendo reinicializado pode sofrer uma falha permanente.

Ao receber uma mensagem *falha*, o objeto FILA_DE_FALHAS_INTERMITENTES adiciona à fila o número de componentes com falha.

A mensagem *reinicialização do sistema* informa a FILA_DE_FALHAS_INTERMITENTES que o sistema foi reinicializado, e portanto, todos os processadores que estão na fila, já foram reinicializados. O objeto então, envia para o CONTROLADOR uma mensagem "processador ok" com o número de processadores que estava na fila.

Para um sistema com dois processadores, onde pelo menos um processador precisa estar operacional para o sistema funcionar, poderíamos ter o seguinte nível de aplicação para o objeto PROCESSADOR do tipo componente_SAVE:

```
TIPO(processador, componente_SAVE).
INICIAL(processador,(1,0)).
COMP(processador,2).
RESERVA(processador,0).
DORMANT_QDO_SISTEMA_FALHO(processador,sim).
TAXA_FALHA_QDO_DORMANT(processador,0).
TAXA_FALHA_QDO_RESERVA(processador,0).
```

```
TAXA_FALHA_QDO_OPERACIONAL(processador,λ).
TAXA_REPARO(processador,1,μP).
PROB_MODO(processador,1,1).
OPERAÇÃO_DEPENDE(processador,[ ]).
REPARO_DEPENDE(processador,[ ]).
REPARO(processador, 1, controlador).
```

Para os outros objetos do sistema seria possível a seguinte definição do nível de aplicação:

```
TIPO(controlador, controlador_sistema).
TIPO(fila_falhas_permanentes, fila_de_reparo).
TIPO(fila_falhas_intermitentes, fila_de_reinicialização).
INICIAL(controlador,(0,0,0,0,0)).
INICIAL(fila_falhas_permanentes,0).
INICIAL(fila_falhas_intermitentes,0).
PROB_FALHA(permanente, p, c).
PROB_FALHA(intermitente, 1 − p, k).
TAXA_REINICIALIZAÇÃO_PROCESSADOR(μPB).
TAXA_REPARO_PROCESSADOR(μP).
TAXA_REINICIALIZAÇÃO_SISTEMA(μCB).
TAXA_RECONFIGURAÇÃO_SISTEMA(μR).
TAXA_FALHA_INTERMITENTE_PARA_FALHA_PERMANENTE(λp).
NUM_MIN_PROCESSADORES(1).
```

A cláusula PROB_FALHA fornece para cada tipo de falha (permanente ou intermitente) a probabilidade da falha ser do tipo especificado e a probabilidade da falha ser *coberta*. As cláusulas TAXA_REINICIALIZAÇÃO_PROCESSADOR, TAXA_REPARO_PROCESSADOR, TAXA_REINICIALIZAÇÃO_SISTEMA e TAXA_RECONFIGURAÇÃO_SISTEMA fornecem respectivamente, as taxas de reinicialização e de reparo dos processadores e as taxas de reinicialização e reconfiguração do sistema. A cláusula TAXA_FALHA_INTERMITENTE_PARA_FALHA_PERMANENTE especifica a taxa com que falhas intermitentes se transformam em falhas permanentes. E NUM_MIN_PROCESSADORES define o número mínimo de processadores que devem estar sem falhas para o sistema poder ser considerado operacional.

É interessante mostrar a rede de Petri estocástica para este exemplo [17] para que possamos ter uma comparação da descrição em rede de Petri estocástica com a descrição feita com a nossa ferramenta. Na figura III.5:

- P_{UP} = número de processadores operacionais;
- P_{PF} = ocorrência de uma falha permanente;

- P_{CPF} = a falha permanente é *coberta*;
- P_{UPF} = a falha permanente é *não coberta*;
- P_{PRP} = término de uma reinicialização do sistema e início de um reparo em um processador;
- P_{TIN} = término de um reparo ou reinicialização de um processador;
- P_{IF} = ocorrência de uma falha intermitente;
- P_{CIF} = a falha intermitente é *coberta*;
- P_{UIF} = a falha intermitente é *não coberta*;
- P_{IRB} = término da reconfiguração do sistema;

Apesar da representação gráfica do modelo ser a mesma para qualquer que seja o número de processadores do sistema, temos uma representação gráfica complexa e que praticamente *impedirá* o analista de determinar o comportamento do sistema pelo simples estudo da figura.

Objetivando simplificar a modelagem deste exemplo em redes de Petri, em [17] foi assumido que nenhuma falha podia ocorrer nos processadores quando o sistema estivesse sendo reconfigurado. Entretanto, esta suposição não corresponde ao modelo real do sistema VAX, e portanto, não foi aqui considerada.

III.3.2 Truncagem de Cadeias de Markov

Um sistema pode possuir centenas de milhares de estados. É interessante portanto que técnicas de redução do número de estados possam ser utilizadas. Nesta seção mostraremos a facilidade com que um método de redução de estados pode ser implementado a nível de definição de tipos de objetos. Como exemplo usaremos a técnica de agregação de estados proposta em [18] para o cálculo da disponibilidade de modelos de confiabilidade.

Os estados da cadeia de Markov de um modelo de confiabilidade representam, basicamente, o número de componentes que estão falhos (precisam de reparo) e o número de componentes que são operacionais. Em sistemas reais, na maior parte do tempo de operação, são poucos os componentes falhos. Assim, a

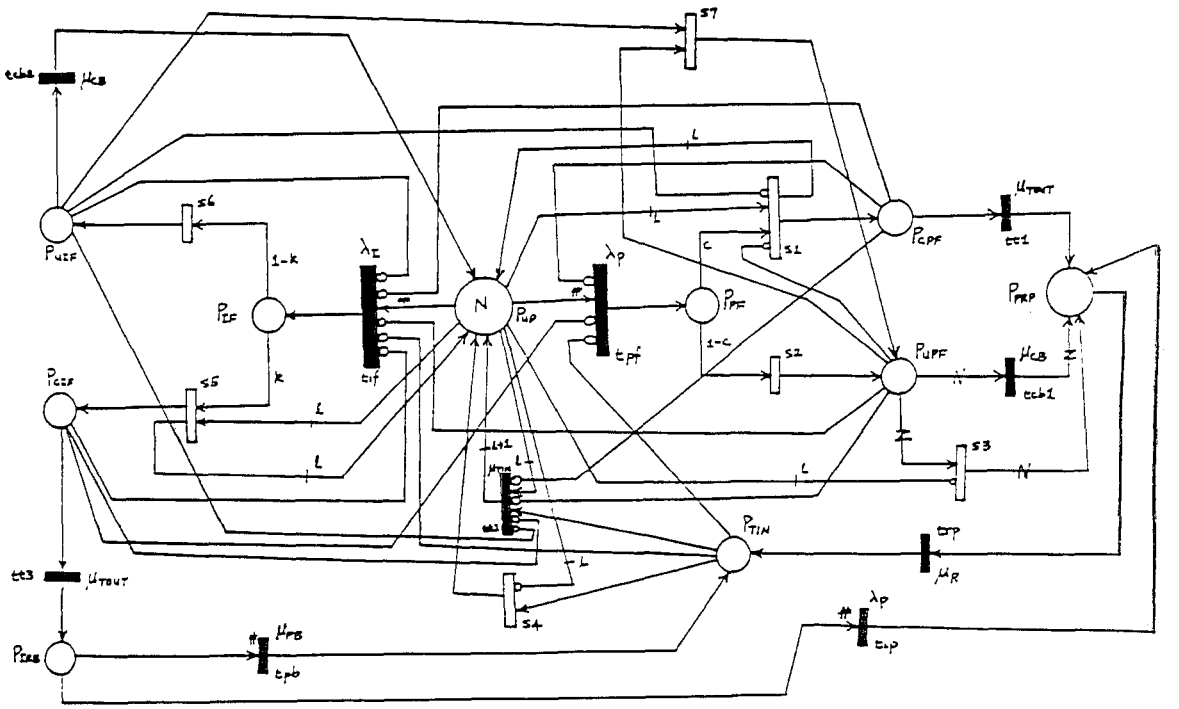

Figura III.5: Rede de Petri para o sistema VAX

probabilidade dos estados se concentra em um número pequeno de estados da cadeia de Markov. Partindo desta observação, Muntz *et al* [18] propõem que seja mantida a descrição detalhada dos estados com poucos componentes falhos e que os demais estados sejam representados de forma simplificada. Esta simplificação é obtida pela agregação dos estados que possuem pelo menos K falhas ($K \geq 0$). A partir da cadeia de Markov gerada são calculados os limites (inferior e superior) da disponibilidade do sistema.

O Modelo

Os estados de um modelo de confiabilidade podem ser classificados em *operacionais* ou *falhos*. Um estado é considerado *falho* quando o sistema não possui o número mínimo de componentes necessários para funcionar ou os componentes operacionais remanescentes não estão conectados de maneira a permitir o

funcionamento do sistema.

Seja F_i o conjunto de todos os estados que possuem i componentes com falhas. Podemos representar este conjunto como um único estado A_i que é chamado de *agregado* (estados não *agregados* são chamados de *detalhados*). A taxa de reparo para um estado *agregado* corresponde à menor taxa de reparo entre todos os objetos do sistema, e a taxa de falhas equivale à soma de todas as taxas de falhas dos estados que compõem o estado *agregado*.

Para a cadeia de Markov de um sistema com N componentes e cujos estados são agregados a partir da K-ésima falha, podemos ter a seguinte representação (fig. III.6):

$$g_0 = \{\bigcup_{i=0}^{F-1} F_i\}$$

$$g_1 = \{\bigcup_{i=F}^{K-1} F_i\}$$

$$g_2 = \{\bigcup_{i=K}^{N} F_i\}$$

onde $0 < F < K$.

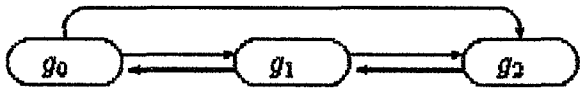

Figura III.6: Cadeia de Markov com matriz de transição g

A matriz g de transição para esta cadeia seria:

$$\begin{vmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & g_{21} & g_{22} \end{vmatrix}$$

Utilizando a metodologia proposta em [18], teríamos a cadeia de Markov mostrada na figura III.7. Basicamente, a nova cadeia possui dois conjuntos de estados, g'_{1d} e g'_{1a}, para representar g_1. O conjunto g'_{1d} possui a descrição detalhada dos estados que possuem de F a K-1 falhas, e g'_{1a} possui a representação

agregada para estes mesmos estados. Se $[\pi_0, \pi_1, \pi_2]$ for a solução de $\pi g = \pi$, e $[\pi'_0, \pi'_{1d}, \pi'_{1a}, \pi'_2]$ for a solução para $\pi' g' = \pi'$; então $\pi_0 = \pi'_0$, $\pi_1 = \pi'_{1d} + \pi'_{1a}$ e $\pi_2 = \pi'_2$. A explicação é simples. Suponha que o sistema inicia com zero componentes falhos (estado F_0). O sistema ficará em um dos estados de g'_0 e g'_{1d}, enquanto o número de falhas do sistema for menor que K. Quando K ou mais falhas ocorrerem, o estado do sistema será representado por um dos estados do conjunto g'_2. Quando o número de falhas voltar a ser menor que K, o sistema estará em g'_{1a} enquanto o número de falhas não for menor que F, quando então, o sistema voltará para g'_0. Observe que de um estado em g'_{1a} o sistema pode ir para mais de um estado em g'_0. Cada uma dessas transições forma um submodelo a ser resolvido da cadeia de Markov. Neste trabalho, tomaremos F = 1, e assim, apenas um submodelo precisará ser solucinado. O estudo de cadeias de Markov, quando F > 1, pode ser encontrado em [18].

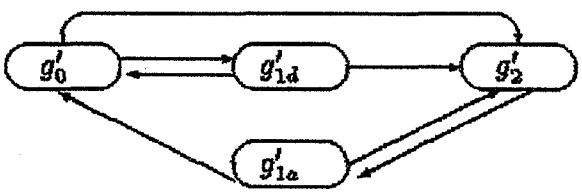

Figura III.7: Cadeia de Markov com matriz de transição g'

A matriz g' de transição para a cadeia de Markov gerada seria:

$$\begin{vmatrix} g_{00} & g_{01} & 0 & g_{02} \\ g_{10} & g_{11} & 0 & g_{12} \\ g_{10} & 0 & g_{11} & g_{12} \\ 0 & 0 & g_{21} & g_{22} \end{vmatrix}$$

Para um sistema com N componentes e cujos estados são *agregados* a partir da K-ésima falha temos dois conjuntos:

$$C_D = \bigcup_{i=0}^{K-1} F_i \quad e \quad C_A = \bigcup_{i=1}^{N} \{A_i\}$$

onde C_D reúne todos os estados cujas transições são representadas com detalhes e C_A é o conjunto de todos os estados *agregados*.

Seja $P(C_D)$ ($P(C_A)$) a probabilidade do estado estacionário de que o sistema esteja em C_D (C_A), e R_D (R_A) a *recompensa* dado que o sistema está em C_D (C_A). Então a disponibilidade pode ser definida como:

$$Disp = P(C_D)R_D + P(C_A)R_A$$

onde R_A é feito igual a zero para se obter o limite inferior da disponibilidade e é feito igual a 1 para se obter o limite superior. Assim,

$$P(C_D)R_D \leq Disp \leq P(C_D)R_D + P(C_A)$$

ou seja, o limite inferior equivale ao somatório das probabilidades estacionárias dos estados que são *detalhados* e *operacionais*, e o limite superior corresponde a este somatório acrescido das probabilidades estacionárias dos estados *agregados*. As provas dos resultados encontra-se em [18].

Implementação

Vejamos agora, como podemos implementar a técnica de agregação, proposta em [18], para o primeiro modelo de confiabilidade discutido na seção anterior. Este sistema é composto por objetos do tipo componente_SAVE e por uma unidade de reparo do tipo servidor_prior.

Visando minimizar as alterações necessárias nesta implementação, um novo objeto será adicionado ao modelo. Este objeto ficará responsável pelas transições dos estados agregados, e por isto será denominado de objeto agregador. O sistema passará a ter o seguinte funcionamento:

> Objetos do tipo componente_SAVE só poderão gerar eventos *falha*, quando o estado do sistema não for agregado. Ao receber uma mensagem *falha* de um componente_SAVE, a unidade de reparo retransmite a mensagem para o objeto agregador e continua a executar suas ações normalmente. Quando da chegada desta mensagem, o objeto agregador verifica se o número de falhas do sistema

já permite classificar o estado como agregado; em caso afirmativo, o objeto agregador transmite uma mensagem para todos os objetos do sistema, informando-os que o estado agora é agregado. A partir deste instante, nenhum evento é gerado pelos objetos do tipo componente_SAVE ou servidor_prior, até que o sistema volte a ter apenas componentes sem falhas (estado inicial).

Antes de um objeto do tipo componente_SAVE falhar ele deve verificar se o estado do sistema não é *agregado*. Esta alteração pode ser feita com a inclusão dos seguintes comandos no início da definição do evento *falha* :

```
EVENTO falha: (Modo,Num_Falhas1) → (Modo,Num_Falhas2);
        CONDIÇÃO : tipo(Objeto_Agregador,"objeto_agregrador"),
                   avalia_estado_global(Objeto_Agregador,Estado_Global),
                   /(),
                   Estado_Global ≠ "agregado",
                   ........
```

A cláusula tipo do nível de aplicação fornece o nome do objeto responsável pelas transições do estado *agregado*. A função avalia_estado_global verifica se o estado do sistema é *agregado*. Esta função pode ser representada pela troca de mensagens entre Nome_Objeto e Objeto_Agregador: o primeiro pede informação sobre o estado global e o segundo informa se é *detalhado* ou se é *agregado*. A cláusula /() impede que sejam reavaliadas as cláusulas do mesmo predicado anteriores a /(). Assim, se o estado for *agregado*, o evento *falha* não ocorre e não há tentativas de encontrar outra resposta. O resto do evento é idêntico ao descrito na seção anterior.

Uma mensagem também é adicionada à descrição do tipo componente_SAVE: *agregado*. Esta mensagem informa ao componente que o estado global do sistema é agregado, o objeto, então, faz o seu estado igual a zero. A explicação para isto é simples. De estados detalhados distintos, pode existir transições para um mesmo estado agregado. Para que estas transições sejam consideradas pelo *núcleo*, como transições para um mesmo estado, apenas uma representação global, para cada estado agregado, deve existir.

```
MENSAGEM {"agregado"} : (Modo,Num_Falhas1) — (Modo,Num_Falhas2);
        AÇÃO : Num_Falhas2 = 0;
```

O objeto do tipo servidor_prior passará a receber as seguintes mensagens:

```
MENSAGEM ("falha", Comp_Falho, Modo_Falho, Num_Falho) : Fila1 — Fila2;
            AÇÃO : adiciona((Comp_Falho,Modo_Falho,Num_Falho),Fila1,Fila2);
                   tipo(Objeto_Agregador,"objeto_agregador"),
                   envia_mensagem(mens("falha",Num), Componente);
MENSAGEM ("agregado") : Fila1 → Fila2
            AÇÃO : Fila2 = [ ];
```

Na primeira mensagem, o objeto adiciona a mensagem *falha* à Fila1, e avisa ao objeto agregador da ocorrência desta falha; e na segunda mensagem, o objeto faz seu estado iqual a [ ] (vazio), pois o estado do sistema é agregado.

O estado do objeto agregador é zero quando o estado global do sistema é *detalhado*, e é igual ao número total de falhas do sistema quando o estado é *agregado*. Este objeto pode gerar dois eventos, *falha* e *reparo*, e pode receber uma mensagem, *falha*. Poderíamos ter a seguinte definição para o objeto agregador:

```
TIPO : objeto_agregador;
NOME : Nome_Objeto;
ESTADO : Num_Comp_Falhas;
EVENTO falha: Num_Comp_Falhos1 → Num_Comp_Falhos2;
        CONDIÇÃO : Num_Comp_Falhos1 > 0,
                   calcula_total_comp(Total_Comp),
                   Num_Comp_Falhos1 < Total_Comp;
        AÇÃO :     prob_modo(Objeto,Modo,Prob),
                   calcula_taxa(Objeto,Modo,T),
                   Num_Comp_Falhos2 := Num_Comp_Falhos1 + 1,
                   afeta_componentes(Objeto,Modo,Lista_de_Objetos_Afetados),
                   avisa_objetos("falha",Lista_de_Objetos_Afetados),
                   compute(set, Componente,tipo(Componente,*),Lista_Objetos),
                   avisa_objetos("agregado",Lista_Objetos);
        TAXA :     T;
EVENTO reparo: Num_Comp_Falhos1 → Num_Comp_Falhos2;
        CONDIÇÃO : Num_Comp_Falhos1 > 0;
        AÇÃO :     verifica_menor_taxa_reparo(Menor_Taxa_Reparo),
                   Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1;
        TAXA :     Menor_Taxa_Reparo;
MENSAGEM ("falha",Num) : Num_Comp_Falhos1 → Num_Comp_Falhos2;
          CONDIÇÃO : Num_Comp_Falhos1 = 0,
                     estado_agregado(K),
                     calcula_total_falhas(Total_Falhas),
                     K ≤ Total_Falhas;
          AÇÃO :     Num_Comp_Falhos2 = Total_Falhas,
                     compute(set,tipo(Objeto,*),Lista_Objetos),
                     avisa_objetos("agregado",Lista_Objetos);
          CONDIÇÃO : Num_Comp_Falhos1 > 0;
          AÇÃO :     Num_Comp_Falhos2 := Num_Comp_Falhos1 + Num;
```

O evento *falha* só ocorre quando o estado do sistema é agregado (Num_Comp_Falho1 > 0) e o número total de componentes do sistema é menor que o número total de componentes com falha (Num_Comp_Falhos1 < Total_Comp). O

número total de componentes do sistema (Total_Comp) é fornecido pela função calcula_total_comp que é apenas a leitura das cláusulas COMP do nível de aplicação. Para calcular todas as possíveis transições de um estado agregado, em decorrência de falhas dos componentes do sistema, foi utilizado o seguinte artifício: o objeto agregador seleciona um objeto do tipo componente_SAVE e um modo de falha deste objeto (na reavaliação das cláusulas todos os objetos e modos de falhas são testados), mensagens são então enviadas para todos os objetos que podem ser afetados por uma falha do objeto selecionado; se algum objeto é afetado, a unidade de reparo avisará o objeto agregador. A cláusula prob_modo seleciona um objeto e um modo de falha deste objeto. A cláusula calcula_taxa calcula a taxa de falha para o modo de falha do objeto selecionado; o cálculo desta taxa é idêntico ao discutido na seção anterior, considerando que o objeto em questão não possui componentes falhos (isto se deve ao fato de não ser possível definir o número de componentes falhos de um objeto em um estado agregado). A cláusula afeta_componentes do nível de aplicação fornece os nomes dos objetos que podem ser afetados pela falha, e a cláusula avisa_componentes envia mensagens *falha* para estes objetos. COMPUTE é uma função do VM Prolog que gera uma lista com os nomes de todos os objetos que estão definidos na cláusula *tipo* do nível de aplicação. A partir desta lista, o objeto agregador informa a todos os objetos do sistema que o estado é agregado (estado igual a zero para objetos do tipo componente_SAVE e estado igual a [ ] para objetos do tipo servidor_prior).

O evento *reparo* só ocorre quando o estado global do sistema é agregado (Num_Comp_Falhos1 > 0). A função verifica_menor_taxa_reparo lê as cláusulas TAXA_REPARO de todos os objetos, e seleciona a taxa de reparo de menor valor para a taxa de ocorrência do evento.

Se a mensagem *falha* é recebida quando o estado do sistema ainda é considerado *detalhado* (Num_Comp_Falhos1 = 0), o objeto agregador verifica se com a nova falha o estado do sistema passa a ser agregado; em caso afirmativo,todos os objetos do sistema são informados. Se a mensagem é recebida quando o estado é agregado (Num_Comp_Falhos1 > 0), apenas o número de novas falhas

é adicionada ao estado do objeto agregador. Esta segunda condição ocorre quando o objeto agregador simula uma falha para calcular as transições dos estados agregados.

Para o nível de aplicação seria necessária apenas a inclusão de mais duas cláusulas:

```
TIPO(objeto_agregador, objeto_agregador).
ESTADO_AGREGADO(2).
```

A primeira cláusula informa o nome do objeto responsável pelas transições dos estados agregados, e a segunda cláusula estabelece que os estados serão agregados a partir da segunda falha.

III.3.3 Redes de Filas

Nesta seção será mostrado como filas e os mecanismos de comunicação entre filas podem ser modelados. Considere uma fila como uma coleção de tarefas, onde cada tarefa corresponde a uma ou mais unidades de trabalho a ser executada por um servidor. Cada fila é composta por dois elementos: 1) um ou mais servidores e 2) uma política de atendimento que especifica qual a próxima tarefa a ser atendida. As tarefas são organizadas em classes que definem o caminho a ser por elas seguido após deixarem uma determinada fila e a quantidade de trabalho a ser realizada durante o serviço. Uma explicação mais detalhada de cada um desses elementos será visto nos exemplos a seguir.

Exemplos

Vejamos inicialmente a definição de filas FCFS (First Come, First Serve): tarefas são processadas na ordem de chegada; uma tarefa só é atendida, quando todas as tarefas que chegaram antes, já não estiverem na fila.

A fila FCFS pode ser modelada por um objeto que pode gerar um evento, *processa*, e pode receber uma mensagem, *tarefa*. Quando o evento *processa* ocorre, uma tarefa é retirada da fila e atendida pelo servidor. Quando uma mensagem *tarefa* é recebida, a mensagem é colocada no final da fila.

```
TIPO : servidor_FCFS;
NOME : Nome_Objeto;
ESTADO : Fila;
EVENTO processa: Fila1 → Fila2;
        CONDIÇÃO : Fila1 = Classe_Inicial.Tail;
        AÇÃO :     taxa(Nome_Objeto, Classe_Inicial, T),
                   capacidade(Nome_Objeto, Capacidade),
                   Fila2 = Tail,
                   caminho(Nome_Objeto, Classe_Inicial, Prob, Destino, Classe_Final),
                   envia_mensagem(mens(tarefa, Classe_Final), Destino);
        TAXA :     Prob * (Capacidade / T);
MENSAGEM (tarefa, Classe): Fila1 → Fila2;
          AÇÃO : adiciona_tarefa(Fila1, Classe, Fila2);
          PROBABILIDADE : 1;
```

No VM Prolog uma fila é definida por um conjunto de elementos separados por ponto (.). Se o estado inicial do objeto (Fila1) poder ser instanciado (igualado) por Classe_Inicial.Tail, Classe_Inicial será o nome do primeiro elemento da fila, e Tail representará o resto da fila (quando a fila possui apenas um elemento, Tail será [ ]). Se Fila1 = [ ] (a fila esta vazia), a condição falha, e o evento não ocorre. A cláusula taxa(Nome_Objeto, Classe_Inicial, T) fornece a taxa de serviço (unidades de trabalho / tarefa) para a classe. A cláusula capacidade(Nome_Objeto, Capacidade) define a capacidade (unidades de trabalho / unidade de tempo) do servidor da fila. A cláusula caminho(Nome_Objeto, Classe_Inicial, Prob, Destino, Classe_Final) do nível de aplicação fornece o nome de um objeto (Destino) com quem a fila está conectada, a probabilidade da tarefa com classe *CLASSE_INICIAL* ser enviada para este objeto (Prob), e a classe da tarefa ao chegar na próxima fila (Classe_Final). Esta cláusula, na realidade, estabelece as ligações entre as filas, e portanto, pode existir mais de uma cláusula para uma mesma fila. Na reavaliação das cláusulas (*backtrack*), todos os caminhos que uma tarefa pode pecorrer são detectados.

Quando uma mensagem *tarefa* é recebida, o objeto coloca a mensagem no final de Fila1: a função *adiciona* junta Fila1 com Classe, e o resultado é colocado em Fila2. A função adiciona_tarefa pode ser definida como:

```
adiciona_tarefa(Classe, [ ], Classe).
adiciona_tarefa(Classe, Classe2.Tail2, Classe2.Novo_Tail) <-
   adiciona_tarefa(Classe, Tail2, Novo_Tail).
```

É interessante observar a facilidade com que uma fila FCFS foi definida. Isto não é verdadeiro para algumas interfaces de modelagem, como por exemplo, redes de Petri.

Vejamos agora como poderíamos implementar filas que possuem diferentes níveis de prioridade. Considere o seguinte exemplo de uma fila de prioridade não preemptiva: uma fila possue N níveis de prioridades numerados de zero a N - 1; quanto menor o número de um nível, maior é a prioridade de atendimento das tarefas que estão neste nível; tarefas dentro de um mesmo nível são atendidas por ordem de chegada.

Similar à fila FCFS, esta fila poderia ser modelada por um objeto que pode gerar um evento, *processa*, e pode receber uma mensagem, *tarefa*. O estado deste objeto seria representado pela fila das tarefas ordenada por prioridade e por ordem de chegada. Quando o evento *processa* ocorre, a primeira tarefa da fila é retirada e atendida pelo servidor. Quando uma mensagem *tarefa* é recebida, a tarefa correspondente é colocada antes da primeira tarefa com menor prioridade.

```
TIPO : servidor_níveis_de_prioridades;
NOME : Nome_Objeto;
ESTADO : Fila;
EVENTO processa: Fila1 → Fila2;
        CONDIÇÃO : Fila1 ≡ (Classe_Inicial, Prior).Tail;
        AÇÃO :     taxa(Nome_Objeto, Classe_Inicial, T),
                   capacidade(Nome_Objeto, Capacidade),
                   Fila2 ≡ Tail,
                   caminho(Nome_Objeto, Classe_Inicial, Prob, Destino, Classe_Final),
                   envia_mensagem(mens(tarefa, Classe_Final), Destino);
        TAXA :     Prob * (Capacidade / T);
MENSAGEM (tarefa, Classe) : Fila1 → Fila2;
          AÇÃO : mapeamento(Classe, Prior),
                 coloca_na_fila((Classe, Prior), Fila1, Fila2);
          PROBABILIDADE : 1;
```

A cláusula mapeamento(Classe, Prior) do nível de aplicação associa às tarefas de uma determinada classe um nível de prioridade. A função coloca_na_fila pode ser definida como:

```
coloca_na_fila((Classe, Prior), [ ], (Classe, Prior)).
coloca_na_fila((Classe, Prior), (Classe2, Prior2).Tail2, (Classe, Prior).(Classe2, Prior2).Tail2) <-
   Prior < Prior2.
coloca_na_fila((Classe, Prior), (Classe2, Prior2).Tail2, (Classe2, Prior2).Novo_Tail) <-
   Prior ≥ Prior2,
   coloca_na_fila((Classe, Prior), Tail2, Novo_Tail).
```

Observe que a descrição para o tipo de objeto servidor_níveis_de_prioridades é a mesma para qualquer que seja o número de níveis de prioridade. Os níveis de prioridades da fila serão definidos pela cláusula MAPEAMENTO do nível de aplicação.

Considere o modelo de redes de filas mostrado na figura III.8. Este exemplo possui dois tipos de filas: uma fila atende as tarefas por ordem de chegada, enquanto a outra fila possui dois níveis de prioridade (níveis zero e um). Utilizando os tipos de objetos acima definidos poderíamos ter o seguinte nível de aplicação para este modelo:

```
TIPO(servidor1,servidor_FCFS).
TIPO(servidor2,servidor_níveis_de_prioridades).

INICIAL(servidor1,[ ]).
INICIAL(servidor2,[(classe_A, 0).(classe_B, 1).(classe_B, 1)]).

TAXA(servidor1, classe_A, μ1 ).
TAXA(servidor1, classe_B, μ2 ).
TAXA(servidor2, classe_A, μ3 ).
TAXA(servidor2, classe_B, μ4 ).

CAPACIDADE(servidor1, c1 ).
CAPACIDADE(servidor2, c2 ).

CAMINHO(servidor1, classe_A, 1, servidor2, classe_A).
CAMINHO(servidor1, classe_B, 1, servidor2, classe_B).
CAMINHO(servidor2, classe_A, 1, servidor1, classe_A).
CAMINHO(servidor2, classe_B, 1, servidor1, classe_B).

MAPEAMENTO(classe_A, 0).
MAPEAMENTO(classe_B, 1).
```

A cláusula INICIAL define que a fila do tipo servidor_FCFS está inicialmente vazia, enquanto a fila do tipo servidor_níveis_de_prioridades possui uma tarefa com prioridade zero (maior prioridade) e duas tarefas com prioridade um.

É importante observar que primitivas de modelação para redes de filas, utilizadas por outras ferramentas, podem facilmente ser definidas. Por exemplo, vejamos a descrição do *nó de alocação* usada na ferramenta RESQ [14]: uma tarefa espera em um nó de alocação até conseguir um determinado número de fichas (recursos). Esta primitiva pode ser modelada por um objeto cujo estado é representado pelo número de fichas que possui e pela fila das tarefas a espera de alocar uma ficha. Duas mensagens podem ser recebidas pelo objeto: *Tarefa* e *ficha*. Quando o objeto recebe uma mensagem *Tarefa*, se possuir pelo menos

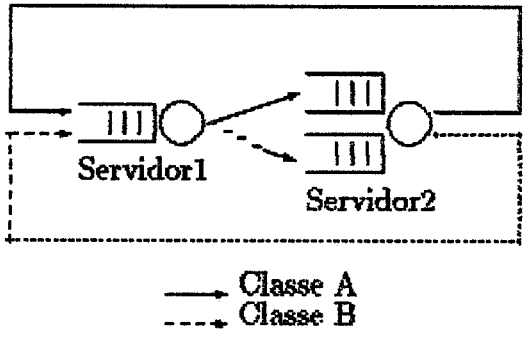

Figura III.8: Um exemplo de redes de filas

uma ficha, o número de fichas do objeto é decrementado e a tarefa é despachada; se o objeto não possuir nenhuma ficha, a tarefa é adicionada à fila. Ao receber uma mensagem *ficha*, se houver alguma tarefa esperando na fila, uma tarefa é depachada; se a fila de espera por ficha estiver vazia, o número de fichas do objeto é incrementado.

```
TIPO : no_de_alocação;
NOME : Nome_Objeto;
ESTADO : Num_Fichas, Fila;
MENSAGEM "ficha": (Num_Fichas1, Fila1) → (Num_Fichas2, Fila2);
            CONDIÇÃO : Fila1 = [ ];
            AÇÃO : Num_Fichas2 := Num_Fichas1 + 1,
                   Fila2 = [ ];
            PROBABILIDADE : 1;
            CONDIÇÃO : Fila1 = (Tarefa, Classe_Inicial).Tail;
            AÇÃO : Num_Fichas2 = 0,
                   Fila2 = Tail,
                   caminho(Nome_Objeto, Classe_Inicial, Prob, Destino, Classe_Final),
                   envia_mensagem(mens(Tarefa, Classe_Final), Destino);
            PROBABILIDADE : Prob;
MENSAGEM (Tarefa, Classe_Inicial): (Num_Fichas1, Fila1) → (Num_Fichas2, Fila2);
            CONDIÇÃO : Num_Fichas1 = 0;
            AÇÃO : Num_Fichas2 = 0,
                   adiciona_tarefa(Fila1, (Tarefa, Classe_Inicial), Fila2);
            PROBABILIDADE : 1;
            CONDIÇÃO : Num_Fichas1 > 0;
            AÇÃO : Num_Fichas2 := Num_Fichas1 - 1,
                   Fila2 = [ ],
                   caminho(Nome_Objeto, Classe_Inicial, Prob, Destino, Classe_Final),
                   envia_mensagem(mens(Tarefa, Classe_Final), Destino);
            PROBABILIDADE : Prob;
```

Observe que não foi definido, para o *nó de alocação*, o número de fichas que cada tarefa deseja alocar (foi assumido que cada tarefa precisa de apenas uma ficha). Entretanto, esta é uma modificação facilmente implementada: na

mensagem *tarefa* pode-se incluir um parâmetro com o número de fichas a serem alocadas, e na mensagem *ficha* pode-se definir uma política de atendimento para quando existem tarefas esperando na fila (Num_Tarefas1 > 0) e não existe um número suficiente de fichas para atender a tarefa.

A descrição do *nó de liberação* utilizada em RESQ [14] é também bastante simples. O objeto recebe apenas uma mensagem, *Tarefa*. Quando do recebimento desta mensagem, uma mensagem *ficha* é enviada para o nó de alocação e a tarefa é despachada. O estado deste objeto não muda, e pode ser representado por [ ].

```
TIPO : no_de_liberação;
NOME : Nome_Objeto;
ESTADO : [];
MENSAGEM (Tarefa, Classe_Inicial): [] → [];
            AÇÃO : caminho(Nome_Objeto, "ficha", *, Destino1, *),
                   envia_mensagem(mens("ficha"), Destino1),
                   caminho(Nome_Objeto, Classe_Inicial, Prob, Destino2, Classe_Final),
                   envia_mensagem(mens(Tarefa, Classe_Final), Destino2);
            PROBABILIDADE : Prob;
```

O asterisco (*) em VM Prolog é um termo *anônimo* que instancia com qualquer valor. Este termo foi acima utilizado na cláusula caminho, para substituir os parâmetro que não são utilizados pela função envia_mensagem.

III.3.4 Protocolos de Comunicação

A maioria das arquiteturas de redes de comunicação existentes são organizadas em um conjunto de níveis [19] como mostrado na figura III.9. O número de níveis, e o nome e a função de cada nível são, em geral, diferentes para arquiteturas distintas. Entretanto, em todas as redes, o objetivo de cada nível é oferecer serviço aos níveis que lhe são superiores. O nível n de um equipamento só troca informações com o nível n de um outro equipamento. As regras e convenções usadas nesta conversação são conhecidas por *protocolos*. Na realidade, nenhum dado é transmitido diretamente do nível n de um equipamento para o nível n de um outro equipamento (exceto no nível inferior). Uma troca de dados entre dois equipamentos sempre é inicializada pelo nível superior de um dos equipamentos. Cada nível, então, passa os dados para o

nível imediatamente abaixo, até que a informação a ser transmitida alcance o nível inferior. Só então, há uma comunicação *física* com o outro equipamento (nos outros níveis a comunicação é virtual). Ao receber uma mensagem, um nível passa a informação para o nível que lhe é imediatamente superior, até que a informação recebida alcance o nível superior do equipamento.

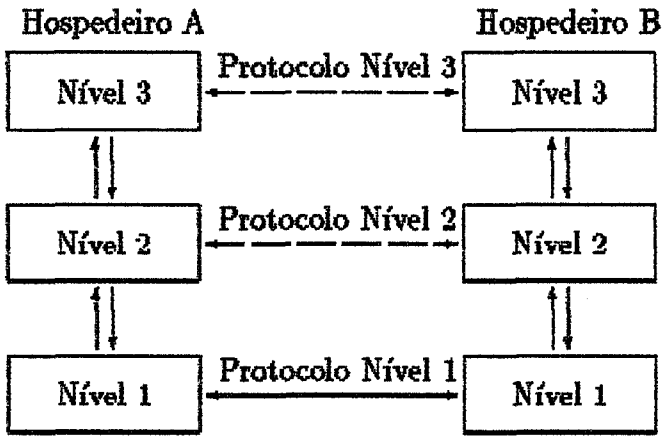

Figura III.9: Arquitetura de redes de comunicação

Muitos fatores influenciam na eficácia de um protocolo, como por exemplo, tempo para uma mensagem chegar a seu destino, erro de transmissão do canal, capacidade do buffer do equipamento receptor, tempo de espera da confirmação do recebimento da mensagem (ack), etc. Nesta seção, mostraremos a facilidade de modelar,estes e outros fatores, utilizados na definição dos protocolos bit alternante [7], go back n [19] e abracadabra [20]. Apesar do bit alternante ser um caso particular do go back n, ele será discutido inicialmente, por ser um exemplo bastante simples.

Bit Alternante

Considere a definição do protocolo do bit alternante utilizada em [7]:

> Mensagens chegam exponencialmente com média $1/\lambda$ no emissor que possui capacidade de guardar apenas uma mensagem. Mensagens

que chegam quando o buffer está ocupado são descartadas. Junto à cada mensagem transmitida é incorporado um bit 0 ou 1. A uma mensagem transmitida *pela primeira vez* é incorporado um bit de valor diferente da última mensagem transmitida. Ao receber uma mensagem o receptor envia a confirmação (ack) e quando da chegada do ack o emissor libera o buffer. Os canais podem, com uma certa probabilidade, transmitir com erro. Mensagens ou acks com erro de transmissão são descartados.

O sistema pode ser modelado por quatro objetos: EMISSOR, RECEPTOR e dois canais, CANAL1 que leva mensagens do emissor ao receptor e CANAL2 que leva acks do receptor ao emissor. Os objetos CANAL1 e CANAL2 representam um canal *full duplex*, onde mensagens são transmitidas em ambas as direções de forma independente. Os tempos de transmissão de mensagens ou confirmações são variáveis aleatórias com distribuição exponencial e parâmetro μ_2 e μ_3 respectivamente. Quando chega mensagem no EMISSOR, ela é colocada no CANAL1 para ser entregue ao RECEPTOR. Após um período de espera pelo ack (*timeout*), o EMISSOR retransmite a mensagem.

O EMISSOR é responsável pela geração de pacotes e recuperação de erros. Identificamos então dois eventos: geração de novos pacotes e ocorrência de *timeouts*. Para cada evento, diferentes ações podem ser tomadas, de acordo com o estado do EMISSOR. Por exemplo, um novo pacote faz com que o transmissor gere uma mensagem para o objeto canal para transmissão.

O EMISSOR entende de 2 tipos de mensagens: uma indica que o canal transmissor está livre para transmitir novo pacote, e a outra indica a chegada de um ack vindo do canal receptor.

O RECEPTOR ao receber a mensagem com o número de seqüência desejado, coloca o ack da mensagem no CANAL2. Este ack corresponde ao número de seqüência da próxima mensagem a ser enviada pelo EMISSOR. Mensagens fora de seqüência são descartadas e o ack da última mensagem correta é retransmitido.

É importante ressaltar que o mapeamento da descrição original do protocolo em quatro objetos é arbitrário. O mesmo comportamento pode ser descrito definindo-se outros tipos de objetos.

A descrição detalhada do EMISSOR é apresentada abaixo:

```
TIPO : transm_1_buffer;
NOME : Nome_Objeto;
ESTADO : (Buffer,Num_Seg,Num_Pend,Sit_Canal,Timeout);
EVENTO chegada de mensagens: (Buffer,Num_Seg,Num_Pend,Sit_Canal,Timeout) →
                             (Buffer2,Num_Seg,Num_Pend2,Sit_Canal2,Timeout);
        CONDIÇÃO : Buffer = "vazio",
                   Sit_Canal = "livre";
        AÇÃO :     Buffer2 = "ocupado",
                   Num_Pend2 = [ ],
                   Sit_Canal2 = "transmitindo",
                   taxa_chegada(Nome_Objeto,T),
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Seg), Canal);
        TAXA :     T;
        CONDIÇÃO : Buffer = "vazio",
                   Sit_Canal = "transmitindo";
        AÇÃO :     Buffer2 = "ocupado",
                   Num_Pend2 = Num_Seg,
                   Sit2_Canal = "transmitindo",
                   taxa_chegada(Nome_Objeto,T);
        TAXA :     T;
EVENTO timeout : (Buffer,Num_Seg,Num_Pend,Sit_Canal,Timeout) →
                 (Buffer,Num_Seg,Num_Pend,Sit_Canal2,Timeout2);
        CONDIÇÃO : Timeout = 1;
        AÇÃO :     Sit_Canal2 = "transmitindo",
                   Timeout2 = 0,
                   taxa_timeout(T),
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Seg), Canal);
        TAXA :     T;
MENSAGEM "livre" : (Buffer,Num_Seg,Num_Pend,Sit_Canal,Timeout) →
                   (Buffer,Num_Seg,Num_Pend2,Sit_Canal2,Timeout2);
            CONDIÇÃO : Buffer = "ocupado",
                       Num_Pend ≠ [ ];
            AÇÃO :     Num_Pend2 = [ ],
                       Sit_Canal2 = "transmitindo",
                       Timeout2 = 0,
                       caminho(Nome_Objeto, Canal, Destino),
                       envia_mensagem(mens(Num_Seg), Canal);
            CONDIÇÃO : Buffer = "ocupado",
                       Num_Pend = [ ];
            AÇÃO :     Num_Pend2 = [ ],
                       Sit_Canal2 = "livre",
                       Timeout2 = 1;
            CONDIÇÃO : Buffer = "vazio";
            AÇÃO :     Num_Pend2 = [ ],
                       Sit_Canal2 = "livre",
                       Timeout2 = 0;
MENSAGEM Ack : (Buffer,Num_Seg,Num_Pend,Sit_Canal,Timeout) →
               (Buffer2,Ack,Num_Pend,Sit_Canal,Timeout2);
            CONDIÇÃO : Ack = Num_Seg;
            AÇÃO :     Buffer2 = "ocupado",
                       Timeout2 = Timeout;
            CONDIÇÃO : Ack := (Num_Seg + 1) mod 2;
            AÇÃO :     Buffer2 = "vazio",
                       Timeout2 = 0;
```

O estado do EMISSOR do bit alternante é definido por (Buffer, Num_Seg, Num_Pend, Sit_Canal, Timeout), onde Buffer para transmissão é *ocupado* quando o EMISSOR possui uma mensagem cujo recebimento ainda não foi confirmado pelo RECEPTOR e é *vazio* quando está pronto para receber uma nova mensagem; Num_Seg é o número de seqüência da próxima mensagem a ser enviada; Num_Pend indica o número de seqüência de uma mensagem para ser transmitida logo que o canal desocupe ou é [ ] quando não exite mensagem pendente; Sit_Canal é *livre* quando CANAL1 está desocupado e é *transmitindo* quando existe mensagem no canal sendo transmitida; e Timeout é 1 quando a retransmissão da última mensagem pode ocorrer e é zero no caso contrário.

Este tipo de objeto pode gerar dois eventos: *chegada de mensagens* e *timeout*. O primeiro evento pode ocorrer com duas condições: o buffer está vazio (Buffer = "vazio") e o canal está desocupado (Sit_Canal = "livre") ou o buffer está vazio (Buffer = "vazio") e o canal está transmitindo alguma mensagem (Sit_Canal = "transmitindo"). Na primeira condição chegadas de novas mensagens são inibidas (Buffer = "ocupado") e a mensagem é colocada no canal para ser transmitida. Na segunda condição a mensagem tem que esperar para ser transmitida logo que CANAL1 desocupe (Num_Pend2 = Num_Seg) e o buffer fica ocupado (Buffer = "ocupado"). Caminho(Nome_Objeto, Canal, Destino) é a leitura de um parâmetro do nível de aplicação que fornece o nome do canal e do objeto receptor da mensagem. A taxa de chegada é dada pelo parâmetro taxa_chegada(Nome_Objeto,T). A taxa de ocorrência deste evento é igual a taxa de chegada das novas mensagens.

No segundo evento é feita a retransmissão das mensagens após um período de espera pelo ack. Este evento só ocorre quando timeout foi habilitado. Note que o timeout só é habilitado quando não existe mensagem pendente a ser transmitida e o buffer de retransmissão está ocupado. A taxa de ocorrência deste evento é dada por taxa_timeout(T).

Dois tipos de mensagens podem ser recebidos pelo objeto: *livre* e *ack*. A primeira proveniente do objeto CANAL1 indica que este está livre para novas transmissões. Ao receber a mensagem *livre* uma nova mensagem é enviada se

Num_Pend for diferente de [ ]. A segunda proveniente do CANAL2, indica a chegada de um ack sem erro. Quando uma mensagem é corretamente recebida o ack é igual ao número da próxima mensagem a ser enviada (Ack := (Num_Seg + 1) mod 2). Qualquer outro tipo de mensagem recebida é ignorado pelo emissor.

O nível de aplicação para um objeto do tipo acima descrito poderia ser:

```
TIPO(emissor,transm_1_buffer).

INICIAL(emissor,(vazio,0,[ ],livre,0)).

TAXA_CHEGADA(emissor,20).

TAXA_TIMEOUT(1).

CAMINHO(emissor, canal1, receptor).
```

A cláusula TIPO especifica o nome do objeto (emissor) e o seu tipo correspondente (transm_1_buffer). A cláusula INICIAL define o estado inicial do objeto (Buffer = "vazio", Num_Seg = 0, Num_Pend = [ ], Sit_Canal = "livre" e Timeout = 0). As cláusulas TAXA_CHEGADA e TIMEOUT determinam as taxas de chegada e de retransmissão das mensagens. E a cláusula CAMINHO define o caminho seguido pela mensagem quando da sua transmissão.

O estado do RECEPTOR é representado pelo número de seqüência da próxima mensagem a ser recebida, pelo número de seqüência do último ack transmitido e por uma variável indicando o estado do canal: *livre* quando CANAL2 está desocupado e *transmitindo* quando CANAL2 está com alguma mensagem.

```
* definição do tipo de objeto recep
TIPO : recep;
NOME : Nome_Objeto;
ESTADO : (Num_Seg,Ultimo_Ack,Sit_Canal);
MENSAGEM Mens : (Num_Seg,Ultimo_Ack,Sit_Canal) — (Num_Seg2,Ultimo_Ack2,Sit_Canal2);
        CONDIÇÃO: Num_Seg = Mens,
                  Sit_Canal = "livre";
        AÇÃO:     Num_Seg2 := (Num_Seg + 1) mod 2,
                  Ultimo_Ack2 = Num_Seg2,
                  Sit_Canal2 = "transmitindo",
                  caminho(Nome_Objeto, Canal, Destino),
                  envia_mensagem(mens(Num_Seg2), Canal);
        PROBABILIDADE : 1;
        CONDIÇÃO: Num_Seg = Mens,
                  Sit_Canal := "transmitindo";
        AÇÃO:     Num_Seg2 := (Num_Seg + 1) mod 2,
                  Ultimo_Ack2 = Ultimo_Ack,
                  Sit_Canal2 = "transmitindo";
        PROBABILIDADE : 1;
```

```
            CONDIÇÃO: Num_Seg ≠ Mens,
                      Sit_Canal = "livre";
            AÇÃO:     Num_Seg2 = Num_Seg,
                      Ultimo_Ack2 = Ultimo_Ack,
                      Sit_Canal2 = "transmitindo",
                      caminho(Nome_Objeto, Canal, Destino),
                      envia_mensagem(mens(Num_Seg), Canal);
            PROBABILIDADE : 1;
MENSAGEM "livre" : (Num_Seg,Ultimo_Ack,Sit_Canal) — (Num_Seg2,Ultimo_Ack2,Sit_Canal2);
            CONDIÇÃO: Num_Seg ≠ Ultimo_Ack;
            AÇÃO:     Sit_Canal2 = "transmitindo",
                      Ultimo_Ack2 = Num_Seg,
                      caminho(Nome_Objeto, Canal, Destino),
                      envia_mensagem(mens(Num_Seg), Canal);
            PROBABILIDADE : 1;
            CONDIÇÃO: Num_Seg = Ultimo_Ack;
            AÇÃO:     Sit_Canal2 = "livre",
                      Ultimo_Ack2 = Ultimo_Ack;
            PROBABILIDADE : 1;
```

O objeto RECEPTOR ao receber uma mensagem com o número de seqüência esperado (Num_Seg = Mens) e estando o CANAL2 desocupado (Sit_Canal = "livre"), atualiza o código da próxima mensagem a ser recebida (Num_Seg2 := (Num_Seg + 1) mod 2) e transmite o ack. Se o CANAL2 já tiver transmitindo um ack a variável Ultimo_Ack não é atualizada, e logo que o canal fique livre o ack é transmitido. Caso a mensagem recebida tenha uma seqüência diferente, ela é descartada e o número de seqüência esperado é retransmitido se o CANAL2 estiver livre. Qualquer outro tipo de mensagem é ignorado.

O estado de cada canal é [ ] quando não estiver transmitindo mensagem ou indica o número de seqüência da mensagem sendo transmitida.

CANAL1 e CANAL2 quando ocupados (N ≠ [ ]) transmitem a mensagem com uma determinada probabilidade de erro. A probabilidade de erro, a taxa de transmissão e o nome do objeto que receberá a mensagem são dados por prob_de_falhar(Nome_Objeto, Prob), taxa(Nome_Objeto, T) e caminho(Origem, Nome_Objeto, Destino) respectivamente.

```
* definição do tipo de objeto canal
TIPO : Canal;
NOME : Nome_Objeto;
ESTADO : (N);
EVENTO transmissão : N → N1;
          CONDIÇÃO : N ≠ [ ];
          AÇÃO :     taxa(Nome_Objeto,T),
                     prob_de_falhar(Canal,Prob),
                     caminho(Origem, Nome_Objeto, Destino),
                     envia_mensagem(mens("erro"), Destino),
                     envia_mensagem(mens("livre"), Origem),
```

```
                N1 = [ ];
        TAXA :  Prob * T;
        CONDIÇÃO : N ≠ [ ];
        AÇÃO :  taxa(Nome_Objeto,T),
                prob_de_falhar(Canal,Prob),
                caminho(Origem, Nome_Objeto, Destino),
                envia_mensagem(mens(N), Destino),
                envia_mensagem(mens("livre"), Origem),
                N1 = [ ];
        TAXA :  (1 - Prob) * T;
MENSAGEM M: N → N1;
        CONDIÇÃO : N = [ ];
        AÇÃO :  N1 = M;
        PROBABILIDADE : 1;

* definição do nível de aplicação
TIPO(receptor, recap).
TIPO(canal1, canal).
TIPO(canal2, canal).

INICIAL(receptor,[0,0,livre]).
INICIAL(canal1,[ ]).
INICIAL(canal2,[ ]).

CAMINHO(emissor, canal1, receptor).
CAMINHO(receptor, canal2, emissor).

TAXA(canal1,μ2 ).
TAXA(canal2,μ3 ).

PROB_DE_FALHAR(canal1,0.05).
PROB_DE_FALHAR(canal2,0.05).
```

É interessante observar que em [7], foi assumido que o *timeout* só ocorreria quando houvesse um erro de transmissão; permitindo assim, a definição de uma rede de Petri com um número finito de estados. Entretanto, esta suposição não é verdadeira para o protocolo do bit alternante, e portanto, não foi aqui considerada.

Go Back N

Considere o protocolo Go Back N como descrito em [19]:

> Mensagens são numeradas de 0 a 2N - 1. Mensagens chegam exponencialmente com média $1/\lambda$ no emissor que possui capacidade de guardar apenas N mensagens. Mensagens que chegam quando o buffer está ocupado são descartadas. O emissor pode transmitir até N mensagens antes que fique bloqueado por falta de acks. Todas as mensagens transmitidas após uma ocorrência de erro no canal

são descartadas pelo receptor até que a mensagem que falta seja retransmitida. Ao receber um ack o emissor considera que todas as mensagens anteriores a este ack foram corretamente recebidas pelo receptor. Mensagens ou acks com erro na transmissão são descartados.

Similarmente ao exemplo anterior, o sistema pode ser modelado por quatro objetos: EMISSOR, RECEPTOR, CANAL1 e CANAL2. O estado do objeto EMISSOR é representado pelo número de mensagens no buffer de transmissão (Buffer); pelo número de seqüência do último ack recebido (Num_Ack); pelo número de seqüência da próxima mensagem a ser enviada (é igual a [ ] quando não existe mensagem a transmitir); pela situação de CANAL1 (*livre / transmitindo*); e por um indicador (Timeout) que é 1 quando o timeout está habilitado e é zero no caso contrário. O estado do RECEPTOR é dado pelo número de seqüência da próxima mensagem a ser recebida, pelo número de seqüência do último ack enviado e pela situação de CANAL2 (*livre / transmitindo*).

O objeto EMISSOR ao receber a mensagem *livre* do CANAL1 transmite a mensagem marcada em Num_Pend, se houver, e habilita o evento *timeout* (Timeout = 1). Ao receber um ack, o EMISSOR diminui do buffer o número de mensagens esperando confirmação. O EMISSOR pode gerar dois tipos de eventos: chegada de novas mensagens e *timeout*. O evento *timeout* nada mais é do que uma indicação para a retransmissão das mensagens que ainda não receberam ack: o número de seqüência da próxima mensagem passa a ser igual ao número de seqüência do último ack recebido.

O objeto RECEPTOR ao receber uma mensagem com o número de seqüência desejado (mens = Num_Pend) transmite o ack se o CANAL2 estiver livre. Mensagens recebidas com número de seqüência diferente do desejado são descartadas e o último ack é retransmitido se CANAL2 estiver desocupado. Mensagens recebidas pelo RECEPTOR quando CANAL2 está ocupado são aceitas se o número de seqüência está correto, mas somente o ack da última mensagem é transmitida quando o canal fica livre. Observe que este objeto é semelhante

ao objeto RECEPTOR do exemplo anterior. A única diferença consiste que o RECEPTOR do GO_BACK(N) se utiliza de uma seqüência de mensagem de módulo 2N no lugar de módulo 2. Portanto, não iremos aqui repetir o tipo definido anteriormente.

CANAL1 e CANAL2 são idênticos ao do exemplo anterior.

```
* definição do tipo de objeto transm_N_buffers
TIPO : transm_N_buffers;
NOME : Nome_Objeto;
ESTADO : (Buffer,Num_Ack,Num_Pend,Sit_Canal,Timeout);
EVENTO chegada : (Buffer,Num_Ack,Num_Pend,Sit_Canal,Timeout) —
                 (Buffer2,Num_Ack,Num_Pend2,Sit_Canal2,Timeout);
        CONDIÇÃO : go_back(N),
                   Buffer < N,
                   Num_Pend ≠ [ ],
                   Sit_Canal = "livre";
        AÇÃO :     Buffer2 := Buffer + 1,
                   Num_Pend2 := (Num_Pend + 1) mod (2 * N),
                   Sit_Canal2 = "transmitindo",
                   taxa_chegada(Nome_Objeto,T),
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Pend), Canal);
        TAXA :     T;
        CONDIÇÃO : go_back(N),
                   Buffer < N,
                   Num_Pend = [ ],
                   Sit_Canal = "livre";
        AÇÃO :     Buffer2 := Buffer + 1,
                   Num_Pend2 := (Num_Ack + 1) mod (2 * N),
                   Sit_Canal2 = "transmitindo",
                   taxa_chegada(Nome_Objeto,T),
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Pend), Canal);
        TAXA :     T;
        CONDIÇÃO : go_back(N),
                   Buffer < N,
                   Sit_Canal = "transmitindo";
        AÇÃO :     Buffer2 := Buffer + 1,
                   Num_Pend2 = Num_Pend,
                   Sit_Canal2 = "transmitindo",
                   taxa_chegada(Nome_Objeto,T);
        TAXA :     T;
EVENTO timeout : (Buffer,Num_Ack,Num_Pend,Sit_Canal,Timeout) —
                 (Buffer,Num_Ack,Num_Pend2,Sit_Canal2,Timeout2);
        CONDIÇÃO : Buffer > 0,
                   Timeout = 1,
                   Sit_Canal = "livre";
        AÇÃO :     Sit_Canal2 = "transmitindo",
                   Timeout2 = 0,
                   taxa_timeout(T),
                   go_back(N),
                   Num_Pend2 := (Num_Ack + 1) mod (2 * N),
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Ack), Canal);
        TAXA :     T;
        CONDIÇÃO : Buffer > 0,
                   Timeout = 1,
                   Sit_Canal = "transmitindo";
        AÇÃO :     Sit_Canal2 = "transmitindo",
                   Timeout2 = 0,
                   Num_Pend2 = Num_Ack,
                   taxa_timeout(T);
        TAXA :     T;
MENSAGEM "livre" : (Buffer,Num_Ack,Num_Pend,Sit_Canal,Timeout) →
```

```
                (Buffer,Num_Ack,Num_Pend2,Sit_Canal2,Timeout2);
        CONDIÇÃO : Buffer > 0,
                   Num_Pend ≠ [ ],
                   go_back(N),
                   (Num_Pend - Num_Ack) mod 2 * N < Buffer;
        AÇÃO :     Sit_Canal2 = "transmitindo",
                   Num_Pend2 := (Num_Pend + 1) mod (2 * N),
                   Timeout2 = 1,
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Pend), Canal);
        PROBABILIDADE : 1;
        CONDIÇÃO : Buffer > 0,
                   go_back(N),
                   Num_Pend = [ ];
        AÇÃO :     Sit_Canal2 = "transmitindo",
                   Num_Pend2 := (Num_Ack + 1) mod (2 * N),
                   Timeout2 = 0,
                   caminho(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mens(Num_Ack), Canal);
        PROBABILIDADE : 1;
        CONDIÇÃO : Buffer > 0;
                   Num_Pend ≠ [ ],
                   go_back(N),
                   (Num_Pend - Num_Ack) mod 2 * N ≥ Buffer;
        AÇÃO :     Sit_Canal2 = "livre",
                   Num_Pend2 = Num_Pend,
                   Timeout2 = 1;
        PROBABILIDADE : 1;
        CONDIÇÃO : Buffer = 0;
        AÇÃO :     Sit_Canal2 = "livre",
                   Num_Pend2 = [ ],
                   Timeout2 = 0;
        PROBABILIDADE : 1;
MENSAGEM Ack : (Buffer,Ultimo_Ack,Num_Pend,Sit_Canal,Timeout) →
               (Buffer2,Ultimo_Ack2,Num_Pend2,Sit_Canal,Timeout2);
        CONDIÇÃO : go_back(N),
                   Buffer - (Ack - Ultimo_Ack) mod 2 * N ≠ 0;
        AÇÃO :     go_back(N),
                   Buffer2 := Buffer - (Ack - Ultimo_Ack) mod 2 * N,
                   Ultimo_Ack2 = Ack,
                   Num_Pend2 = Num_Pend,
                   Timeout2 = Timeout;
        PROBABILIDADE : 1;
        CONDIÇÃO : go_back(N),
                   Buffer - (Ack - Ultimo_Ack) mod 2 * N = 0;
        AÇÃO :     Buffer2 = 0,
                   Ultimo_Ack2 = Ack,
                   Num_Pend2 = [ ],
                   Timeout2 = 0;
        PROBABILIDADE : 1;

* definição do nível de aplicação
GO_BACK(2).

TIPO(emissor,transm_N_buffers).
TIPO(receptor,recep_N).
TIPO(canal1, canal).
TIPO(canal2, canal).

INICIAL(emissor,(0,0,[ ],livre)).
INICIAL(receptor,(0,0,livre)).
INICIAL(canal1,[ ]).
INICIAL(canal2,[ ]).

CAMINHO(emissor, canal1, receptor).
CAMINHO(receptor, canal2, emissor).

TAXA_CHEGADA(emissor,λ).
TAXA_TIMEOUT(μ1).
```

TAXA(canal1,μ_2).
TAXA(canal2,μ_3).

PROB_DE_FALHAR(canal1,p).
PROB_DE_FALHAR(canal2,k).

É fácil verificar a semelhança entre as especificações do EMISSOR do bit alternante e do go_back_N. As diferenças são devidas a que, no go_back_N, podemos ter uma fila para transmissão de mensagens, e existe a possibilidade de ocorrer timeout antes de N mensagens serem transmitidas. A cláusula GO_BACK(N) especifica o número máximo de mensagens que o emissor transmite antes de se bloquear. Para N = 1, temos o bit alternante descrito na seção anterior. Observe que podemos conseguir o protocolo go back N para vários valores de N pela simples mudança do valor de N na cláusula GO_BACK(N).

ABRACADABRA

Em [20] é proposto o protocolo ABRACADABRA que captura o comportamento básico do modelo OSI. Este protocolo tem importantes características dos protocolos normalizados como as fases de estabelecimento e de liberação de conexão.

A especificação do protocolo propõe uma conexão entre dois usuários UA e UB como mostrada na figura III.10. O pedido para estabelecimento da conexão é feito por um usuário à entidade com a qual se conecta pelo canal UCEP. A entidade transmite o pedido para a outra entidade pelo uso de um canal bidirecional que possue uma determinada probabilidade de transmitir com erro. Confirmado o estabelecimento da conexão os dados são transmitidos. No término da transmissão dos dados é pedido o encerramento da conexão.

As unidades de dados do protocolo (ou PDUs) são de 3 tipos:

1. CR (pedido de conexão) e CC (confirmação de conexão) na fase de estabelecimento de conexão;
2. DT (dado) e AK (ack) na fase de transferência de dados;

3. DR (pedido de desconexão) e DC (confirmação de desconexão) na fase de liberação da conexão;

As PDU's DT e AK são enviadas junto com um número de seqüência de módulo 2. A cada nova conexão esta seqüência é inicializada.

As PDU's CR, DT e DR são retransmitidas pela entidade após um período de espera pela resposta. Um número máximo de tentativas por PDU é estabelecido pelo protocolo.

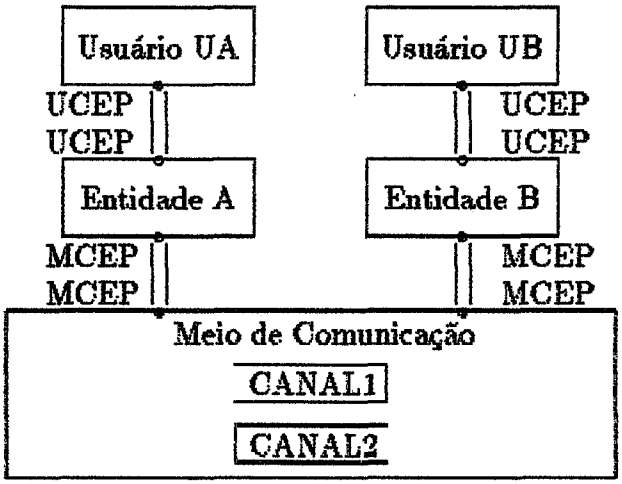

Figura III.10: Arquitetura do protocolo ABRACADABRA

O sistema pode ser modelado por seis objetos relacionados aos módulos da figura III.10 (entretanto, apenas três tipos de objetos são necessários): USUÁRIO_A, USUÁRIO_B, ENTIDADE_A, ENTIDADE_B, CANAL1 E CANAL2. O objeto USUÁRIO_A (USUÁRIO_B) inicializa pedindo a ENTIDADE_A (ENTIDADE_B) que estabeleça uma conexão com o USUÁRIO_B (USUÁRIO_A). Mensagens são enviadas da ENTIDADE_A (ENTIDADE_B) para a ENTIDADE_B (ENTIDADE_A) através do CANAL1 (CANAL2). Respostas da ENTIDADE_B (ENTIDADE_A) são transmitidas para a ENTIDADE_A (ENTIDADE_B) pelo CANAL2 (CANAL1). CANAL1 e CANAL2 são do mesmo tipo usado nos dois exemplos anteriores. Ao receber a confirmação da conexão a ENTIDADE_A (ENTIDADE_B) informa ao USUÁRIO_A (USUÁRIO_B) e este transmite os primeiros dados. A cada recebimento correto de

um ack o USUÁRIO_A (USUÁRIO_B) é informado e novos dados são transmitidos ou o fechamento da conexão é pedido. Qualquer entidade pode pedir o fechamento da conexão se um erro for detectado. A ENTIDADE_A (ENTIDADE_B) retransmite a última PDU após um determinado período de espera. O número de retransmissões permitidas para cada PDU é dada pelo parâmetro NUM_RETRANSMISSÕES.

São estados possíveis para o usuário que está transmitindo uma mensagem:

1. (CLOSED, 0) - o componente está livre de qualquer conexão;
2. (OPEN, CR) - o componente está aguardando a resposta de um pedido de abertura de conexão;
3. (OPEN, AK) - o componente está esperando pelo ack de uma mensagem;
4. (OPEN, DT) - o componente pode transmitir novos dados ou fechar conexão;

O usuário que está recebendo uma mensagem possue apenas dois estados: (CLOSED, 0) e (OPEN,0). No primeiro estado o componente está livre de qualquer conexão e no segundo estado está aguardando o envio das mensagens. As três últimas definições de mensagens do objeto de tipo *usuário_ABRA* (especificado abaixo) são utilizadas na recepção.

```
TIPO : usuário_ABRA;
NOME : Nome_Objeto;
ESTADO : M;
EVENTO abertura de conexão : Sit → Sit2;
        CONDIÇÃO : Sit = (CLOSED,0);
        AÇÃO :     Sit2 = (OPEN, CR),
                   taxa_abertura(Nome_Objeto,T),
                   ucep(Nome_Objeto, Destino),
                   envia_mensagem(ucep(CR), Destino);
        TAXA :     T;
EVENTO transmite :  (OPEN,K) → (OPEN,K1);
        CONDIÇÃO : K = DT;
        AÇÃO :     K1 = AK,
                   taxa_processamento(Nome_Objeto,T),
                   prob_novos_dados(Nome_Objeto,Prob),
                   ucep(Nome_Objeto,Destino),
                   envia_mensagem(ucep(DT), Destino);
        TAXA :     Prob * T;
EVENTO fechamento da conexão : (OPEN,K) → (K1,0);
        CONDIÇÃO : K = DT;
        AÇÃO :     K1 = CLOSED,
                   taxa_processamento(Nome_Objeto,T),
                   prob_novos_dados(Nome_Objeto,Prob),
                   ucep(Nome_Objeto,Destino),
```

```
                    envia_mensagem(ucep(DR), Destino);
        TAXA :      (1 - Prob) * T;
MENSAGEM ucep(CC): (OPEN,K) — (OPEN,K1);
            CONDIÇÃO : K = DR;
            AÇÃO :     K1 = AK,
                       ucep(Nome_Objeto,Destino),
                       envia_mensagem(ucep(DT), Destino);
MENSAGEM ucep(AK): (OPEN,K) → (OPEN,K1);
            CONDIÇÃO : K = AK;
            AÇÃO :     K1 = DT;
MENSAGEM ucep(DR) : (OPEN) — (CLOSED);
MENSAGEM ucep(CR): (CLOSED,0) → (OPEN,0);
            AÇÃO : ucep(Nome_Objeto,Destino),
                   envia_mensagem(ucep(CC), Destino);
MENSAGEM ucep(DR): (OPEN,0) → (CLOSED,0);
            AÇÃO : ucep(Nome_Objeto,Destino),
                   envia_mensagem(ucep(DC), Destino);
MENSAGEM ucep(DT) : M — M;
```

São estados possíveis da entidade ABRACADABRA:

1. (CLOSED,0) - o componente está livre de qualquer conexão;
2. (WFCC,N) - o componente está esperando a resposta de um pedido por conexão (Wait For CC), onde N é o número de retransmissões feitas do CR;
3. (WFAK,M(N, E, R, K)) - o componente está esperando pelo ack do dado enviado (Wait For Ack); N é o número de retransmissões feitas do DT, E é o número de seqüência do último dado transmitido, R é o número de seqüência do último dado recebido e K é *true* quando nenhum DT ou AK foi recebido pelo componente;
4. (OPEN,M(0, E, R, K)) - o componente está esperando por dados;
5. (CLOSING,N) - o componente está esperando a resposta de um pedido por liberação de conexão, onde N é o número de retransmissões feitas do DR;
6. (WFUR,0) - o componente está esperando a resposta do usuário (Wait For User Response). Este estado é transitório.

```
* definição do tipo de objeto ent (entidade do ABRACADABRA)
TIPO : ent;
NOME : Nome_Objeto;
ESTADO : (M,N);
EVENTO timaout : (K1,N1) → (K2,N2);
        CONDIÇÃO : K1 = CLOSING,
                   num_retransmissões(Nome_Objeto,NUM),
```

```
                    N1 ≤ NUM;
          AÇÃO :    K2 = CLOSING,
                    N2 := N1 + 1,
                    taxa(Nome_Objeto,T),
                    mcep(Nome_Objeto, Canal, Destino),
                    envia_mensagem(mcep(DR), Canal);
          TAXA :    T;
          CONDIÇÃO : K1 = WFCC,
                    num_retransmissões(Nome_Objeto,NUM),
                    N1 ≤ NUM;
          AÇÃO :    K2 = WFCC,
                    N2 := N1 + 1,
                    taxa(Nome_Objeto,T),
                    mcep(Nome_Objeto, Canal, Destino),
                    envia_mensagem(mcep(CR), Canal);
          TAXA :    T;
          CONDIÇÃO : K1 = WFAK,
                    N1 = M(N,E,R,K),
                    num_retransmissões(Nome_Objeto,NUM),
                    N ≤ NUM;
          AÇÃO :    K2 = WFAK,
                    N2 = (N + 1,E,R,K),
                    taxa(Nome_Objeto,T),
                    mcep(Nome_Objeto, Canal, Destino),
                    envia_mensagem(mcep(DT,E), Canal);
          TAXA :    T;
          CONDIÇÃO : K1 = WFCC,
                    num_retransmissões(Nome_Objeto,NUM),
                    N1 > NUM;
          AÇÃO :    K2 = CLOSING,
                    N2 = 0,
                    taxa(Nome_Objeto,T),
                    ucep(Nome_Objeto, Destino1),
                    envia_mensagem(ucep(DR), Destino1),
                    mcep(Nome_Objeto, Canal, Destino2),
                    envia_mensagem(mcep(DR), Canal);
          TAXA :    T;
          CONDIÇÃO : K1 = WFAK,
                    N1 = M(N,E,R,K),
                    num_retransmissões(Nome_Objeto,NUM),
                    N > NUM;
          AÇÃO :    K2 = CLOSING,
                    N2 = 0,
                    taxa(Nome_Objeto,T),
                    ucep(Nome_Objeto, Destino1),
                    envia_mensagem(ucep(DR), Destino1),
                    mcep(Nome_Objeto, Canal, Destino2),
                    envia_mensagem(mcep(DR), Canal);
          TAXA :    T;
          CONDIÇÃO : K1 = CLOSING,
                    num_retransmissões(Nome_Objeto,NUM),
                    N1 > NUM;
          AÇÃO :    K2 = CLOSED,
                    N2 = 0,
                    taxa(Nome_Objeto,T);
          TAXA :    T;
MENSAGEM ucep(CR): (CLOSED,0) → (WFCC,0);
          AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                 envia_mensagem(mcep(CR), Canal);
MENSAGEM mcep(K): (WFCC,N) → (OPEN,M(0,0,0,true));
          CONDIÇÃO : K = CC ou K = CR;
          AÇÃO : ucep(Nome_Objeto,Destino),
                 envia_mensagem(ucep(CC), Destino);
MENSAGEM mcep(DR): (WFCC,N) → (CLOSED,0);
          AÇÃO : ucep(Nome_Objeto,Destino1),
                 envia_mensagem(ucep(DR), Destino1),
                 mcep(Nome_Objeto, Canal, Destino2),
                 envia_mensagem(mcep(DC), Canal);
MENSAGEM ucep(DR): (WFCC,N) → (CLOSING,0);
```

```
            AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mcep(DR), Canal);
MENSAGEM mcep(CR): (CLOSED,0) → (WFUR,0);
            AÇÃO : ucep(Nome_Objeto,Destino),
                   envia_mensagem(ucep(CR), Destino);
MENSAGEM mcep(DR): (CLOSED,0) → (CLOSED,0);
            AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mcep(DC), Canal);
MENSAGEM ucep(CC): (WFUR,0) → (OPEN,M(0,0,0,true));
            AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mcep(CC), Canal);
MENSAGEM ucep(DR): (WFUR,0] → (CLOSING,0);
            AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mcep(DR), Canal);
MENSAGEM ucep(DT): (OPEN,M(N,E,R,K)) → (WFAK,M(0,E,R,K));
            AÇÃO : mcep(Nome_Objeto, Canal, Destino),
                   envia_mensagem(mcep(DT,E), Canal);
MENSAGEM mcep(AK,E2): (WFAK,M(N,E,R,K)) → (K2,N2);
            CONDIÇÃO : E2 := (E + 1) mod 2;
            AÇÃO :     K2 = OPEN,
                       N2 = M(0,E2,R,K),
                       ucep(Nome_Objeto,Destino1),
                       envia_mensagem(ucep(AK), Destino1);
            CONDIÇÃO : E2 ≠ (E + 1) mod 2;
            AÇÃO :     K2 = CLOSING,
                       N2 = 0,
                       ucep(Nome_Objeto,Destino1),
                       envia_mensagem(ucep(DR), Destino1),
                       mcep(Nome_Objeto, Canal, Destino2),
                       envia_mensagem(mcep(DR), Canal);
MENSAGEM mcep(DT,Q): (sit,M(N,E,R,K)] → (sit,M(N,E,R2,false));
            CONDIÇÃO : sit = WFAK ou sit = OPEN,
                       Q = R;
            AÇÃO :     R2 := (R + 1) mod 2,
                       ucep(Nome_Objeto,Destino1),
                       envia_mensagem(ucep(DT), Destino1),
                       mcep(Nome_Objeto, Canal, Destino2),
                       envia_mensagem(mcep(AK,R2), Canal);
            CONDIÇÃO : sit = WFAK ou sit = OPEN,
                       Q ≠ R;
            AÇÃO :     R2 = R,
                       mcep(Nome_Objeto, Canal, Destino),
                       envia_mensagem(mcep(AK,R), Canal);
MENSAGEM mcep(CR): (sit,M(N,E,R,true)) → (sit,M(N,E,R,true));
            CONDIÇÃO : sit = WFAK ou sit = OPEN;
            AÇÃO :     mcep(Nome_Objeto, Canal, Destino),
                       envia_mensagem(mcep(CC), Canal);
MENSAGEM mcep(DR): (sit,M(N,E,R,K)) → (CLOSED,0);
            CONDIÇÃO : sit = WFAK ou sit = OPEN;
            AÇÃO :     ucep(Nome_Objeto,Destino1),
                       envia_mensagem(ucep(DR), Destino1),
                       mcep(Nome_Objeto, Canal, Destino2),
                       envia_mensagem(mcep(DC), Canal);
MENSAGEM ucep(DR): (sit,K] → (CLOSING,0);
            CONDIÇÃO : sit = WFAK ou sit = OPEN;
            AÇÃO :     mcep(Nome_Objeto, Canal, Destino),
                       envia_mensagem(mcep(DR), Canal);
MENSAGEM mcep(M): (CLOSING,N) → (CLOSED,0);
            CONDIÇÃO : M = DC ou M = DR;
```

O nível de aplicação utiliza os objetos acima definidos. As entidades A e B são do tipo *ent*, e usuário_A e usuário_B são do tipo *usuário_ABRA*. É importante notar também a definição das ligações como mostra a Figura III.10.

* definição do nível de aplicação

```
TIPO(usuário_A,usuário_ABRA).
TIPO(usuário_B,usuário_ABRA).
TIPO(entidade_A,ent).
TIPO(entidade_B,ent).
TIPO(canal1, canal).
TIPO(canal2, canal).

INICIAL(usuário_A,(closed,0)).
INICIAL(usuário_B,(closed,0)).
INICIAL(entidade_A,(closed,0)).
INICIAL(entidade_B,(closed,0)).
INICIAL(canal1,[ ]).
INICIAL(canal2,[ ]).

NUM_RETRANSMISSÕES(entidade_A,1).
NUM_RETRANSMISSÕES(entidade_B,1).

UCEP(usuário_A,entidade_A).
UCEP(usuário_B,entidade_B).

MCEP(entidade_A, canal1, entidade_B).
MCEP(entidade_B, canal2, entidade_A).

TAXA_ABERTURA(usuário_A,μ1).
TAXA_PROCESSAMENTO(usuário_A,μa).
TAXA(canal1,c1).
TAXA(canal2,c2).

PROB_NOVOS_DADOS(usuário_A,p1).
PROB_DE_FALHAR(canal1,p2).
PROB_DE_FALHAR(canal2,p3).
```

Observe que para o usuário_A foram especificadas a taxa de abertura de conexão, a taxa de processamento de mensagens e a probabilidade de transmitir novos dados após o recebimento do ack da última mensagem. Estes parâmetros não foram definidos para o usuário_B. Como conseqüência, o usuário_B não poderá iniciar uma conexão. Entretanto, esta é uma opção feita para diminuir o número de estados da cadeia de Markov, e pode portanto, ser facilmente alterada.

Como comentário final observamos que não houve preocupação na escolha das variáveis de estado de cada objeto (nos exemplos acima), de maneira a tornar a descrição de seu comportamento mais compacta. Outras descrições poderiam ser feitas desde que não alterassem o comportamento final do objeto.

Capítulo IV

Interface com o Usuário

Para auxiliar o analista tanto na definição de modelos quanto na criação de novos tipos de objetos foi desenvolvida uma interface com dois módulos (fig. IV.1): definição de modelos e definição de tipos de objetos. No primeiro módulo temos a utilização da biblioteca de tipos existentes. Nesta biblioteca estão disponíveis quinze tipos de objetos: quatro para a área de confiabilidade, quatro para a descrição de modelos de filas e sete para a área de protocolos de comunicação. No segundo módulo é fornecida uma biblioteca de funções para serem usadas na definição de novos tipos de objetos. Esta biblioteca é composta de cinquenta funções do Prolog IBM e doze funções criadas especialmente para o usuário da interface. Este último tipo consiste de rotinas encontradas com freqüência nas definições de tipos de objetos e por isto colocadas disponíveis a todos os objetos. A seguir comentaremos com detalhes cada um desses módulos e falaremos sobre a definição do nível de aplicação para objetos criados pelo usuário.

IV.1 Definição de Modelos

Inicialmente o usuário fornece o nome do modelo (fig. IV.2) que deseja especificar. Modelos anteriormente definidos podem ser alterados. Após especificar o nome do modelo são pedidos os nomes e tipos de objetos que compõe o sistema (fig. IV.3). Para cada objeto são solicitados os parâmetros correspondentes ao tipo especificado. Se este componente já tiver sido definido, a tela é mostrada

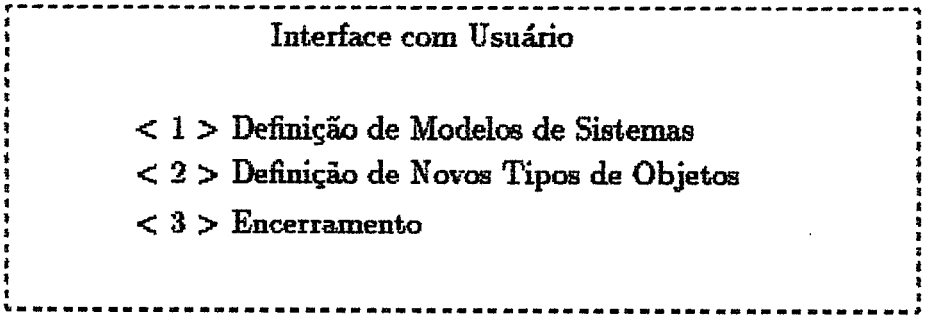

Figura IV.1: Tela inicial

com os parâmetros já preenchidos, podendo ser modificados, de outra forma, a tela é mostrada com os campos em branco.

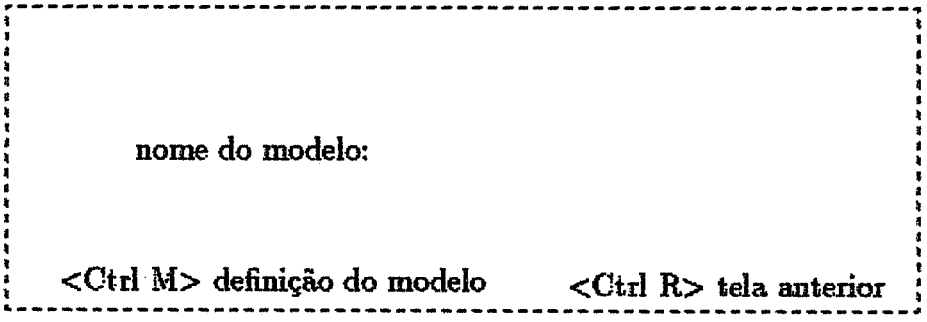

Figura IV.2: Definição do nome do modelo

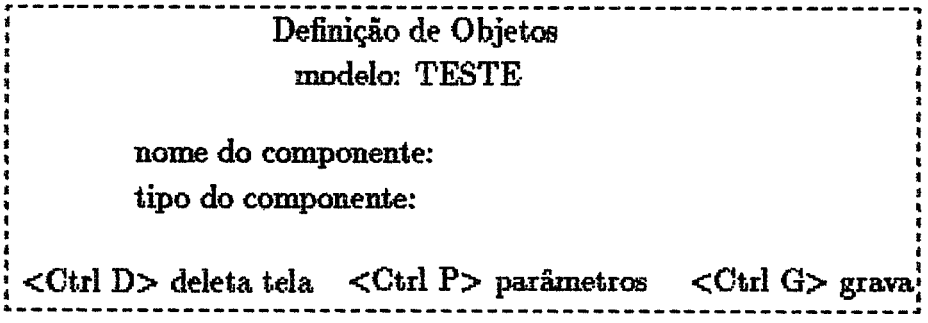

Figura IV.3: Definição dos nomes e tipos de objetos

Um exemplo deste módulo é mostrado na figura IV.4, onde são especificados os parâmetros para o EMISSOR do bit alternante discutido no capítulo anterior. O primeiro parâmetro fornece o nome do canal a ser utilizado pelo EMISSOR

para a transmissão das mensagens. O segundo parâmetro especifica o nome do objeto que receberá as mensagens do EMISSOR. O terceiro e o quarto parâmetros fornecem respectivamente a taxa de chegada das mensagens no EMISSOR e a taxa com que as mensagens são retransmitidas.

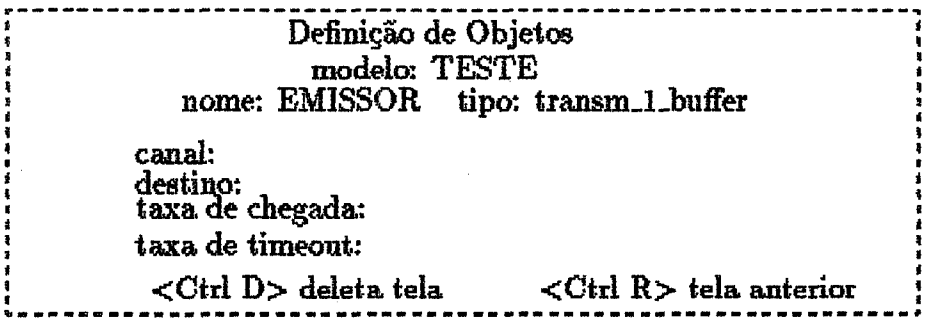

Figura IV.4: Definição dos parâmetros do bit alternante

No encerramento deste módulo é criado um arquivo com as cláusulas do nível de aplicação para cada objeto definido. Para o exemplo acima discutido, poderíamos ter a seguinte definição:

```
TIPO(emissor, transm_1_buffer).
CAMINHO(emissor, canal1, receptor).
TAXA_CHEGADA(emissor,0.5).
TIMEOUT(emissor,0.4).
INICIAL(emissor,(vazio,0,[ ],livre)).
<- RECONSULT(transm_1_buffer).
```

A cláusula RECONSULT carrega na memória do prolog a definição do tipo de objeto *transm_1_buffer*. As outras cláusulas já foram comentadas no capítulo anterior. Observe que o estado inicial não foi pedido ao usuário, pois é assumido que o objeto não possui nenhuma mensagem para transmitir e que o canal está livre.

Fazem também parte da biblioteca de tipos de objetos:

1. modelos de confiabilidade:
   - componente_save;
   - servidor_prior;
   - controlador_sistema;

- servidor_prior especial: idêntico ao tipo servidor_prior, exceto que, o evento *reparo* só ocorre quando o sistema não possui o número mínimo de componentes para continuar funcionando;

2. modelos de redes de filas:
   - fila_de_reparo;
   - fila_de_reinicialização;
   - servidor_infinito;
   - scheduler: retransmite uma mensagem recebida para a fila que tem menor tamanho dentre as filas com que está conectado;
3. modelos de protocolos de comunicação:
   - canal;
   - emissor e receptor do bit alternante;
   - emissor e receptor do go back n;
   - usuário e entidade do ABRACADABRA;

Para definir modelos de sistemas utilizando a interface desenvolvida neste trabalho, o usuário só dispõe dos quinze tipos de objetos acima descritos. Ao criar um novo tipo de objeto, poderá ser necessária a inclusão de uma nova tela neste módulo para permitir que o usuário final se abstrai dos níveis inferiores da ferramenta. Esta inclusão pode ser feita de maneira simples e rápida.

A nível de programação, a definição de um modelo é composta por um conjunto de registros de vários tipos, onde os registros guardam os parâmetros digitados pelo usuário referentes ao objeto. Para cada objeto definido, um ou mais registros são criados. No encerramento do módulo de definição de modelos, os registros são lidos, e um arquivo com as cláusulas do nível de aplicação é gravado. Uma operação inversa é feita quando o usuário pede para alterar um modelo de sistema anteriormente definido: o arquivo com as cláusulas do nível de aplicação é lido e os registros com as definições dos objetos são gravados. Assim, para incluir uma nova tela o usuário deverá:

1. alterar o programa AREA1: este programa possui a definição de todos os

tipos de registros. Na realidade, AREA1 funciona como uma área comum a todos os programas da interface. Deve-se incluir neste programa um novo tipo de registro para o objeto criado, caso não seja possível utilizar um dos tipos de registros existentes.

2. criar um programa com a descrição da nova tela: baseando-se nos programas já escritos, o usuário pode facilmente definir uma nova tela. Todos os programas feitos para a interface são basicamente divididos em dois módulos: parâmetros a serem recebidos e gravação destes parâmetros. No primeiro módulo, o usuário deve definir o formato da nova tela: quais os parâmetros a serem recebidos e quais teclas o usuário pode digitar para cada um dos parâmetros (números / letras / caracteres especiais). E no segundo módulo, as informações digitadas pelo usuário são gravadas nos registros.

3. alterar o programa TELA1: este programa recebe o nome e o tipo do objeto a ser definido (fig. IV.3). Apenas os quinze tipos de objetos acima descritos são considerados válidos pelo programa. O usuário deverá então incluir: (a) o nome do novo tipo de objeto na crítica feita pelo programa; (b) a chamada para a nova tela quando este tipo de objeto for especificado e o usuário teclar <Ctrl P>.

4. alterar o programa MODELO: este programa recebe o nome do modelo (fig. IV.2), e quando este modelo já existe, faz a leitura do arquivo com as cláusulas do nível de aplicação e grava os registros com as definições dos objetos. No arquivo lido, a definição de cada objeto é iniciada por uma cláusula TIPO, e só termina quando uma nova cláusula TIPO é lida ou quando é detectado o fim do arquivo. Ao ler uma cláusula TIPO, o programa formata o registro correspondente ao tipo de objeto especificado, e a cada nova cláusula do nível de aplicação lida, os campos correspondente no registro são preenchidos. Assim, o usuário deverá: (a) incluir na leitura da cláusula TIPO, a formatação do registro para o novo tipo de objeto; (b) incluir a leitura das cláusulas do nível de aplicação que só são geradas por este tipo de objeto.

5. alterar o programa MODULO1: este programa é responsável pela gravação do arquivo com as cláusulas do nível de aplicação. O usuário deverá, se especificou um novo tipo de registro, definir a gravação do nível de aplicação para cada um dos parâmetros recebidos.

IV.2 Definição de Tipos de Objetos

Este módulo tem por objetivo ajudar ao usuário que possui pouco conhecimento de Prolog. Inicialmente o usuário fornece o nome do tipo de objeto a ser definido (fig. IV.5). Então, são solicitados os eventos que o objeto pode gerar e as mensagens que pode receber. Isto é feito perguntando ao usuário se ele deseja definir um evento ou uma mensagem. Para cada evento são pedidos: o estado do objeto para o evento ocorrer, o estado do objeto após o evento ter sido gerado, as funções a serem executadas e a taxa de ocorrência do evento. Para cada mensagem a ser recebida são pedidos: os dados recebidos na mensagem, o estado do objeto quando a mensagem é recebida, o estado do objeto após o recebimento da mensagem, as funções a serem executadas e a probabilidade destas funções serem executadas. Esta probabilidade é 1 quando o objeto reage apenas de uma maneira ao recebimento de uma mensagem. Quando existem K maneiras distintas do objeto reagir, para cada uma dessas K maneiras, o usuário deve definir uma mensagem com um mesmo estado inicial e os mesmos dados recebidos; a soma dessas probabilidades deve necessariamente ser 1. Na realidade, o usuário não precisa repetir estas informaçõs, basta digitar a cláusula = para estado inicial e parâmetros recebidos, e a interface considera a última definição fornecida para estas duas cláusulas.

Cada função definida pelo usuário corresponde a uma ação a ser executada pelo objeto (ex: a leitura de um parâmetro do nível de aplicação) ou a uma condição para determinadas ações serem executadas (ex: o buffer do bit alternante não está vazio). O usuário terá acesso a definição das funções existentes e a criação de novas funções ao teclar <Ctrl F> na tela de definição de eventos e mensagens. Neste submódulo é solicitado o nome da função que o usuário

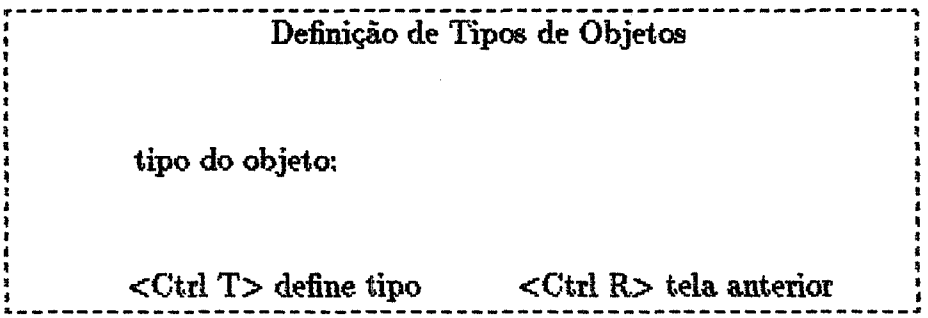

Figura IV.5: Definição de tipos de objetos

deseja ver ou definir (fig. IV.6). O não preenchimento deste campo e a digitação do <Ctrl F> implica na mostragem, em ordem alfabética, de todas as funções disponíveis na biblioteca. Caso o usuário forneça o nome de uma função da biblioteca, será mostrada a tela correspondente a definição da função, podendo o usuário pecorrer a biblioteca das funções a partir desta função. Caso o usuário peça uma função que não existe na biblioteca, será mostrada a tela para definição de funções. Se esta função já tiver sido definida anteriormente, a tela é mostrada preenchida. Para cada função são pedidos: os parâmetros utilizados na função e as funções executadas dentro desta função. Estas funções são agrupadas em conjuntos, onde cada conjunto corresponde a uma resposta possível a execução da função. É importante observar que o Prolog admite recursividade, podendo assim, a função chamar ela própria.

Um exemplo do uso deste submódulo é mostrado abaixo, onde é apresentada a definição da função ADIÇÃO.

```
parâmetros : A
             B
             C

funções 1 :  NUMB(A)
             NUMB(B)
             /()
             C := A + B

funções 2 :  A = 0
             /()
             C = B

funções 3 :  B = 0
             /()
             C = A

funções 4 :  C = A + B
```

Esta função faz a soma de dois valores, A e B, e coloca o resultado em C. Se A e B forem valores numéricos, C será igual a soma destes dois números. A cláusula /() impede que a função teste as outras respostas para a função ADIÇÃO se a primeira foi sucesso. Se A e/ou B não forem numéricos é testado se A não é zero, e depois se B não é zero. Caso nenhuma das condições anteriores tenham sido satisfeitas, a função retorna a equação A + B. Observe que o símbolo := em Prolog faz com que o lado direito da equação seja resolvido antes de unificar o resultado com a variável C, e o símbolo = unifica a equação com C sem resolvê-la.

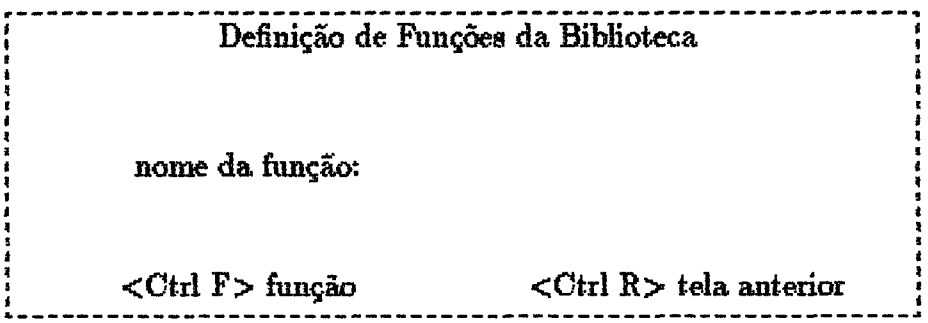

Figura IV.6: Definição de funções

Ao término deste submódulo é criado um arquivo em Prolog com a definição da função:

```
ADIÇÃO (A, B, C) <-
        NUMB(A),
        NUMB(B),
        /(),
        C := A + B .
ADIÇÃO (A, B, C) <-
        A = 0,
        /(),
        C = B .
ADIÇÃO (A, B, C) <-
        B = 0,
        /(),
        C = A.
ADIÇÃO (A, B, C) <-
        C = A + B .
```

Suponha agora a definição do tipo de objeto *fila_de_reinicialização* utilizado no segundo modelo de confiabilidade do capítulo anterior. O usuário entraria com os seguintes comandos:

```
Evento / Mensagem :   E
estado inicial :      Num_Comp_Falhos1

estado final :        Num_Comp_Falhos2

funções :             gt(Num_Comp_Falhos1,0)
                      taxa_reinicialização_processador(T)
                      Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1
                      tipo(Objeto,"controlador_sistema")
                      mensagem(mens("processador ok",1), Objeto)

taxa :                Num_Comp_Falhos1 * T

Evento / Mensagem :   E
estado inicial :      Num_Comp_Falhos1

estado final :        Num_Comp_Falhos2

funções :             gt(Num_Comp_Falhos1,0)
                      taxa_falha_intermitente_para_falha_permanente(T)
                      Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1
                      tipo(Objeto,"servidor_FCFS")
                      mensagem(mens("falha",1), Objeto)

taxa :                Num_Comp_Falhos1 * T

Evento / Mensagem :   M
parâmetros recebidos : "falha"
                       Num

estado inicial :      Num_Comp_Falhos1

estado final :        Num_Comp_Falhos2

funções :             adição(Num_Comp_Falhos1,Num,Num_Comp_Falhos2)

probabilidade :       1

Evento / Mensagem :   M
parâmetros recebidos : "reinicialização do sistema"

estado inicial :      Num_Comp_Falhos1

estado final :        Num_Comp_Falhos2

funções :             Num_Comp_Falhos2 = 0
                      tipo(Objeto,"controlador_sistema")
                      mensagem(mens("processador ok",Num_Comp_Falhos1), Objeto)

probabilidade :       1
```

Para enviar mensagens, deve ser utilizada a função mensagem(mens(x),destino) da biblioteca de funções, onde *x* representa os parâmetros a serem enviados, e *destino* é o nome do objeto que receberá a mensagem.

Observe que na definição da primeira mensagem é utilizada a função ADIÇÃO

comentada anteriormente. Esta função poderia ser substituida, neste exemplo, por Num_Comp_Falhos2 := Num_Comp_Falhos1 + Num.

Ao término deste módulo é gerado o seguinte programa em Prolog:

```
EVENTO (Nome_Objeto,Taxa,Estado,Novo_Estado) <-
        tipo(Nome_Objeto,"fila_de_reinicialização"),
        subst( f(Nome_Objeto,Num_Comp_Falhos1),Estado,
               f(Nome_Objeto,Num_Comp_Falhos2),Estado2),
        gt(Num_Comp_Falhos1,0),
        taxa_reinicialização_processador(T),
        Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1,
        tipo(Objeto,"controlador_sistema"),
        mensagem(mens("processador ok",1),Objeto,Prob_Mens1,Estado2,Estado3),
        Taxa = Prob_Mens1 * Num_Comp_Falhos1 * T,
        Novo_Estado = Estado3.
EVENTO (Nome_Objeto,Taxa,Estado,Novo_Estado) <-
        tipo(Nome_Objeto,"fila_de_reinicialização"),
        subst( f(Nome_Objeto,Num_Comp_Falhos1),Estado,
               f(Nome_Objeto,Num_Comp_Falhos2),Estado2),
        gt(Num_Comp_Falhos1,0),
        taxa_falha_intermitente_para_falha_permanente(T),
        Num_Comp_Falhos2 := Num_Comp_Falhos1 - 1,
        tipo(Objeto,"servidor_FCFS"),
        mensagem(mens("falha",1),Objeto,Prob_Mens1,Estado2,Estado3),
        Taxa = Prob_Mens1 * Num_Comp_Falhos1 * T,
        Novo_Estado = Estado3.
MENSAGEM (mens("falha",Num),Nome_Objeto,Prob,Estado,Novo_Estado) <-
            tipo(Nome_Objeto,"fila_de_reinicialização"),
            subst( f(Nome_Objeto,Num_Comp_Falhos1),Estado,
                   f(Nome_Objeto,Num_Comp_Falhos2),Estado2),
            adição(Num_Comp_Falhos1,Num,Num_Comp_Falhos2),
            Prob = 1,
            Novo_Estado = Estado2.
MENSAGEM (mens("para reinicialização"),Nome_Objeto,Prob,Estado,Novo_Estado) <-
            tipo(Nome_Objeto,"fila_de_reinicialização"),
            subst( f(Nome_Objeto,Num_Comp_Falhos1),Estado,
                   f(Nome_Objeto,Num_Comp_Falhos2),Estado2),
            Num_Comp_Falhos2 = 0,
            tipo(Objeto,"controlador_sistema"),
            mensagem(mens("processador ok",Num_Comp_Falhos1),Objeto,Prob_Mens1,Estado2,Estado3),
            Prob = Prob_Mens1,
            Novo_Estado = Estado3.

ADIÇÃO (A, B, C) <-
        NUMB(A),
        NUMB(B),
        /(),
        C := A + B .
ADIÇÃO (A, B, C) <-
        A = 0,
        /(),
        C := B .
ADIÇÃO (A, B, C) <-
        B = 0,
        /(),
        C = A.
ADIÇÃO (A, B, C) <-
        C = A + B .
```

A variável *Estado* representa o estado global do sistema antes de um evento ser gerado ou de uma mensagem ser recebida, e a variável *Novo_Estado* é o estado final do sistema. A função *tipo* verifica se o componente é do tipo

fila_de_reinicialização para poder gerar o evento e receber as mensagens. A função *subst* substitui o estado F(Nome_Objeto, Num_Comp_Falhos1) do componente por F(Nome_Objeto, Num_Comp_Falhos2), gerando um novo estado global. Se mensagens são enviadas para outros objetos, a probabilidade de cada possível reação em um objeto receptor é incluida no cálculo final da ocorrência do evento. Observe que a função ADIÇÃO é adicionada ao final do arquivo, pois ela não faz parte da biblioteca de funções disponíveis ao usuário, e portanto, é desconhecida pela ferramenta.

Para adicionar à biblioteca de funções uma função criada pelo usuário, deve-se incluir a definição Prolog da função no programa BASICS do VM PROLOG, e a descrição resumida do que faz a função no programa FUNÇÃO.TXT da interface. O primeiro programa possui a descrição Prolog das funções que são comuns a todos os objetos, e o segundo programa serve de consulta para o usuário da interface.

A descrição das funções disponíveis ao usuário se encontram no apêndice A.

Apesar deste módulo ter sido incluido na interface com o usuário, ele não está conectado ao nível de aplicação como sugerido na figura III.2 Considerando apenas os módulos desenvolvidos neste trabalho, temos, na realidade, os níveis descritos na figura IV.7.

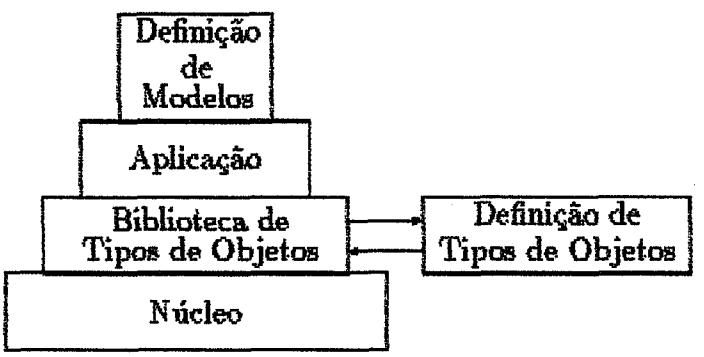

Figura IV.7: Níveis do Sistema

IV.3 Nível de Aplicação

Na definição de modelos ou na definição de tipos de objetos ou de funções, o usuário pode ter cometido erros. Para cada definição, é criado, em separado, um arquivo com os erros e os possíveis erros (advertências) cometidos pelo usuário. Um exemplo de erro é a digitação em um campo de probabilidade de um valor numérico maior que um. Um exemplo de advertência é a leitura de parâmetros do nível de aplicação (a interface não diferencia uma função de uma leitura de parâmetro). O nome do arquivo com os erros e advertências é então mostrado na tela para a verificação do usuário.

No exemplo da seção anterior, o usuário faz três leituras de parâmetros do nível de aplicação: TAXA_REINICIALIZAÇÃO_PROCESSADOR, TIPO e TAXA_FALHA_INTERMITENTE_PARA_FALHA_PERMANENTE. Como a ferramenta não encontra as definições de funções com estes nomes, um arquivo é gerado com advertências para estas três cláusulas.

A definição do nível de aplicação para objetos que não estão na biblioteca é da responsabilidade do usuário. Para o exemplo acima comentado, o usuário poderia criar o seguinte arguivo:

```
TIPO(cpu1,fila_de_reinicialização).
CAMINHO(cpu1,terminais,1).
TAXA(cpu1,μ5)

<-RECONSULT(fila_de_reinicialização).
<-RECONSULT(NUCLEO).
<-RECONSULT(BASICS).
```

A cláusula TIPO especifica o tipo do objeto. A cláusula RECONSULT carrega os programas na memória do Prolog, onde *fila_de_reinicialização* possui a definição criada pelo usuário, NUCLEO possui o programa que funciona como o nível *núcleo* da ferramenta, e BASICS possui as funções criadas especialmente para o usuário. Caso o sistema utilize mais de um tipo de objeto, mesmo que este esteja na biblioteca, o usuário deve utilizar a cláusula RECONSULT para carregá-lo na memória, além de definir os parâmetros correspondentes do nível de aplicação.

Capítulo V

Análise de Resultados

Ao final da execução, a ferramenta gera dois arquivos: ESTADOS e TAXAS. O primeiro arquivo possui a descrição simbólica dos estado globais, e o segundo arquivo possui as transições da cadeia de Markov: número do estado inicial, número do estado final e a taxa de transição. É interessante observar que as taxas de transição podem ser geradas de forma simbólica (valores não numéricos), permitindo, assim, analisar o comportamento de um modelo para diferentes parâmetros, sem que seja necessário reconstruir a cadeia de Markov. A geração em forma simbólica é importante também para a análise de sensibilidade. A partir destes dois arquivos é possível fazer a análise do sistema. Três programas-exemplo estão disponíveis para o uso do usuário: os dois primeiros programas lêem o arquivo com as transições da cadeia de Markov e calculam as probabilidades dos estados estacionários e as probabilidades transientes [21]; e o terceiro programa (para modelos de confiabilidade) lê o arquivo das probabilidades estacionárias e o arquivo com a descrição simbólica dos estados e calcula os limites, inferior e superior, da disponibilidade dos estados estacionários [18].

Na definição de modelos de confiabilidade o usuário especifica as condições pelas quais o sistema é considerado operacional; ao ler a descrição simbólica de um estado, o programa verifica se alguma dessas condições é satisfeita. Esta verificação é feita de forma simples. Cada condição de operabilidade do sistema é definida por uma cláusula CONDIÇÃO_DE_OPERAÇÃO. Esta cláusula fornece a lista dos objetos com os respectivos números mínimos de

componentes para que o sistema seja considerado operacional. Para cada um dos estados, o programa lê as cláusulas CONDIÇÃO_DE_OPERAÇÃO uma a uma e verifica se o sistema pode ser considerado operacional. Estas cláusulas simulam o diagrama de confiabilidade (Reliability Block Diagram). É fácil observar que da mesma maneira, *recompensas* podem ser definidas em alto nível para cada estado.

Como exemplo, a cadeia de Markov gerada a partir do protocolo do bit alternante, definido no capítulo III, possui 28 estados, e tem a seguinte descrição simbólica:

(EMISSOR,(vazio,0,[ ],livre,0)), (CANAL1,[ ]), (RECEPTOR,(0,0,livre)), (CANAL2,[ ])

⇓

(EMISSOR, (ocupado,0,[ ],transmitindo,0)), (CANAL1,0), (RECEPTOR,(0,0,livre)), (CANAL2,[ ]));

taxa = taxa_chegada

onde o primeiro estado corresponde ao estado inicial do protocolo do bit alternante: o EMISSOR está com o buffer vazio, a próxima mensagem a ser transmitida terá bit zero, não existe nenhuma mensagem esperando a liberação do canal para ser transmitida ([ ]), o canal que o EMISSOR utiliza está livre e o *timeout* não foi ativado; o CANAL1 está desocupado; o RECEPTOR está esperando por uma mensagem com bit zero, o último ack enviado pelo RECEPTOR tinha bit zero e o canal que o RECEPTOR utiliza está livre; o CANAL2 está desocupado. Após a chegada de uma mensagem o sistema passa para o segundo estado com taxa *taxa_chegada*, onde o EMISSOR tem o buffer ocupado e o CANAL1 possui uma mensagem para transmitir, com bit zero.

Seja ρ a probabilidade do estado estacionário que o sistema esteja com o buffer cheio. No modelo as mensagens chegam com taxa Poisson λ e são descartadas se o buffer já está ocupado. Portanto, o throughput do sistema é $\lambda(1 - \rho)$.

Considere um sistema que usa este protocolo e que tem canais com capacidade de 9600 e probabilidade de erro de 5 por cento e pacotes de 1024 bits. Assuma que a taxa de timeout é de 1 msg/seg. Variando a taxa de λ temos o gráfico do throughput pela taxa de chegada como mostrada na figura V.1.

Na figura V.2 é traçado o gráfico da probabilidade de ocorrência de timeout por erro de transmissão do canal em função do tempo.

O segundo exemplo, consiste no primeiro modelo de confiabilidade descrito no capítulo III. O sistema possui uma cadeia de Markov com 14 estados e um número máximo de três falhas concorrentes. A disponibilidade dos estados estacionários para este modelo é de 0.99884643453 quando os parâmetros do nível de aplicação são: pf = 1/720, dbf = 1/(2 × 720), fef = 1/(4 × 720), pr1 = 1, pr2 = 1, fer = 1, dbr = 1, p = 0.6 e c = 0.9. Na tabela V.1 é mostrado o cálculo da disponibilidade para as possíveis formas de agregação dos estados do sistema ([18]).

estados detalhados até K falhas	disponibilidade mínima	disponibilidade máxima	número de estados
0	0.99606824	1	5
1	0.99872860781	0.99896407961	9
2	0.99884629394	0.99884674152	16

Tabela V.1: Disponibilidade do sistema

Como exemplo do tempo que a ferramenta gasta para gerar a cadeia de Markov, considere o modelo de confiabilidade acima discutido. Novos modelos podem ser gerados a partir do aumento do número de processadores e do número de base de dados (neste exemplo o número de processadores e de base de dados foi multiplicado por dois, quatro, oito, dessesseis e trinta e dois). Os mesmos exemplos foram definidos utilizando a ferramenta SAVE. Os tempos de CPU necessários para a geração da cadeia de Markov nas duas ferramentas estão indicados no gráfico V.3. É importante observar a diferença significativa encontrada em favor da ferramenta desenvolvida em Prolog. Uma possível explicação para isso é o grande número de I/Os executado por SAVE. O tempo gasto inclui o tempo de CPU gasto com I/O (mas não o tempo de I/O), e esperamos que este tempo esteja causando um impacto significativo. Para uma melhor comparação sugerimos uma mudança nas duas ferramentas no sentido de se evitar I/Os, para que uma comparação mais refinada possa ser feita.

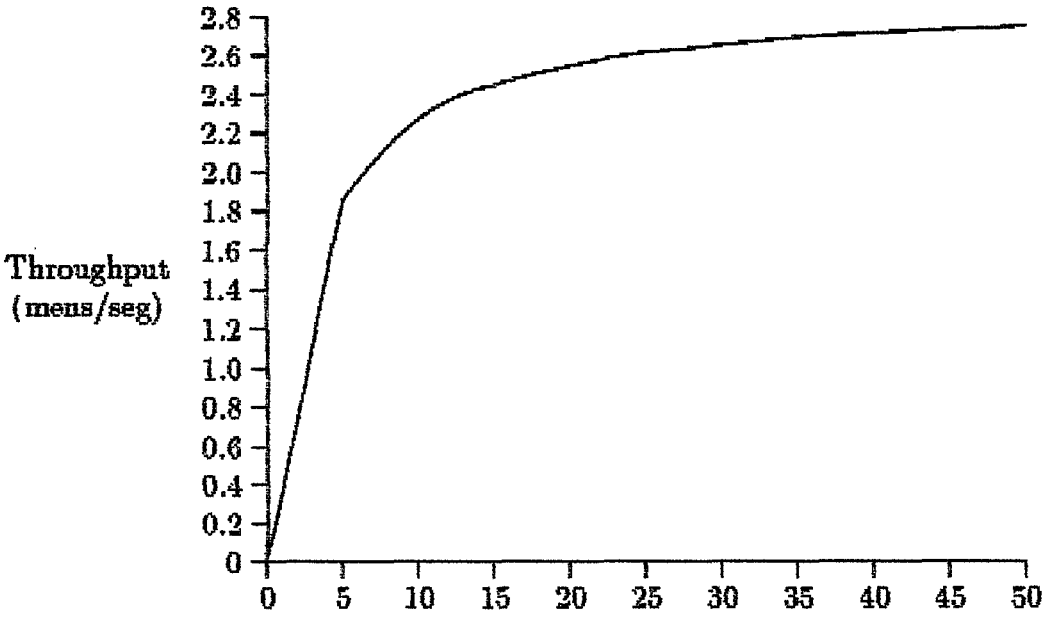

Figura V.1: Throughput do Bit Alternante

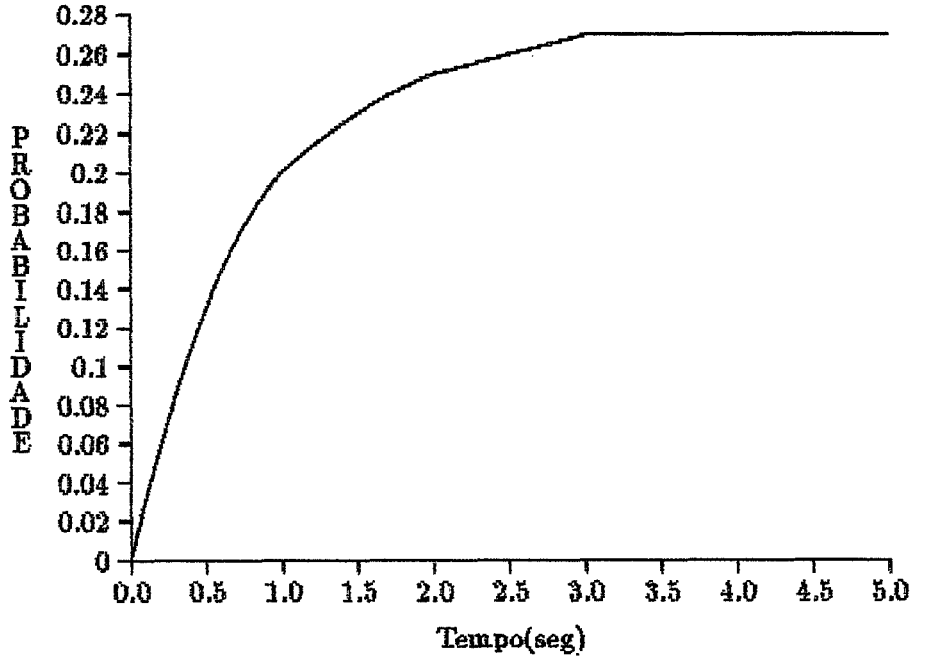

Figura V.2: Probabilidade de timeout por erro de transmissão

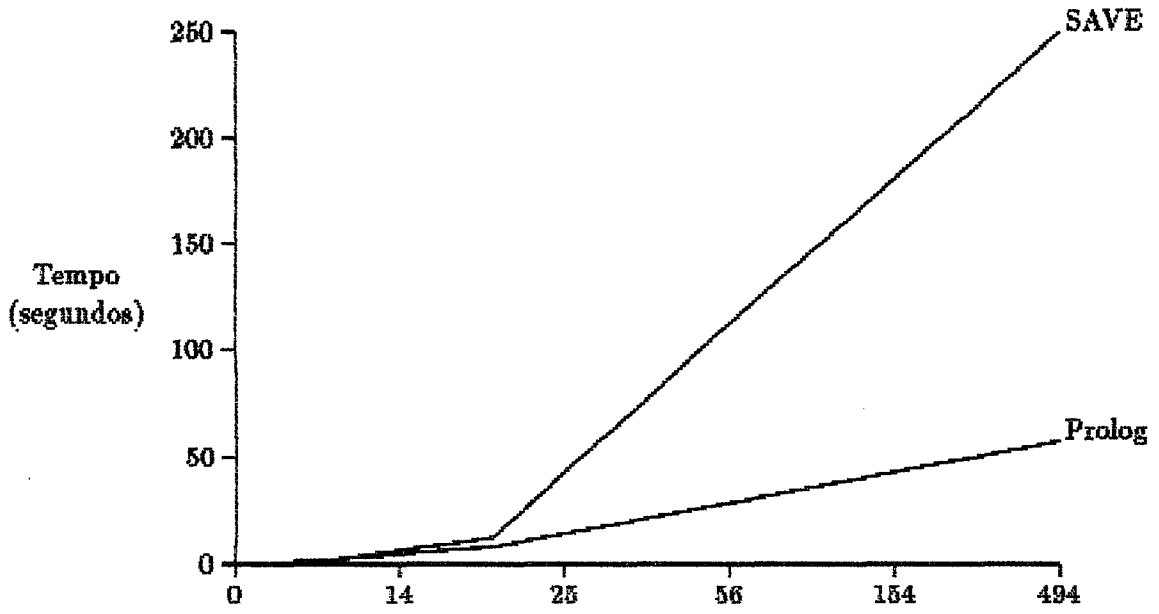

Figura V.3: Estados gerados por segundo

Entretanto, uma conclusão pode ser tirada com os exemplos: o tempo usado pela ferramenta Prolog é perfeitamente compatível com a ferramenta SAVE, o que mostra a grande utilidade da primeira, mesmo para modelos grandes.

O terceiro exemplo é o do protocolo ABRACADABRA descrito no capítulo III e que gerou 431 estados. Considere o estado onde usuário_A e entidade_A têm estados internos (CLOSED,0), o usuário_B tem estado OPEN e a entidade_B tem estado OPEN,M(0,0,0,FALSE). Este estado pode ocorrer quando há uma abertura de conexão e troca de mensagens, mas a entidade_A não consegue fechar a conexão. O fechamento da conexão pode ter sido pedido pelo usuário_A quando do fim da transmissão dos dados ou pela entidade_A devido a detectação de erro no recebimento dos acks. A probabilidade estacionária deste estado quando $\mu_1 = 20$, $\mu_2 = 20$, $p_1 = 0.7$, $p_2 = 0.5$, $p_3 = 0.5$, $c_1 = 9.375$ e $c_2 = 9.375$ (estes parâmetros estão definidos no nível de aplicação) é de 0.39×10^{-4}.

Um outro exemplo é o do modelo de redes de filas mostrado no capítulo III. Este sistema é composto por uma fila do tipo servidor_FCFS e por uma fila do tipo servidor_de_níveis_de_prioridade com dois níveis de prioridade, zero e um, e duas classes, A e B. As tarefas, neste exemplo, não mudam de classe. Para

o modelo que possui uma tarefa na classe A (maior prioridade) e duas tarefas na classe B, temos uma cadeia de Markov com nove estados. Podemos definir o throughput da tarefa da classe A como:

$$T = (c_1/\mu_1) \times \sum Pr_{A\ na\ frente} = (c_2/\mu_3) \times \sum Pr_{existe\ A}$$

onde $Pr_{A\ na\ frente}$ é a probabilidade da tarefa da classe A ser a próxima tarefa a ser atendida pelo servidor_FCFS, (c_1 / μ_1) é a taxa de serviço de servidor_FCFS para a classe A, $Pr_{existe\ A}$ é a probabilidade da tarefa da classe A está esperando na fila servidor_de_níveis_prioridade, e (c_2 / μ_3) é a taxa de serviço de servidor_níveis_de_prioridade para a classe A. O throughput para a tarefa da classe A, é então, 0.0993928, quando os parâmetros para o nível de aplicação são: c_1 / $\mu_1 = 0.2$, c_1 / $\mu_2 = 0.2$, c_2 / $\mu_3 = 0.3$ e c_2 / $\mu_4 = 0.1$.

Finalmente, considere o exemplo de redes de filas analisado em [22]. Este sistema é composto por uma fila com infinito servidores (centro de serviço 1) e por uma fila com dois servidores FCFS (centro de serviço 2). O sistema possui duas classes de tarefas (1 e 2), onde cada classe possui uma taxa específica de serviço em cada uma das filas. Tarefas não mudam de classe e possuem um número fixo dentro do sistema. Este modelo possui uma cadeia de Markov com 83 estados. Nas tabelas V.2 e V.3 são mostrados a utilização, tamanho médio da fila e tempo médio de serviço para cada uma das classes no centro de serviço 2 (FCFS) quando são usados os seguintes parâmetros (em [22] o modelo é resolvido por um método de aproximação proposto e os resultados são comparados a simulação; abaixo estão alguns resultados exatos):

- tempo médio de serviço (x_{ab} : a = classe e b = centro) : $x_{11} = 6$, $x_{21} = 3$, $x_{12} = 1$, x_{22} = variável;
- número de tarefas (N_a : a = classe) : $N_1 = 5$, $N_2 = 2$.

É interessante observar que a cadeia de Markov do sistema acima descrito foi gerada apenas uma vez. O estudo paramêtrico do modelo foi feito a partir da substituição das variáveis da descrição simbólica por valores numéricos.

x_{22}	Utilização	Tamanho médio da fila	Tempo médio na fila
2	0.33632	1.04444	1.55276
11	0.22358	2.37005	5.30025
20	0.16678	3.03814	9.10807

Tabela V.2: Classe 1

x_{22}	Utilização	Tamanho médio da fila	Tempo médio na fila
2	0.37194	0.87310	2.34738
11	0.71931	1.60768	12.29264
20	0.80600	1.75576	21.78358

Tabela V.3: Classe 2

Capítulo VI

Conclusões

A nossa experiência mostrou que a metodologia para definição de modelos é simples e poderosa. Utilizando-se de objetos previamente especificados, o usuário tem a liberdade de descrever modelos para diversas aplicações. Para facilitar a descrição do modelo, uma interface interativa para o usuário foi desenvolvida. A interface é útil não só na fase de construção do modelo, mas também na fase de construção de tipos de objetos. Para a construção de tipos de objetos, entretanto, o usuário necessita de um mínimo de conhecimento de Prolog. Porém, é importante ressaltar que o usuário final não precisa destes conhecimentos, uma vez que ele acessaria apenas a biblioteca de objetos previamente construídos.

A ferramenta gera a cadeia de Markov do sistema sendo modelado de forma simbólica, permitindo que o usuário possa selecionar transições ou estados a partir de uma descrição de alto nível. As transições e os estados selecionados servem tanto para a ferramenta de solução como para visualizar o comportamento do sistema.

Este trabalho foi inicializado a partir das definições de modelos propostas em [2]. Podemos dividir o desenvolvimento da ferramenta nas seguintes etapas:

1. especificação e teste do *núcleo*;
2. desenvolvimento de objetos para as áreas de confiabilidade, redes de filas e protocolos de comunicação;

3. análise de características estacionárias e transientes de modelos gerados a partir da biblioteca de tipos de objetos;
4. implementação da técnica de agregação de estados proposta em [18];
5. estudo comparativo com modelos gerados por outras ferramentas;
6. desenvolvimento da interface: definição de modelos de sistemas e de novos tipos de objetos em uma linguagem de alto nível;

Como pesquisa futura pode ser desenvolvido um banco de dados dos objetos e funções disponíveis ao usuário. Este banco de dados teria a descrição de todos os objetos e das suas funções, facilitando assim o uso da ferramenta. Outra sugestão seria a alteração do núcleo para simular o comportamento do sistema, mais especificamente, simular a cadeia de Markov. Atualmente, está sendo desenvolvida na UCLA uma linguagem que permita extrair resultados importantes a partir da descrição simbólica da cadeia de Markov, e está sendo estudada na UFRJ, a integração desta ferramenta com uma ferramenta de especificação formal de protocolos (ESTELLE), visando a análise de desempenho de protocolos especificados em ESTELLE.

Referências Bibliográficas

[1] GOYAL, A., CARTER, W.C., DE SOUZA E SILVA, E. e LAVENBERG, S.S. e TRIVEDI, K.S., *The System Availability Estimator*, Digest of the 16th Annual Symposium on Fault-Tolerant Computing, (Jul.), IEEE, New York, pp. 84-89, 1986.

[2] BERSON, S., DE SOUZA E SILVA, E. e MUNTZ, R.R., *An Object Oriented Methodology for Specification of Markov Models*, First International Workshop on the Numerical Solution of Markov Chains, North Carolina, janeiro 1990.

[3] JOHNSON, A. M. e MALEK, M., *Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability*, ACM Computing Surveys, vol. 20, No. 4, dezembro 1988.

[4] KLEINROCK, Leonard, *Queueing Systems, Volume I: Theory*, Wiley - Interscience Publication, 1975.

[5] ROSS, Sheldon, *A First Course in Probability*, Macmillan Publishing Co., Inc., NY, 1976.

[6] PETERSON, J.K., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ. USA, 1981.

[7] MOLLOY, Michael K., *Performance Analysis Using Stochastic Petri Nets*, IEEE Transactions of Computers, vol. c-31, No. 9, setembro 1982.

[8] CONWAY, A. E. e GOYAL, A., *Monte Carlo Simulation of Computer System Availability/Reliability Models*, IBM Research Report,RC 12459 (#55619), 12/15/86.

[9] SHAHABUDDIN, P., NICOLA, V. F., HEIDELBERGER, P., GOYAL, A. e GLYNN, Peter W. , *Variance Reduction in Mean Time to Failure Simulations*, IBM Research Report,RC 13910 (#62486), 8/10/88.

[10] NG, Ying W. e AVIZIENIS, A.A., *A Unified Reliability Model for Fault-Tolerant Computers*, IEEE Transactions on Computers, vol. c-29, No. 11, novembro 1980.

[11] GEIST, Robert M. e TRIVEDI, Kishor S., *Ultrahigh Reliability Prediction for Fault-Tolerant Computer Systems*, IEEE Transactions on Computers, Vol. c-32, N.12, dezembro 1983.

[12] TRIVEDI, Kishor, GEIST, Robert, SMOTHERMAN, Mark, e DUGAN, Joanne Bechta, *Hybrid Reliability Modeling of Fault-Tolerant Computer Systems*, Int. J. Comput. Electr. Eng., N.11, 2/3, 87-108, 1984.

[13] CARRASCO, J.A. e FIGUERAS, J., *METFAC: Design and Implementation of a Software Tool for Modeling and Evaluation of Complex Fault-Tolerant Computing Systems*, Digest of the 16th Annual Symposium on Fault-Tolerant Computing, IEEE, New York, pp. 424-429, julho 1986.

[14] SAUER, Charles H., MACNAIR, Edward A., e KUROSE, James F., *Queueing Network Simulations of Computer Communication*, IEEE Journal on Selected Areas in Communications, vol. sac-2, No. 1, janeiro 1984.

[15] CIARDO, G., MUPPALA, J. e TRIVEDI, K.S., *SPNP: Stochastic Petri Net Package*, Proceedings of the Third International Workshop on Petri Nets and Performance Models, Japão, Dezembro 1989.

[16] GOYAL, A., *System Availability Estimator (SAVE), User's Manual*, Thomas J. Watson Research Center, outubro 1988.

[17] IBE, Oliver C., SATHAYE, Archana, HOWE, Richard C. e TRIVEDI, K.S., *Stochastic Petri Net Modeling of VAXcluster System Availability*, Proceedings of the Third International Workshop on Petri Nets and Performance Models, Japão, Dezembro 1989.

[18] MUNTZ, R.R., DE SOUZA e SILVA, E. e GOYAL, A., *Bounding Availability of Repairable Computer Systems*, IEEE Transactions on Computers, Vol. c-38, N.12, pp. 1714 - 1723, dezembro 1989, também em Revista Brasileira de Computação, Vol. 5, N.2, pp. 19 - 30, outubro - dezembro 1989.

[19] TANENBAUM, A.S., *Computer NetWorks*, PRENTICE-HALL, INC., Engledwood Cliffs, New Jersey 07632, 1981.

[20] ISO/SC2/Wg1/FDT-A, *Guidelines for the application of Estelle, LOTOS and SDL*, Project/97.21.9/Q48.2 CCITT/SG/X/Q7, agosto 1987.

[21] DE SOUZA E SILVA, E. e GAIL, H.R., *Calculating Cumulative Operational Time Distributions of Repairable Computer Systems*, IEEE Transactions on Computers, Vol. c-35, N.4, abril 1986.

[22] DE SOUZA E SILVA, E. e MUNTZ, R.R., *Approximate Solutions for a Classe of Non-Product Form Queueing Network Models*, Performance Evaluation, Vol.7, 1987.

Apêndice A

Funções

Fazem parte da biblioteca de funções:

1. Funções Prolog IBM:
   - ABS(X): valor absoluto de X.
   - ATOM(X): sucede se X é um átomo.
   - ATOMIC(X): sucede se X é um átomo ou um número.
   - CALL(X) ou CALL(X,pref) ou CALL(X,pref,ind): X é um termo funcional ou um átomo; pref é um átomo ou um string vazio; ind é um número ou uma variável livre.
   - COMPUTE(type,exp,goal,varlist,list): unifica LIST com uma lista de valores de EXP no contexto das soluções do predicado GOAl.
   - COS(X): o cosseno de X. X deve estar em radianos.
   - CUT(lab) ou CUT(lab,n): pesquisa pelo n-ésimo label (iniciando do mais recente) que pode ser unificado com LAB (se não existe nenhum, um erro ocorre), e suprime o ponto de *backtrack* criado após a avaliação da regra contendo este label. CUT() ou /() corresponde que nenhuma cláusula anterior a este comando, dentro do mesmo predicado, deve ser reavaliada.
   - DIFF(X,Y,Z): resolve a equação Z = X - Y. Se um dos argumentos é um número de ponto flutuante, a operação é considerada de ponto flutuante. Pelo menos dois argumentos devem ser números. Se Z e X são valores numéricos e Y é uma variável livre, a diferença de X

e Z é unificada com Y. Se Z e Y são valores numéricos e X é uma variável livre, a soma de Z e Y é unificada com X.

- DIGIT(N): sucede se N é um inteiro entre 0 e 9.
- EQ(X,Y): sucede se e somente se X é igual a Y. X e Y são átomos, strings ou números.
- ERROR(): força um erro ocorrer.
- EXP(X): e^X.
- FAIL(): força um erro ocorrer.
- FL_TO_INT(F,I): converte um número de ponto flutuante F para um inteiro I.
- FLOATP(U): sucede se U é um número de ponto flutuante.
- GE(X,Y): sucede se e somente se X é maior ou igual a Y. X e Y são átomos, strings ou números.
- GT(X,Y): sucede se e somente se X é maior que Y. X e Y são átomos, strings ou números.
- INT(U): sucede se U é um número inteiro.
- LE(X,Y): sucede se X é menor ou igual a Y. X e Y são átomos, strings ou números.
- LOG(X): Log_e^X.
- LT(X,Y): sucede se X é menor que Y. X e Y são átomos, strings ou números.
- MAX(X,Y): o máximo de X e Y.
- MIN(X,Y): o mínimo de X e Y.
- NE(X,Y): sucede se e somente se X não é igual a Y. X e Y são átomos, strings ou números.
- NL() ou NL(arquivo) ou NL(arquivo,N): passa para o próximo registro do arquivo que pode estar sendo lido ou impresso.
- NUMB(U): sucede se U é um número.
- PI(): o número pi.

- PROD(X,Y,Z): unifica Z com o produto de X por Y. Se X e Y são inteiros, um produto inteiro é computado; de outra forma o produto é um número de ponto flutuante. O primeiro e o segundo argumento devem ser números e o terceiro argumento deve ser um número ou uma variável livre.
- QUOT(X,Y,Z): unifica Z com o quociente de X por Y. Se X e Y são inteiros, um quociente inteiro é computado;de outra forma o quociente é um número de ponto flutuante. O primeiro e o segundo argumento devem ser números e o terceiro argumento deve ser um número ou uma variável livre.
- RANDOM(X,Y,Z): unifica Z com um número aleatório entre X e Z. X e Y são números inteiros e Y é maior que X.
- READ(U) ou READ(U,arquivo) ou READ(U,arquivo,N): unifica U com um termo terminado por um ponto e lido de um arquivo.
- READAT(U) ou READAT(U,arquivo) ou READAT(U,arquivo,N): unifica U com um átomo,um caracter string ou um número lido de um arquivo.
- READCH(U) ou READCH(arquivo) ou READCH(U,arquivo,N): unifica U com um caracter lido de um arquivo.
- READEMPTY(): sucede se o proximo READCH provoca a leitura de uma nova linha.
- READLI(U) ou READLI(arquivo) ou READLI(U,arquivo,N): unifica U com uma linha lida de um artigo.
- REM(X,Y,Z): unifica Z com o resto da divisão de X por Y. Se X e Y são inteiros, um resto inteiro é computado; de outra forma o resto é zero com ponto flutuante. O primeiro e o segundo argumento devem ser números e o terceiro argumento deve ser um número ou uma variável livre.
- SIN(X): o seno de X. X deve estar em radianos.

- ST_TO_AT(S,A): converte um string S em um número ou um átomo e vice versa.
- ST_TO_LI(X,Y): converte um string S em uma lista Y de caracteres ou dígitos.
- ST_TO_NB(S,N): converte um string S em um número N.
- STLEN(X,Y): unifica Y com o tamanho do string X.
- STRINGP(U): sucede se U é um string.
- SUBSTRING(ST,SST,I,L): unifica SST com o substring de ST de tamanho L que começa na posição I.
- SUM(X,Y,Z): resolve a equação Z = X + Y. Se um dos argumentos é um número de ponto flutuante, a operação é considerada de ponto flutuante. Pelo menos dois argumentos devem ser números. Se Z e X são valores numéricos e Y é uma variável livre, a diferença entre Z e X é unificada com Y. Se Z e Y são valores numéricos e X é uma variável livre, a diferença entre Z e Y é unificada com X.
- TAB(col) ou TAB(col,arquivo): posiciona a próxima saida de um arquivo na coluna COL. Se COL é menor que a posição atual, um erro ocorre.
- TRUE(): o predicado sempre sucede.
- VAR(U): sucede se U for uma variável livre.
- WRITE(U) ou WRITE(U,arquivo): imprime a expressão U e termina com um ponto e uma nova linha.
- WRITES(U) ou WRITES(U,arquivo): imprime a expressão U sem terminar com um ponto ou com uma nova linha.
- WRITEX(U) ou WRITEX(U,arquivo): imprime a expressão U sem terminar com um ponto ou com uma nova linha. Átomos são impressos sem aspas (").

2. Funções especiais criadas:
   - ADD(X,Y,Z): soma X com Y e unifica o resultado com Z. X e Y podem ser números ou átomos e Z deve ser uma variável livre.

- APPEND(X,Y,Z): junta duas listas, X e Y, e unifica o resultado com Z. X e Y devem ser listas e Z deve ser uma variável livre. Os elementos da lista Y são colocados na lista Z após todos os elementos de X.
- COMBINATION(X,Y,Z): Z é unificado com a combinação de X em Y. X e Y devem ser números inteiros e Z uma variável livre.
- DIVIDE(X,Y,Z): Divide X por Y e unifica o resultado com Z. X e Y podem ser números ou átomos e Z deve ser uma variável livre.
- FATORIAL(X,Y): o fatorial de X é unificado com Y. X deve ser um número inteiro e Y uma variável livre.
- MEMBER(X,Y): sucede se X é um elemento da lista Y. X deve ser um número, string ou átomo e Y deve ser uma lista.
- MESSAGE(mens(dado1,dado2,...,dadon), comp): envia a mensagem mens(dado1,dado2,...,dadon) para o objeto COMP.
- MULTIPLY(X,Y,Z): multiplica X por Y e unifica o resultado com Z. X e Y podem ser números ou átomos e Z deve ser uma variável livre.
- NONMEMBER(X,Y): sucede se X não é um elemento da lista Y. X deve ser um número,string ou átomo e Y deve ser uma lista.
- SUBST(X,Y,Z,K): substitui o elemento X da lista Y pelo elemento Z, a nova lista é unificada com K. X e Z devem ser números,strings ou átomos, Y deve ser uma lista e K uma variável livre.
- SUBT(X,Y,Z): retira o elemento X da lista Y, e a nova lista é unificada com Z. X deve ser um número,string ou átomo, Y deve ser uma lista e Z uma variável livre.
- SUBTRACT(X,Y,Z): subtrai Y de X e unifica o resultado com Z. X e Y podem ser números ou átomos e Z deve ser uma variável livre.

Apêndice B

Implementação IBM

Para o correto uso da ferramenta, o usuário precisa inicialmente preparar o ambiente em que a ferramenta irá ser executada. Os dois arquivos gerados pelo *núcleo* são colados em um minidisco temporário B. Assim, o usuário deve criar e formatar este minidisco.

```
define T3375 200 5
format 200 b
```

O primeiro comando define um minidisco temporário com endereço virtual 200 ocupando 5 cilindros. Este minidisco será automaticamente removido quando o usuário encerrar a sessão (LOGOFF). O segundo comando formata o minidisco anteriormente definido. Minidiscos recém criados devem ser formatados antes de serem usados.

São necessários pelo menos 2 megabytes de memória virtual para carregar o VM Prolog. Para o usuário que é autorizado a definir uma memória virtual de até 10 megabytes, ele pode entrar com os seguintes comandos:

```
def stor 10m
cp ipl cms
vmprolog gs 5100k ls 3500k trail 800k
```

O primeiro comando redefine a memória virtual para 10 megabytes (o sistema assume uma memória virtual de 1 megabytes para cada usuário). Após a

reconfiguração, o usuário deve digitar o segundo comando para recarregar o CMS, pois o anterior é destruido pelo comando DEF STOR. O terceiro comando não só carrega o VM Prolog, como também define os valores que as pilhas de memória devem possuir. Estes valores foram considerados ideais para muito dos exemplos executados, mas fica sob a responsabilidade do usuário alterá-los quando necessário.

Para gerar a cadeia de Markov e a representação simbólica dos estados de um modelo de sistema, cujas cláusulas do nível de aplicação se encontram no arquivo TESTE, o usuário deve entrar com os seguintes comandos após carregar o VM Prolog:

```
<- new().
<- consult(teste).
<- expand(*).
<- fin.
```

O primeiro comando limpa a memória do Prolog de qualquer cláusula que possa ter. Este comando não é necessário quando anteriormente nenhum programa foi colocado na memória. O segundo comando faz com que o Prolog carregue o arquivo TESTE em sua memória. O terceiro comando chama o nível *núcleo* para a execução da ferramenta. E o quarto comando sai do VM Prolog.

Para ver os arquivos gerados pela interface o usuário deve digitar FLIST * * B. Pelo menos dois arquivos deverão ser mostrados: TAXAS e ESTADOS. O primeiro arquivo guarda as transições de estado do modelo e o segundo arquivo possui a definição simbólica dos estados.

O usuário deve digitar STSTAV, no CMS, para gerar as probabilidades dos estados estacionários, e STSTAV2 para gerar as probabilidades dos estados em relação a um parâmetro de tempo. Antes de usar estes dois programas o usuário deve alterar o tamanho das matrizes que guardam os estados da cadeia, e para o segundo programa, deve, também, alterar o parâmetro de tempo para o qual serão geradas as probabilidades de estados. No final é

fornecido ao usuário o arquivo EIGEN no minidisco temporário B com as probabilidades dos estados. Estes programas foram feitos em FORTRAN que não é capaz de fazer execuções quando uma memória virtual de mais de 8 megabytes foi definida. Neste caso, o usuário deve redefinir a memória virtual para no máximo 8 megabytes.

Para calcular a disponibilidade de um modelo de confiabilidade cujos arquivos TAXAS, ESTADOS e EIGEN já foram gerados, o usuário deve entrar com os seguintes comandos:

```
prolog
<- consult(dispon).
<- dispon.
<- fin.
```

O primeiro comando chama o Prolog que já foi carregado anteriormente, o segundo comando carrega o programa DISPON, e o terceiro comando calcula a disponibilidade. O resultado do cálculo é colocado no arquivo BOUNDS no minidisco temporário B.

Para o correto uso da ferramenta, o usuário precisa inicialmente preparar o ambiente em que a ferramenta irá ser executada. Os dois arquivos gerados pelo *núcleo* são colados em um minidisco temporário B. Assim, o usuário deve criar e formatar este minidisco.

```
define T3375 200 5
format 200 b
```

O primeiro comando define um minidisco temporário com endereço virtual 200 ocupando 5 cilindros. Este minidisco será automaticamente removido quando o usuário encerrar a sessão (LOGOFF). O segundo comando formata o minidisco anteriormente definido. Minidiscos recém criados devem ser formatados antes de serem usados.

São necessários pelo menos 2 megabytes de memória virtual para carregar o VM Prolog. Para o usuário que é autorizado a definir uma memória virtual de até 10 megabytes, ele pode entrar com os seguintes comandos:

```
def stor 10m
cp ipl cms
vmprolog gs 5100k ls 3500k trail 800k
```

O primeiro comando redefine a memória virtual para 10 megabytes (o sistema assume uma memória virtual de 1 megabytes para cada usuário). Após a reconfiguração, o usuário deve digitar o segundo comando para recarregar o CMS, pois o anterior é destruído pelo comando DEF STOR. O terceiro comando não só carrega o VM Prolog, como também define os valores que as pilhas de memória devem possuir. Estes valores foram considerados ideais para muito dos exemplos executados, mas fica sob a responsabilidade do usuário alterá-los quando necessário.

Para gerar a cadeia de Markov e a representação simbólica dos estados de um modelo de sistema, cujas cláusulas do nível de aplicação se encontram no arquivo TESTE, o usuário deve entrar com os seguintes comandos após carregar o VM Prolog:

```
<- new().
<- consult(teste).
<- expand(*).
<- fin.
```

O primeiro comando limpa a memória do Prolog de qualquer cláusula que possa ter. Este comando não é necessário quando anteriormente nenhum programa foi colocado na memória. O segundo comando faz com que o Prolog carregue o arquivo TESTE em sua memória. O terceiro comando chama o nível *núcleo* para a execução da ferramenta. E o quarto comando sai do VM Prolog.

Para ver os arquivos gerados pela interface o usuário deve digitar FLIST * * B. Pelo menos dois arquivos deverão ser mostrados: TAXAS e ESTADOS. O

primeiro arquivo guarda as transições de estado do modelo e o segundo arquivo possui a definição simbólica dos estados.

O usuário deve digitar STSTAV, no CMS, para gerar as probabilidades dos estados estacionários, e STSTAV2 para gerar as probabilidades dos estados em relação a um parâmetro de tempo. Antes de usar estes dois programas o usuário deve alterar o tamanho das matrizes que guardam os estados da cadeia, e para o segundo programa, deve, também, alterar o parâmetro de tempo para o qual serão geradas as probabilidades de estados. No final é fornecido ao usuário o arquivo EIGEN no minidisco temporário B com as probabilidades dos estados. Estes programas foram feitos em FORTRAN que não é capaz de fazer execuções quando uma memória virtual de mais de 8 megabytes foi definida. Neste caso, o usuário deve redefinir a memória virtual para no máximo 8 megabytes.

Para calcular a disponibilidade de um modelo de confiabilidade cujos arquivos TAXAS, ESTADOS e EIGEN já foram gerados, o usuário deve entrar com os seguintes comandos:

```
prolog
<- consult(dispon).
<- dispon.
<- fin.
```

O primeiro comando chama o Prolog que já foi carregado anteriormente, o segundo comando carrega o programa DISPON, e o terceiro comando calcula a disponibilidade. O resultado do cálculo é colocado no arquivo BOUNDS no minidisco temporário B.

Apêndice C

Problemas de Implementação

No desenvovimento da ferramenta foram encontrados alguns problemas durante a fase de implementação. Era nossa intenção inicial que a ferramenta fosse toda implementada em um micro computador PC, tendo o *núcleo* e os primeiros tipos de objetos sidos escritos em ARIT Prolog. Mas a impossibilidade de gerar grandes números de estados nos levou a converter os programas para o Prolog do IBM 4381. No ARIT Prolog só quinze estados de um determinado modelo de rede de filas foram gerados antes de ocorrer um erro por falta de memória. No Prolog IBM, o mesmo modelo, nos permitiu gerar mais de cinco mil estados.

Mas o prolog IBM também apresentou limitações. Este *software* trabalha com quatro pilhas de memória: *Global* que guarda as variáveis globais; *Local* que guarda as variáveis locais; *Trail* que guarda informações sobre as variáveis utilizadas pelo Prolog durante o *backtracking*; e a *área das cláusulas*, onde as cláusulas e as constantes são armazenadas. A alocação de memória é feita de maneira estática: não há transferência de memória livre entre as pilhas. Uma vez esgotada a área de uso de uma pilha, o programa é abortado por falta de memória, mesmo que as outras pilhas possuam grandes áreas de memória disponíveis. Como o uso destas pilhas depende do programa sendo executado, fica impossível determinar uma alocação que satisfaça a todos os modelos de sistemas.

Uma outra limitação imposta é a do uso máximo de 15 Megabytes de memória

virtual por usuário do IBM 4381. Assim o máximo de 15M podem ser distribuidos entre as três pilhas de memória.

Como exemplo da alocação de memória, utilizamos um modelo de rede de filas que gera uma cadeia de Markov com 10.626 estados. Inicialmente foi usada a alocação padrão do Prolog IBM (Global: 80K; Local: 380K; Trail: 20K; área das cláusulas: 400K) que permitiu a geração de apenas setenta estados. A partir desta alocação inicial foram geradas cadeias de Markov multiplicando os valores das pilhas por 2 até 30. Na figura C.1 é traçado o gráfico de alocação de memória por número de estados deste exemplo.

Após vários testes chegamos a conclusão que, para este exemplo, a alocação ideal da memória era de 7290K para a pilha Global, 4950K para a pilha Local e 720K para a pilha Trail. A partir destes dados foi traçado o gráfico da figura C.2.

Para os dois gráficos foi mantida sempre a alocação de 400K para a área das cláusulas, pois esta área guarda o programa que está sendo executado, e portanto, é uma área que não necessita de alocar memória durante a execução.

Observe nos gráficos, que diferentes alocações de memória para as pilhas, podem fornecer diferentes números de estados, apesar de um mesmo total de memória usada.

Um outro problema encontrado no uso do Prolog IBM foi a precisão de apenas 6 casas decimais na resolução de equações numéricas. Para o cálculo das probabilidades dos estados estacionários era necessário que uma linguagem com maior precisão fosse utilizada. A linguagem escolhida foi FORTRAN, pois fornece uma precisão de até 32 casas decimais. Entretanto, este programa não faz parte da ferramenta, mas é um dos programas que colocamos disponíveis ao usuário para a análise da cadeia de Markov.

E por último, tivemos dificuldade na escolha da linguagem para a implementação da interface com o usuário. Não havia, na época que foi iniciado o desenvolvimento deste nível da ferramenta, um software disponível no IBM 4381 que satisfizesse todas as exigências para manipulação de telas e de ar-

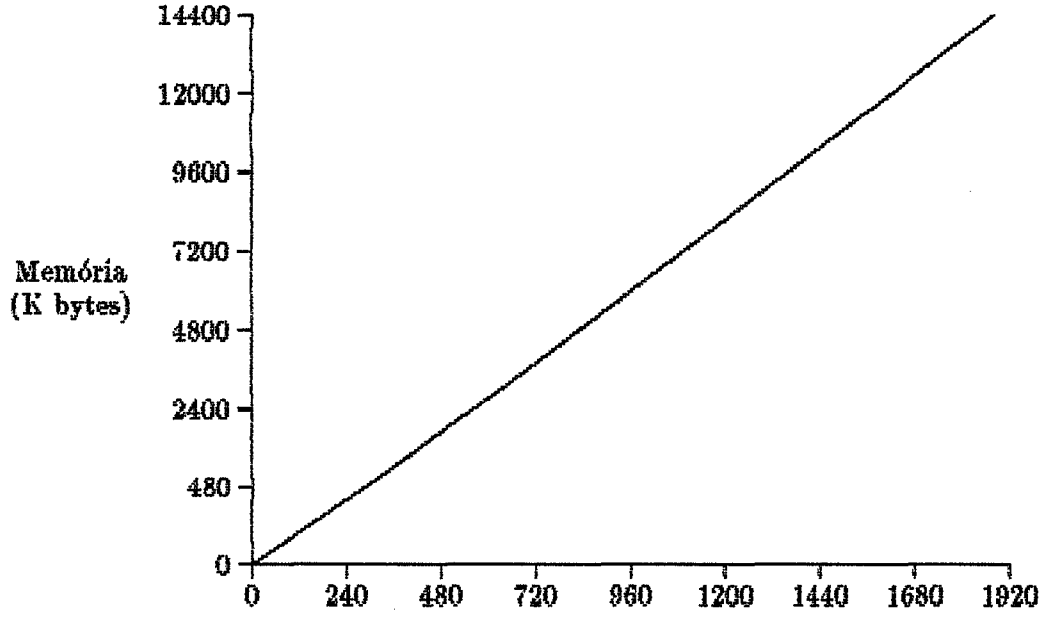

Figura C.1: Estados gerados por alocacão de memória

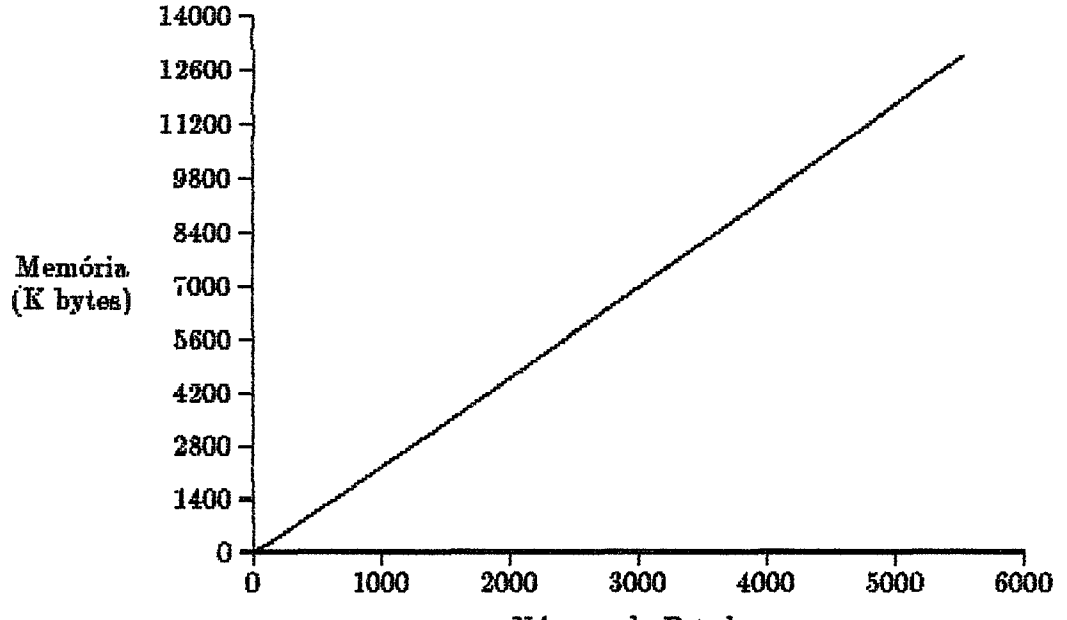

Figura C.2: Estados gerados por alocacão de memória

quivos. A interface foi então desenvolvida utilizando o turbo Pascal 5.0 para micros PC. A transferência dos arquivos do micro PC para o IBM 4381 fica sob a responsabilidade do usuário.

Apêndice D

Núcleo

```
/* ************************************************************** */
/* ROTINA: NUCLEO                                                 */
/* OBJETIVO: COMANDAR A EXECUCAO DAS ROTINAS                      */
/* ************************************************************** */

<- MODULE(*).


/* CRIA HASH TABLE E COMANDA EXECUCAO */
EXPAND(*HASH_TABLE) <-
        INITIAL(*S) &
        SKEL(H,220,*VISITED) &
        SYSTEM('ACCESS 200 B',*) &
        SYSTEM('ERASE RATES PROLOG B',*) &
        SYSTEM('ERASE STATES PROLOG B',*) &
        DCIO(FILE1,OUTPUT,FILE,'RATES',PROLOG,B,V,80) &
        DCIO(FILE2,OUTPUT,FILE,'STATES',PROLOG,B,V,80) &
        BLOC_DELETE(REDUCED_STATE) &
        NEXISTS(*S,*VISITED,0) &
        TRACE(*S,X(0,*S),1,*Z - *Z,*VISITED) &
        DCIO(FILE1, CLOSE) &
        DCIO(FILE2, CLOSE) &
        /() & FAIL.


/* COMANDA OS EVENTOS EM CADA COMPONENTE DOS ESTADOS */
TRACE(NIL,*_STATE,*_NUMBER,*Z1 - *Z2,*_VISITED) <-
        *Z1 == *Z2 &
        /().
TRACE(NIL,*_STATE,*NUMBER,(X(*NUM,*STATE).*TAIL) - *Z,*VISITED) <-
        TRACE(*STATE,X(*NUM,*STATE),*NUMBER,*TAIL - *Z,*VISITED).
TRACE(*COMP.*TAIL,X(*S_NUM,*STATE),*NUMBER,*FRONT - *BACK,*VISITED) <-
        XCOMPUTE(LIST,R(*NEW_STATE,*RATE),REACT(*COMP,*RATE,*STATE,
                *NEW_STATE),NIL,*STATE_LIST) &
        UNDUP(*S_NUM,*STATE,*STATE_LIST,*VISITED,*NEW_STATE_LIST,*NUMBER,
                *NEW_NUMBER) &
        APPEND(*NEW_STATE_LIST,*NEW_BACK,*BACK) &
        TRACE(*TAIL,X(*S_NUM,*STATE),*NEW_NUMBER,*FRONT - *NEW_BACK,
                *VISITED).


/* COMANDA A IMPRESSAO E A NUMERACAO DOS ESTADOS */
UNDUP(*C_NUM,*,NIL,*VISITED,NIL,*NUMBER,*NUMBER).
UNDUP(*C_NUM,*STATE,(R(*HEAD,*RATE).*TAIL),*VISITED,(X(*NUMBER,*HEAD).
        *NEW_TAIL),*NUMBER,*NEW_NUMBER) <-
        NEXISTS(*HEAD,*VISITED,*NUMB) &
        PRINT_TRAN(*C_NUM,*NUMB,*NUMBER,*RATE) &
        /() &
        *NUM := *NUMBER + 1 &
        UNDUP(*C_NUM,*STATE,*TAIL,*VISITED,*NEW_TAIL,*NUM,*NEW_NUMBER).
```

```
UNDUP(*C_NUM,*STATE,(*_HEAD.*TAIL),*VISIT,*NEW_TAIL,*NUMBER,*NEW_NUMBER)
        <- UNDUP(*C_NUM,*STATE,*TAIL,*VISIT,*NEW_TAIL,*NUMBER,*NEW_NUMBER).


/* IMPRIME ESTADOS E TAXAS */
PRINT_TRAN(*C_NUM,*N1,*N1,*RATE) <-
        /() &
        TAB(2,FILE1) &
        WRITEX(*C_NUM,FILE1) &
        TAB(9,FILE1) &
        WRITEX(*N1,FILE1) &
        TAB(16,FILE1) &
        WRITEX(*RATE,FILE1) &
        NL(FILE1).
PRINT_TRAN(*C_NUM,*N1,*,*RATE) <-
        TAB(2,FILE1) &
        WRITEX(*C_NUM,FILE1) &
        TAB(9,FILE1) &
        WRITEX(*N1,FILE1) &
        TAB(16,FILE1) &
        WRITEX(*RATE,FILE1) &
        NL(FILE1) &
        FAIL.


/* VERIFICA SE UM ESTADO GERADO JA EXISTE NA HASH TABLE */
NEXISTS(*STATE,*VISITED,*NUMBER) <-
         HASH(*STATE,1,*BUCKET) &
         REM(*BUCKET,220,*REMAINDER) &
         SUM(*REMAINDER,1,*NEW_BUCKET) &
         ARG(*NEW_BUCKET,*VISITED,*LIST) &
         QMEMBER(*STATE,*LIST,*NUMBER).


/* VERIFICA SE POSICAO NA HASH TABLE ESTA VAZIA */
QMEMBER(*STATE,*NEX,*NUMBER) <-
            VAR(*NEX) &
            /() &
            *NEX = (X(*NUMBER,*STATE).*) &
            TAB(0,FILE2) &
            WRITEX(*STATE,FILE2) &
            WRITEX(" .",FILE2) &
            NL(FILE2).

/* VERIFICA SE ESTADO EH IGUAL AO ESTADO DA HASH TABLE
QMEMBER(*STATE,(X(*NUM,*STATE).*_TAIL),*NUMBER) <-
            /() &
            *NUM = *NUMBER.

/* VERIFICA PROXIMO ESTADO DA POSICAO DA HASH TABLE */
QMEMBER(*STATE,(*_HEAD.*_TAIL),*NUMBER) <-
            QMEMBER(*STATE,*_TAIL,*NUMBER).


<- DCIO(*, CLOSE).

/* ********************** FIM DO JOB ********************************** */
```