Um Ambiente de Edição e Simulação de Estadogramas

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela Sra. **Valéria Gonçalves S. Elias** e aprovada pela Comissão Julgadora.

Campinas, 09 de dezembro de 1992

Prof. Dr. ____________________
Hans Kurt Edmund Liesenberg

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do título de MESTRE em Ciência da Computação.

UNICAMP
BIBLIOTECA CENTRAL

A meus pais.
A Ivana, a Luciana.
A Caroline e a seu irmãozinho.
A Glêdson.
A Deus.

Conteúdo

Agradecimentos vii

Resumo viii

1 Introdução 1
- 1.1 O Paradigma de Estados/Transições . . . 1
  - 1.1.1 Um modelo de estados/transição: Redes de Petri . . . 2
- 1.2 Sistemas Reativos . . . 5
- 1.3 Estadogramas . . . 7
- 1.4 Motivação para o Trabalho de Tese . . . 7
- 1.5 Organização dos capítulos . . . 8

2 Editores Gráficos 9
- 2.1 TEDMOS - Um Sistema de CAD para ensino de projeto de circuitos microeletrônicos de alta integração . . . 9
  - 2.1.1 Como especificar um projeto de grande complexidade . . . 9
  - 2.1.2 Visão geral do TEDMOS . . . 10
  - 2.1.3 O Editor Gráfico . . . 10
  - 2.1.4 O Verificador de Regras de Projeto . . . 11
  - 2.1.5 O Simulador . . . 12
- 2.2 Ferramenta CAD para projetos de sistemas de irrigação - IRRIGRAPHOS . . . 12
- 2.3 LACSUZ - Um Sistema Editor de Microcircuitos . . . 14
  - 2.3.1 Como se faz a indicação de pontos . . . 14
  - 2.3.2 Facilidades de Visualização . . . 15
  - 2.3.3 Facilidades de Armazenamento . . . 15
  - 2.3.4 Desenho estruturado . . . 15
- 2.4 Editores Gráficos para Projeto de Circuitos Integrados . . . 15
  - 2.4.1 Visão Geral do EDMOS . . . 15
  - 2.4.2 Visão Geral do EDCI . . . 17
- 2.5 Editor Diagramático para DFD, baseado em formalismos gramaticais . . . 18
  - 2.5.1 Especificação da Sintaxe . . . 18
  - 2.5.2 Especificação da Semântica . . . 19
  - 2.5.3 Aspectos de Implementação . . . 20
- 2.6 Um Editor de Diagrama de Blocos para Projeto de Sistemas de Controle . . . 20
  - 2.6.1 Implementação do Editor de Diagrama de Blocos . . . 21

2.6.2 Interface Homem-Máquina . . . 23
2.6.3 Interface Gráfica . . . 24
2.7 Estudo Comparativo dos Editores Gráficos . . . 25

3 Estadogramas: Estrutura e Funcionamento **27**
3.1 Descrição Gráfica . . . 27
3.1.1 Comunicação em Estadogramas . . . 29
3.1.2 History . . . 29
3.1.3 O History de Retorno . . . 31
3.1.4 Ações e Atividades . . . 33
3.2 O formalismo dos estadogramas . . . 33
3.2.1 Sintaxe . . . 36
3.2.2 Semântica . . . 37
3.3 Ferramentas para Descrição de Sistemas Reativos através de Estadogramas . . . 40
3.3.1 STATEMATE . . . 40
3.3.2 Um Editor Gráfico para Estadogramas . . . 41
3.3.3 Gerador de Gerenciadores de Sistemas Reativos . . . 42

4 O Editor de Estadogramas **47**
4.1 Necessidade da Ferramenta . . . 47
4.2 Passos de uma especificação . . . 48
4.3 A Ferramenta . . . 48
4.3.1 Editor Gráfico de Estadogramas . . . 48
4.3.2 Simulador de Estadogramas . . . 58
4.4 Arquitetura do Sistema . . . 58
4.5 Estruturas de Dados Internas . . . 58
4.6 Detalhes de Implementação . . . 65
4.6.1 Alterações realizadas no módulo de Edição . . . 65
4.6.2 Alterações realizadas no módulo da Simulação . . . 66
4.7 Como foi feita a comunicação por *Sockets* . . . 67
4.7.1 O Modelo Cliente/Servidor . . . 67

5 Um Exemplo de uma Aplicação de nosso Editor **69**
5.1 Objetivos . . . 69
5.2 O Editor Topológico para a LegoShell . . . 69
5.3 Descrevendo o editor através de estadogramas . . . 72
5.4 Estadogramas Principais do Editor da LegoShell . . . 74

6 Conclusões **80**
6.1 Contribuições . . . 80
6.2 Trabalhos Futuros . . . 81

Bibliografia **82**

A Interpretações diferentes para Histories **86**

B Sintaxe de LEG 95

C Estruturas de Dados de descrição de estadogramas 99

Lista de Figuras

1.1 Sistema de Mensagem 2
1.2 Uma Rede de Petri 3
1.3 Conjunto de Alcançabilidade da RP da Figura 1.2 4
1.4 Um Modelo de RP de um Multiprocessador 5
1.5 Sistema transformacional e Sistema Reativo 6
1.6 Contexto no qual se insere o Editor Gráfico 8

2.1 Editor Diagramático para DFD 19
2.2 Diagrama de Blocos 21
2.3 Símbolos para Construção de um Diagrama de Blocos 22
2.4 Conversão Diagrama de Blocos - Grafo de Fluxo de Sinal 23
2.5 Pré-redução do grafo 23
2.6 Áreas de Operação do Editor 24

3.1 Exemplo de um Estadograma 28
3.2 Diagrama de Estados 29
3.3 Estadograma equivalente à Figura 3.2 30
3.4 Comunicação em estadogramas 30
3.5 History Simples 31
3.6 History de Profundidade 32
3.7 History de Retorno 32
3.8 Condição ny 35
3.9 Condição ny 35
3.10 Definição de *status* 38
3.11 Transições Conflitantes 39
3.12 Estadograma de um Cronômetro 43

4.1 Arcos criados com as opções de iniciar e finalizar 51
4.2 Interface do Editor no momento da criação de uma bolha 52
4.3 Tela Principal do EGREST 53
4.4 Edição de Estadogramas 54
4.5 Edição de Bolhas 54
4.6 Criar Bolha 55
4.7 Edição de Arcos 55
4.8 Criar Arcos 56
4.9 Descrição Textual 57

4.10 Arquivos 57
4.11 Interface do Simulador no momento do disparo de um evento 59
4.12 Simulação por Disparo de Eventos 60
4.13 Manipulação de Arquivos 61
4.14 Informações 61
4.15 Janela de Eventos 62
4.16 Logfile 62
4.17 Descrição de Rótulos e Eventos 62
4.18 Simulação com Execução Paralela 63
4.19 Arquitetura do Sistema 64

5.1 Exemplo de Abstração de Computação 71
5.2 Tela Principal do Editor 74
5.3 Janela de Cabeçalho 75
5.4 Janela de Visualização 75
5.5 Área de Trabalho 76
5.6 Inclusão de Objetos 77
5.7 Edição 78
5.8 Seleção de Objetos 79

A.1 Inicialização por *default* 86
A.2 Ocorrência do evento a, entrada por *default* 87
A.3 Segunda ocorrência do evento a 88
A.4 Ocorrência do evento c 89
A.5 Ocorrência do evento a, entrada por *history* 90
A.6 Ocorrência do evento a 91
A.7 Ocorrência do evento d 92
A.8 Ocorrência do evento b.History associado a arco: entrada por default 93
A.9 Ocorrência do evento b. History associado à bolha: entrada por history 94

Agradecimentos

Ao Prof. Dr. Hans Kurt Liesenberg, pela orientação sempre clara e objetiva. Pela compreensão nos momentos difíceis.

Ao Prof. Dr. Paulo César Masiero pela gentileza de ter-nos cedido o código fonte do EGS.

Ao Antônio Figueiredo, pelas informações sempre úteis ao desenvolvimento deste trabalho.

Ao CNPq e a FAPESP que forneceram apoio financeiro para realização do Mestrado.

A meus pais e irmãs que souberam estar sempre presentes, apesar da distância. Obrigada pelo amor e carinho.

A Glêdson, pelo interesse, participação e sugestões.

A Caroline, por sua alegria.

A Cecil, Chris, Inés, Ivonne, Silvinha e Marquinhos pelo apoio, amizade e compreensão.

○ ○ ○

Resumo

Este trabalho apresenta um Editor Gráfico de Estadogramas. Estadogramas são uma extensão de diagramas de estados convencionais, acrescidos de conceitos de hierarquia, concorrência e comunicação. Apresentamos uma interface gráfica particular, que permite especificar sistemas reativos através da notação de estadogramas. No final de uma sessão de edição, fornecemos a opção de simulação de estadogramas. Nessa simulação são destacados os estados em que o sistema se encontra e são efetuadas as mudanças de estados que ocorrerem em decorrência de eventos. A cada estadograma editado, é gerada uma descrição textual passível de ser convertida em um programa em C que se comporta de forma funcionalmente equivalente ao estadograma. Paralelamente à execução deste programa, pode-se executar a simulação do estadograma original, onde através do mecanismo de *sockets*, o programa envia ao simulador os identificadores dos eventos ocorridos para serem realizadas as mudanças de estados decorrentes. Com este tipo de simulação fica disponível uma maneira de depurar programas a nível de estadogramas.

Abstract

A Graphic Editor of Statecharts is presented. Statecharts are an extension of conventional state diagrams where the concepts of hierarchy, concurrency and communication have been added. We present a particular graphic interface which supports the specification of reactive systems by means of the statechart notation. At the end of an editting session we offer a statechart simulation option. During a simulation the states, which the system is at, are highlighted and the state swappings due to event occurrences are performed. A textual description of an edited statechart may as well be generated. This description is liable to be converted into an executable program which behaves in a functionally equivalent manner. It is possible to execute this program at the same time the simulation of the original statechart is running. The program sends to the simulator the events identifiers, by means of the sockets mechanism, as soon as they occur. The simulator then applies the state swapping operation. With this kind of simulation it is possible to provide some support to debug programs at the statechart level.

Capítulo 1

Introdução

1.1 O Paradigma de Estados/Transições

Muitos métodos são utilizados para modelar sistemas através de estados e transições. Características comuns a estes métodos são:

- Eventos internos e externos que afetam o comportamento do sistema modelado;
- Uma ou mais variáveis de estados, que representam possíveis contextos em que o sistema pode-se encontrar;
- Funções de transição;
- Um estado inicial.

Podemos ver cada evento como um disparo possível de ativar uma função de transição que mapeia o estado corrente para um novo estado. Uma máquina de estados/transição típica, começa sua operação em um estado inicial [Sun 82]. Em tempos indeterminados, o estado da máquina é transformado por uma função de transição. A máquina poderá executar potencialmente por tempo infinito, ou pode terminar de operar se atingir um estado final.

Todos nós temos uma idéia intuitiva do que seja um evento. Grosseiramente, podemos dizer que um evento é um acontecimento atômico. Isto, de maneira alguma, implica dizer que o evento não tenha uma estrutura interna, ele pode até resultar de um processo complexo que poderá ser analisado em algum nível de abstração.

Modelos de estados/transição são freqüentemente descritos graficamente, com círculos ou quadrados representando estados e arcos representando transições. Cada arco é rotulado por um ou mais eventos que disparam a transição correspondente. A geração de eventos internos em função de uma transição também pode ser representada junto ao arco correspondente, bem como valores de variáveis que mantém informações relevantes para o sistema modelado. A Figura 1.1 representa um modelo de estados/transição para um sistema simples de troca de mensagens, permitindo o transporte de uma única mensagem do transmissor para o receptor.

Modelos de estados/transições têm sido usados para especificar protocolos de comunicação, incluindo Redes de Petri, linguagens formais, expressões seqüenciais e linguagens de programação.

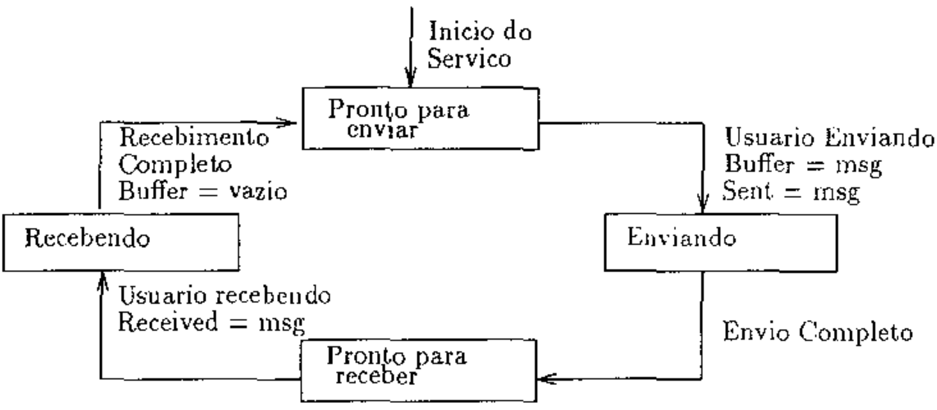

Figura 1.1: Sistema de Mensagem

Nas próximas seções, falaremos sobre como especificar um sistema através de uma Rede de Petri e como é o seu funcionamento e a seguir, descreveremos os sistemas reativos destacando suas diferenças com os sistemas transformacionais [Mar 90, Pnu 86, Har 85]

1.1.1 Um modelo de estados/transição: Redes de Petri

Modelos gráficos têm sido propostos como uma ferramenta útil para a análise de características peculiares de sistemas de computação, tal como, concorrência, comunicação e cooperação entre sub-sistemas [Mar 84]. Uma Rede de Petri é um modelo formal de uma computação. Sua maior aplicabilidade refere-se aos comportamentos constituídos por uma seqüência de eventos que podem acontecer concorrentemente e cujas ocorrências estão, normalmente, sujeitas a regras de precedência e/ou concorrência.

As Redes de Petri (RP) foram concebidas originariamente para se estudar a sincronização e a comunicação entre processos, ou seja, o fluxo de informações em um sistema.

Graficamente definimos uma RP como sendo um multigrafo bipartite e dirigido, isto é, um grafo cujo conjunto dos vértices pode ser particionado em dois subconjuntos distintos: o conjunto dos lugares e o conjunto das transições. Suas arestas unem exclusivamente elementos destes dois conjuntos, imprimindo um sentido de orientação a esta ligação. A união entre dois vértices pode ser feita por mais de uma aresta. Os lugares são representados por círculos e as transições por barras. Arcos conectam transições a lugares e lugares a transições. Lugares podem conter fichas[1], que são representados como pontos negros. O estado de uma RP é definido pelo número de fichas contidos em cada lugar e é denotado por um vetor, onde o i-ésimo componente representa o número de fichas na i-ésima posição. O estado de uma RP é chamada marca da RP.

O conceito de marcação, associado à definição de RP, consiste numa distribuição de fichas pelos diversos lugares da rede [Men 89]. Assim sendo, define-se uma RP como uma quádrupla RP = (P,T,A,MO), onde P,T,A são respectivamente os conjuntos de lugares, das transições e das arestas e onde MO representa a marcação inicial.

[1] tokens, em inglês

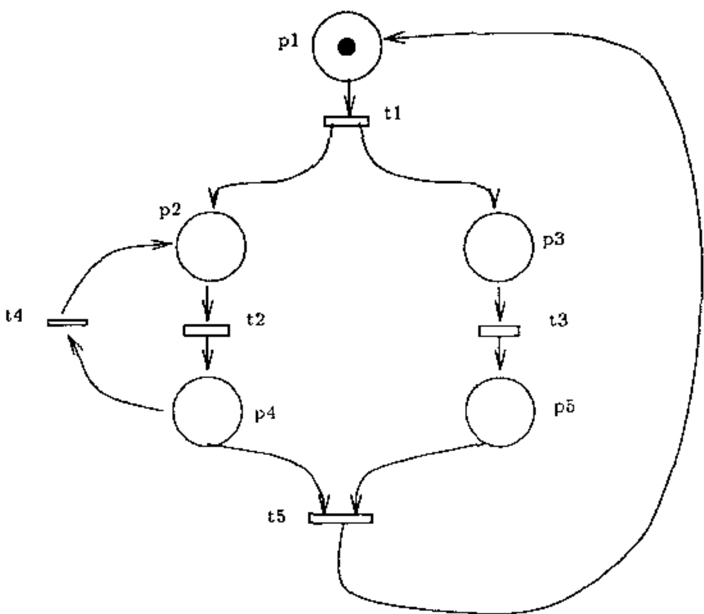

Figura 1.2: Uma Rede de Petri

A rede representada na Figura 1.2 exemplifica um processo p1 que após uma execução seqüencial t1, divide-se em dois outros processos que executam concorrentemente (p2 e p3) e que ao final da computação paralela (t2 e t3) esperam o término de seu "irmão" para reiniciarem o processamento seqüencial (t5). Um dos filhos pode experimentar uma outra contenção (t4) além da espera pelo seu "irmão".

O funcionamento de uma RP está determinado pela movimentação das fichas pelos lugares da rede. Para que se possa ter uma mudança na distribuição das mesmas, deve ocorrer um disparo de uma transição. Por outro lado, o disparo de uma transição só pode ocorrer, quando a mesma está habilitada, ou seja, quando o número de fichas em cada lugar de entrada é igual ou superior ao número de arcos que vão deste lugares para a transição em questão. Como exemplo de habilitação, verificamos que na Figura 1.2 apenas a transição t1 está habilitada a disparar, pois o número de fichas em p1, seu único lugar de entrada, é igual ao número de arcos que vão de p1 a t1.

Na ocorrência de uma transição, são retiradas tantas fichas dos lugares quantos são os arcos que vão destes lugares para a transição e são colocadas em cada lugar de saída tantas fichas quantos são os lugares de saída da transição.

O funcionamento da rede prossegue indefinidadmente; porém, para que se possa avaliar as propriedades de um sistema modelado por RP's, torna-se necessário enumerar o conjunto mínimo de marcações através do qual o "funcionamento" da rede possa ser caracterizado. Este conjunto é denominado conjunto de alcançabilidade [Men 89].

Faz-se necessário estabelecer algumas definições básicas a fim de enunciarmos o processo de geração de um conjunto de alcançabilidade. Uma marcação é dita morta quando não habilita nenhuma transição. Um lugar é k-limitado se o número máximo de fichas observável no mesmo, ao longo do funcionamento é igual a k. Uma rede que somente possua lugares k-limitados é de-

nominada k-limitada. Em contraposição ao lugar k-limitado, encontra-se o lugar ilimitado, que é caracterizado por possuir uma transição entrada com habilitação permanente, que acarreta um aumento incessante do número de fichas do lugar.

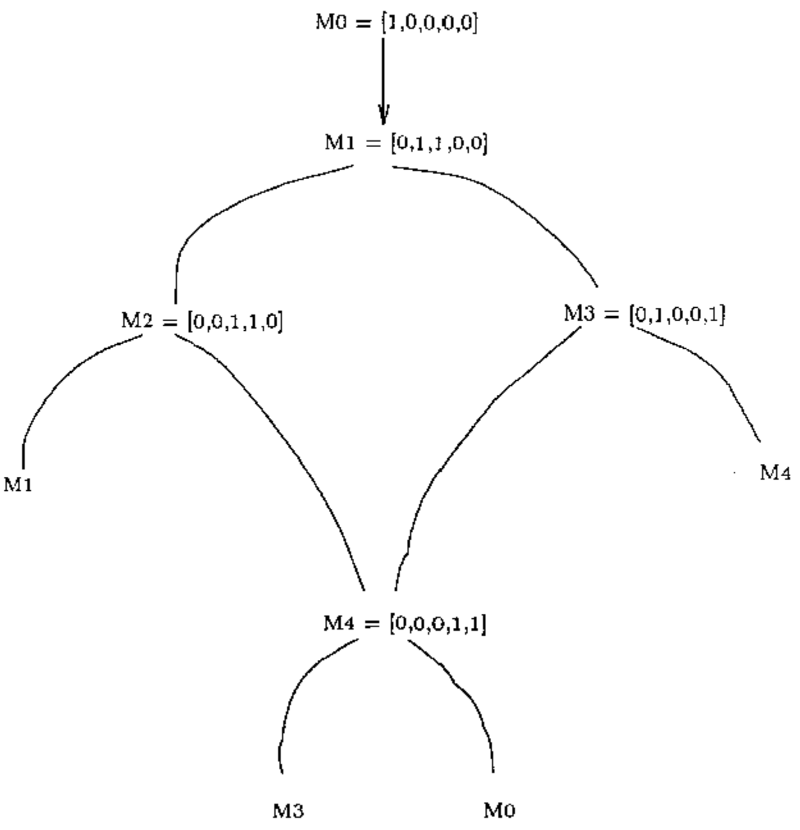

Figura 1.3: Conjunto de Alcançabilidade da RP da Figura 1.2

A enumeração do conjunto de alcançabilidade começa pela marcação inicial e, a partir desta, gera-se as próximas marcações, explorando-se a possibidade de disparo de cada transição habilitada. A cada marcação gerada, repete-se o processo de geração de novas marcações, desde que a marcação corrente não seja uma marcação morta ou duplicata.

Ao se encontrar uma marcação caracterizada pelo aumento do número de fichas em um determinado lugar e também pelo aumento do número total de fichas da rede, passa-se a caracterizar o lugar onde houve o aumento por um símbolo especial, que indica que o mesmo admite um número infinito de fichas. A Figura 1.3 ilustra o conjunto de alcançabilidade da Figura 1.2.

Pode-se atribuir diversos significados aos elementos de uma RP como, por exemplo, lugares podem indicar condições e transições a ocorrência de eventos. Dentro do contexto de Sistemas de Computação, esta interpretação poderia ser uma tarefa em execução (condição) e o fim da sua execução (evento).

A simplicidade do mecanismo de funcionamento de uma RP aliada à flexibilidade de interpretação da sua estrutura fazem desta uma poderosa ferramenta para a modelagem de sistemas.

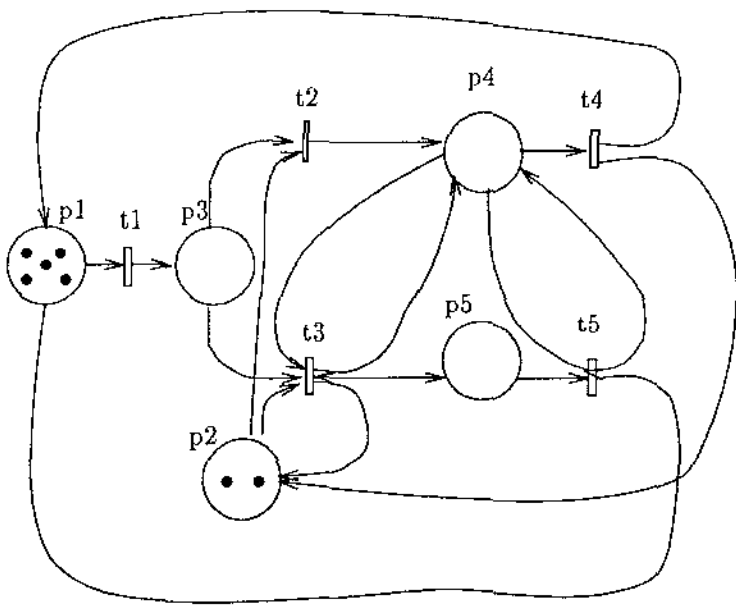

Figura 1.4: Um Modelo de RP de um Multiprocessador

O exemplo dado [Men 89], mostrado na Figura 1.4, refere-se a um sistema multiprocessador com cinco processadores, três memórias globais e dois barramentos. Neste sistema, os processadores permanecem ativos durante um certo período, acessando apenas a sua memória local. Terminado este período, o processador faz uma referência à uma memória global, porém antes disto deve estar de posse de um barramento. A Figura 1.4 ilustra a dinâmica de compartilhamento das memórias globais. O número de fichas nos lugares p1 e p2 indicam o número de processadores acessando a sua memória local e o número de barramentos disponíveis respectivamente. Após o término da execução local (t1), o processador tenta efetuar um acesso ao módulo de memória global desejado. Caso a memória escolhida esteja ocupada (t3), o processador espera pelo referido módulo (p5). Caso o módulo desejado esteja livre (t2), o processador efetua o acesso à memória global. As transições t5 e t4 representam o término do acesso a módulos de memória onde existem e onde não existem respectivamente requisições pendentes.

1.2 Sistemas Reativos

Existem basicamente duas visões diferentes de sistemas de computação. A primeira delas considera programas como funções que convertem um estado inicial em um estado final. Denomina-se este tipo de programas de transformacionais [Mar 90, Pnu 86, Har 85]. Exemplos típicos destes sistemas são: *batch*, processamento de dados *off-line* e outros. Para programas transformacionais, descrições abstratas adequadas e completas bem como ferramentas de especificação são providos por uma semântica denotacional baseada em funções de estados e transformadores de predicados (Dijkstra) [Pnu 86].

Por outro lado, existem sistemas que não se encaixam na visão transformacional. Alguns siste-

mas tais como sistemas operacionais, programas de controle de processos, sistemas de reserva aérea, idealmente nunca terminam. Além disso, o propósito da execução dos mesmos não é obter um resultado final, mas manter alguma interação com seu ambiente. Refere-se a este tipo de sistemas como Sistemas Reativos. Esta classe de sistemas está sendo considerada particularmente problemática quando tenta-se achar métodos satisfatórios para a sua descrição comportamental. Sistemas reativos geralmente não podem ser descritos adequadamente referindo-se apenas aos seus estados iniciais e finais. Uma descrição adequada deve fazer referência continuamente ao seu comportamento, ao longo de uma escala de tempo possivelmente infinita.

Numa outra tentativa de caracterizar esta classe de sistemas mais difíceis, que requerem descrição comportamental, identifica-se usualmente alguns termos tais como: tempo real, encapsulamento, concorrência, distribuição, etc. Nós vemos que nenhuma destas características, no entanto, captura precisamente a essência da reatividade: a interação natural entre o sistema e o seu ambiente. A sua interação não está restrita a aceitar entradas na inicialização e produzir saídas ao seu término. Em particular, ele permite que algumas entradas dependam de saídas intermediárias. Concorrência ou distribuição, por outro lado, se referem a organização interna de um sistema. Um sistema reativo pode, igualmente, ser bem implementado por uma arquitetura seqüencial assim como uma arquitetura concorrente e distribuída.

Tentaremos ilustrar a distinção entre sistemas reativos e transformacionais. Um sistema transformacional recebe dados de entrada, processa-os e então produz dados de saída (ver Figura 1.5).

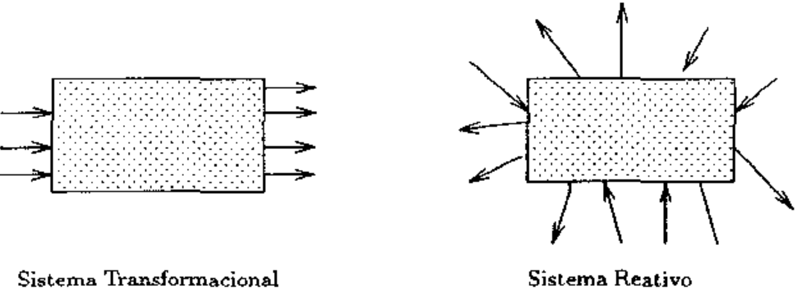

Figura 1.5: Sistema transformacional e Sistema Reativo

Também incluimos na definição de sistema transformacional, aqueles sistemas que podem pedir entradas adicionais, e que libera saídas ao longo de sua execução. Os sistemas reativos, por outro lado, estão sempre respondendo a entradas externas.

Um Sistema Reativo, em geral, não computa ou realiza uma função, mas admite-se que o mesmo mantém uma certa relação contínua com o seu ambiente. Em comum a todos estes sistemas, está a noção de responder ou reagir a estímulos externos, sejam estes gerados de forma previsível pelo usuário (como o pressionamento de uma tecla), ou pelo ambiente (como a elevação da temperatura detectada por um sensor), ou de forma anormal (tal como queda de força).

1.3 Estadogramas

O método de estadogramas[2] [Har 87a, Har 87b, Har 87c] foi introduzido recentemente como um formalismo bidimensional para especificar o comportamento de sistemas reativos complexos. O processo de representar um sistema através de estadogramas é denominado estadificação e consiste em descrever hierarquicamente o comportamento do sistema em termos de estados, eventos e transições sujeitas eventualmente a condições restritivas com relação ao seu disparo. Ambos, estados e condições, podem ser associados a atividades, que podem ser executadas por ocasião de um disparo de uma transição ou por ocasião de entrada, saída ou permanência em um dado estado. No caso de transições, estamos falando de ações de duração muito pequena. Para estados, temos a ação que é disparada na entrada do estado, uma atividade processada enquanto o sistema permanecer neste estado, e a ação executada na saída do estado. Falaremos com mais detalhes de ações e atividades na Seção 3.1.4.

A notação de estadogramas foi proposta inicialmente por David Harel [Har 87a, Har 87b, Har 87c] para descrever a estrutura de controle de sistemas reativos. Esta descrição consiste em contínuas mudanças de estados ocasionados pela ocorrência de eventos, sejam eles internos ou externos. A notação de estadogramas foi derivada a partir de diagramas de estados/transições convencionais acrescidos de conceitos de hierarquia, concorrência e comunicação. Algumas características dos estadogramas são:

- Encapsulamento de estados, permitindo características de refinamento e abstração.
- Transições interníveis, ou seja, as transições entre os estados não estão restritas a um mesmo nível de hierarquia.
- Ortogonalidade de estados, ou seja, o sistema em um determinado instante pode estar em mais de um estado, em um mesmo nível, ao mesmo tempo.
- Mecanismo de *history*, fazendo com que os estadogramas "memorizem" estados recentemente visitados, para um possível retorno a estes estados.

Nosso objetivo em apresentar as Redes de Petri foi apenas o de mostrar um tipo de modelo de estados/transições, alternativo à estadogramas, mas elas apresentam características que não necessitamos no momento e não apresentam algumas capacidades dos estadogramas desejáveis ao nosso trabalho, tais como: hierarquia de estados, mecanismos de comunicação e capacidade de *history*.

1.4 Motivação para o Trabalho de Tese

Existe uma dificuldade generalizada no projeto e desenvolvimento de sistemas reativos devido ao comportamento dinâmico dos mesmos, característica esta que torna sua descrição formal mais complexa. Como vimos anteriormente, a notação de estadogramas foi criada com o objetivo de descrever abstratamente este tipo de sistemas.

[2] denominados *statecharts* por Harel

Nossa proposta é fornecer ao projetista de sistemas reativos, um "Ambiente de Edição e Simulação de Estadogramas", a fim de apoiar o processo de criação e especificação de estadogramas oferecendo ainda a oportunidade de simulá-los após sua completa descrição para validá-lo.

Nosso trabalho consiste em uma adaptação do Editor Gráfico de Statecharts, desenvolvido na USP/São Carlos [Bat 91]. Apresentamos uma versão estendida e o inserimos no contexto descrito na figura 1.6.

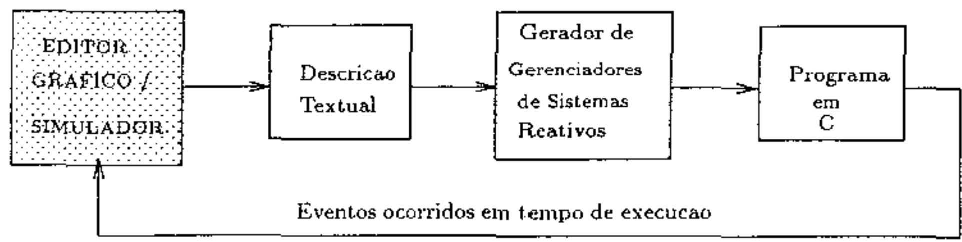

Figura 1.6: Contexto no qual se insere o Editor Gráfico

Ou seja, o Editor Gráfico de estadogramas gera uma descrição textual dos estadogramas, ao final de cada sessão de edição. Esta descrição textual é a entrada do Gerador de Gerenciadores de Sistemas Reativos (GGSR) [Fig 91]. O GGSR lê esta descrição textual e a traduz em um programa em C funcionalmente equivalente ao estadograma editado.

No final o programa em C, gerado pelo GGSR, ainda poderá se comunicar com o simulador de estadogramas através do mecanismo de *sockets* (ver seção 4.7). Desta forma, fornecemos ao usuário um mecanismo de depuração de seu sistema através de uma animação do estadograma concorrente com a execução do programa em C.

1.5 Organização dos capítulos

Fizemos um levantamento bibliograáfico sobre alguns editores gráficos descritos na literatura técnica e no Capítulo 2 escolhemos alguns deles que apresentavam algumas características mais semelhantes ou distintas ao nosso.

No Capítulo 3 apresentamos com maiores detalhes em que consiste a notação de estadogramas. Ao final relatamos alguns trabalhos existentes que utilizam a notação de estadogramas para descrever sistemas reativos.

No Capítulo 4 especificamos a ferramenta, objeto deste trabalho, demonstrando como são manipuladas internamente as informações, e apresentamos alguns detalhes de implementação especificando cada módulo do sistema.

No Capítulo 5 mostramos um exemplo de utilização de nossa ferramenta.

Ao final, no Capítulo 6, apresentamos uma conclusão do nosso trabalho, suas contribuições e algumas sugestões para trabalhos futuros.

Capítulo 2

Editores Gráficos

Neste capítulo apresentamos descrições de alguns editores gráficos com características bem particulares, que foram pesquisados com o objetivo de auxiliar o processo de especificação e desenvolvimento da ferramenta objeto deste trabalho.

2.1 TEDMOS - Um Sistema de CAD para ensino de projeto de circuitos microeletrônicos de alta integração

O projeto de um circuito integrado de baixa complexidade, por exemplo, transistores e portas lógicas, pode perfeitamente ser realizado de forma manual. As etapas de um projeto deste porte são basicamente:

- Definição do circuito;
- Esboço do traçado;
- Cálculo dos parâmetros elétricos;
- Cálculo do desempenho do circuito;

A dificuldade do projeto manual aumenta de acordo com o aumento da complexidade do sistema.

2.1.1 Como especificar um projeto de grande complexidade

O problema principal do projeto de um circuito integrado de alta densidade é a gerência de sua complexidade. Da mesma maneira que num projeto de software, o projeto de um circuito de grande porte deve usar uma metodologia estruturada. O projeto vai sendo desenvolvido com refinamentos sucessivos, a partir de especificações gerais que vão sendo progressivamente detalhadas [Bor 87].

Em geral, um projeto segue os seguintes passos:

1. Especificação Funcional:

   Neste passo, se define o circuito integrado em termos de suas características de entrada e saída: os pinos e sua função, a potência dissipada, as relações lógicas e temporais dos sinais.

2. Detalhamento Lógico:

   Quando a definição funcional está pronta, o próximo passo consiste em fazer o detalhamento dos circuitos internos em termos de portas lógicas.

3. Especificação da rede de transistores:

   O nível de refinamento seguinte é a criação de uma rede de transístores equivalente à rede lógica.

4. Traçado:

   Neste passo, cada um dos transistores é dimensionado, alocado e interconectado com o resto do circuito. Esta é a fase mais demorada do projeto.

5. Verificações Geométricas:

   Quando o layout está concluído é submetido a uma verificação das dimensões e espaçamento dos elementos.

Algumas das etapas podem ser executadas em paralelo, e dependendo do resultado de alguma etapa algum passo anterior pode ter que ser reexecutado.

2.1.2 Visão geral do TEDMOS

O TEDMOS é um sistema que foi criado especialmente para o ensino de projeto de circuitos integrados e utilização em multiprojetos de universidades.

Através do TEDMOS é possível fazer o desenvolvimento de células razoavelmente complexas, sua verificação geométrica e simulação lógica, num único pacote integrado.

O TEDMOS é composto das seguintes partes principais:

- Editor Gráfico,
- Verificador de regras geométricas,
- Extrator de circuitos,
- Simulador a nível de chaves.

O sistema provê funções extras para a manipulação de arquivos, edição de texto integrado, geração automática de PLAs e impressão gráfica. As próximas versões também incorporarão um editor hierárquico para montagem das células para formar um chip completo e uso opcional de *mouse* para a entrada de dados gráficos e seleção de funções.

2.1.3 O Editor Gráfico

Introdução

O Editor Gráfico é o núcleo do TEDMOS. Através dele, o projetista pode criar e modificar traçados de células de circuitos integrados na tela do computador.

A tela do computador representa um trecho deste traçado. O projetista move um cursor retangular sobre a tela e, nos pontos desejados, manda o editor preencher este retângulo com um padrão hachurado ou colorido, que representa uma das máscaras de fabricação.

Funções do Editor

- pintura e apagamento de retângulos
- movimentação, ajuste e cópia de trechos
- movimentação da janela de edição
- visibilidade parcial de máscaras
- guardar e trazer do disco, traçados já criados.

Estrutura de Dados Interna

A representação da geometria das máscaras em editores gráficos para circuitos integrados é normalmente feita de uma das seguintes maneiras: lista de polígonos, lista de retângulos ou mapa de bits. Dentre estes foi escolhido o mapa de bits devido a extrema simplicidade dos algoritmos. A quantidade de memória básica do computador é suficiente para a edição de um circuito relativamente grande.

O desenho é definido sobre uma matriz. Cada elemento da matriz é composto de um conjunto de bits. Cada bit indica a presença ou ausência de uma área opaca da máscara correspondente naquele ponto.

Na tela é feita a representação visual dessa matriz. Cada elemento da matriz é mapeado num conjunto de pontos na tela, com aparência aproximadamente quadrada (pixel). O mapeamento entre cada elemento possível e sua representação na tela é fornecido em um arquivo de configuração.

2.1.4 O Verificador de Regras de Projeto

A verificação geométrica torna-se cada vez mais importante com o crescimento da complexidade dos circuitos. Esta só pode ser feita de forma confiável por um programa de computador que faça a verificação sistemática de todas as formas geométricas do circuito.

O Extrator de Circuitos

O Extrator de Circuitos converte a descrição geométrica do circuito (mapa de bits) na rede de transistores equivalentes. Basicamente, o problema consiste no seguinte:

- localizar os transistores
- descobrir os fios de ligação entre os transistores.

O extrator recebe como entrada o mapa de bits do editor e dá como saída, um arquivo que irá alimentar o simulador.

2.1.5 O Simulador

Uma vez criado o traçado e verificadas as suas regras geométricas, não se pode simplesmente enviar o projeto criado para fabricação. É necessário saber se o traçado implementa a função lógica desejada. Como o projeto foi criado manualmente, a probabilidade de que haja erros de desenho ou de lógica é muito alta.

Os transistores podem ser modelados com diferentes graus de refinamento. Quanto mais refinado o modelo dos transistores mais precisos serão os resultados, porém maior será o esforço gasto na simulação.

Interface com Usuário

Um dos elementos principais na atividade de simulação é a interação do usuário com o simulador e a apresentação visual dos dados simulados.

Foi definida uma série de comandos através dos quais o usuário pode atribuir seqüências de valores de entrada a cada ponto do circuito, especificar os pontos de monitoração e comandar o andamento do processo de simulação.

Foi também criada uma interface visual na qual é apresentada na tela um gráfico emulando um osciloscópio de 10 canais. Através deste gráfico, a visualização do comportamento simultâneo de pontos do circuito é muito facilitada.

2.2 Ferramenta CAD para projetos de sistemas de irrigação - IRRIGRAPHOS

A ferramenta IRRIGRAPHOS é um editor gráfico, que suporta comandos de desenho, edição, visualização e averiguação, além de permitir a criação, alteração e manipulação de uma biblioteca de símbolos mais comumente utilizados para representar ou legendar os elementos hidráulicos de um projeto de irrigação. Essa biblioteca que é modularizada por sistema de irrigação, ou seja, gotejamento, micro-aspersão, aspersão, sulco etc, e pode ser ampliada ou alterada pelo projetista [ArA 91].

A metodologia utilizada na implementação do IRRIGRAPHOS visa observar as características básicas de um bom aplicativo na área de computação gráfica. As rotinas incluídas neste sistema estão agrupadas nos seguintes módulos:

- Diálogo:

  Módulo que se encarrega da comunicação entre o operador e a CPU.

- Criação da Geometria:

  Módulo responsável pela construção das entidades geométricas.

- Aplicação:

  Os módulos de aplicação são aqueles que a partir de uma geometria previamente construída, operam rotinas específicas de cálculo, simulação e análise.

- Estruturador de Dados:

  Este módulo trata dos mecanismos, parâmetros, ponteiros, estruturas, enfim todas as informações que o sistema necessita.

- Controle Geral:

  Módulo composto pelas rotinas de supervisão que garantem a integração de cada um dos outros módulos.

Através de comandos interativos gráficos de mesa digitalizadora ou *mouse*, o projetista define, estuda, simula e altera o traçado do projeto de irrigação, que poderá ser posteriormente impresso e analisado em impressora matricial gráfica, impressora a laser e em traçador gráfico.

Um recurso importante do IRRIGRAPHOS é a sua comunicação ou interface com o software PSI - Projeto de Sistemas de Irrigação, ferramenta CAE desenvolvida pela TECNAL com finaciamento da FINEP, a qual permite a automação de todas as fases de cálculo de um projeto de irrigação. Esta interface com o PSI permite, após o traçado concluído no IRRIGRAPHOS, a geração de arquivos de entrada de dados do dimensionamento do projeto de engenharia do sistema de irrigação, quando serão transportadas todas as informações de número de unidades e sub-unidades de rega na irrigação localizada, número de sub-módulos na irrigação por aspersão, número de baterias na irrigação por sulco, comprimento e desníveis topográficos em todas as tubulações, etc. Esta mesma interface também permitirá a transferência de informações no sentido contrário, ou seja, do PSI para o IRRIGRAPHOS, possibilitando que o desenho do projeto seja concluído com os dados do dimensionamento de forma totalmente automatizada.

Recursos de Software

- MENU DESENHO:

  Com as opções de ANEL, ARCO, CÍRCULO, HACHURAS, LINHA, PLACA, POLÍGONO, POLILINHA, PONTO e RISCO.

- MENU DE EDIÇÃO:

  Com as opções de ALONGA, ARREDONDA, CÓPIA, CORTA, DESFAZ, DIVIDE, ESCALA, ESPELHA , MUDA, MOVE, QUEBRA, RETIRA e ROTAÇÃO.

- MENU DE VISUALIZAÇÃO:

  Com as opções de APROXIMA, DESLOCA, LIMPA e REGENERA.

- MENU FERRAMENTAS:

  Com as opções de COORDENADAS, CORES, GRADE, ORTOGONAL e PRECISÃO.

- MENU TEXTO:

  Com as opções de ESTILO, TAMANHO, TEXTO e TIPO.

- MENU COMUNICAÇÃO:

  Com as opções de ABANDONA, ASCII, DATA, DOS, HORA, FINALIZA, IMPRESSORA, MESA e PLOTTER.

- MENU IRRIGAÇÃO:

  Com as opções de ASPERSÃO, GOTEJAMENTO, MICROASPERSÃO, PROPRIEDADE e SULCO.

- MENU OBRAS HIDRÁULICAS:

  Com as opções de BARRAGENS, CANAIS, DRENAGEM, POÇOS e SISTEMATIZAÇÃO.

2.3 LACSUZ - Um Sistema Editor de Microcircuitos

O Objetivo deste sistema é o auxílio ao projeto de microcircuitos através da utilização de um hardware de baixo custo (microcomputador pessoal). Seu uso será feito por estudantes de cursos de graduação, em disciplinas de introdução ao projeto de circuitos integrados [Can 85].

O sistema aqui descrito parte do princípio que um sistema de CAD para microeletrônica não é necessariamente oneroso. Procurou-se desenvolvê-lo num microcomputador de 8 bits (Apple-2) com o intuito de dispor de ferramentas suficientemente acessíveis para o ensino de microeletrônica em cursos de graduação.

Foi feita uma divisão em 3 sub-sistemas:

- subsistema de edição de arquivos
- subsistema de manipulação de arquivos
- subsistema de controle dos periféricos

As seguintes funções estão disponíveis ao usuário:

- fazer objetos (desenhos) com o uso do computador
- armazenar os objetos em disco
- editar (alterar) objetos armazenados
- obter cópias em papel de objetos existentes
- fazer programas que acessem os arquivos para gerar novos formatos.

O usuário final será sempre um profissional de microeletrônica.

2.3.1 Como se faz a indicação de pontos

Os dois tipos possíveis de objetos de trabalho são os retângulos e as retas, ambos definidos por dois pontos.

O modo como foi definido a interação com o usuário baseia-se em dois elementos, correspondentes aos dois pontos de trabalho: a "ampulheta" e a "borboleta". A ampulheta fica posicionada no ponto ao qual o usuário está se referindo e a borboleta fica posicionada no outro ponto de trabalho.

Pode-se re-editar as posições de borboleta e ampulheta até que se esteja satisfeito com as suas posições, que irão definir a reta ou o retângulo. A interação com o usuário pode ser feita através do "paddle" ou através do teclado.

2.3.2 Facilidades de Visualização

A janela de seleção refere-se a uma parte do universo com o qual se deseja operar, pois nem sempre se deseja operar com todo o circuito. A janela de exibição especifica que será usada somente parte da tela. Eventualmente, pode ser importante operar simultaneamente sobre mais de um detalhe, ou com vista de todo o circuito e dois detalhes, etc. Para satisfazer a este tipo de exigência foram implantados quatro conjuntos seleção-exibição de janelas. É possível especificá-los, bem como torná-los ativos ou inativos. Com esta estrutura fica facilitada a localização em circuitos muito grandes, bem como a especificação de linhas longas, como as de alimentação, por exemplo.

2.3.3 Facilidades de Armazenamento

Paralelamente ao editor gráfico, existe um sistema de arquivos que permite armazenar os circuitos construídos. Este é constituído por um diretório, sobre o qual é possível efetuar operações sobre arquivos.

Os arquivos são usados para armazenar em disco todas as informações gráficas necessárias: fazer um novo desenho é criar um novo arquivo; alterá-lo é alterar este mesmo arquivo.

2.3.4 Desenho estruturado

A segunda parte do sistema editor de microcircuitos diz respeito a "desenho estruturado". Aqui os elementos de trabalho não são retas ou retângulos, mas circuitos. É possível, por exemplo, projetar uma determinada céclula CEL0, como um conjunto de retas e retângulos e posteriormente criar a célula CEL1, constituída por três células CEL0, dispostas lado a lado. A seguir, pode-se criar a célula CEL2, constituída por duas células CEL1 e uma célula CEL0 girada de 90 graus, por exemplo, e assim sucessivamente.

A filosofia hierárquica adotada evita o dispêndio de esforços repetitivos. Neste contexto, é permitido, no sistema de arquivos, listar os pais de uma célula, os ancestrais, e os filhos, com a finalidade de localização dentro da estrutura hierárquica.

2.4 Editores Gráficos para Projeto de Circuitos Integrados

O projeto de circuitos integrados está intimamente ligado ao uso de um editor gráfico onde se possa fazer operações de criação e montagem das partes componentes. Iremos mostrar algumas características de dois editores gráficos para projeto VLSI que foram produzidos no NCE-UFRJ [Bor 85].

O primeiro editor é o EDMOS, que já vem sendo utilizado desde 1983 e utiliza um microcomputador nacional (SDE-42) com vídeo gráfico e disquete como estação de trabalho. O segundo editor é o EDCI, um editor hierárquico que visa acompanhar o projeto desde a alocação inicial de áreas até a montagem completa.

2.4.1 Visão Geral do EDMOS

O EDMOS é um Editor Gráfico de células. O EDMOS utiliza a memória local do vídeo gráfico para armazenamento do traçado (96kbytes). Não existem estruturas de dados intermediárias, ou seja,

o layout é feito sobre a própria memória da tela, acendendo ou apagando os pontos convenientes como se estivéssemos colorindo um papel. Quando o desenho está completo a memória de tela é varrida, extraindo-se o desenho. Uma nova versão do EDMOS incorpora para armazenamento das informações uma estrutura de dados armazenada através de um processo semelhante a memória virtual e que pode ser transportada facilmente para qualquer equipamento.

Entre as principais facilidades do EDMOS, temos a criação e retirada de retângulos coloridos, ajuste de trechos, cópia, zoom, replicação de trechos, espelhamento, rotação etc., que provêem a maioria das facilidades necessárias para a edição de células. Entretanto, o EDMOS não é capaz de editar traçados maiores que o tamanho da tela.

Alguns Comandos do EDMOS

- O cursor e sua movimentação:

  O cursor pode ser movido, expandido e encolhido. O cursor é alterado usando-se exclusivamente o teclado.

- Modos de operação:

  Existem dois modos de operação: rápido e lento.

- Comandos de movimentação de áreas:

  Existe uma série de comandos cuja finalidade é corrigir o traçado através da cópia ou movimentação de trechos.

- Visibilidade:

  Uma das operações mais importantes é a visibilidade seletiva, ou seja, ver-se somente algumas camadas do circuito integrado.

- Pegar e guardar um layout no disquete:

  Os layouts são gravados em CIF (Caltech Intermediate Format) e podem ser trazidos para a posição do cursor, rodados ou espelhados.

- DRC online:

  O EDMOS está conectado diretamente ao verificador de regras de projeto (DRC - Design Rules Checker) o que permite a verificação imediata dos traçados durante o processo de edição. Neste caso são mostrados piscando na tela os trechos os quais apresentam algum tipo de violação geométrica.

O EDMOS prevê saídas para impressoras alfanumérica e gráfica. A vantagem em utilizar impressora gráfica é grande. A rapidez e a relativa qualidade do desenho obtido permitem que se evite tirar saídas em traçadores gráficos, que apresenta uma certa dificuldade de visualização e requer um tempo enorme de desenho.

O EDMOS trabalha preenchendo áreas da tela com cores e consultando o conteúdo da tela. Isso, na tecnologia utilizada é extremamente simples, pois basta consultar diretamente a memória da tela.

A extração do desenho é feita varrendo-se a área delimitada por cursor, linha a linha, de baixo para cima, localizando-se as sequências de pontos com a mesma cor. É mantida uma lista com as sequências da linha anterior.

2.4.2 Visão Geral do EDCI

O EDCI foi projetado para poder ser executado tanto no computador principal, utilizando a estação de trabalho apenas como unidade de saída, ou executar mesmo na própria estação de trabalho, de maneira *stand-alone*. Essa independência é obtida pela utilização de rotinas gráficas modulares e pelo fato do editor ser escrito em C padrão.

Ele utiliza uma tela alfanumérica e uma tela gráfica. Na tela alfanumérica se mantém todo o diálogo com o operador, e são mostrados todos as variáveis de status da edição. Na tela gráfica, aparece o traçado que se está projetando, numa escala arbitrária. Essa separação de telas possibilitou o uso de terminais gráficos com resolução bastante modesta.

No EDCI, um chip é tratado como um caixote visto do alto (caixa preta). Dentro desta caixa, podem existir três tipo de coisas: retângulos coloridos, textos ou outros caixotes. Dentro dos caixotes internos podem existir outros retângulos, textos e caixotes, e assim sucessivamente.

Foi minimizada a quantidade de coisas que o editor desenha automaticamente. Como agora estamos tratando de traçados completos, o tempo de desenho pode ser muito grande. É melhor, então, que seja opção do operador, por exemplo, mostrar um símbolo apenas através de seu contorno e de seus pontos de conexões, do que todo o interior expandido. Além disso é permitido o uso de escala menor que 1, ou seja, pode-se ter uma visão macroscópica do circuito integrado, e nesse caso, o número dos elementos desenhados na tela é enorme, caso se deseje ver o conteúdo de todos os seus subsímbolos.

Operação do EDCI

Inicialmente é mostrada na tela o contorno de um retângulo, representando o circuito integrado, e dentro dele, outros retângulos vazios, representando o nível imediatamente inferior de células e os retângulos coloridos deste nível.

Existe um cursor retangular que pode ser movido semelhantemente ao EDMOS para se posicionar dentro do traçado. Eventualmente o operador vai querer entrar num subsímbolo, alterar seu conteúdo, entrar num subsímbolo ainda mais interno, sair dele, e assim por diante, navegando de forma razoavelmente confortável dentro do traçado do circuito integrado. É possível isolar um trecho do traçado ou um símbolo para edição em escala maior, alteração de tamanho dos símbolos ou seu posicionamento, inclusão, criação e retirada de elementos no circuito, etc.

Os comandos do EDCI implementados em sua primeira versão são os seguintes:

- movimenta cursor
- cria, remove, entra e sai de símbolo
- cria e remove contorno
- ajusta tamanhos e posições de símbolos
- expande chamadas de símbolos até um nível específico

- isola, amplia e reintegra símbolos
- cria matrizes de símbolos
- salva e recupera o conteúdo do cursor
- cria e remove hierarquias
- traz símbolo em CIF para o conjunto de arquivos ou dele extrai um arquivo em CIF
- mostra, altera, manuseia tabela de símbolos
- seleciona janela de edição, visibilidade seletiva, escala
- comando de ajuda "on-line" para todos os comandos
- possui interface para conexão com o DRC e o gerador de rotas automático.

Os algoritmos deste editor e suas principais estruturas de dados são baseados em uma árvore que define as relações de hierarquia e o conteúdo das células. No manuseio desta árvore, são utilizados algoritmos clássicos de busca, manuseio, inclusão e exclusão em árvores aplicadas a processamento gráfico.

2.5 Editor Diagramático para DFD, baseado em formalismos gramaticais

Apresentamos aqui a construção de um Editor de DFD(Diagramas de Fluxos de Dados), a partir de uma especificação formal do diagrama. Utiliza-se técnicas apoiadas em formalismos gramaticais, utilizadas na construção de editores dirigidos por sintaxe [Fav 89].

A especificação de diagramas segue a seguinte orientação: o nível léxico é representado por uma gramática regular, o nível sintático por uma gramática livre de contexto estendida com o conceito de compartilhamento de nodos, o nível semântico por uma gramática de atributos estendida e por tabelas relacionais.

2.5.1 Especificação da Sintaxe

O Editor representa, internamente, os diagramas em árvores sintáticas abstratas (ASAs). Uma árvore de reconhecimento ou derivação representa o caminho de derivação de uma sentença a partir de uma gramática livre do contexto. Uma ASA é uma árvore de reconhecimento abstraída de detalhes da estrutura sintática que não representam informações consideradas relevantes.

A partir da Figura 2.1, observa-se:

1. Uma página lógica de um DFD é composta por um identificador (nome da página), por um conjunto de entidades (quadrados), por um conjunto de processos (bolinhas), por um conjunto de depósitos (retângulos) e por um conjunto de ligações (arcos).

2. Cada entidade, depósito ou processo pode ser considerado como um item léxico, isto é, um objeto fechado para o nível sintático.

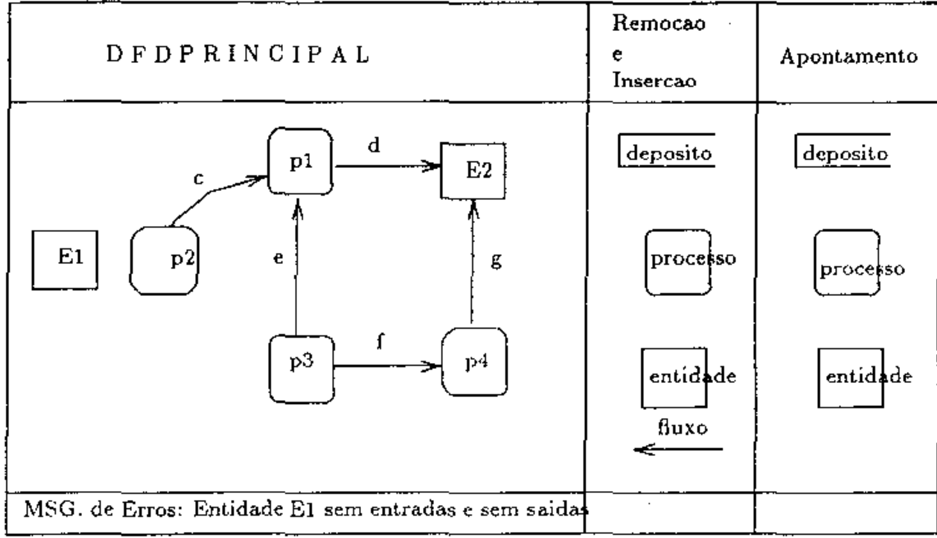

Figura 2.1: Editor Diagramático para DFD

3. Uma ligação ocorre entre dois elementos do tipo entidade, processo, ou depósito. Um elemento pode participar de várias ligações e em cada ligação deve existir pelo menos um processo.

Na notação BNF, não existe o conceito de conjunto, porém pode-se descrever um conjunto por uma lista de elementos e por uma regra semântica que verifica a unicidade dos elementos. Também sugere-se uma extensão da notação BNF, indicando que os nodos são compartilhados.

Um documento DFD compreende várias páginas hierarquizadas com forma de uma árvore, onde cada processo pode ser refinado (explodido) em outra página lógica. Esta estrutura de árvore é representada como:

```
Processo --> ProcessoGrafico SubProcessoOpcional
SubProcessoOpcional --> Pagina
Pagina --> NomeDePagina ListadeEntidades ListadeProcessos
            ListadeDepositos ListadeLigacoes
```

Os objetos gráficos da Figura 2.1 podem ser identificados como elementos léxicos: Entidades (Eg), Processo (Pg), Depósitos (Dg), Fluxos (Fg), e Nome da Página do diagrama (Ng). Cada um desses elementos é construído a partir de elementos geométricos primitivos, assim como, cada *token* léxico de uma linguagem de programação é construído a partir de uma seqüência de caracteres de um dado alfabeto.

2.5.2 Especificação da Semântica

A semântica compreende o aspecto da semântica estática e de geração de código intermediário. Para o nível semântico cada elemento é visto e tratado como uma produção de GA (Gramática de

F / F	FUNCAO DE TRANS-FERENCIA. ENTRADA PELA DIREITA OU ESQUERDA
+ - + -	SOMADORES COM SAIDA PARA A ESQUERDA
+ - - +	SOMADORES COM SAIDA PARA A DIREITA
	CONEXAO TIPO "T"
	CONEXAO TIPO "L"
	CONEXAO VERTICAL E HORIZONTAL

Figura 2.3: Símbolos para Construção de um Diagrama de Blocos

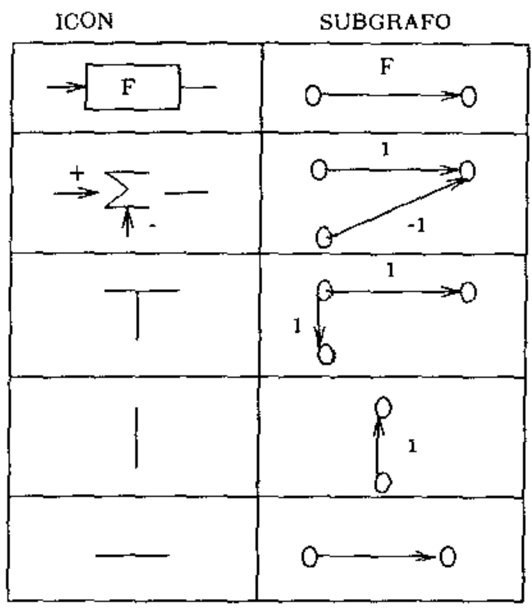

Figura 2.4: Conversão Diagrama de Blocos - Grafo de Fluxo de Sinal

Como pode ser observado na Figura 2.4, os grafos assim obtidos tendem a conter muitos arcos redundantes com transmitância unitária. Para resolver este problema é feito uma pré-redução do grafo, eliminando arcos com transmitância unitária, ou seja, para todos os arcos (I,J) com transmitância unitária da forma mostrada na Figura 2.5. Note que não pode haver um arco de I para K ou de M para I para que esta redução seja feita.

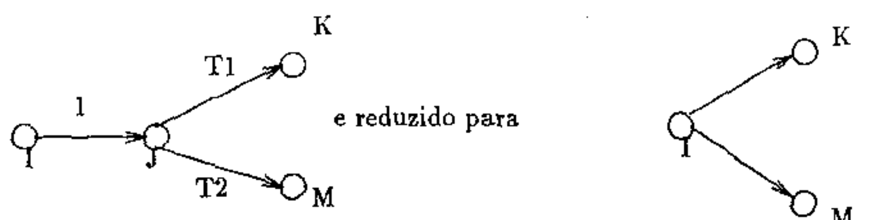

Figura 2.5: Pré-redução do grafo

2.6.2 Interface Homem-Máquina

Com um pequeno conjunto de comandos é possível criar, modificar, salvar e carregar um diagrama de blocos. A construção do diagrama inicia-se pelo posicionamento do cursor na tela, teclando-se "I", o usuário escolhe o tipo do item que deseja inserir, e RETURN para colocar o elemento no diagrama. Repete-se o procedimento acima até que o diagrama esteja completo.

O diagrama pode ser salvo em disco com o comando "S" e recuperado com o comando "C".

2.7 Estudo Comparativo dos Editores Gráficos

Apresentamos neste capítulo uma série de Editores Gráficos de aplicações bem variadas. As características de implementação e de utilização de cada um deles diferem bastante uns dos outros. Por causa disto, fizemos uma tabela que mostra um resumo das características principais de cada um deles, de maneira a tornar mais fácil uma comparação entre eles.

	TEDMOS	IRRIGRAFOS	LACSUZ
Área de Aplicação do Editor	Projeto de circuitos integrados	Auxílio no projeto de irrigação	Projeto de microcircuitos
Estruturas de dados	Mapa de bits para representar a geometria das máscaras.	Estrutura de dados diversas.	Sistema de arquivos.
Funções Principais	Pintura e apagamento de retângulos. Movim., ajuste e cópia de trechos. Movim. janela. Visibilidade parcial.	Desenho, edição, visualização, averiguação, criação, alteração e manipulação de símbolos.	Criar, armazenar, editar e obter cópia de objetos.
Interface com o Usuário	Tela gráfica para edição de retângulos e Tela gráfica emulando osciloscópio.	Diversos menus de edição.	Duas janelas gráficas com dois elementos auxiliares na tela de edição. Interação via "paddle" e teclado.
Tipos de desenhos	retângulos hachurados	elementos hidáulicos	retângulos e retas

	EDMOS/EDCI	ED-DFD	EDB
Área de Aplicação do Editor	Projeto de circuitos integrados.	Edição de diagramas de fluxos de dados.	Modelagem de sistemas em termos de diagramas de blocos.
Estruturas de Dados	EDMOS: Armaz. do traçado na mem. local do vídeo gráfico. EDCI:Árvore definindo relações de hierarquia e contendo células.	Árvores sintáticas abstratas.	Grafos de fluxos de sinal.
Funções Principais	EDMOS: Criação e retirada de retângulos. Ajuste de trechos. Cópia e "zoom". EDCI: Isola, amplia e reintegra símbolos.Cria e remove hierarquias	Edição e manipulação dos símbolos de DFD.	Inserir item. Salvar diagrama. "Hardcopy" em impressora e apagar elemento.
Interface com o usuário	EDMOS: Preenchim. de áreas da tela.EDCI: Tela alfanum. e graf. Janela de edição,visibilidade e escala.	Manipulação de símbolos por meio de menus.	Área de diálogo Área de símbolos Área de edição Área de comandos
Tipos de desenhos	EDMOS: Retângulos coloridos. EDCI: Retângulos representando componentes de um circuito integrado.	Entidades, processos, depósitos e fluxos.	Caixa com entradas e saídas. Conexões e somadores.

Capítulo 3

Estadogramas: Estrutura e Funcionamento

3.1 Descrição Gráfica

O conceito de *statecharts*, ou estadogramas[1], cunhado por Harel [Har 87a, Har 87b, Har 87c, Har 89, Har 90], foi originado do aperfeiçoamento de técnicas já existentes tais como: o formalismo de grafos, somado com noções de círculos de Euler e diagramas de Venn [Har 87a, Har 87b, Har 87c].

Os estadogramas, assim como máquinas de estados finitos, são baseados em estados, eventos e condições com combinações dos dois últimos que causam transições entre os primeiros. Os estadogramas são uma extensão dos diagramas de estados/transições convencionais, acrescidos de conceitos de hierarquia, concorrência e comunicação.

Em uma hierarquia de estados, representados graficamente por retângulos de cantos arredondados e também conhecidos por simplesmente *bolhas*, o aninhamento de bolhas representa as relações hierárquicas entre as mesmas. Por definição de nível hierárquico, dizemos que, na Figura 3.1, os elementos A e J tem nível superior aos elementos D e G.

As transições dentro de um estadograma são interníveis, isto é, podem ligar estados de quaisquer níveis. Uma transição é representada por um arco direcionado, o qual será rotulado com um identificador de um evento seguido, opcionalmente, de uma condição entre parênteses. No caso da existência de uma condição relacionada a um evento, a ocorrência efetiva da transição se realiza se e somente se a condição for verdadeira.

Um estado *default* de uma bolha decomposta é representado por uma setinha tendo um pontinho negro como origem e significa que toda vez que é ativado o seu estado pai e se não houver transição específica para nenhum outro subestado que causou esta ativação, então o subestado *default* é ativado.

Um estado poderá representar uma decomposição de sub-estados do tipo ou-exclusivo (XOR) ou do tipo AND. Na Figura 3.1, temos o estado C que representa o ou-exclusivo dos estados G e H. Isto significa que estar em C implica estar no estado G ou no estado H, mas nunca em ambos. Por outro lado, o estado A é o produto ortogonal dos estados D, E e F com G e H. Seus componentes, B e C, são separados por uma linha tracejada, significando que estar em A, implica estar em alguma combinação de estados dos componentes B e C.

[1] tradução feita por Antônio Figueiredo, mestre em Ciência da Computação, pelo DCC-Unicamp

são representados como estados na Figura 3.2 visto que a hierarquia em diagramas de estados convencionais é "achatada".

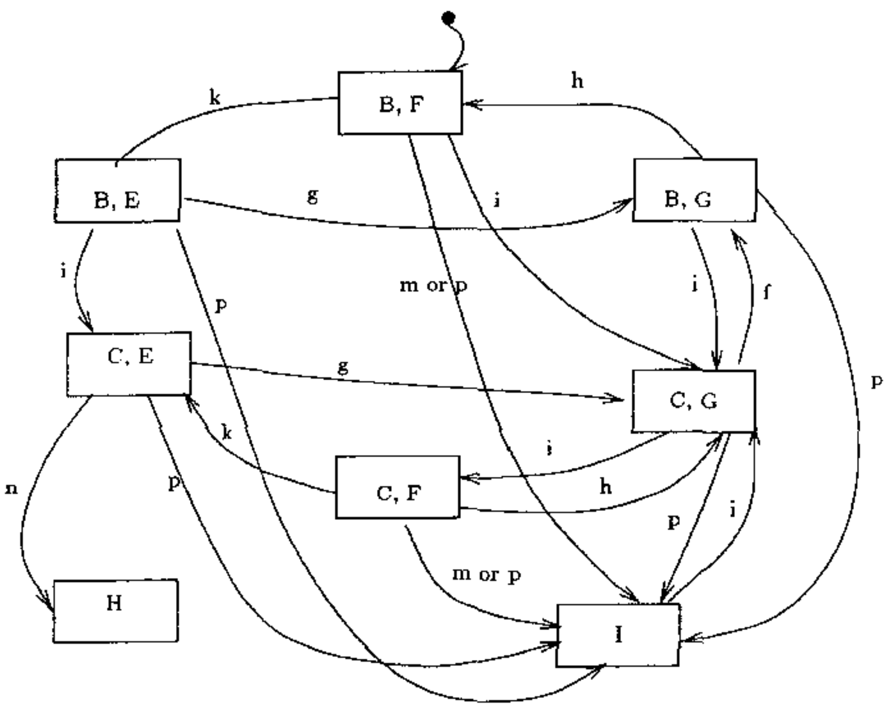

Figura 3.2: Diagrama de Estados

3.1.1 Comunicação em Estadogramas

Até agora, diante do que apresentamos, os componentes ortogonais apenas se sincronizam através de eventos comuns [como o evento i em componentes ortogonais, na Figura 3.3] e afetam uns aos outros através de condições do tipo [se em (estado)]. Uma inovação permitida aos estadogramas é a de gerar eventos internos. Um evento interno pode ser associado opcionalmente a um evento associado a uma transição e é gerado por ocasião do disparo desta transição. A ocorrência de um evento interno causa transições em todos os componentes nos quais ele seja relevante. Por exemplo, se ocorre o evento a, e uma transição rotulada como a/b é disparada, então o evento b ocorre, e é tratado.

A Figura 3.4, apresenta um exemplo deste mecanismo. Se estivermos nos estados (Y, A, D, H, B,F,J) e ocorrer o evento m, a próxima configuração de bolhas ativas será (Y, A, D, H, C,G,I), isto porque o evento ev, gerado no componente H, causa transições nos componentes A e D.

3.1.2 History

Uma outra característica bastante interessante dos estadogramas, é a capacidade de "lembrança" a respeito de uma visita anterior a um estado. A esta capacidade é dado o nome de *history*. É uma

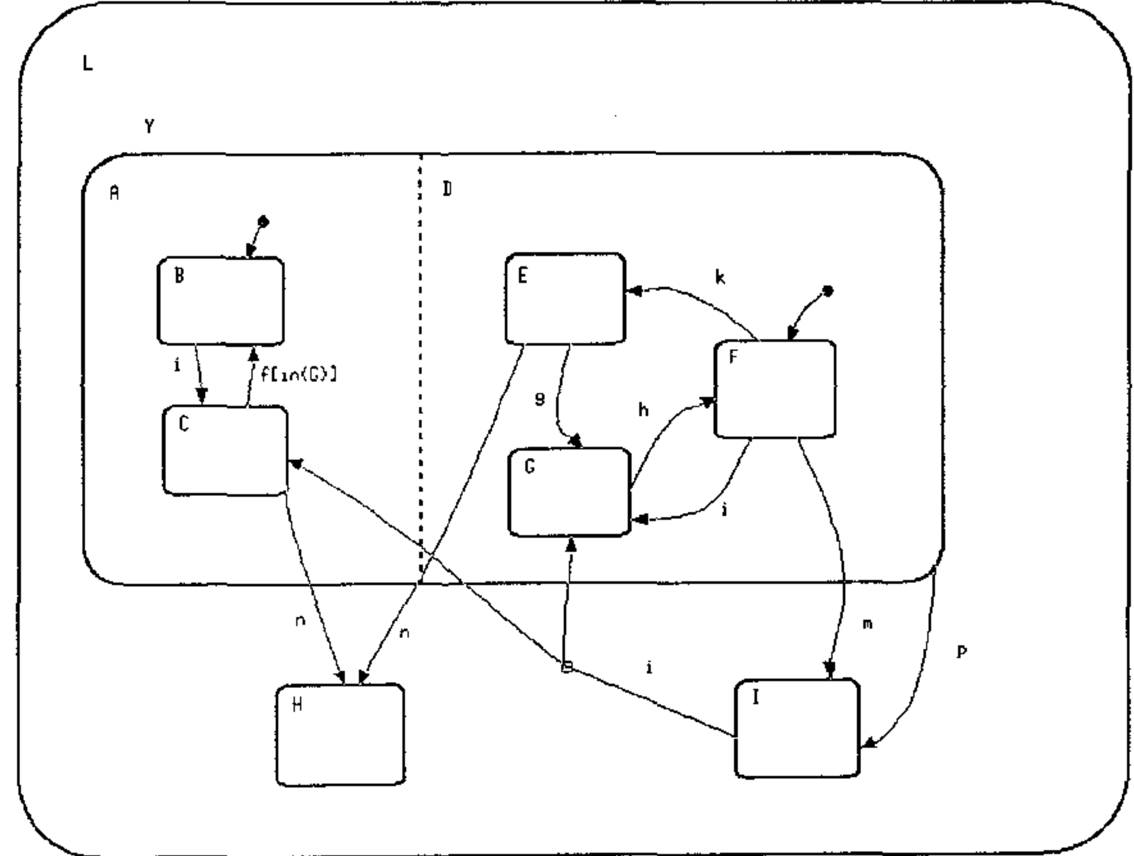

Figura 3.3: Estadograma equivalente à Figura 3.2

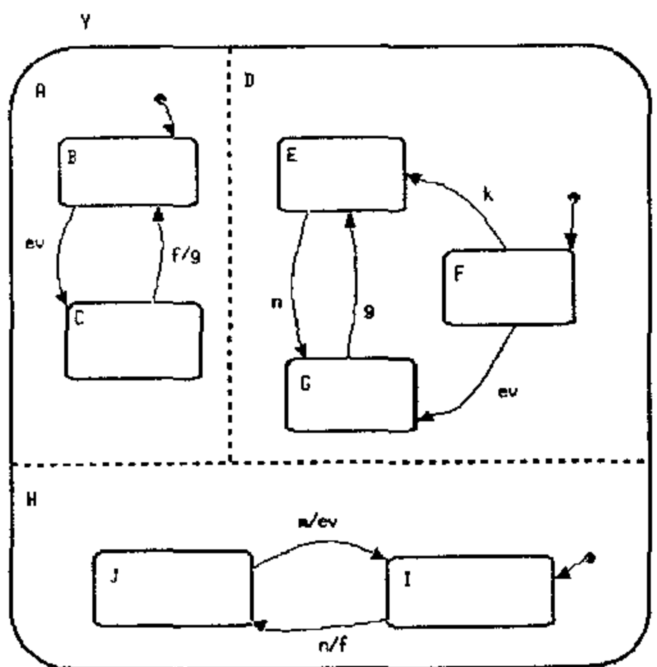

Figura 3.4: Comunicação em estadogramas

maneira bem diferente de se entrar em um grupo de estados. "Entrar-por-*history*"significa entrar no estado mais recentemente visitado. Normalmente o *history* H é aplicado apenas às bolhas do nível em que o mesmo foi definido.

Vemos um exemplo de uma definição de *history* simples na Figura3.5. Nesta Figura o *history* aparece na bolha K, de modo que, quando o sistema entra nesta bolha, dependendo da última "visita" do sistema à bolha K, ocorrerá a entrada em G ou F. Se o sistema estiver entrando em K pela primeira vez, então a bolha *default*, no caso, G, será a escolhida.

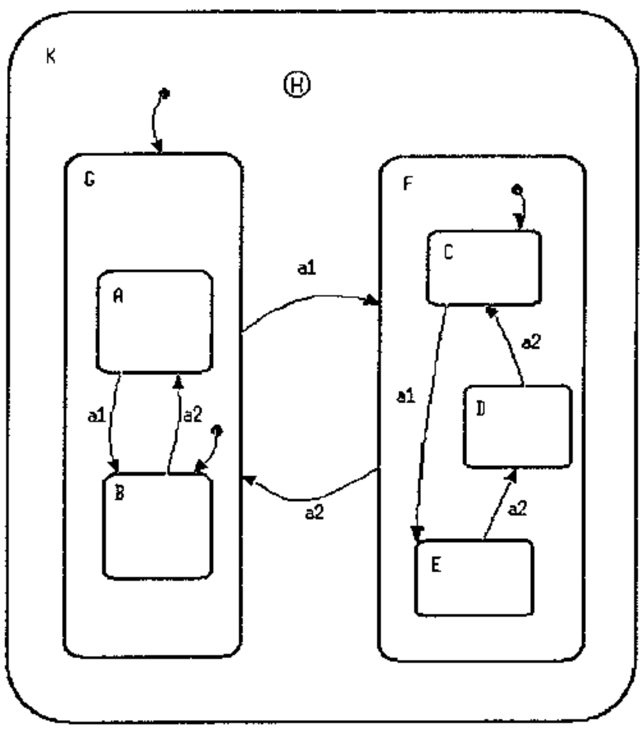

Figura 3.5: History Simples

O conector H pode vir seguido de um asterisco significando que o retorno ao mais recentemente visitado se aplica a todos os subníveis a partir do ponto em que for especificado, a não ser que seja cancelado mais abaixo na hierarquia. Na Figura 3.6, o H* faz com que sejam mantidas as informações da subbolha mais recentemente visitada das bolhas K, G, F. Em caso de uma nova ativação de K esta informação será utilizada para a ativação da subbolha.

3.1.3 O History de Retorno

Já falamos anteriormente de dois tipos básicos de *history*, ambos idealizados pelo Harel, que são o history simples e o history de profundidade.

O *history* de retorno nasceu de uma necessidade real da aplicação de estadogramas na descrição de sistemas dentro do ambiente A-HAND [Fig 91].

Em uma bolha do tipo ou-exclusivo, com vários subestados, pode existir, em um determinado momento, a necessidade de se sair de uma determinado subestado e de voltar para o estado "irmão"

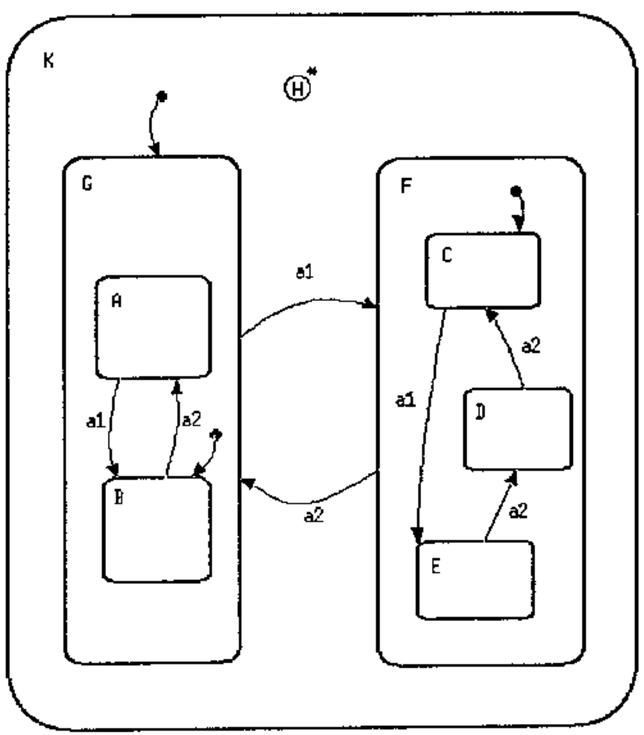

Figura 3.6: History de Profundidade

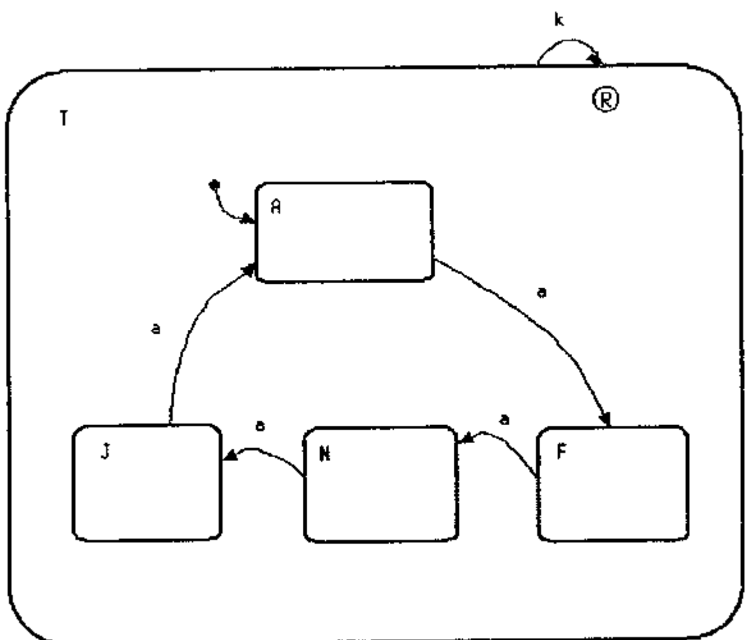

Figura 3.7: History de Retorno

UNICAMP
BIBLIOTECA CENTRAL

anteriormente visitado (caso típico onde este *history* é útil: navegação em menus). Para implementar esta necessidade, criou-se o *history* de retorno.

Quando um bolha com este tipo de *history* é ativada, ocorre conceitualmente o seguinte: uma pilha vazia, com capacidade limitada para conter identificadores é criada. Enquanto a bolha estiver ativada, todos os identificadores de seus descendentes diretos que forem visitados, exceto via *history* de retorno, são empilhados. Uma inserção na pilha ocorre quando um descendente direto é desativado e um novo é ativado. O identificador a ser inserido é o da bolha que acaba de ser desativada. Caso a pilha esteja cheia por ocasião de uma inserção, o elemento do fundo da pilha é descartado para abrir um novo espaço para o identificador a ser inserido. Esta pilha é destruída por ocasião da desativação da bolha à qual está associada.

Caso ocorra uma transição à qual está associada o *history* de retorno, ocorre o seguinte:

1. Estando o sistema em qualquer uma das bolhas componentes da bolha T, segundo a figura 3.7, ocorrendo o evento k, o sistema voltará para o irmão anteriormente visitado desta bolha descendente. Ou seja, o sistema volta para a bolha cujo identificador está no topo da pilha.

2. O passo 1 pode ser executado várias vezes, desta forma, o sistema retrocederá o percurso de navegação dentro da bolha T, até esvaziar a pilha ficando esta modalidade de *history* sem efeito a partir deste ponto enquanto a pilha continuar vazia. Isto é, caso disparada a transição associada ao history de retorno e a pilha estando vazia, tudo se passa como se nada tivesse acontecido.

3.1.4 Ações e Atividades

Os estadogramas descrevem o mecanismo de controle de um sistema, o qual é responsável por tomar decisões dependentes de ocorrências de eventos externos e internos que influenciam o comportamento do sistema como um todo. A reatividade do sistema é expressa pela mudança de estados internos da configuração do sistema. Falta descrevermos a habilidade de gerar eventos ou executar ações que eventualmente afetem condições restritivas de disparo de transições.

A notação ../S pode ser associada a uma transição, significando que uma ação S deve ser executada por ocasião do disparo da transição. Por definição, dizemos que uma ação está relacionada com acontecimentos "instantâneos", que tomem um tempo idealmente zero.

Necessitamos também definir as operações executadas em cada estado. As atividades, tem um tempo de execução maior que as ações, por definição. Elas se referem às funções, ou seja, a computações propriamente ditas.

Basicamente existem três tipos básicos de ações e atividades que se repetem para cada bolha: ação *on_entry* que é executada por ocasião da entrada do sistema na respectiva bolha, ação *on_exit* que é executada em sua saída, e a atividade *throughout*, que é a operação computacional executada enquanto o sistema permanecer em uma determinada bolha.

3.2 O formalismo dos estadogramas

As características de um estadograma muitas vezes provocam problemas semânticos bem particulares. David Harel e outros em [Har 87c] definiram uma sintaxe formal e a semântica operacional de um estadograma, introduzindo, uma sintaxe detalhada e poderosa da notação de estadogramas

completa, com condições e eventos compostos e variáveis compartilhadas que podem ser definidas e testadas.

Podemos dizer que uma "reação" são acontecimentos paralelos em componentes ortogonais. Para exemplificarmos, analisemos a Figura 3.4. Suponhamos primeiramente que o sistema esteja na configuração (Y, A, D, H, B, F, J). Quando ocorre o evento externo m, a próxima configuração do sistema, ou seja, o próximo conjunto de estados ativados do sistema seria (Y, A, D, H, C, G, I) em virtude da ação ev, disparada pela ocorrência do evento m na componente H, disparar duas transições nas componentes A e D.

Sejam e e f expressões booleanas, então, e $\vee$ f e e $\wedge$ f são eventos e também podem ser condições restritivas. Existem diferentes tipos de eventos e condições. Por exemplo, en(s) e ex(s) são eventos que ocorrem na entrada e na saída de um estado S, respectivamente, e in(S) é a condição correspondente.

Para expressões algébricas τ e σ, nós temos condições do tipo, $\tau = \sigma$ e $\tau < \sigma$, assim como, eventos do tipo ch(v) que ocorre na mudança do valor de v. Qualquer condição está relacionada com dois eventos: tr(c) e fs(c), que ocorrem respectivamente quando a condição c muda de falsa para verdadeira e vice-versa. Uma condição especial é denominada ny(e) que significa que o evento e ainda não ocorreu. Esta condição se refere unicamente a um acontecimento dentro da reação que está sendo avaliada correntemente.

Em estadogramas nós temos eventos em níveis altos, concorrência em todos os níveis e entrada por *default* e *history* que são usados no controle de ativação de bolhas em todos os níveis. Também temos reações simultâneas de eventos, e variáveis compartilhadas.

Apresentamos uma semântica operacional formal para estadogramas, conforme definida por Harel. Para tornar nossa explicação mais simples, assumimos que estamos numa configuração C e que um evento externo e tenha ocorrido. Nossa intenção é definir um conjunto possível das próximas configurações next(C), que são os elementos que são as saídas legais de todas as transições que tenham sido disparadas por e, seguida de todas as consequências, ou seja, transições recentemente habilitadas, seguida de suas consequências, etc.

Define-se micro-passo como sendo um passo "real" sendo executado numa sequência máxima de passos. Micro-passos podem capturar a ordem em que transições simultâneas e ações de um simples passo sejam carregadas. Micro-passos não são diretamente transparentes para o usuário, no entanto, os construtores ny e cr podem ser usados para explorá-los beneficamente. Em um micro-passo o sistema seleciona não-deterministicamente algum subconjunto de transições habilitadas, e executa as mesmas numa única e mesma hora. Após isso, adiciona-se suas ações imediatamente conseqüentes, para eventos e transições esperando execução e manipulando o conjunto restante para o próximo micro-passo a ser executado.

A seguir mostraremos alguns exemplos simples. A Figura 3.9 mostra um tipo simples de não-determinismo estrutural onde, se o sistema estiver em B e o evento e ocorrer, duas possibilidades diferentes de mudança de estados em componentes ortogonais serão possíveis. Nesta Figura temos um não-determinismo causado pela liberdade da escolha de subconjuntos em micro-passos. Isso porque, se o sistema estiver em (Y, F1, F2, F3, A, B, C), e o evento ev ocorrer, o sistema poderá chegar em (Y, F1, F2, F3, D, B, F), se as transições tiverem ocorrido na ordem A$\rightarrow$D, B$\rightarrow$E, C$\rightarrow$F, já que a condição no segundo componente será falsa. Contudo, (Y, F1, F2, F3, D, E, F) também poderá ser possível, se as transições forem consideradas na ordem B$\rightarrow$E, A$\rightarrow$D, C$\rightarrow$F. Note que o evento ev$\wedge$f ocorre em ambos os casos. Alguns outros casos em que o sistema age não-deterministicamente

causando resultados, muitas vezes, conflitantes são relatados em [Har 87c].

Na Figura 3.8 abaixo temos um exemplo bastante interessante da condição ny (not-yet-ocurred), onde vemos que a ocorrência do evento ev causa a transição de A para B e a ocorrência do evento f causa a transição de A para C. No caso, porém da ocorrência simultânea dos dois eventos, a condição ny, faz com que o evento ev tenha prioridade, e desta forma o sistema irá para B.

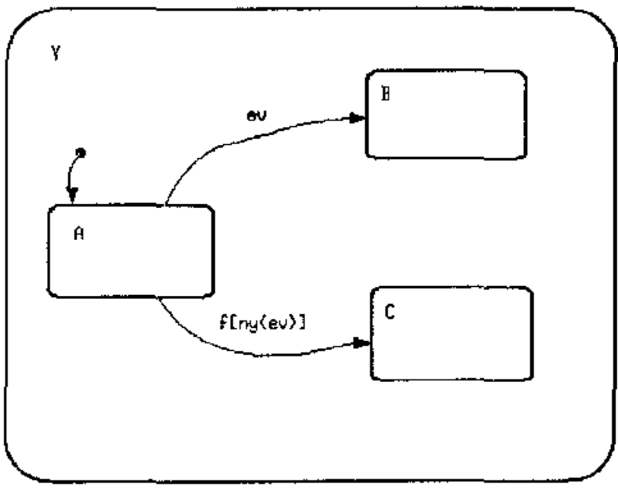

Figura 3.8: Condição ny

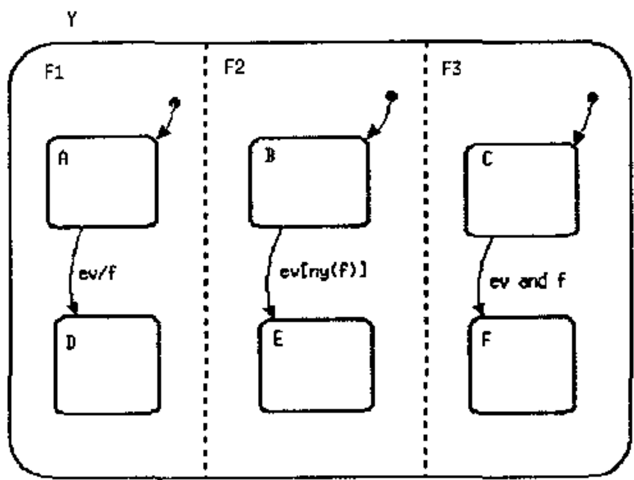

Figura 3.9: Condição ny

A seguir, apresentamos um resumo da sintaxe e da semântica dos estadogramas propostas por Harel em [Har 87c].

3.2.1 Sintaxe

A sintaxe dos estadogramas foi definida baseada nos seguintes conjuntos de elementos: estados, transições, eventos primitivos, condições primitivas e variáveis. Utilizando estes conjuntos básicos, definimos conjuntos estendidos de eventos, condições, expressões e rótulos de transições e suas conexões entre si.

- **Estados:** O conjunto de estados S é definido junto com uma função de hierarquia ρ, uma função de tipo ψ, um conjunto de símbolos *history* H e uma função de *default* δ. A função de hierarquia ρ, define para cada estado, o conjunto de seus subestados. Se $\rho(x) = \rho(y)$ então x = y. Existe um único estado $r \in S$ tal que $\forall s \in S$, $r \notin \rho(s)$ chamado raiz do estadograma.

  A função de tipo $\psi : S \rightarrow (AND, OR)$ define para cada estado o seu tipo. Se $\rho(s) \neq \emptyset$ e $\psi(s) = OR$, então $\rho(s)$ é uma decomposição do tipo ou-exclusivo de s. Isto significa que o sistema em um estado s está em um e apenas um de seus subestados diretos. Se $\rho(s) \neq \emptyset$ e $\psi(s) = AND$ então $\rho(s)$ é uma decomposição do tipo AND de s. Isto significa que o sistema estando no estado s está simultaneamente em todos os seus subestados diretos.

  O conjunto de símbolos *history*, H, é relacionado com o conjunto de estados pela função $\gamma : H \rightarrow S$, tal que, $\gamma(h1) = \gamma(h2)$ implica h1=h2 e $\gamma(H)$ é um subconjunto de estados **ou**.

  A função de *default*, δ, define para um estado s, um conjunto de estados e símbolos *history* que são contidos no estado. $\delta(s)$ é o conjunto *default* para s.

- **Expressões:** O conjunto de variáveis é denotado por Vp. O conjunto de expressões V é definido indutivamente como segue:

  1. Se K é um número, então $K \in V$.
  2. Se $v \in Vp$, então $v \in V$.
  3. Se $v \in V$, então current(v)$\in$V (current(v) pode ser abreviado por cr(v)).
  4. Se $v1, v2 \in V$ e op é uma operação algébrica, então $op(v1, v2) \in V$.

- **Condições:** O conjunto de condições primitivas é denotado por Cp. O conjunto de condições C é definido indutivamente como segue:

  1. $T, F \in C$, onde T e F correspondem a verdadeiro e falso respectivamente.
  2. Se $c \in Cp$, então $c \in C$.
  3. Se $s \in S$, então $in(s) \in C$.
  4. Se e$\in$ E, então not_yet(e)$\in$ C
  5. Se u,v $\in$ V, R $\in$ $\{=, >, <, \neg, \leq, \geq\}$ então uRv $\in$ C.
  6. Se $c \in C$, então current(c)$\in$C
  7. Se c1, c2 $\in$ C, então $c1 \vee c2$, $c1 \wedge c2$, $\neg c1 \in C$.

- **Eventos:** O conjunto de eventos primitivos é denotado por Ep. O conjunto de eventos E é definido indutivamente como segue:

1. $\lambda \in E$, onde λ é o evento nulo.
2. Se $e \in Ep$, então $e \in E$.
3. Se $v \in V$, então changed(v)$\in$ E.
4. Se $s \in S$, então exit(s), entered(s) $\in$ S
5. Se $e1, e2 \in$ E, então $e1 \vee 2$, $e1 \wedge 2 \in E$.
6. Se c $\in$ C, então tr(c), fs(c) $\in$ E
7. Se $e \in E, c \in C$, então $e \mid c \mid \in E$.

- **Ações:** O conjunto de ações A é definido indutivamente como segue:

1. $\mu \in A$, onde μ é a ação nula.
2. Se $c \in Cp, d \in C$, então $c := d \in A$
3. Se $v \in Vp, u \in V$ então $v := u \in A$
4. $a_i \in A$, i=0,..,n então $a_o, .., a_n \in$ A

- **Rótulos:** O conjunto de rótulos L é o conjunto de pares $E \times A$, e para l = (e,a) nós escrevemos e/a. Informalmente, se e/a é um rótulo de uma transição **t**, então **t** é provocado por **e**, e **a** é executada quando **e** é disparada.

- **Transições:** Uma transição t = (x,l,y) é composto de: o conjunto origem x e o conjunto destino y, denotado por origem(t), destino(t), respectivamente, e um label l. Informalmente, se l=e/a, o sistema está em X e **e** ocorre, então t é habilitada e **a** é disparada, e no final o sistema muda de estado.

3.2.2 Semântica

A Semântica dos estadogramas é baseada segundo uma seqüência de instantes de tempo (σ_i's),i$\geq$ 0 correspondendo à taxa do Sistema Sob Descrição (SSD). Os intervalos de tempo básicos são definidos por $I_i = [\sigma_i, \sigma_{i+1})$. Em σ_{i+1} o SSD reage a estímulos externos que ocorrem no intervalo I_i. A semântica de estadogramas é definida para prover uma definição formal para taxas que ocorrem no SSD como uma reação a estímulos externos. Um estímulo externo associado a σ_{i+1} e um terno (π, θ, ϵ) onde π é um conjunto de condições primitivas cujos valores são verdadeiros em $[\sigma, \sigma_{i+1})$ para algum $\sigma >= \sigma_i$, sendo ϵ uma função determinada pelo ambiente tal que, para a variável v, $\epsilon(v) = x$, se o valor de v é x em $[\sigma, \sigma_{i+1})$ para algum $\sigma > \sigma_i$. A configuração do sistema associado com um instante σ_{i+1} é uma quádrupla (X,π, θ, ϵ) onde X é uma configuração maximal de estados do estado raiz e (π, θ, ϵ) é um estímulo externo associado com σ_{i+1}.

A reação do sistema em algum instante é composta pelo conjunto de transições que ocorrem naquele instante e o conjunto de eventos gerados quando estas transições são feitas. Então, a reação do sistema é o par (Υ,Π') onde Υ é um conjunto de transições chamados de passo e Π' é o conjunto de eventos atômicos gerados por Υ. Dada a configuração do sistema cs = (X, Π, θ, ϵ), o conjunto de SR = (Υ, Π') é definido pelas possíveis reações do sistema para X.

Intuitivamente dizemos que um passo é um conjunto de transições consistentes que são estruturalmente relevantes para uma dada configuração do sistema e habilitada sob um dado estímulo

interno ou externo. O conjunto de eventos gerados é o conjunto de eventos que ocorrem como resultados de passos de transições que são alcançadas e execução de ações associadas com estas transições. Todas as transições que participam de um passo Υ são conceitualmente executadas de forma simultânea. Na definição formal de um passo, seqüencia-se o conjunto de transições tomadas em micro-passos, cada um deles sendo um subconjunto de Υ. Então, um passo é definido como sendo uma seqüência de micro-passos. Isto é importante para entender que apesar de micro-passos serem essenciais para a definição precisa e o entendimento da semântica, eles são considerados como um mecanismo interno para passos computacionais e de fato podem ser ocultados do usuário.

Segundo especificação de semântica feita no STATEMATE [i-L 89a], podemos admitir as definições que seguem. Primeiro, definimos o *status* inicial do sistema, seguido de passos para cada mudança de *status*. Então:

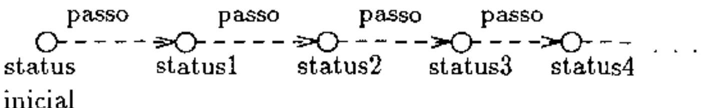

Figura 3.10: Definição de *status*

Um *status* contém as seguintes informações:

- A configuração de estados em que o sistema se encontra;
- Os valores dos itens de dados;
- Os valores-verdades das condições;
- As atividades que estão ativas;
- Listas internas de eventos;
- Informações de *history*.

Existem três tipos de estados em um *statechart*: OR estados, AND estados e estados atômicos. Estados OR, possuem subestados que se relacionam uns com os outros por ou-exclusivo, ou seja, estar em um estado OR significa estar em um e apenas um de seus subestados diretos. Por sua vez estados AND possuem componentes ortogonais que se relacionam por um AND, ou seja, estar em um estado AND significa estar em todos os seus componentes ao mesmo tempo. Estados atômicos são aqueles que se situam no final da hierarquia de estados, ou seja, não possuem subestados. O estado que se situa no nível mais alto do estadograma, que não possui estado pai, é chamado de estado raiz.

A configuração de um sistema é o conjunto máximo de estados em que um sistema pode estar ao mesmo tempo. Mais precisamente, uma configuração é um conjunto de estados C que obedeça as seguintes regras:

- C contém o estado raiz.

- Se C contiver um estado A do tipo OR, então ele deve conter exatamente um dos subestados de A.

- Se C contiver um estado A do tipo AND, então ele deve conter exatamente todos os seus subestados.

Uma configuração é denominada básica quando ela é formada apenas pelos estados atômicos de uma configuração.

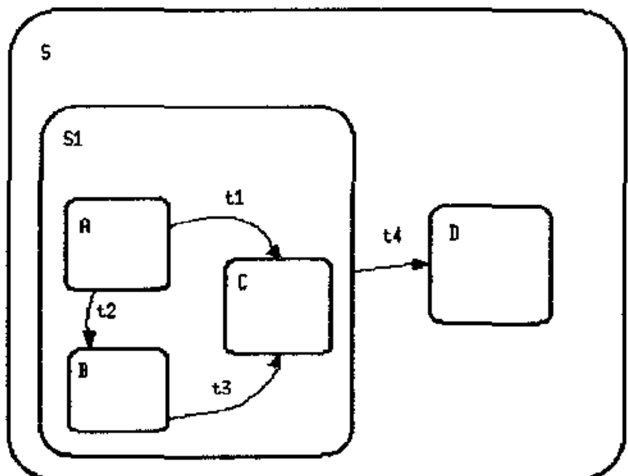

Figura 3.11: Transições Conflitantes

Podemos encontrar em determinados casos transições que são conflitantes entre si. Como exemplo disto podemos mostrar o estadograma da Figura 3.11. Nesta figura temos que t1 e t2 são conflitantes entre si, porque cada uma delas implica na saída do estado A. Também podemos dizer que t4 está em conflito com t1, t2 e t3, pois a ocorrência de t4 só terá efeito quando o sistema estiver em S1, e isto implica dizer estar em algum de seus subestados. Suponhamos então que t1 e t4 ocorram ao mesmo tempo. As duas transições não podem ser executadas em um mesmo passo. Contudo, a semântica trata este tipo de conflito de forma diferente. No primeiro caso, quando se habilitam as transições t1 e t2, se as duas estiverem "prontas" no início do passo, nós dizemos que o sistema está diante de um não-determinismo, já que não existem razões de preferência por uma ou outra. A situação é diferente diante da habilitação de t1 e t4 em um mesmo passo. Neste caso, nós dizemos que a transição t4 tem prioridade sobre a transição t1 (e também sobre t2 e t3), e desta forma o não determinismo não existe.

As definições da semântica dos estadogramas são muito extensas. Pode-se encontrar mais detalhes em [i-L 89a, Har 87c].

3.3 Ferramentas para Descrição de Sistemas Reativos através de Estadogramas

3.3.1 STATEMATE

STATEMATE [i-L 87] é uma ambiente de desenvolvimento visual de sistemas complexos. Nesta seção apresentaremos as principais características deste sistema.

No desenvolvimento de sistemas complexos existem diversos aspectos que o projetista precisa definir e que são particulares ao Sistema Sob Descrição (SSD). No entanto existem algumas características que são comuns na maior parte dos sistemas, que são: módulos, atividades, item de dados, estados, eventos e condições.

Os módulos são a parte mais concreta de todos. Eles representam os componentes físicos do SSD, provendo uma descrição da estrutura de implementação do próprio sistema. Um outro conceito um tanto mais abstrato que os módulos, refere-se às atividades. As atividades podem ser associadas às funções do aplicativo propriamente dito de um sistema computacional. A noção mais abstrata de todas, no entanto, fica para os estados, eventos e condições. Um estado é entendido como um modo de um sistema. Em um estado típico podem existir diversas atividades, da mesma forma que uma atividade pode estar associada com a passagem dos sistema por diversos estados.

O STATEMATE divide os sistemas em três visões:

- A visão comportamental,
- A visão funcional,
- A visão estrutural.

A visão comportamental é a visão suportada pela linguagem gráfica de estadogramas e descreve o SSD através de estados, eventos, condições e transições.

A visão funcional é suportada pela linguagem gráfica de diagramas de atividades e descreve o sistema através de fluxo de informações entre os mesmos.

A visão estrutural é suportada pela linguagem gráfica de diagramas de módulos que descreve o sistema através de componentes físicos e suas ligações entre os mesmos.

Todas essas linguagens são baseadas em descrições de sistemas complexos de forma hierárquica. Elas descrevem o sistema com profundidade, com descrições multiníveis, apresentando a capacidade de *zoom* para descrições visuais.

O comportamento dinâmico comumente relacionado com aspectos de controle, são tratados em STATEMATE pelos estadogramas.

Em STATEMATE, os estadogramas e os diagramas de atividades estão altamente integrados. Associado com cada nível do diagrama de atividades, existe usualmente um estadograma, chamado de atividade de controle, cuja função é controlar as atividades e os fluxos de dados deste nível, da mesma forma como os estadogramas são capazes de gerar diversos tipos de ações que podem afetar as atividades. Por exemplo, eles podem enviar às atividades as instruções de começar, parar ou retomar suas tarefas.

Desta forma podemos afirmar que os diagramas de atividades são incompletos como modelos de um sistema, já que eles não especificam o controle do sistema. Assim como os estadogramas

são incompletos como especificações, pois sem atividades eles não tem o que controlar. Juntos, o diagrama de atividades detalhado e seu estadograma de controle, provêem um modelo conceitual.

O STATEMATE possui um editor gráfico para cada uma destas linguagens, e de um outro editor não gráfico, que juntos permitem projetar um sistema.

3.3.2 Um Editor Gráfico para Estadogramas

O Editor Gráfico para Statecharts, EGS [Bat 91] foi desenvolvido por João do E. S. Batista Neto, sob orientação do Prof. Dr. Paulo César Masiero, no Instituto de Ciências Matemáticas de São Carlos, da Universidade de São Paulo. O EGS permite a criação de estadogramas incluindo todas as definições propostas por David Harel em [Har 87a, Har 87b, Har 87c], com funções de expansão de bolhas, e posterior simulação do estadograma editado.

O EGS foi desenvolvido segundo o tipo particular de *Higraphs* propostos por Harel em [Har 87a], que são os estadogramas. Baseado na semântica formal de estadogramas e de *Higraphs*, implementou-se as principais características dos mesmos, contidos nos artigos [Har 87a, Har 87b, Har 87c]. A partir disto desenvolveu-se a ferramenta.

O EGS permite a construção de um estadogramas por meio de menus que indicam as operrações de manipulação de bolhas e arcos. Cada uma delas consiste de uma agregação de atributos que o usuário pode manipular.

Na edição de uma bolha, há a opção de defini-la como sendo do tipo OR ou do tipo AND. A opção dada como *default* é o tipo OR. Caso seja escolhido o tipo AND, o usuário deve informar quantos são os componentes, e depois indicar no desenho a posição dos diversos componentes. Depois é solicitado o nome da bolha e a seguir a localização do mesmo no diagrama.

Na edição de um arco, pode-se defini-lo como sendo um arco *default*, um arco 1 para 1 e um arco M para N. Todos os eles, com exceção do arco *default*, são acompanhados de um rótulo que pode ser simples ou composto. O EGS associa a definição de *history*, feita pelo Harel, aos arcos, de modo que uma bolha pode ser afetada po mais de um *history*. Assim sendo, uma bolha β com uma transição rotulada com o evento γ associada a um *history*, funciona da seguinte forma:

- Na ocorrência de qualquer outro evento diferente de γ, que habilite uma transição qualquer, sem *history* associado, que tenha como bolha destino a bolha β, a entrada em β implicará na ativação da bolha *default* de β.

- Na ocorrência do evento γ, a entrada na bolha β será feita através da modalidade de *history* associada à transição.

Também é permitido a definição de variáveis que podem estar relacionadas com eventos, condições e ações, sendo estas últimas associadas a estados ou transições. O usuário poderá realizar operações de consulta e de atualização destas variáveis.

Após finalizado o desenho do estadograma, o EGS interage com um outro software que efetua uma simulação de estadogramas da seguinte forma: a partir de uma lista de eventos fornecida pelo sistema, o usuário vai disparando os eventos que deseja, para verificar visualmente a mudança de estados.

3.3.3 Gerador de Gerenciadores de Sistemas Reativos

O Gerador de Gerenciadores de Sistemas Reativos (GGSR) ou Sistemas de Controle [Fig 91], também utiliza-se da notação de estadogramas. Esta ferramenta objeto da tese, intitulada " **Um Processo de Síntese de Sistemas Reativos**" foi desenvolvida por Antônio Figueiredo Filho, no Departamento de Ciência da Computação da Unicamp, sob orientação do Prof. Dr. Hans Kurt Edmund Liesenberg.

O GGSR utiliza-se de um arquivo contendo uma descrição textual de um estadograma, escrita na linguagem LEG idealizada pelo próprio autor, e a partir disto gera um programa em C funcionalmente equivalente ao estadograma original. Os programa em C funcionam como gerenciadores da estrutura de controle de um sistema.

O GGSR comporta-se como um tradutor, pois inicialmente ele traduz um programa escrito em LEG para a linguagem C e depois é efetuada a compilação do programa gerado.

Detalhes de Implementação

O processo de tradução da linguagem LEG para a linguagem C está dividido em duas fases: a fase de análise e a fase de síntese. Na fase de análise, são realizadas a análise léxica e análise sintática e, na fase de síntese, é feita a geração de código propriamente dita.

Os analisadores léxicos e sintáticos foram gerados a partir das ferramentas *Lex* e *Yacc*.

Durante o processo de tradução são criados estruturas de dados intermediárias que permitem a transmissão de informações do primeiro para o segundo passo. Essas estruturas são tabelas de símbolos, tabelas de transições e uma árvore sintática que representa a estrutura textual do estadograma a ser convertido em um programa em C. Este processo de tradução baseia-se em técnicas padrões de compilação. A seguir descreveremos algumas dessas estruturas.

1. Tabela de Símbolos

   Armazena-se nesta estrutura algumas informações necessárias para a geração de código, tais como: identificadores, categoria associada e, no caso de bolhas, um índice que servirá como seu identificador interno.

2. Tabela de Transições

   Essa estrutura é semelhante a tabela de símbolos, obviamente com campos diferentes. A tabela de transições permite manter informações necessárias para efetuar as mudança de estados.

3. Árvore Descritiva da Hierarquia do Estadograma

   Essa estrutura é gerada concomitantemente à geração da tabela de símbolos e a tabela de transições, e representa de forma abstrata a hierarquia do estadograma a ser convertido em programa.

Estrutura dos Programas Gerados

Um gerenciador construído pelo tradutor, objeto do GGSR, constitui-se de uma ferramenta que efetua mudanças do estado global de um estadograma (status) em função da ocorrência de eventos

e executa as ações associadas às bolhas envolvidas nas mudanças efetuadas. Essas mudanças são feitas internamente por meio da manipulação das tabelas acima descritas.

- Estrutura dos Programas Fonte

  Programas em LEG (linguagem de estadogramas) têm basicamente uma estrutura aninhada que reflete a hierarquia de estadogramas. Para melhor entendimento dos programas escritos em LEG, ver Apêndice B.

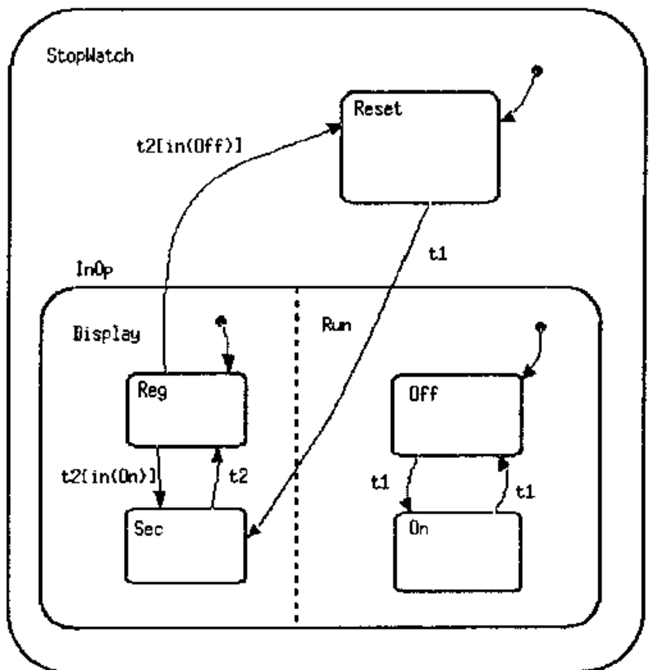

Figura 3.12: Estadograma de um Cronômetro

Descrição em LEG do estadograma da figura 3.12.

```
main blob StopWatch
{
  on_entry : [DrawWatch();];
  on_exit  : [ByeWatch();];
  blob Reset
  {
    on_entry : [ResetTime();];
    Transition
    {
      on_event(t1)  to Sec
    }
  }
  blob InOp
  {
    blob Display
    {
      blob Reg
      {
        on_entry : [freg();];
        on_exit  : [fexitreg();};
        Transition
        {
          on_event(t2[in(Off)])  to Reset ;
          on_event(t2[in(On)])  to Sec
        }
      }
      blob Sec
      {
        on_entry : [fsec();];
        Transition
        {
          on_event(t2)  to Reg
        }
      }
    } |
    blob Run
    {
      blob Off
      {
        on_entry : [foff();];
        Transition
```

```
          {
            on_event(t1)  to On
          }
        }
        blob On
        {
          on_entry : [fon();];
          Transition
          {
            on_event(t1)  to Off
          }
        }
      }
    }
  }
```

Sempre que existirem comandos do tipo on_entry, throughout, on_exit, têm-se chamadas de funções em C, que deverão estar em outros arquivos definidos pelo usuário para serem posteriormente ligados, num segundo passo, durante o processo de compilação. A seguir descreve-se o código gerado pelo tradutor a partir de uma especificação dada por um programa LEG.

- O Código Gerado

Iremos agora descrever parte da estrutura do código gerado. O programa principal indica que, primeiramente, é gerado o estado (ou configuração) *default* inicial, através da função `InitState()` e seguido da invocação da função `BlobMainLoop()`, que pode depender da plataforma de software e hardware para a qual a aplicação tiver sido gerada.

```
main()
{
current_state = InitState();
BlobMainLoop();
}
```

```
BlobMainLoop()
{
do {
   traverse(event = GetEvent());
} while(current_state != NULL);
}
```

No corpo desta função é executado um comando repetitivo onde, em cada iteração, é obtido o "próximo" evento e, em função do mesmo, é analisado o estado corrente para ver se este é afetado.

No corpo da função `traverse()` são incorporadas as transições entre as bolhas em função dos eventos e condições definidas no estadograma que serviu de especificação para o gerenciador gerado. A configuração da função, portanto, é dependente do estadograma apresentado ao tradutor.

A função `traverse()` percorre a lista ligada que mantém o estado global do sistema para verificar se uma dada bolha, cujo identificador se encontra nesta lista, é afetada pelo evento corrente. Em caso afirmativo o estado corrente do estadograma implementado é atualizado.

Tanto para as ações *on_entry* tanto como para as ações *on_exit* gera-se arquivos responsáveis pelo comportamento das mesmas. Assim sendo, um mesmo bloco de programa, através de um comando de seleção, chama as diversas funções *on_entry* (*on_exit*), para cada estado.

Podemos dizer que o código gerado da aplicação constitui-se de duas partes independentes. A primeira delas, que foi a apresentada anteriormente depende da plataforma sobre a qual será executado o programa, mas é invariante com relação ao estadograma submetido ao processo de transformação.

A segunda, que entre outros, verifica a sensibilidade de bolhas a eventos e executa ações associadas, é dependente do estadograma transformado. A principal função da parte invariante denomina-se `fromBlobtoBlob()` e implementa o disparo de transições. Ela possui dois parâmetros que são a bolha origem e a bolha destino. Ao ser invocada a função `fromBlobtoBlob()` realiza-se uma busca do ancestral comum mais próximo das bolhas origem e destino. No caminho do descendente atômico da bolha origem até o ancestral comum são disparadas as funções *on_exit* de todas as bolhas encontradas, e na descida do ancestral comum até a bolha destino, são disparadas todas as funções *on_entry* das bolhas. O processo de ativação é continuado obedecendo-se agora eventuais *histories* associados até se alcançar uma bolha atômica.

Capítulo 4

O Editor de Estadogramas

Como já foi dito anteriormente no Capítulo 1, é grande a dificuldade encontrada na tarefa de especificação e desenvolvimento de sistemas reativos.

No projeto deste tipo de sistema, o projetista muitas vezes se depara com suas próprias dúvidas sobre como organizar o sistema. A técnica de estadogramas permite que se identifique e especifique os estados de um sistema, ou seja, todos as etapas de execução deste sistema para, a partir disto, relacioná-los entre si através das transições disparadas por eventos.

4.1 Necessidade da Ferramenta

Na revisão bibliográfica descrita no Capítulo 2 tivemos oportunidade de verificar que os Editores Gráficos relatados estão relacionados com diferentes áreas de projeto de sistemas, alguns com Banco de Dados, outros, em sua maioria, relacionados com projetos de microcircuitos, outros com modelagem de sistemas. No final do Capítulo 3 vimos outros editores que estão relacionados com descrição de sistemas reativos e utilizam-se da notação de estadogramas, tais como o STATEMATE e o EGS. Verificamos, então, que existem poucas ferramentas desenvolvidas que se utilizam de estadogramas para modelar sistemas.

Por causa disto, tivemos necessidade de especificar uma ferramenta nova, que suportasse algumas características específicas do nosso projeto. A título de comparação, descrevemos a ferramenta com os mesmos tópicos descritos na Tabela do Capítulo 2:

- **Área de Aplicação da Ferramenta:** Descrição e Projeto de Sistemas Reativos; Depuração de Programas.

- **Estruturas de Dados:** Estruturas de Dados convencionais com manipulação de listas, vetores e arquivos.

- **Funções Principais:** Edição de Estadogramas, Geração de uma Descrição Textual de Estadogramas, Simulação de Estadogramas e Depuração de Programas.

- **Interface com o Usuário:** Manipulação de diversas janelas e menus criados com o auxílio do XView [Hel 90].

- **Tipos de Desenhos:** Bolhas, Arcos, conectores Histories, linhas tracejadas e setas.

A ferramenta a ser descrita consiste em um "Ambiente de Edição e Simulação de Estadogramas", onde são fornecidos mecanismos que auxiliam o projetista de sistemas reativos na difícil tarefa de "estadificação".

Depois de especificado o estadograma detalhado do sistema, é oferecida também a oportunidade do usuário simular a especificação. Durante o processo de simulação, tem-se a visualização gráfica do que aconteceria em termos de mudanças de estados, no caso da ocorrência de uma seqüência de eventos.

Também há uma outra forma de simulação que será apresentada na seção 4.3.2.

4.2 Passos de uma especificação

O projetista deve, em primeiro lugar, antes de começar a especificar o sistema, conhecer o estado principal de seu sistema correspondente ao módulo de execução principal. Esta deve ser a primeira bolha a ser representada na especificação.

A partir disto vão-se criando novas bolhas até que finalmente se chegue às bolhas atômicas do sistema, ou seja, bolhas terminais, que não possuam descendentes. Nesta etapa, especificam-se também as transições rotuladas com os seus respectivos eventos.

Depois de concluída a especificação, no momento de gravar as informações geradas em um arquivo, obtém-se uma descrição textual do estadograma editado.

Esta descrição textual será útil para a comunicação com o GGSR, que posteriormente lerá esta descrição e fornecerá como saída um programa em C funcionalmente equivalente ao estadograma editado (ver seção 3.3.3).

4.3 A Ferramenta

A ferramenta, objeto deste trabalho, subdivide-se nos seguintes módulos:

- Editor de Estadogramas
- Gerador da Descrição Textual
- Simulador de Estadogramas

4.3.1 Editor Gráfico de Estadogramas

Durante a fase de especificação de nosso sistema, ficamos sabendo da existência do EGS, "Editor Gráfico de Statecharts", o qual se comunica com o "Analisador de Statecharts" (ou Simulador) [Bat 91]. Através dos próprios autores do EGS, tivemos acesso aos códigos fontes dos dois sistemas de modo que pudemos aproveitar grande parte de suas funções.

Funções

O Editor Gráfico de Estadogramas apresenta as funções básicas de edição de bolhas, arcos, e manipulação de arquivos.

1. Bolhas:

   A edição de uma bolha se realiza de forma simples, através de seu posicionamento na tela gráfica, e da definição de seus atributos.

   Uma bolha pode ser do tipo AND ou OR. Uma bolha do tipo AND tem componentes concorrentes como descendentes diretos. Deve ser fornecido, logo em seguida à especificação da bolha, o número de componentes que a compõe. Depois disso, deve-se indicar os pontos onde serão traçadas as linhas tracejadas, que indicarão os limites entre componentes. Cada componente passa então a ser decomposto em bolhas OR.

   Quando estamos em uma bolha AND, significa que estamos em todos os seus componentes ao mesmo tempo. Quando estamos em uma bolha OR, significa que estamos em um e apenas um de seus filhos.

   A uma bolha pode estar associado um símbolo de *history*, o qual pode ser simples ou de profundidade, significando que a mesma terá uma "memória" em relação a ativação recente de seus filhos (ver Capítulo 3).

   Também pode-se fornecer ações e atividades associadas a esta bolha, ou seja, uma ação *on_entry*, uma atividade *throughout*, e uma ação *on_exit*. Na ferramenta aqui descrita, estas ações e atividades devem ser invocações a funções em C, devidamente especificadas em algum arquivo. O sistema não se incumbe de verificar a existência ou não dessas funções.

   Uma bolha pode ser alterada depois que sua especificação estiver finalizada. Para que isso aconteça, basta que o usuário indique qual a bolha cujos atributos deseja alterar. A alteração dos atributos é quase idêntica à definição dos mesmos, com exceção do atributo tipo, que não pode ser alterado. A alteração deste atributo acarretaria na mudança de sua apresentação e, por causa disto, preferimos não suportá-la.

   Uma bolha também poderá ser removida. Para isto basta que se indique qual a bolha que se deseja remover, valendo salientar que todas as suas bolhas filhas bem como as transições associadas a estas bolhas também serão removidas.

   No momento da edição de um estadograma, pode-se necessitar incluir subbolhas em uma determinada bolha, e esta operação ser dificultada pelas suas dimensões no traçado corrente. No momento da seleção da bolha para sua devida expansão, abre-se uma janela auxiliar onde será feito um novo traçado da bolha em questão. As transições anteriormente definidas que a tinham como bolha origem ou destino, permanecem inalteradas. Como também suas bolhas vizinhas. Na verdade, ficam disponíveis então dois traçados da mesma bolha, simulando-se um "zoom" da mesma. Por convenção, diz-se que a cada nova expansão aumenta-se o nível de visualização da figura. Em cada bolha expandida têm-se dois desenhos definidos em dois níveis de visualização diferentes. Depois de criada a janela de expansão pode-se traçar a bolha, com a dimensão que se desejar, e depois disso inserir na mesma, todas as subbolhas e transições desejáveis.

   Em um estadograma completo podem estar definidos vários níveis de visualização.

2. Arcos:

   A edição de um arco é feita apontando-se quais os pontos que se deseja interligar para representar uma transição entre duas bolhas. Depois de finalizado o desenho, os atributos do arco

devem ser fornecidos.

Um arco pode ser *default*, 1 para 1 e M para N. (Na versão original do EGS, definiu-se arcos com M bolhas origem e N bolhas destino). Pode-se definir um arco como primitivo ou composto (esta é uma definição feita pelos autores do EGS). Um arco primitivo é aquele que só tem o rótulo associado a ele. Um arco composto é aquele que possui o rótulo (identificador que acompanha o desenho do arco), especificação do evento, a ação a ser disparada ao final da execução da transição e uma condição a ser testada para habilitar a execução da transição.

A ação que acompanha a definição de um arco é, na verdade, um identificador de um evento que já deve ter sido anteriormente especificado em alguma outra transição. Normalmente relacionam-se duas transições situadas em componentes ortogonais "irmãos". Deste modo uma transição dispara uma outra transição situada em outro componente ortogonal. Este mecanismo é conhecido como a capacidade de "broadcast" dos estadogramas, ou "reação em cadeia", ou seja, indução de acontecimentos paralelos em componentes ortogonais (ver seção3.1.1).

A condição deve ser sempre do tipo "in(Blob)", o que significa que ela está dependendo do estado global do sistema, ou seja, se ele se encontra na bolha Blob, ou não.

Também pode-se definir um símbolo de *history* para o arco. Este pode ser simples, de profundidade ou de retorno (ver seção 3.1.3).

O *history* de retorno só pode ser especificado quando a bolha origem e a bolha destino, do arco a que ele está associado, forem iguais. Além disso, a bolha não pode ser do tipo AND e deve obrigatoriamente ter bolhas filhas. Este teste é feito no momento de sua definição.

Um arco 1 para 1 pode ser expandido, ou seja, pode estar definido em mais de um nível de visualização. A expansão de arcos consiste na existência de várias subdivisões do mesmo arco, onde cada uma destas partes poderá estar situada em qualquer um dos níveis de visualização existentes para aquela determinada especificação, desde que sejam níveis de visualização adjacentes (no caso de bolhas expandidas, não temos partes da bolha em níveis diferentes, mas, dois desenhos completos da mesma bolha, sendo um deles mais detalhado que o outro). A existência de um arco expandido requer a definição anterior de bolhas expandidas. Um arco pode "entrar" em uma bolha expandida, indo terminar sua definição em alguma subbolha da mesma, terminal ou não. Um arco pode, também, "começar" em uma bolha filha de uma bolha expandida, saindo da mesma e terminando em outra bolha em algum outro nível de visualização. Um arco expandido pode passar por vários níveis de visualização.

Um dos problemas encontrados na definição de arcos consiste em conectar um componente AND a alguma transição. Isto acontece porque a definição gráfica das bordas do componente AND se confunde com as bordas da própria bolha AND, ancestral direto do componente. Nós estendemos a função já implementada de finalizar arco, para contemplar também arcos expandidos. Havendo a necessidade então de se criar uma transição com origem ou destino no componente pode ser resolvido com as opções de "Iniciar" ou "Finalizar". Deste modo, posiciona-se o cursor no interior do componente, e depois disso, solicita-se as opções de "Iniciar", e o sistema assume que o arco está começando naquele ponto, tendo como bolha origem o componente; ou "Finalizar", onde o sistema finaliza o arco tendo como bolha destino o componente. A Figura 4.1 ilustra um exemplo de utilização deste procedimento.

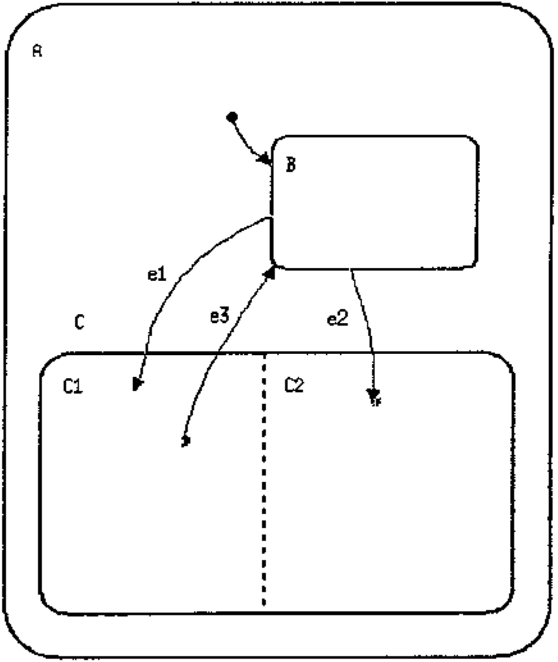

Figura 4.1: Arcos criados com as opções de iniciar e finalizar

Um arco pode ser removido. Para isto basta posicionar o cursor em cima do rótulo do mesmo, após selecionada a opção de remoção.

3. Arquivos:

   As manipulações de arquivos são as convencionais em qualquer sistema de edição, ou seja, salvar um (novo) estadograma, carregar um estadograma, abandonar a edição.

   O arquivo de descrição textual é gerado todas as vezes que se salva o estadograma. Um exemplo de uma descrição textual é dada em 3.3.3.

   Não está incluído na descrição textual os *histories* associados a arcos, pois os mesmos não estão implementados no GGSR, exceto o *history* de retorno.

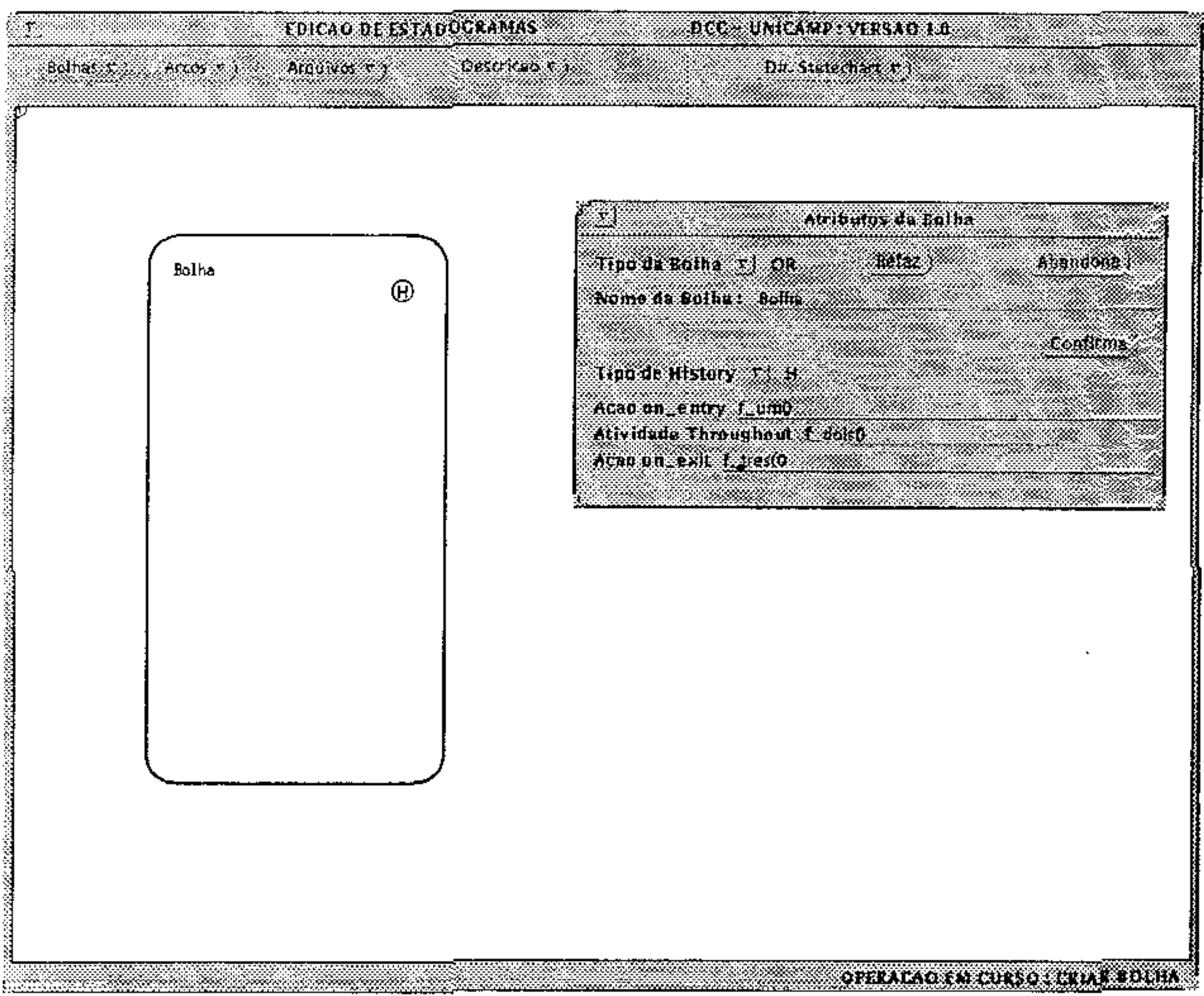

Figura 4.2: Interface do Editor no momento da criação de uma bolha

Estadogramas do Editor

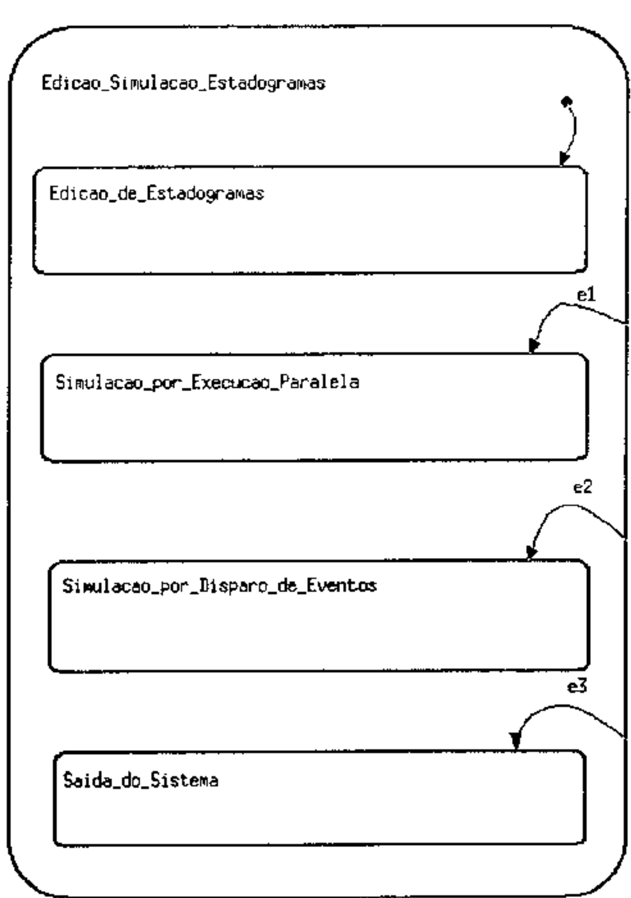

Figura 4.3: Tela Principal do EGREST

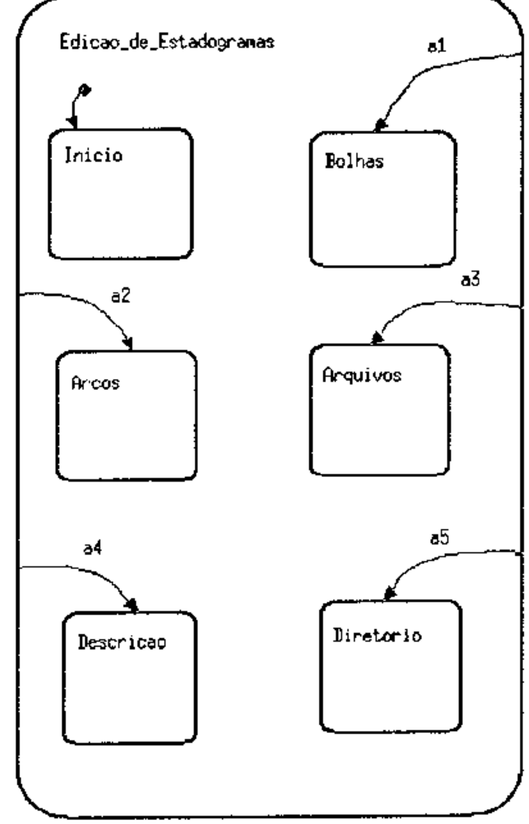

Figura 4.4: Edição de Estadogramas

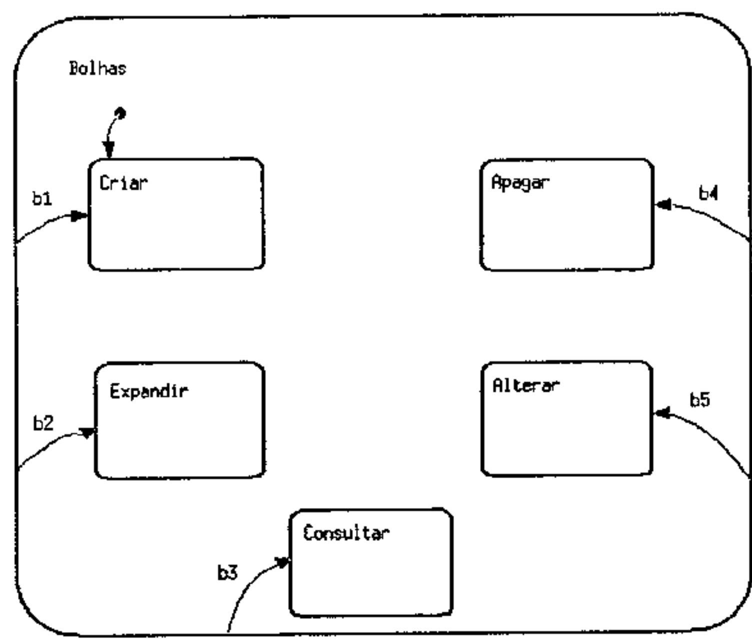

Figura 4.5: Edição de Bolhas

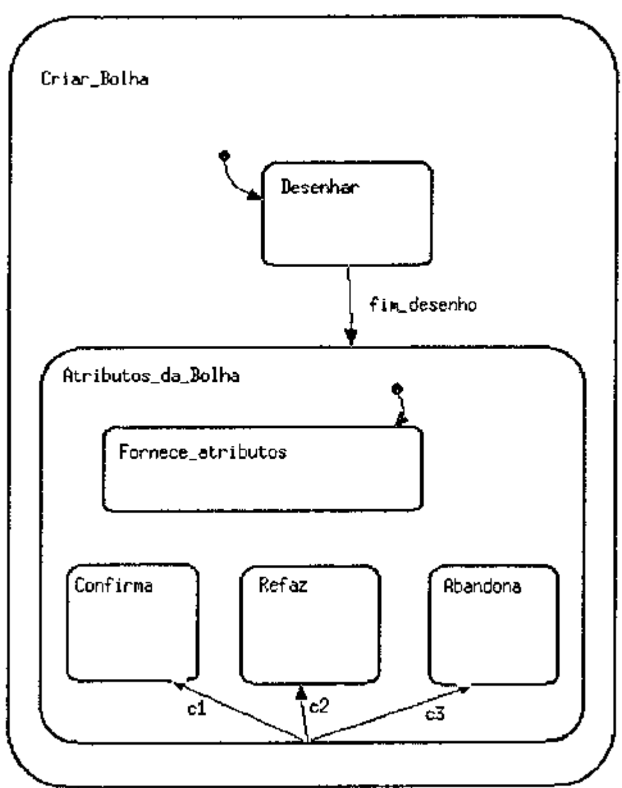

Figura 4.6: Criar Bolha

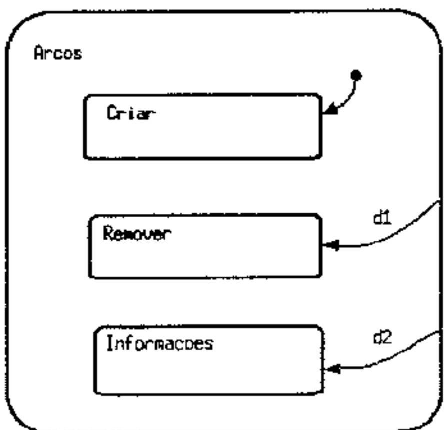

Figura 4.7: Edição de Arcos

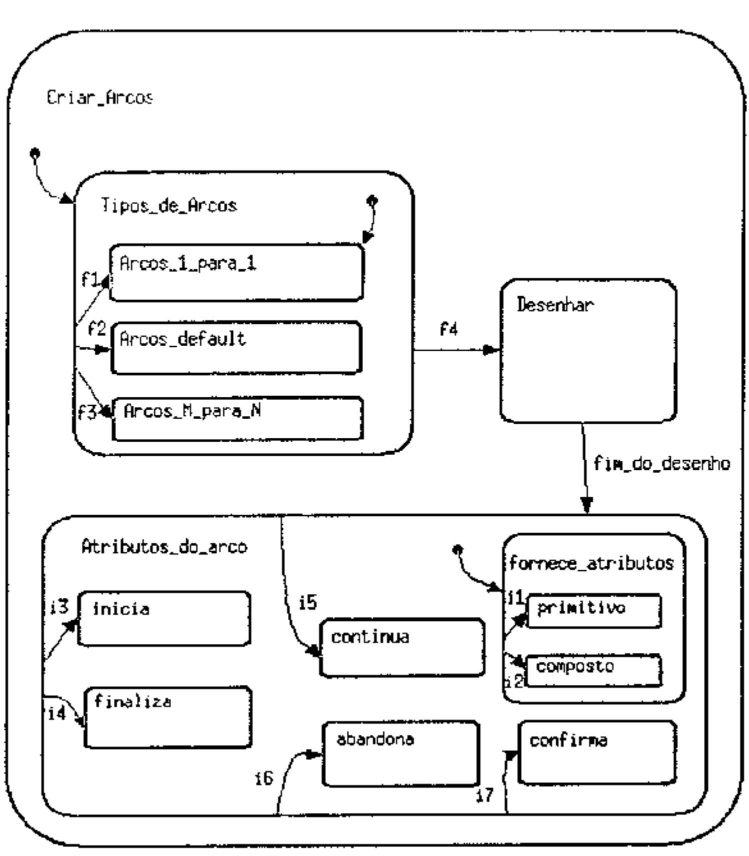

Figura 4.8: Criar Arcos

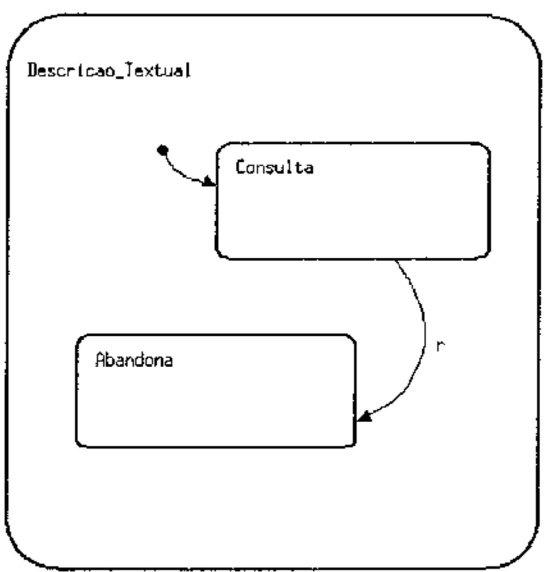

Figura 4.9: Descrição Textual

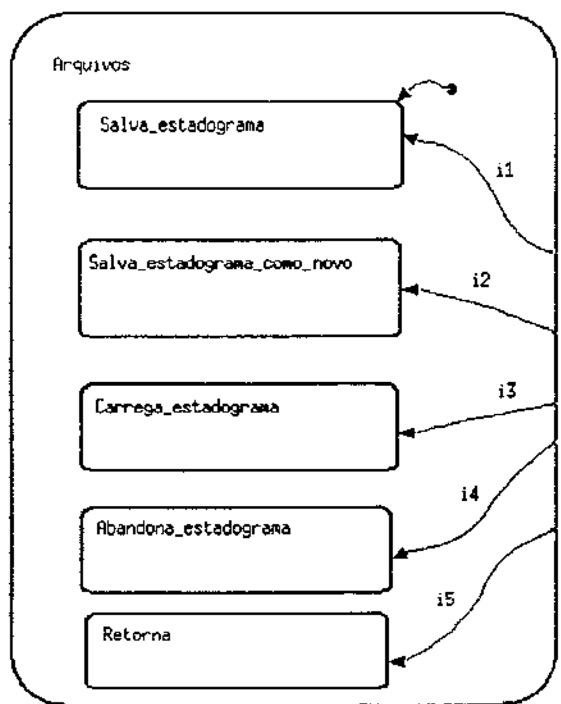

Figura 4.10: Arquivos

4.3.2 Simulador de Estadogramas

Neste módulo apresenta-se uma animação de estadogramas simulando-se as mudanças de estados em resposta a ocorrência de eventos.

Em qualquer um dos modos de simulação, a animação do estadograma é feita destacando-se a(s) bolha(s) ativadas após cada disparo de evento. Para isso, associamos a cor de "background" às bolhas correntes, contrária das demais.

Esta simulação pode ser feita de duas maneiras diferentes:

- A primeira delas consiste numa janela com a lista de eventos possíveis que foram definidos para o estadograma a ser animado. A cada evento escolhido deve-se acionar a tecla de "disparo" para a efetivação da mudança de bolhas correntes.

- A segunda maneira é através da simulação paralela à execução do programa funcionalmente equivalente em C, que foi gerado pelo GGSR (ver seção 3.3.3). Nosso objetivo é oferecer ao usuário uma facilidade de depuração do programa em C no nível do estadograma que deu origem a este programa.

  A cada ocorrência de um evento, o programa "envia" ao simulador o identificador deste evento. A partir desta informação a animação da especificação é feita de forma semelhante à descrita anteriormente.

  Esta comunicação foi realizada através do mecanismo de *sockets* [NPG 90]. Na seção 4.7 demonstramos como foi feita esta comunicação.

A simulação dispõe também de uma mecanismo de "log" onde são registrados os eventos ocorridos e as mudanças de estados correspondentes.

Pode acontecer que a ocorrência de um evento não acarrete em nenhuma mudança de estados. Isto ocorre toda vez que a transição em questão tiver como origem um estado que no momento não esteja ativado.

Estadogramas do Simulador

4.4 Arquitetura do Sistema

A Figura 4.19 mostra os módulos componentes do **Ambiente de Edição e Simulação de Estadogramas**, suas comunicações entre si e os arquivos de manipulação de informações.

4.5 Estruturas de Dados Internas

A manipulação dos dados é feita internamente através das diversas estruturas criadas, que definem as bolhas, os arcos e suas interrelações. A especificação completa das estruturas encontra-se no Apêndice C (Os campos que possuem um asterisco no início, são os que nós modificamos).

Uma bolha possui nome, um ponteiro para sua bolha pai, um ponteiro para sua primeira bolha filha, um ponteiro para sua proxima bolha irmã, e um ponteiro para sua bolha irmã anterior. Além disso, possui também as informações gráficas da mesma, tipo da bolha, tipo e valor do atributo history, bem como as ações e atividades correspondentes.

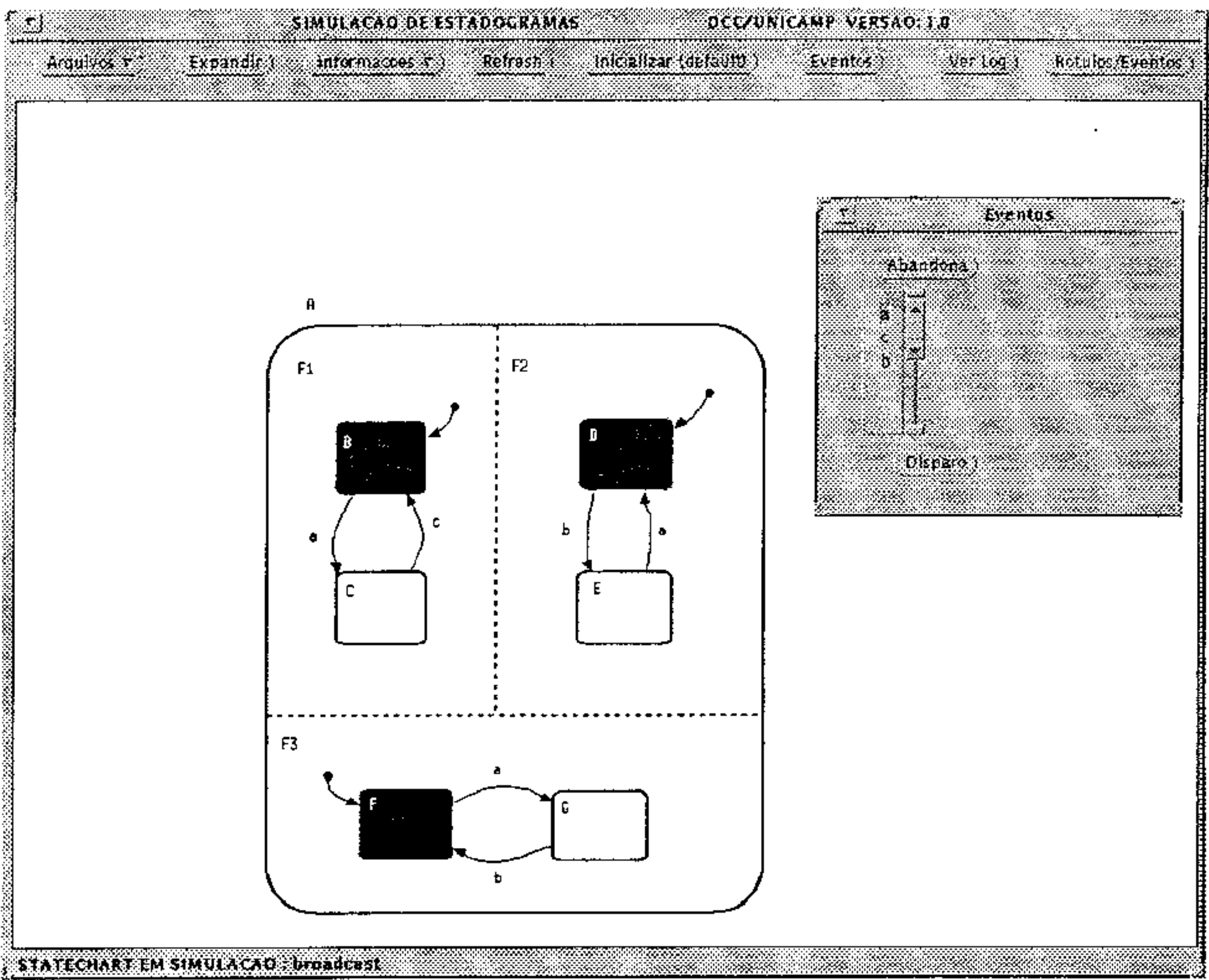

Figura 4.11: Interface do Simulador no momento do disparo de um evento

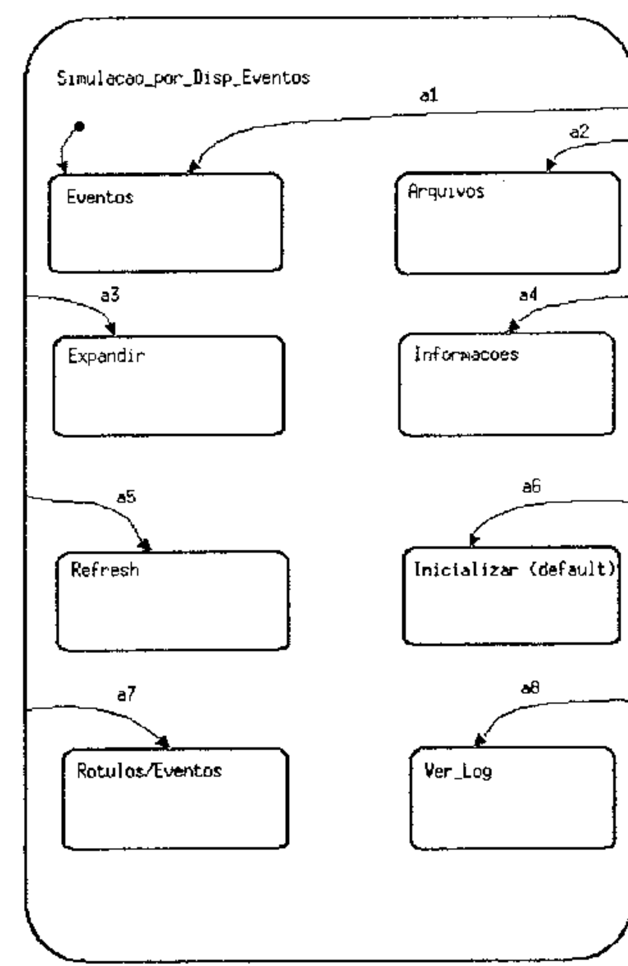

Figura 4.12: Simulação por Disparo de Eventos

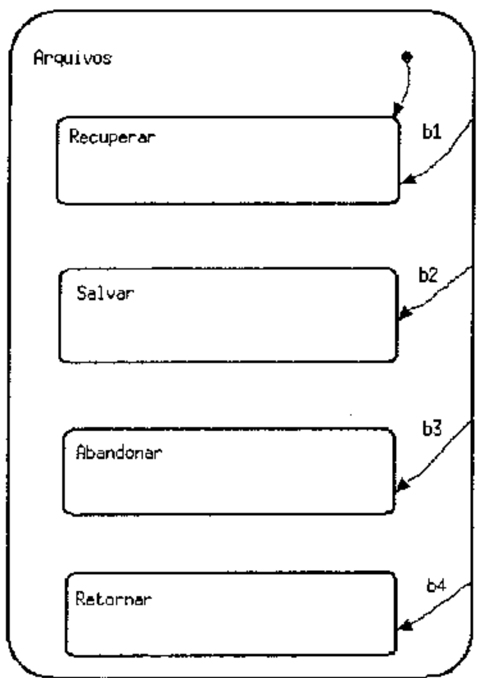

Figura 4.13: Manipulação de Arquivos

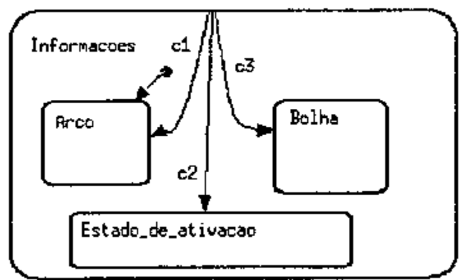

Figura 4.14: Informações

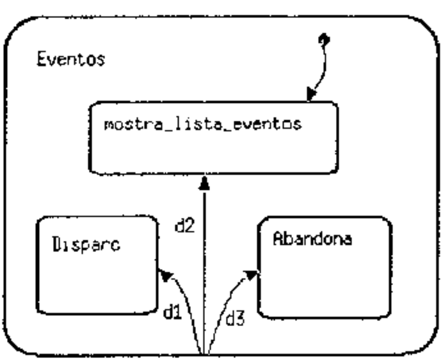

Figura 4.15: Janela de Eventos

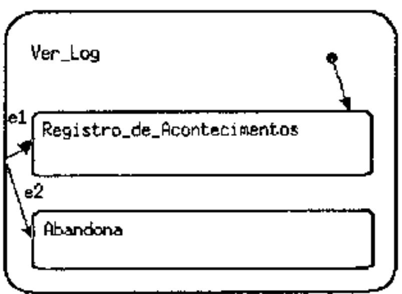

Figura 4.16: Logfile

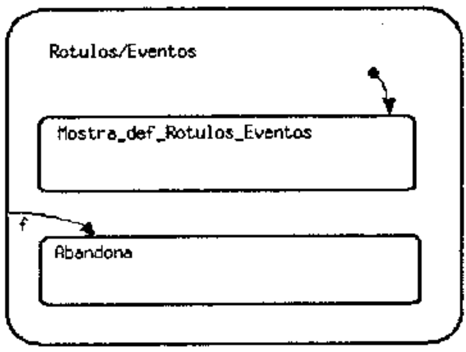

Figura 4.17: Descrição de Rótulos e Eventos

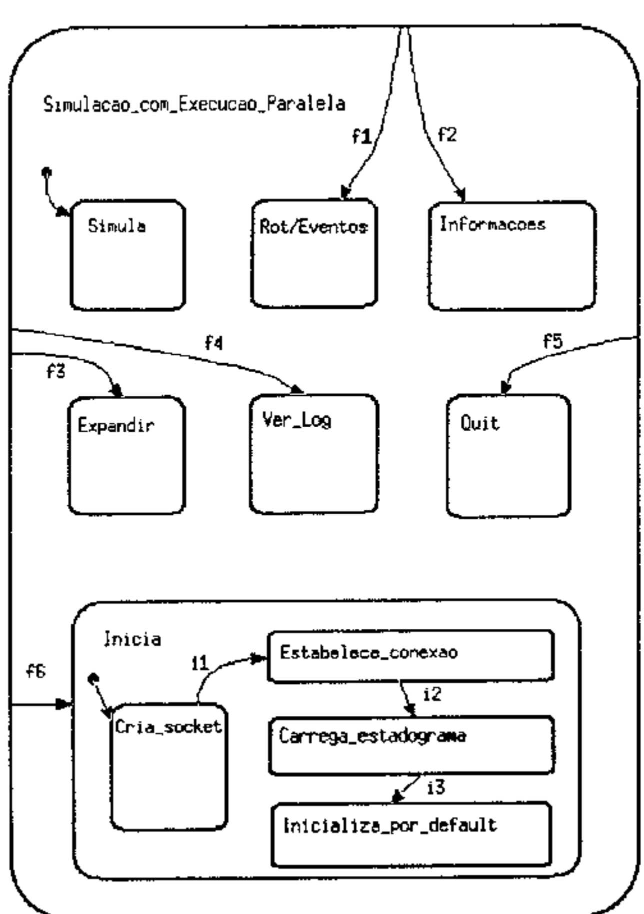

Figura 4.18: Simulação com Execução Paralela

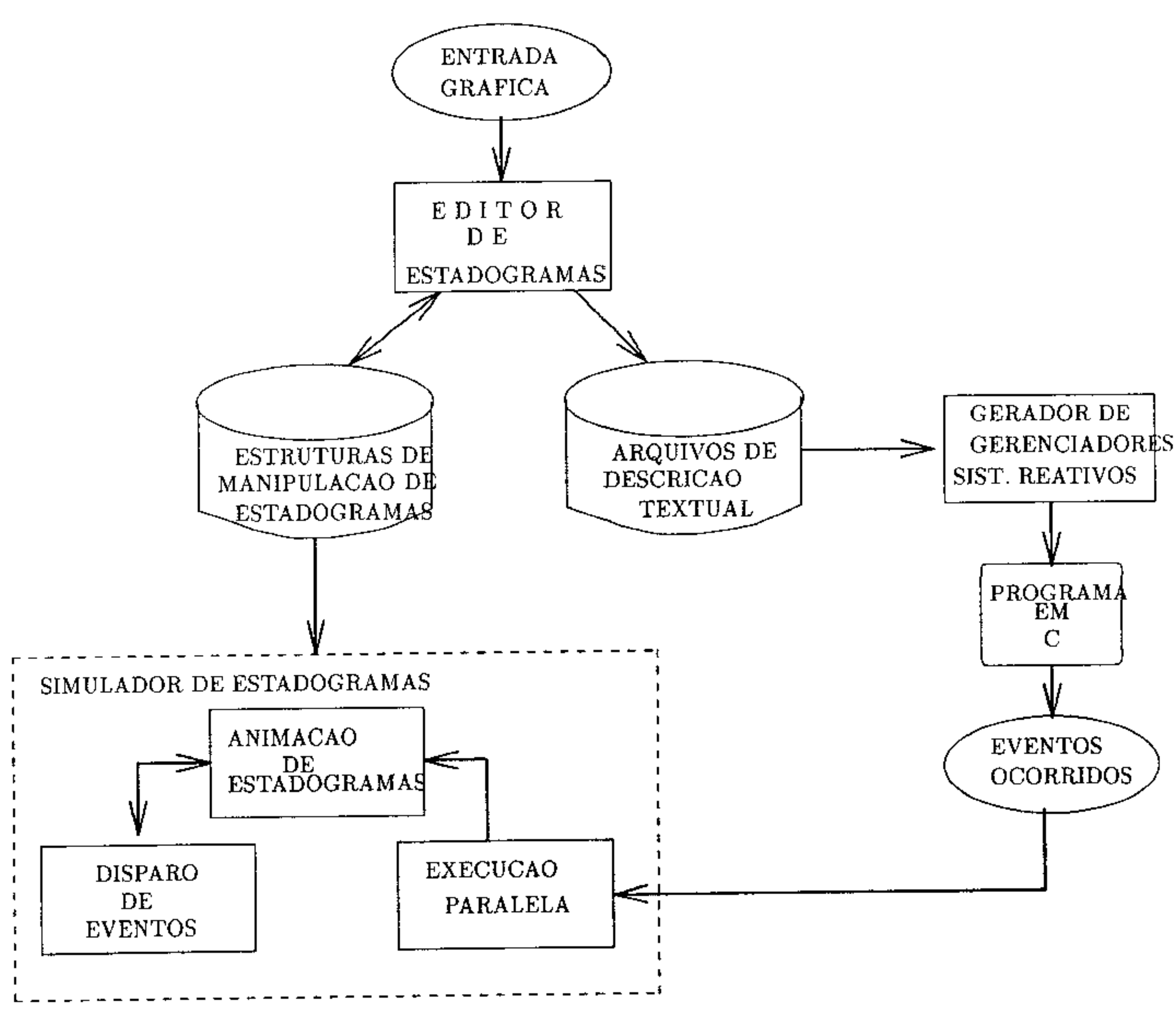

Figura 4.19: Arquitetura do Sistema

Um arco possui um rótulo,um eventoe evntualmente uma ação e uma condição; uma referência à bolha origem e outra à bolha destino, suas informações gráficas, o tipo e valor do atributo history bem como o tipo do arco. Definimos uma estrutura auxiliar que "guarda" as informações das expansões dos arcos. Então para cada arco expandido, temos a estrutura inicial e, em acréscimo, as informações das partes situadas em níveis de visualização diferentes.

Existem muitas outras estruturas internas, que facilitam o acesso das informações nas diversas rotinas.

4.6 Detalhes de Implementação

Procuramos manter sempre o mesmo estilo de programação original do EGS nas diversas funções novas que criamos, de modo a facilitar a compatibilidade destas com as funções já existentes, que utilizamos.

Alguns módulos foram criados, outros foram adaptados para atender às nossas necessidades, e outros foram modificados a fim de realizar novas funções.

As funções de Geração da Descrição Textual e da apresentação da mesma, durante a execução, em uma janela auxiliar, por exemplo, foram módulos novos criados independentes dos demais.

As funções de criação e manipulação de bolhas e arcos foram alteradas, para atender as mudanças feitas em suas estruturas. Foi criada também a função de alterar uma bolha, em módulo separado. A seguir mostramos algumas das alterações que fizemos nos diversos módulos.

4.6.1 Alterações realizadas no módulo de Edição

- Inicialmente tínhamos a definição de *history* relacionados aos arcos. Associamos esta definição de *history*, também às bolhas. Modificamos um pouco a definição feita pelos autores do EGS, que interpretaram a definição de *history* de maneira um pouco diferente, isto é, associaram o *history* aos arcos. Sendo assim, dependendo da transição através da qual se entrava em uma bolha, se entrava "por-history" ou não. Nós mantivemos os dois tipos de definição. No Apêndice A fazemos uma demonstração do que seria a execução dos dois tipos de *history*.

- Os atributos de uma bolha inicialmente se restringiam a seu nome e a seu tipo. Associamos às bolhas a especificação de uma ação *on_entry*, uma atividade *throughout*, e uma ação *on_exit*. Essa alteração foi necessária porque além de complementar os atributos da bolha, o GGSR utiliza estas definições para a devida tradução da descrição textual para o programa em C. Na versão original do editor, não existe definição de ações e atividades para uma bolha.

- Não estava disponível a alteração dos valores dos atributos de uma bolha. Acrescentamos ao editor a facilidade de alterar os atributos de uma bolha, depois da mesma já ter sido definida. A alteração só não é permitida no atributo tipo.

- Restringimos a definição de condição associada a um arco para ser apenas do tipo "in(Blob)". Isto ocorreu devido ao GGSR possuir apenas esta definição de condição. Na versão original do Editor, existem definições de outros tipos de condições.

- Os tipo de *history* associados aos arcos inicialmente eram dos tipos simples e de profundidade. Acrescentamos a definição de *history* de retorno aos arcos. O *history* de retorno foi implementado também devido ao GGSR utilizar esta definição (ver seção 3.1.3).

- Um arco, inicialmente só podia ser definido em um mesmo nível de visualização, ou seja, não era permitido a um arco entrar em uma bolha expandida. Agora na versão atual, um arco pode ser expandido em vários níveis de visualização. Esta facilidade muitas vezes se faz necessária quando se deseja ligar duas bolhas que estejam em níveis de visualização diferentes.

- Acrescentamos um novo tipo de arco que tem como origem uma bolha não atômica, e como destino um seu descendente direto ou indireto. Este tipo de arco é necessário no caso de querermos que a ocorrência da transição não dependa da ativação de nenhum descendente específico desta bolha, no momento da ocorrência do evento.

- Criamos a opção de "iniciar" a continuação de um arco expandido, partindo do meio de uma bolha, desde que ela seja um componente ortogonal expandido, como foi apresentado na seção 4.3.1, na especificação de arcos. Esta função anteriormente não existia no Editor original.

- Permitimos "finalizar" o desenho de um arco expandido no meio de uma bolha, desde que ela seja um componente ortogonal expandido, como foi apresentado na seção 4.3.1, na especificação de arcos. Originalmente existia a opção de finalizar um arco, e nós estendemos esta função para permitir a expansão do mesmo.

- Acrescentamos a opção de "continuar" o desenho de um arco expandido. Quando finalizamos o desenho de um arco em um determinado nível visual, precisamos escolher a opção "Continua" para podermos desenhar a continuação do mesmo em outro nível de visualização, e assim sucessivamente. Esta função foi criada devido a definição de arcos expandidos.

4.6.2 Alterações realizadas no módulo da Simulação

- Assim como permitimos a definição de um novo tipo de *history* chamado de *History* de Retorno, implementamos também a sua semântica durante a simulação. Isto é, toda vez que disparamos o evento relacionado ao *history* de retorno, o sistema vai fazendo exatamante o percurso de ida, em sentido contrário, até chegar na primeira bolha visitada.

- Realizamos a mudança de estados a partir de um arco expandido. Não só criamos a noção de expansão de um arco, como também o executamos na simulação.

- Realizamos o mecanismo de "broadcast" executando eventos paralelos em componentes ortogonais. Ou seja, como falamos na seção 4.3.1, quando executamos um evento que tem uma ação relacionada a ser disparada ao final da transição, nós realizamos o disparo do evento especificado na definição da ação em questão.

- Realizamos a mudança de estados de pai para filho. Tentando especificar melhor, nós sabemos que para que uma transição ocorra, é necessário que a bolha origem desta esteja ativada. Se o arco parte da bolha pai, para uma bolha filha, isto quer dizer que já estamos dentro desta bolha pai, do tipo OR não atômica, como também em uma de suas bolhas filhas. O que nós

desejamos é que a transição dependa apenas da bolha pai, independente da bolha filha que estiver ativada no momento. O que acontece então é apenas o "desligamento" do filho atual, e o "ligamento" do filho que se deseja ir, ficando o pai na mesma situação, ou seja "ligado".

- Fazemos a simulação também por *sockets*. Conforme relatado na seção 4.3.2.

4.7 Como foi feita a comunicação por *Sockets*

A interface com o usuário da Simulação com Execução Paralela, é um pouco diferente da Simulação por Disparo de Eventos. Não é por meio do simulador que o usuário escolhe qual o estadograma que deseja simular. A simulação vai se realizar com o estadograma que o GGSR requisitar. Na tela inicial de simulação, apresentamos a opção de Iniciar a simulação. Esta opção é a encarregada de criar o *socket*. De acordo com o modelo Cliente/Servidor, no momento da escolha desta opção, o simulador de estadogramas, torna-se o servidor da comunicação. Depois disto o servidor requisita a função de **accept()**, ou seja, fica bloqueado esperando a conexão desejada.

Para evitar que outros processos se conectem ao *socket* criado, que não o **cliente**, que no nosso caso, será o programa em C gerado pelo GGSR, criamos um mecanismo de "senhas". Desta forma, quando um processo tenta estabelecer uma conexão com o **servidor**, ele deverá mandar uma mensagem afirmando que ele é o **cliente** esperado. Se o **servidor** não reconhecer a mensagem, a conexão é rejeitada. Da mesma forma, quando o servidor recebe o pedido de conexão e a aceita, então envia uma mensagem confirmando que a conexão foi estabelecida.

O cliente, no nosso caso, não precisa se preocupar quanto a criação do *socket*. Basta chamar uma única função já por nós codificada, chamada de **MandaEvento()**, que se encarrega de realizar todo o processo de criação do *socket*. Esta função cria o *socket* quando é invocada pela primeira vez. Nas chamadas seguintes, já estando o *socket* criado, a função apenas manda a mensagem ao servidor.

O cliente, ou seja, o programa em C, deverá, na primeira chamada à função, mandar o nome do estadograma que deseja simular. Nas chamadas seguintes o cliente deve mandar os identificadores dos eventos ocorridos.

Por outro lado, no momento que o servidor receber o nome do estadograma a ser simulado, ele carregará o mesmo na memória, e logo o inicializará por *default*. A partir daí fica esperando os eventos ocorridos, para efetivar as mudanças de estados.

A simulação acaba no momento que em que o **cliente** envia um evento especial denominado "fim". Neste momento então os dois *sockets* são fechados. Mas, o usuário, se quiser, ainda pode ter acesso às informações da simulação ocorrida através do mecanismo de "log", ou até mesmo, se desejar, pode iniciar uma nova simulação.

Esta comunicação se dá de forma muito rápida, sendo até possível que não se consiga acompanhar todas as mudanças de estados, passo-a-passo.

4.7.1 O Modelo Cliente/Servidor

O paradigma mais comumente utilizado na construção de aplicações distribuídas é o modelo Cliente/Servidor. Neste esquema, as aplicações Clientes requerem serviços do processo Servidor.

O modelo Cliente/Servidor requer um conjunto de convenções pré-determinadas antes que os serviços possam ser prestados. Este conjunto de convenções consiste em um protocolo que devem ser implementados em ambos os lados da conexão.

Um processo servidor normalmente possui um endereço conhecido e fixo para o requerimento de serviços. Um servidor fica bloqueado até uma conexão ser requerida por um processo, para o endereço do servidor. No momento do pedido de conexão pelo processo, o servidor "acorda" e a conexão é estabelecida. A partir daí chamamos o processo de cliente. O servidor então executa todos os serviços que o cliente requisitar.

A comunicação continua até que o Cliente encerre a comunicação. Desta forma, o servidor pode ficar esperando por outra solicitação de conexão.

Capítulo 5

Um Exemplo de uma Aplicação de nosso Editor

5.1 Objetivos

Nosso objetivo é fazer uma aplicação prática e funcional de todo o "Ambiente de Edição e Simulação de Estadogramas". Para isso ser feito, é necessário que se descreva todo um sistema por meio de estadogramas, inclusive que se defina as funções de cada estado, colocando suas chamadas nos locais adequados. A partir da conclusão da edição o Gerenciador de Gerenciadores de Sistemas Reativos lê a descrição textual resultante da edição e gera o programa em C funcionalmente equivalente ao estadograma editado. E por fim, paralelamente a execução do programa gerado, pode-se simular o estadograma alimentando-se o simulador com eventos ocorridos durante a execução do programa gerado.

O Sistema escolhido foi o **Editor Topológico para a Linguagem de Especificação de Computações - LegoShell** [Piñ 90], pois para esse sistema já foi realizada a tradução da descrição textual do estadograma do mesmo para um programa em C, pelo GGSR. O que restava-nos fazer era editar os estadogramas, e conferir a descrição textual gerada. E no final fazer a execução paralela do programa com o simulador.

5.2 O Editor Topológico para a LegoShell

A *LegoShell* é uma linguagem gráfica de especificação de comandos complexos chamados computações. Ela possibilita a composição de programas através da conexão de suas portas de entrada e saída. Ela estende o conceito de "pipe" de UNIX, restrita a uma dimensão só, para grafos multidimensionais compostos de programas, dispositivos periféricos, arquivos, conectores, etc. Estes grafos, chamados de computações, podem ser abstraídos como programas e depois utilizados nas computações indistintamente de programas reais. Qualquer dos componentes de uma computação pode residir ou ser executado em qualquer uma das máquinas da rede local, possibilitando assim, a especificação de computações distribuídas.

Por sua natureza, a LegoShell é inerentemente uma linguagem de estrutura topológica. Um editor apropriado deve ser capaz de manipular estruturas topológicas e representá-las de forma gráfica. A LegoShell é uma das linguagens do "Ambiente de desenvolvimento de software baseado

em Hierarquias de Abstração em Níveis Diferenciados" (A_HAND), atualmente sendo desenvolvido no Departamento de Ciência da Computação da Unicamp.

A linguagem *LegoShell* constitui-se dos seguintes objetos:

- Programas Cm
- Arquivos
- Dispositivos Periféricos
- Conectores: "pipe","broadcast" e "mailbox"

A construção de computações consiste em interligar as portas dos objetos utilizando os conectores. Os programas Cm e conectores especificam as suas próprias portas e os arquivos e dispositivos periféricos têm portas de entrada e saída padrão.

A *LegoShell* é, então, uma linguagem gráfica que implementa processos que se comunicam, segundo o paradigma de "streams", generalizando o conceito de "pipe" de UNIX através de ligações multidimensionais (conectores). Temos como exemplo a computação MSort, constituída de dois componentes `Sort` e `Merge`. Note que esta computação pode ser abstraída, tornando-se componente e peça básica para outras computações. Dessa forma podemos construir computações sem restrições quanto ao número de níveis de abstrações.

Podemos desprezar os dados associados a uma porta, conectando-a a um "buraco negro". Seu propósito é fechar uma porta cujo acesso deva ser negligenciado, como na saída de dados duplicados do programa `Sort`.

Especificaremos sucintamente os vários objetos da *LegoShell*:

- Programas Cm: A linguagem Cm (C modular e polimórfica) foi projetada para ser a linguagem básica do ambiente A_HAND(em [Fur 90] apresenta-se uma descrição desta linguagem).

- Arquivos: Um arquivo ou um depósito de dados possui portas de entrada e saída de acordo com a permissão de leitura e escrita que o usuário que o utiliza possui. Estas portas são virtuais e só passam a existir quando são ligadas a um conector.

- Dispositivos Periféricos: Os dispositivos periféricos podem ser vistos como objetos produtores ou consumidores de dados e detentores de portas de entrada e saída pré-definidas de acordo com o seu tipo.

- Conectores: Os conectores são objetos concentrados e distribuidores de dados. Cada conector será associado a um buffer no qual armazena os dados que consumiu e ainda não distribuiu. Os conectores utilizados na LegoShell são: pipe, broadcast e mailbox.

Os objetos da LegoShell podem ser agrupados segundo três classificações:

- Composto/Primitivo: Se um objeto é construído a partir de outros objetos ou envolve outros em sua definição, ele é dito composto. Por outro lado, objetos primitivos são imutáveis, não possuem dependências com outros objetos e têm um mapeamento direto para objetos físicos. Exemplo de objetos compostos: Computações e programas Cm. Exemplos de objetos primitivos: periféricos, arquivos, pipe, mailbox e broadcast.

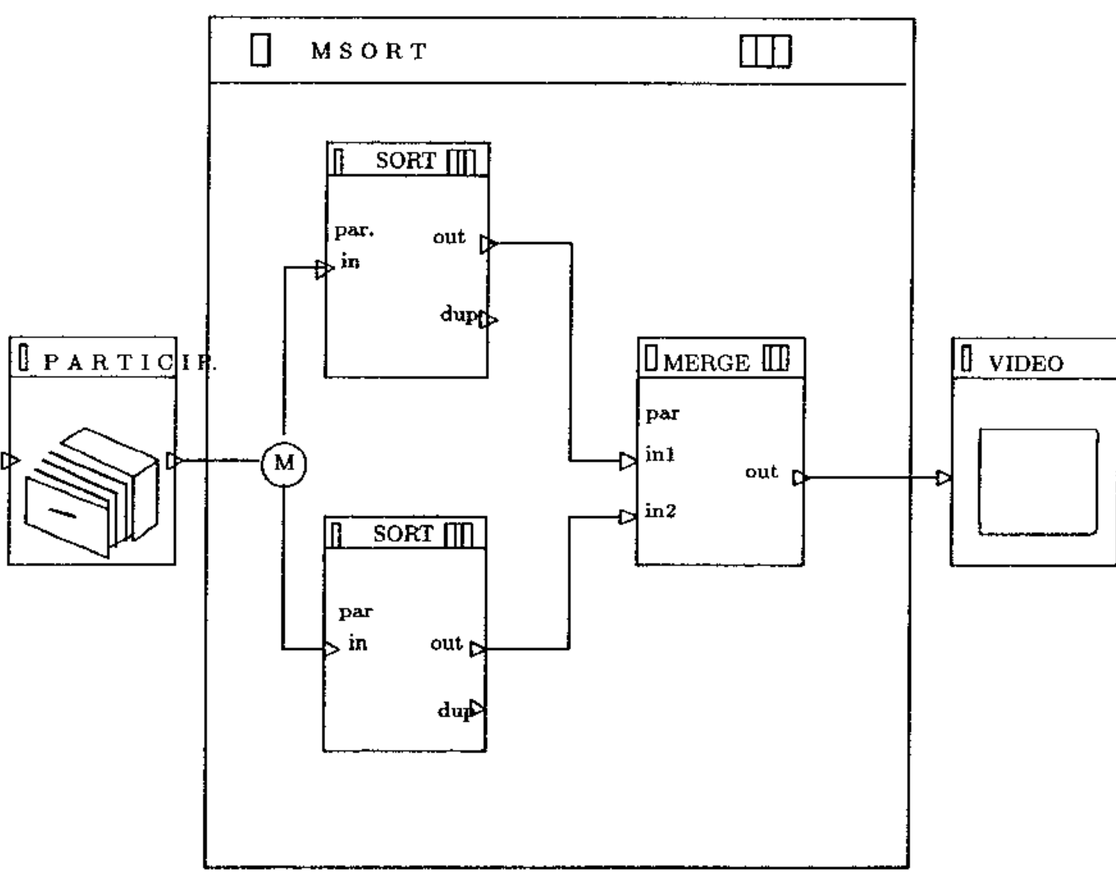

Figura 5.1: Exemplo de Abstração de Computação

- Ativo/Passivo: Objetos ativos são geradores e/ou consumidores de dados, enquanto que objetos passivos são fornecedores e/ou receptores e trabalham sob demanda. Exemplos de objetos ativos: computações, programas Cm e periféricos. Objetos passivos são: arquivos, pipe, mailbox e broadcast.
- Objetos com conteúdo semântico/conectores: Esta classificação distingue objetos que possuem conteúdo semântico por si mesmos daqueles que só têm significado dentro de uma computação da LegoShell, interligando os demais objetos. Exemplos de objetos com conteúdo semântico: computações, programas Cm, periféricos, arquivos. Exemplos de conectores: pipe, mailbox e broadcast.

A tela inicial do Editor de LegoShell, consiste de cinco áreas básicas:

- Cabeçalho: Composto por uma pilha contendo o nome dos objetos sendo editados, além do nível de edição do momento.
- Menu de Barras: Contém as principais funções do Editor.
- Área de Trabalho: Nela será desenvolvido o trabalho de edição.
- Menu de Ícones: Nesta área estão dispostos os ícones dos objetos do ambiente que podem ser selecionados e incluídos na área de trabalho durante uma edição.
- Área de Visualização: Corresponde a uma janela de visualização total do objeto, contendo uma janela de *zoom*, que mostra a parte do objeto sendo representada na área de trabalho.

5.3 Descrevendo o editor através de estadogramas

Ao se executar o Editor, imediatamente entramos no estado Editor, que por sua vez possui três componentes ortogonais, como se vê na Figura 5.2. Na área de trabalho, imediatamente entramos no estado de criação CREATE.

- CREATE

  Criar uma computação consiste conceitualmente em definir quais são os seus componentes e como eles estão conectados. No processo de criação, poderemos ter ações de inclusão de novos componentes, edição da configuração do objeto, edição dos parâmetros do objeto e dos componentes e o armazenamento do objeto criado.

  Podemos especificar mais formalmente uma sequência de ações do CREATE como sendo:

  1. início
  2. INCLUDE
  3. MOVE
  4. EDIT Componente
  5. EDIT Configuração
  6. EDIT Parâmetros Componente

7. EDIT Parâmetros Objeto
8. Atribuição de NOME
9. STORE
10. fim

- Armazenamento

  Há três funções relacionadas com o armazenamento dos objetos do ambiente: STORE, QUIT e EXIT.

  Se o usuário terminar de criar ou editar um objeto, ele pode querer armazená-lo utilizando para tal a função STORE. Se ele quiser armazenar o objeto e também sair do nível de edição para um nível mais alto ou para fora do ambiente, então usará a função EXIT. Se ele quiser abortar a edição, abandonando este nível para um nível superior, usará a função QUIT.

  A função INCLUDE permite a inclusão de componentes em uma computação correspondente à escolha do tipo de menu de ícones (computação, programa, periférico, arquivo ou conector), a sua identificação, no caso de objetos de conteúdo semântico, e finalmente, o seu posterior posicionamento dentro da computação.

- Alteração

  - MOVE: Possibilita que a posição de objetos dentro de uma computação possa ser alterada.
  - EDIT: Alteração dos componentes.
  - LOAD: Corresponde à edição de um objeto, caso ele já exista, ou criação do mesmo, caso contrário.
  - SELECT: Há casos em que se deseja realizar a mesma operação, por exemplo, exclusão, sobre vários objetos.
  - EXCLUDE: Exclusão dos componentes de uma computação.
  - CLEAR: Como o nome já indica, consiste em excluir todos os objetos componentes da computação.
  - REDRAW: Genericamente significa voltar a desenhar todos os objetos que precisam ser representados na área de trabalho.
  - UNDO: Volta ao desenho anterior, abandonado o estado atual e eliminado ou revertendo seus efeitos.
  - ABSTRACT: Esta função consiste em criar uma nova computação composta pelos objetos selecionados através da função SELECT, substituindo-os pela mesma na computação corrente.
  - EXPAND: É a função oposta da abstração e consite em substituir uma computação por seus componentes.
  - CONSIST: A verificação da consistência através desta função é feita somente no nível de edição corrente e não inclui a geração de código.

5.4 Estadogramas Principais do Editor da LegoShell

A seguir temos os estadogramas principais do Editor, onde descrevemos seus principais componentes.

A Figura 5.3 mostra a área de cabeçalho com a função de nomear os objetos que irão sendo editados. A transição rotulada com a1, consiste no evento denominado nome_ok e assim por diante, ou seja, o que apresentamos nas figuras são apenas os rótulos das transições e os nomes dos eventos propriamente ditos são manipulações internas do sistema.

Nós realizamos todo o percurso especificado no início deste Capítulo e o resultado obtido foi equivalente a simulação por Disparo de Eventos, ou seja, conseguimos atingir o objetivo proposto.

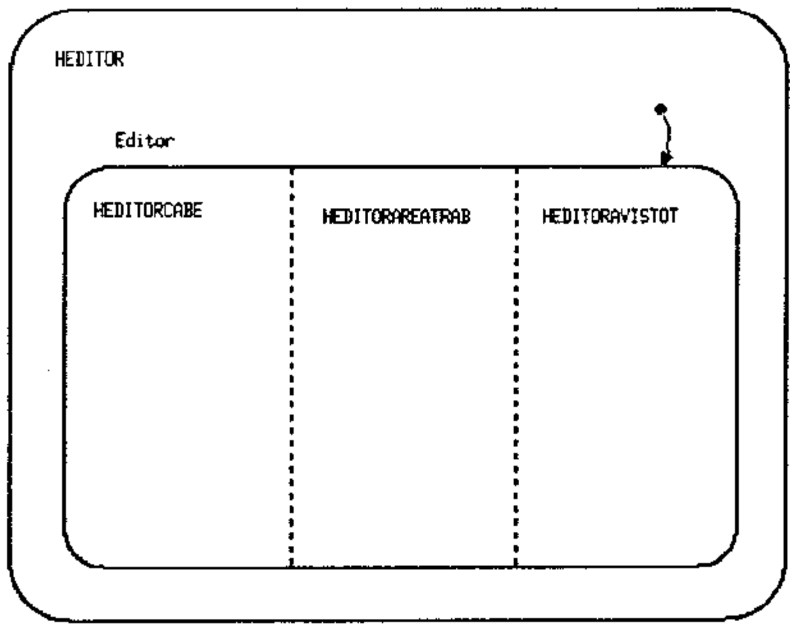

Figura 5.2: Tela Principal do Editor

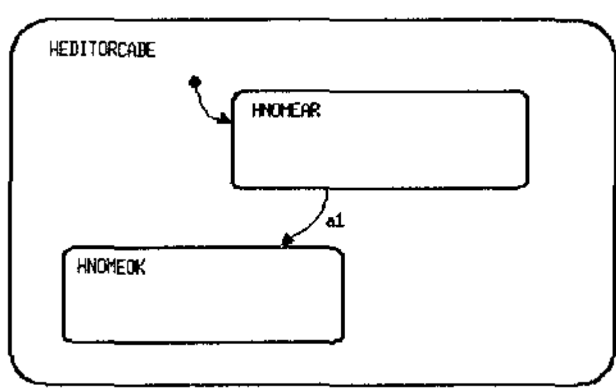

Figura 5.3: Janela de Cabeçalho

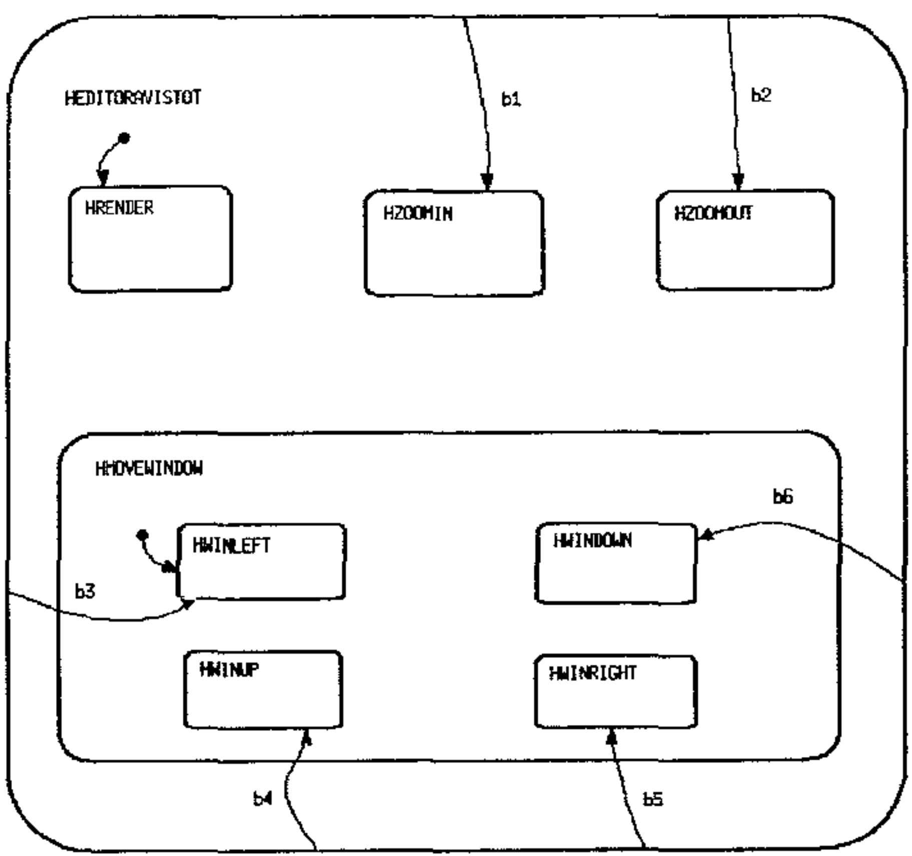

Figura 5.4: Janela de Visualização

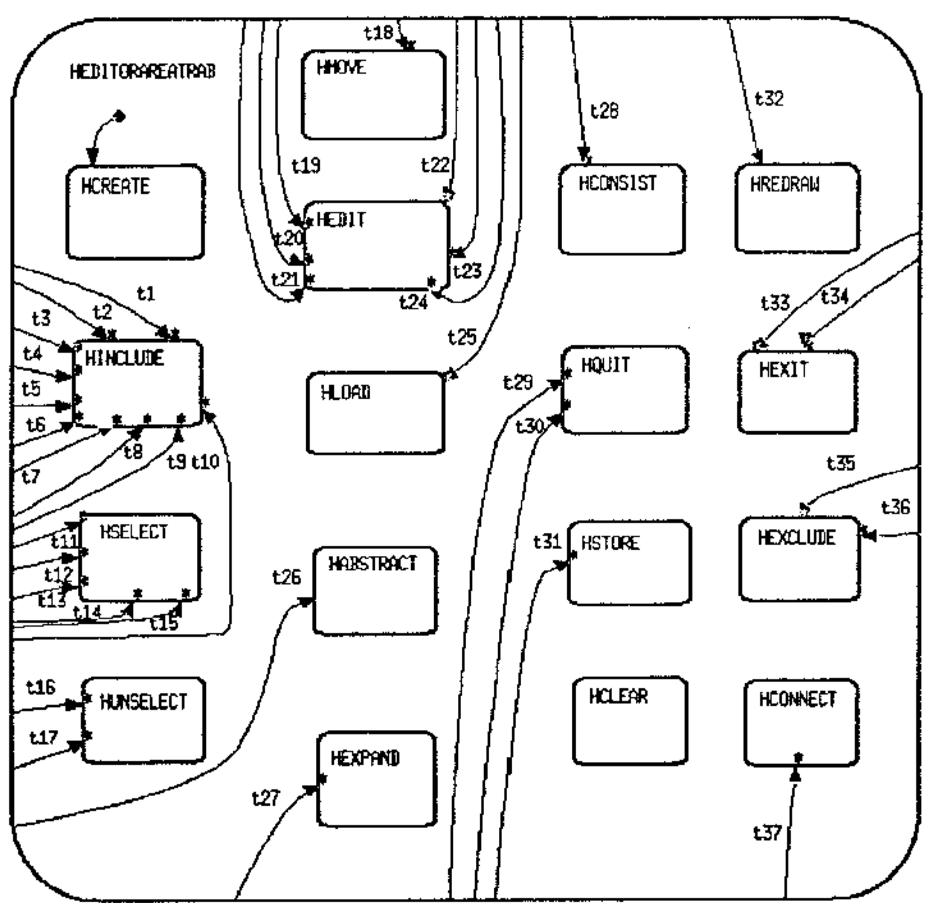

Figura 5.5: Área de Trabalho

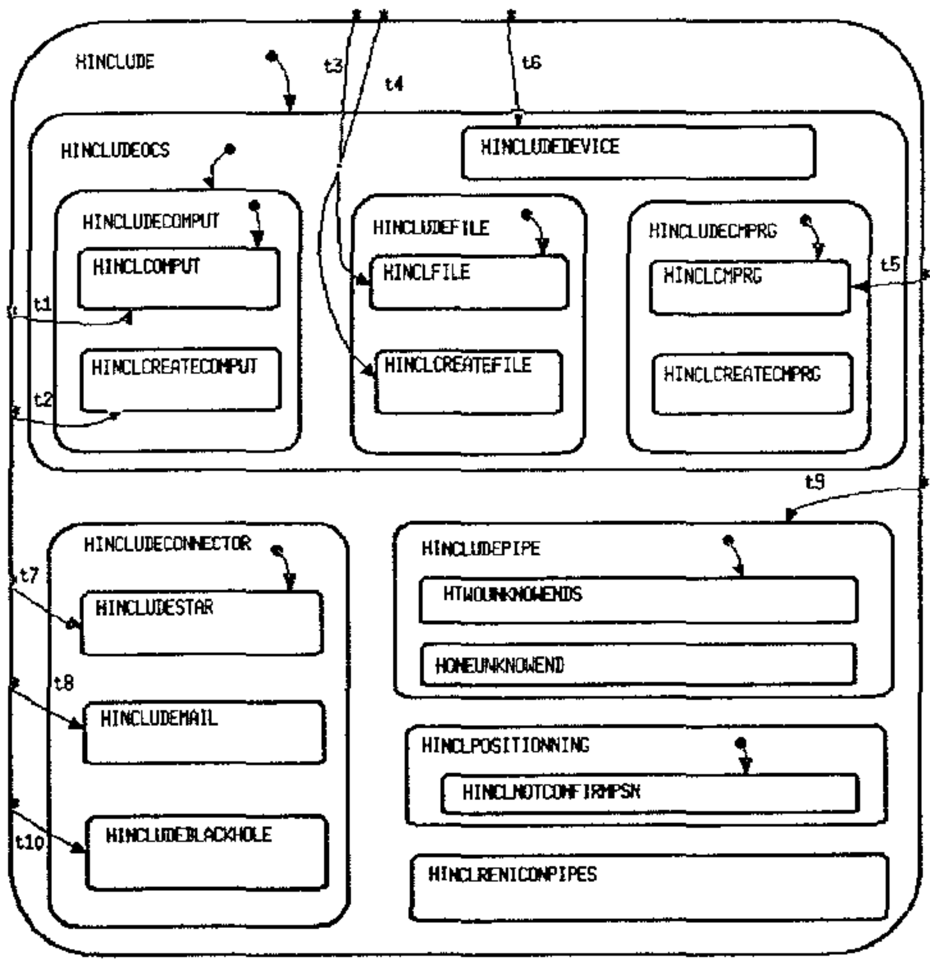

Figura 5.6: Inclusão de Objetos

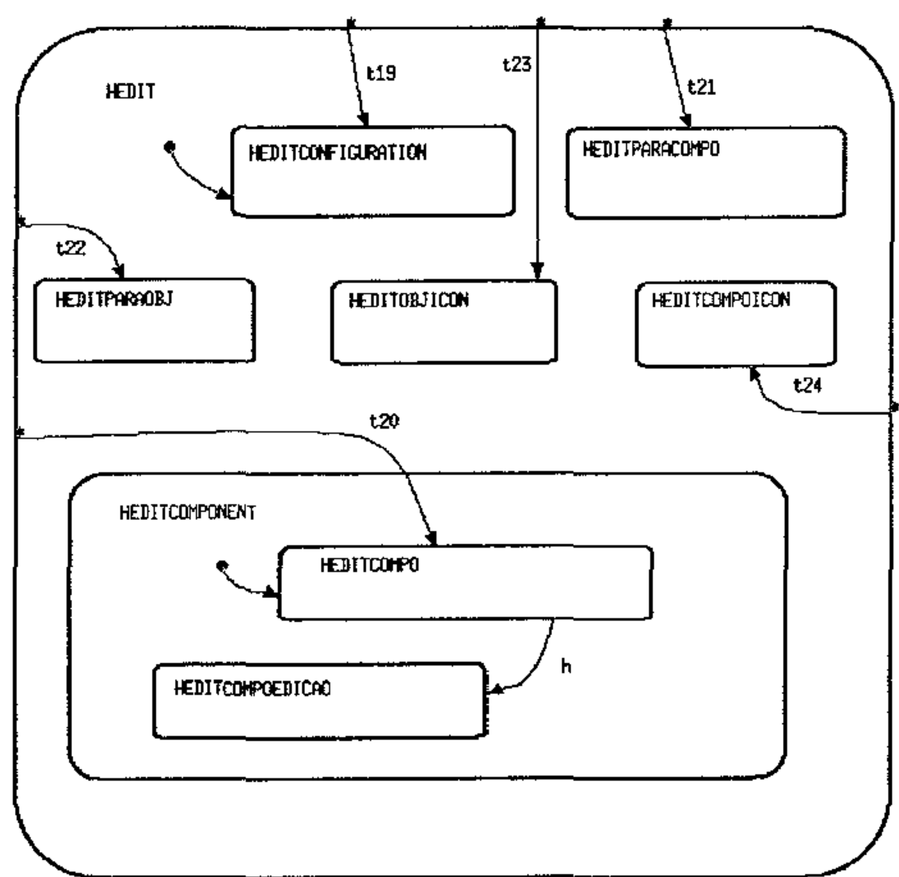

Figura 5.7: Edição

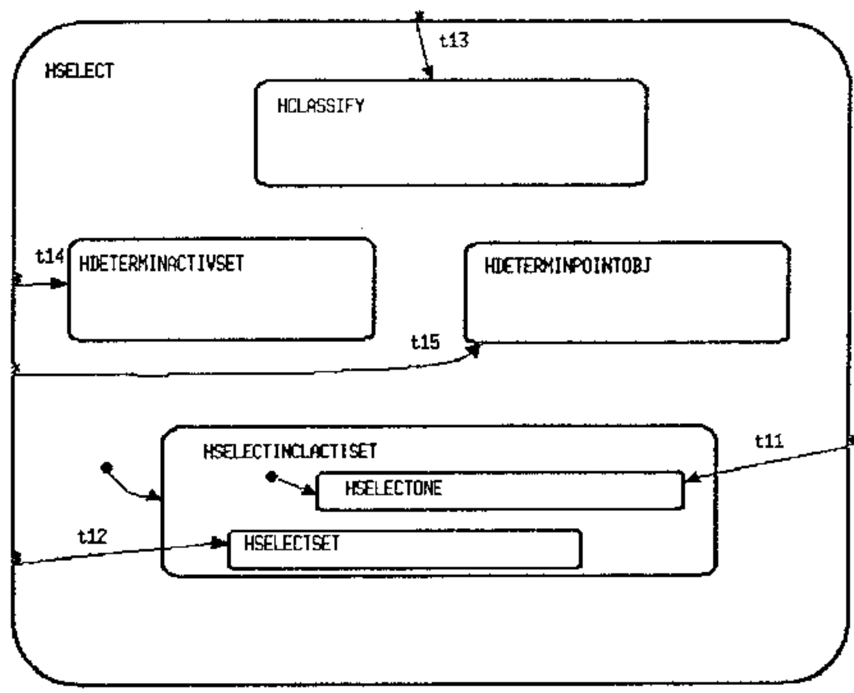

Figura 5.8: Seleção de Objetos

Capítulo 6

Conclusões

Durante a especificação e desenvolvimento da ferramenta, objeto deste trabalho, nós estivemos sempre analisando nossa proposta inicial diante da real situação disponível, tendo em vista a adaptação de um editor já existente, com funções e estruturas já definidas. Muitas vezes foi necessário modificar objetivos anteriormente propostos para adaptá-los a nova situação encontrada.

Por outro lado, acrescentamos à nossa ferramenta o aplicativo de simulação de estadogramas. Isto é, depois de alterar muitas funções do editor, acrescentando ao mesmo características adicionais às existentes, adaptamos também o software de simulação. Ou seja, todas as alterações realizadas no módulo de edição estão compatíveis com as funções do simulador, de modo que as informações manipuladas nos dois módulos estão coerentes. Além disso, conseguimos unir o sistema GGSR com o Simulador de Estadogramas visando oferecer uma facilidade adicional de depuração em nível de estadograma. No momento da execução do programa o projetista pode verificar, através das mudanças de estados ocorridas no estadograma, como o seu programa está se comportando e, caso ocorra uma parada anormal do programa, saber em que estado isto ocorreu.

Por fim, podemos afirmar que os resultados obtidos foram bastante satisfatórios e que a funcionalidade e praticidade da ferramenta dá condições ao usuário de desenvolver sistemas reativos de maneira mais simples, com acesso inclusive a visualização gráfica da execução final do seu programa.

Podemos ver então que nossa ferramenta se aplica muito bem em projetos de sistemas reativos. O projeto de um sistema deste tipo pode ser resumindo da forma descrita a seguir. Depois de definido o estadograma de um sistema, pode-se primeiro simulá-lo por meio do disparo de eventos, até conseguir-se alcançar digamos que o "estadograma ótimo". A partir daí, aciona-se o GGSR que lê a descrição textual gerada, transformando-a num programa em C. Depois, pode-se animar outra vez o estadograma, só que desta vez, paralelamente à execução do programa em C gerado.

6.1 Contribuições

- Permitimos a espécifição detalhada de sistemas reativos através da técnica de estadogramas.
- Permitimos um mecanismo de validação de estadogramas, através da simulação dos mesmos.

- Criamos um maneira diferente de depuração de programas a nível de estadogramas.

Este trabalho consistiu de uma união de diferentes sistemas em um único produto final. Inicialmente alterou-se o "Editor Gráfico de Statecharts" desenvolvido na USP de São Carlos [Bat 91], de modo que pudesse ser realizado uma comunicação deste sistema com o "Gerador de Gerenciadores de Sistemas Reativos" [Fig 91]. Para isso, foi necessário que todos esses sistemas manipulassem os mesmos tipo de informações. Esta tarefa não foi muito fácil de ser realizada devido aos sistemas descritos terem sidos feitos independentes e por autores diferentes.

Nosso objetivo era unir as características importantes e necessárias ao nosso projeto, de cada um desses sistemas, e torná-las coerentes entre si. Para isso, acrescentamos novas funções aos sistemas de Edição e Simulação de Estadogramas, de modo a tornar possível a geração de uma descrição textual dos estadogramas editados, na linguagem LEG (ver Apêndice B). Esta descrição serviria, então, como entrada para o sistema Gerador de Gerenciadores de Sistemas Reativos (GGSR). Ao final o GGSR traduziria esta descrição para um programa em C funcionalmente equivalente ao estadograma editado. Modificamos também a geração do programa em C de modo a permitir que sua execução posterior pudesse se realizar paralelamente ao simulador de estadogramas. Esta execução paralela foi possível comunicando os dois sistemas com o mecanismo de *sockets*, onde o programa em C gerado envia ao simulador os identificadores dos eventos ocorridos e o simulador executa as mudanças de estados decorrentes. Esta é uma maneira diferente de depuração de programas através da visualização de sua execução por meio da animação do estadograma equivalente.

6.2 Trabalhos Futuros

Como trabalhos futuros, pretendemos aumentar as funções disponíveis durante a edição dos estadogramas, tais como:

- Copiar objetos
- Mover objetos
- Melhorar a técnica de "zoom"

Durante a simulação, muitas vezes o usuário necessita utilizar o mecanismo de "log" para se informar em que estado(s) o sistema se encontra em um determinado momento. O que nós pretendemos fazer é o próprio sistema mostrar as bolhas que estão "ligadas" a cada mudança de estados, de modo a facilitar o entendimento e visualização da animação.

Uma outra alteração bastante interessante é na simulação por execução paralela. Como falamos anteriormente, na Seção 4.3.2, essa simulação às vezes acontece tão rapidamente que se torna difícil o acompanhamento de todas as mudanças de estados. Nossa idéia é registrar em um arquivo a seqüência de eventos ocorridos e deixar disponível uma opção de reiníciar a simulação a partir de um determinado evento ocorrido, e a partir disto, acompanhar, passo a passo, as mudanças de estados deixando à cargo do usuário o disparo dos eventos.

Bibliografia

[ArA 91] Araújo, V. P. A. e Almeida, A. A. M.; *Ferramenta CAD para projetos de sistemas de irrigação (IRRIGRAFOS)*; Fundação Parque Tecnológico da Paraíba, IV Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI, 1991.

[Bat 91] Batista Neto, João do E.S.; *Um Editor Gráfico Para Statecharts*; USP - São Carlos; Abril 1991.

[Bea 88] Beaudouin - Lafon, M.; *User Interface Support for the Integration of Software Tools: an Iconic Model of Interaction*; ACM,, 1988.

[Ber 91] Berthomieu,B. and Diag,M.; *Modelling and Verification of Time Dependent Systems Using Time Petri Nets*; IEEE Transactions on Sotware Engineering, Vol. 17, No. 3, March 1991

[Boc 87] Bochmann, G.V. and Verjus, J.P.; *Some Comments on "Transition-Oriented" Versus "Structured" Specification of Distributed Algorithms and Protocols*; IEEE Transactions on Software Engineering, Vol. SE-13, No. 4, April 1987, Correspondence.

[Bor 85] Borges, J.A.S.;*Editores Gráficos para Projeto de Circuitos Integrados*; V Congresso da Sociedade Brasileira de Computação; Porto Alegre; 1985.

[Bor 87] Borges, J. S. e Schmitz, E. A.; *TEDMOS - Um Sistema de CAD para ensino de projeto de circuitos microeletrônicos de alta integração*; XX Congresso de Informática; SUCESU; São Paulo, 1987.

[Can 85] Canto Filho, A. B.; *LACSUZ - Um Sistema Editor de Microcircuitos*; V Congresso da Sociedade Brasileira de Computa,c ao; Porto Alegre; 1985.

[CER 90] Czejdo, B.; Elmasri, R.; Rusinkiewics, M.; *A Graphical Data Manipulation Language for an Extended Entity-Relationship Model*; Computer; March 1990.

[CoH 86] Cox, B. and Hunt, B.; *Objects, Icons and Software ICS*; Byte, August 1986.

[Cos 91] Cosar,A. and Özgüç, B.;*A Text, Image and Graphics Editor*; Computers & Graphics; Vol. 15; No. 1; pp. 57-66; 1991.

[Dru 89] Drusinsky,Doron and Harel,David; *Using Statecharts for Hardware Description and Synthesis*; IEEE Transactions on Computer-Aided Design, vol. 8, no. 7, July 1989.

[Eli 92] Elias, Valéria G. S. e Liesenberg, Hans K. E.;*Debugging Aids for Statechart-Based Systems*; Relatório Técnico, DCC-11/92.

[Fav 89] Favero, E. L. e Price, R. T.;*Editor Diagramático para DFD*; Anais do IX Congresso da SBC, Julho de 1989.

[Fei 87] Feiler, P.; *User Interface Technology Survey*; Techinical Report, April 1987.

[Fig 90] Figueiredo, Antônio G. e Liesenberg, Hans K. E.; *Geração de Gerenciadores de Sistemas Reativos*; Relatório Técnico no. 20/90.

[Fig 91] Figueiredo, Anônio G.; *Um Processo de Síntese de Sistemas Reativos*, Tese de Mestrado, DCC - IMECC - UNICAMP, Dezembro 1991.

[Fiu 89] Fiume, E.; *Active Objects in the Construction of Graphical User Interfaces*; University of Toronto; Canada, 1989.

[Fon 88] Fontanini, W.; *Um Editor de Diagrama de Blocos para Projeto de Sistemas de Controle*; VII Congresso Brasileiro de Automática; São José dos Campos - SP; 1988.

[Ful 88] Fulton, J.; *X Window System*; Release Notes, 1988.

[Fur 90] Furuti, C.A., *Implementação do Tradutor Cm - C*, Tese de Mestrado, DCC-IMECC-Unicamp, Campinas-SP, 1990.

[Gai 89] Gaifman,H.; *Modeling Concurrency by Partial Orders and Nonlinear Transition Systems*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency; Lecture Notes in Computer Science, Vol. 354, pp 467-488, 1989.

[Har 85] Harel,D. and Pnueli,A.; *On the Development of Reactive Systems*; January 1985.

[Har 87a] Harel,David; *STATECHARTS: A Visual Formalism for Complex Systems*; Science of Computer Programming, vol. 8,no. 3, June 1987,pp. 231-274.

[Har 87b] Harel,David; *On Visual Formalisms*; Comm. ACM, vol. 31, no. 5, May 1987, pp. 514-530.

[Har 87c] Harel,D. ; Pnuelli,A. ; Schmidt,J.P. e Sherman,R.; *On the Formal Semantics of Statecharts* (Extended Abstract), Proceeding 2nd IEEE Symposium on Logic in Computer Science, 1987.

[Har 89] Harel,David and Kahana C. A.; *On Statecharts with Overlapping*; Department of Applied mathematics and Computer Science; April 1989.

[Har 90] Harel,D ; Lachover,H ; Naamad, A ; Pnueli, A. ; Politi, M. and Shtul-Trauring; *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*; IEEE Transaction Software Engineering, April, 1990.

[Hel 90] Heller,D. ;*XView - Programming Manual*; California, O'Reilly & Associates, Vol. 7, July, 1990.

[Hol 87] Hollyday,M.A.; Vernon,M.K. ;*A Generalized Timed Petri Net Model for Performance Analysis*; IEEE Transactions on Software Engineering, Vol. SE-13, No.12 - December 1987

[i-L 87] *The Languages of STATEMATE; Documentation for the Statemate System*; i-Logix Inc.; Burlington, MA; 1987.

[i-L 89a] *Semantics of Statecharts*; i-Logix Inc.; Burlington,MA; August 1989.

[i-L 89b] *The STATEMATE Approach to Complex Systems*; i-Logix Inc. Burlington, MA; 1989.

[Jon 89] Jones, O.; *Introduction to the X Window System*, Prentice Hall,Inc; 1989.

[KS 89] Kantorowitz, E. and Sudarsky, O.; *The Adaptable User Interface*; Communications of the ACM, Vol. 32, No. 11, November 1989.

[LVC 89] Linton, M. A.; Vlissides, J. M.; Calder, P.; *Composing User Interfaces with Interviews*; Computer; February 1989; pp. 2,22.

[Mag 85] Maguire, M.C.; *A Review of Human Factors Guidelines and Techniques from the Design of Graphical Human Interfaces*; Computers & Graphics; Vol. 9; No. 13; pp. 221-235; 1985.

[Mar 90] Maraninchi,F.; *Argonaute: Graphical Description, Semantics and Verification of Reative Systems by Using a Process Algebra*, in Automatic Verification Methods for Finite State Systems; Lecture Notes in Computer Science, Vol. 407, pp 38-53, 1990.

[Mar 84] Marson, M.A.; Conte,G.; Balbo, G.;*A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems*; ACM Transactions On Computer Systems, Vol.2, No.2, May 1984, Pages 93-122.

[Mas 91] Masiero, P. C.; Fortes, R. P. M. e Batista, J. E. S.; *Edição e Simulação do Aspecto Comportamental de Sistemas de Tempo Real*; XVIII Seminário Integrado de Software e Hardware, Santos, 1991, pp.45-61.

[Men 89] Menascé,D.A. and Fonseca,N.L.S.;*Redes de Petri Estocásticas*; Techinical Report; August 1989.

[NPG 90] *Network Programming Guide*; Sun Microsystems, Inc., 1990.

[Nye 87] Nye, A.; *The X Window System*; Programming Manual, 1987.

[Per 88] Perkusich, A.;*Interface Gráfica Homem-Máquina para Aplicação em Automação e Controle de Processos*; VII Congresso Brasileiro de Automática; São José dos Campos - SP; 1988.

[Pet 82] Petri C. A.; *State-Transition Structures in Physics and in Computation*, International Journal of Theoretical Physics, Vol. 21, No.12 (1982).

[Piñ 90] Piñon A. H.; *Editor Topológico para a Linguagem de Especificação de Computações - LegoShell*, Dissertação de Mestrado, DCC - IMECC - UNICAMP, dezembro de 1990.

[Pnu 86] Pnueli,A.;*Aplications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends*; in Current Trends in Concurrency; Lecture Notes in Computer Science, Vol. 224, pp. 510-584, 1986.

[Roz 86] Rozemberg,G., Thiagarajan,P.S.; *Petri Nets: Basic Notions, Structure Behaviour*, in Current Trends in Concurrency; Lecture Notes in Computer Science, Vol. 224, pp 585-668, 1986.

[Sil 90] Silva, G.B.; *FARPE - Uma Ferramenta para a análise de Modelos de Redes de Petri Estocásticas e Generalizadas*; Tese de Mestrado; Departamento de Ciência da Computação, UFMG, Março de 1990.

[Sun 82] Sunshine, C.A. ;Thompson,D.H.; Erickson, R.W.; Gerhart, S.L.; Schwabe,D. ;*Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models"*; IEEE Transactions on Software Engineering, Vol. SE-8, No. 5, September 1982.

[SzM 88] Szekely, P. A. and Myers, B. A.; *A User Interface Toolkit Based on Graphical Objects and Constraints*; ACM, September 1988.

[Tak 89] Takahashi, Tadao; *O Paradigma de Objetos: Introdução e Tendências*; Projeto ETHOS; Julho 1989.

[Thi 85] Thimbleby, H. ;*Failure in the Technical User-Interface Design Process*; Computers & Graphics; Vol.9, No. 3; 1985.

[Wal 89] Walcélio, L. M. e Price, R. T.; *Aumentando a produtividade através de um Editor de Diagramas Generalizado*; XXII Congresso Nacional de Informática; São Paulo; 1989.

[Web 85] Weber, H. R.; *Meditation on Man-Machine Interfaces or our Personal Role in Graphics Dialogue Programming*; Computers & Graphics; Vol. 9; No. 3; pp. 237-245; 1985.

[Wil 90] Willson,R.G. and Krogh,B.H.;*Petri Net Tools for the Specification and Analysis of Discrete Controllers*, IEEE Transactions on Software Engineering, Vol. 16, No.1, January 1990.

[Win 89] Winskel, G.; *An Introduction to Event Structures*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency; Lecture Notes in Computer Science, Vol. 354, pp 364-397, 1989.

Apêndice A

Interpretações diferentes para Histories

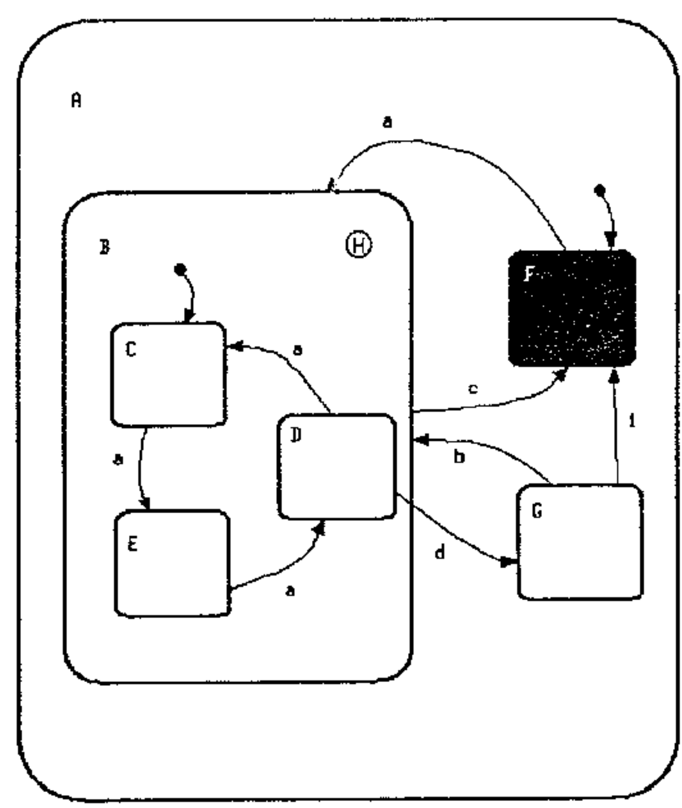

Figura A.1: Inicialização por *default*

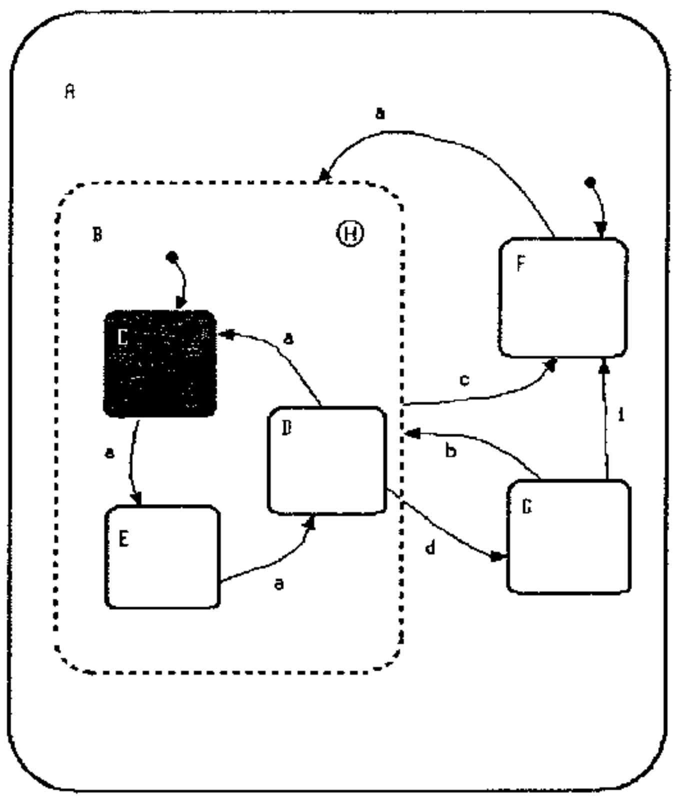

Figura A.2: Ocorrência do evento a, entrada por *default*

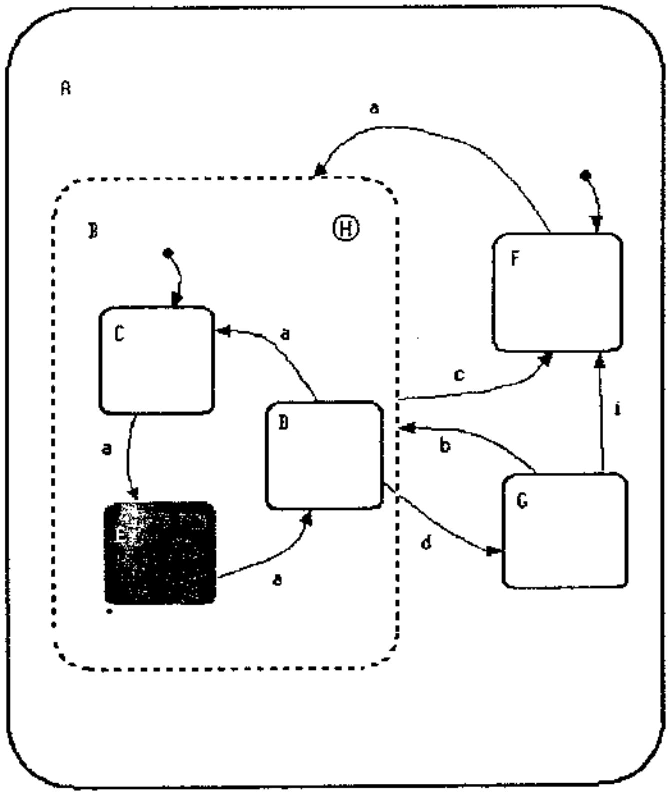

Figura A.3: Segunda ocorrência do evento a

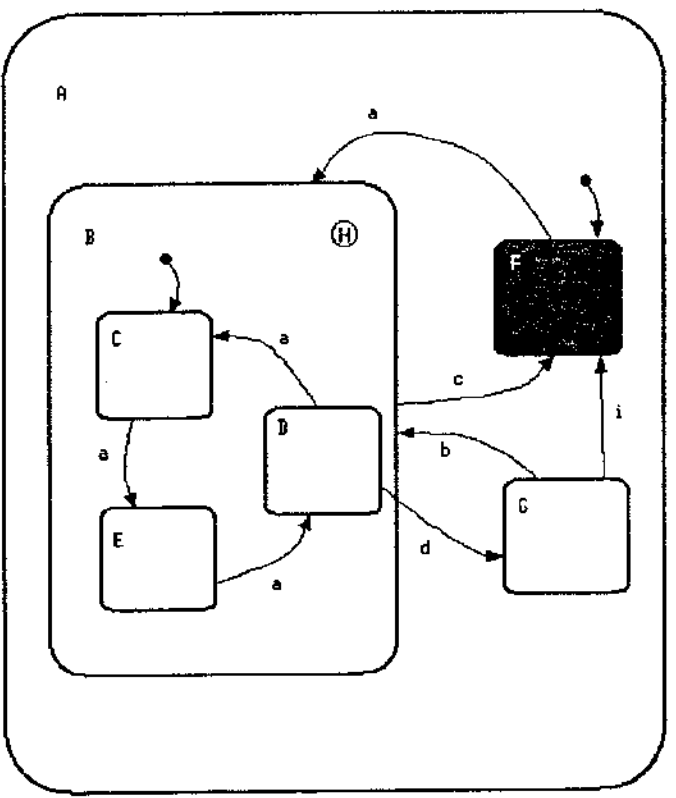

Figura A.4: Ocorrência do evento c

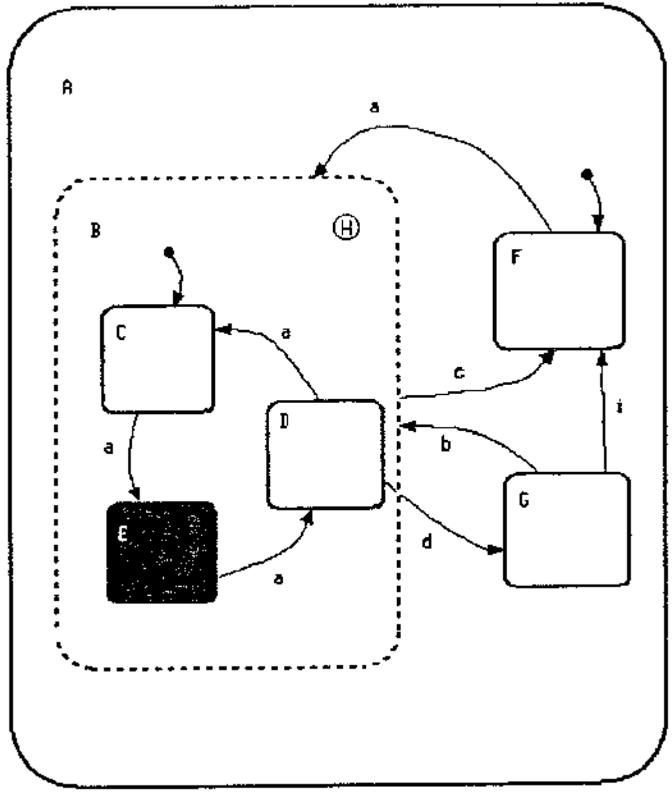

Figura A.5: Ocorrência do evento a, entrada por *history*

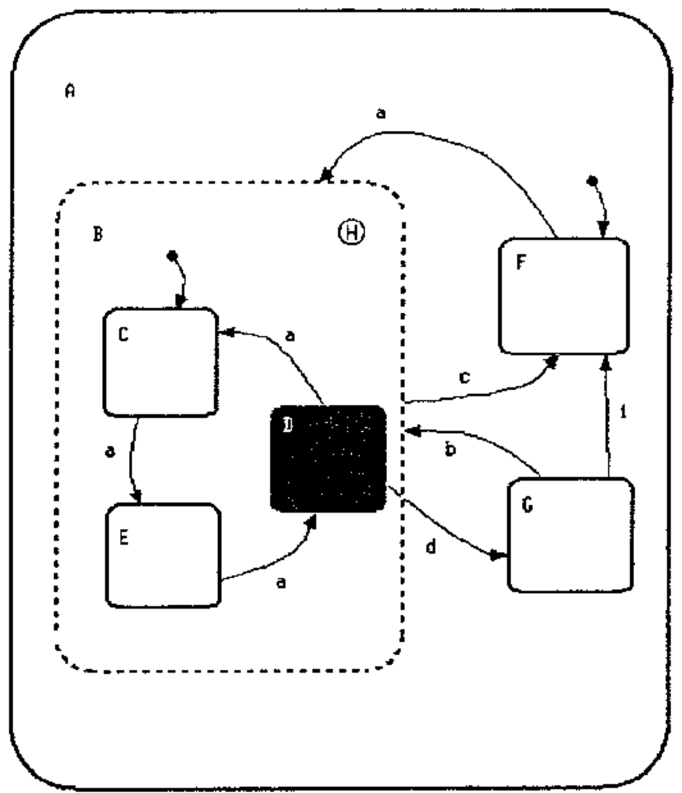

Figura A.6: Ocorrência do evento a

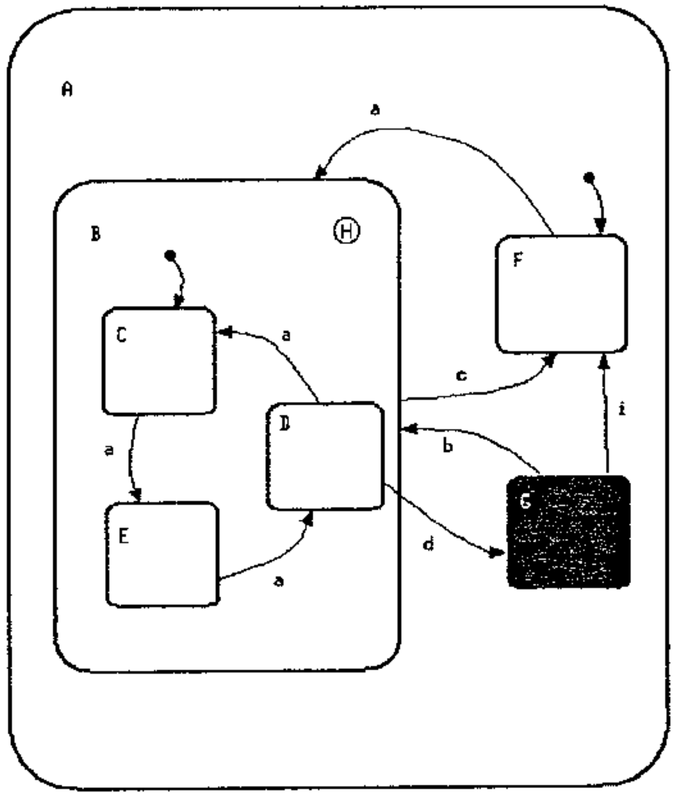

Figura A.7: Ocorrência do evento d

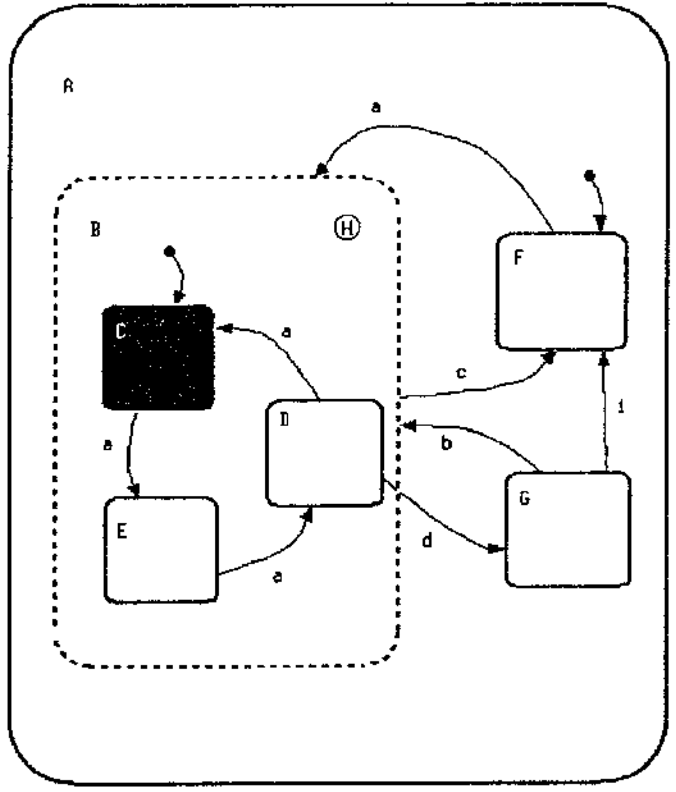

Figura A.8: Ocorrência do evento b.History associado a arco: entrada por default

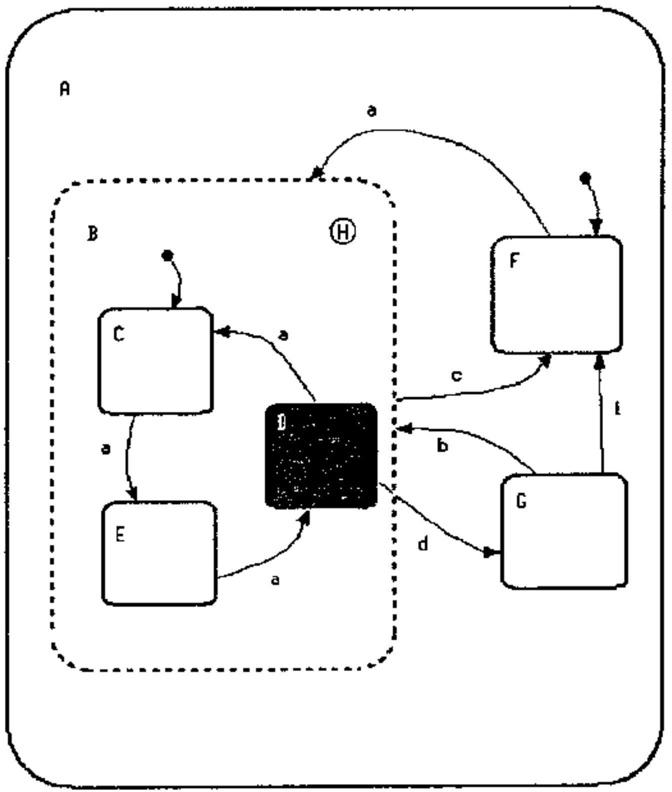

Figura A.9: Ocorrência do evento b. History associado à bolha: entrada por history

Apêndice B

Sintaxe de LEG

O autor do GGSR definiu toda a sintaxe da linguagem LEG, a qual nós mostramos aqui bem como seus símbolos básicos.

- **Letra**

  <letra> ::= a | ... | z | A | ... | Z

- **Dígitos**

  <digitos> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- **Delimitadores**

  <delimitadores> ::= { | } | |

- **Identificadores**

  - **Sintaxe**

    <identicadores> ::= <letra> { <letra> | <digito> }

- **Expressões**

- **Comandos gerais**

  - **Sintaxe**

    <comandos gerais> ::= [<history>]{<ações>}[<transições>]

- **Comando History**

  Este comando serve para associar a presença do mecanismo de *history* a uma determinada bolha.

  - **Sintaxe**

    <history> ::= history <tipos de history> ;
    <tipos de history> ::= <tipo de history> | <tipo de history> , <tipo de history>
    <tipo de history> ::= deep | shallow | pred

- **Comandos de ações**

  Como visto anteriormente, a cada bolha poderão estar associados ações que servem para executar funções escrita em C.

  - **Sintaxe**

    <ações> ::= (on_entry | throughout | on_exit) : [<código C>] ;

  - **Exemplo**

    on_entry : [circle();] ;
    on_exit : [clear();] ;
    throughout : [UpdateChart();];

- **Comandos de controle**

  - **Sintaxe**

    <transição> ::= transition { <corpo da transição> }
    <corpo da transição> ::= <evento> { ; <evento>}
    <evento> ::= on_event (<identificador de evento>) [<comando if>] to <identificador de bolha> [<comando do>]
    <comando if> ::= if [<teste>]
    <comando do> ::= do [<código C>]
    <identificador de bolha> ::= <identificador>
    <identificador de evento> ::= <identificador>

  - **Exemplo**

    transition {
    on_event(t1) **to** EDIT;
    on_event(t2) **to** DRAW **do** [f();];
    on_event(t3) **if** [`in_blob`(G) && `in_blob`(H)]
    to CREATE **do** [f();];
    }

- **Definição de uma bolha**

  - **Sintaxe**

    <bolha> ::= <bloco em LEG> | <chamada>
    <bloco em LEG> ::= blob <identificador> <corpo>
    <corpo> ::= { [<comandos gerais>] [<bolha/componente>] }
    <bolha/componente> ::= <bolha> | <componente>
    <componente> ::= <bolha> | <bolha> { | <bolha>}
    <chamada> ::= call <identificador> ;

- **O programa principal**

- **Sintaxe**
  <bolha principal> ::= main <bolha em LEG>
- **Exemplo**

```
main blob Menu
{
      .
      .
      .
}
```

- **Nota**
  Obrigatoriamente todo programa em LEG, deverá ter uma bolha principal que é identificado pela palavra chave main.

- **Um programa em LEG**
  - **Sintaxe**
    <programa> ::= {<include>} {<bloco em LEG>} <programa principal>
    <include> ::= **#include** "<arquivo>"
  - **Exemplo**

```
#include "X.b"

blob F
{

}

main blob Menu
{
  blob A
  {
    blob B
    {
    } |
    blob C
    {
    }
  }
  call F;
  call X;
}
```

- **Bolha "default"**

  A bolha *default* é descrita em LEG da seguinte forma: toda vez que uma bolha B, for bolha *default* e filha de uma bolha S qualquer, ela será sempre a primeira bolha a ser descrita, entre as bolhas filhas de S.

  - **Exemplo**

```
blob Worksheet
{
  blob Processing
  {
  }
  blob Graphics
  {
  }
  blob Exit
  {
  }
}
```

Apêndice C

Estruturas de Dados de descrição de estadogramas

```
typedef char *CADEIA;

/*--------------------------------------------------------------------*/

/* -----      TIPOS E ESTRUTURAS DA INTERFACE COM USUARIO -------- */

/* Par FRAME_CANVAS. Transfere para uma janela filho, qual o canvas
    e o frame da janela pai */

typedef struct estr1
    {
      Frame fr;
      Canvas can;
    }PAR;

/* ---------------- TIPOS E ESTRUTURAS GRAFICAS  ---------------- */

/************* Ponto na tela. x -> Eixo X; y -> Eixo Y ************/
typedef struct coorde
    {
       int x,y;
    }COORDENADA;

/******** COORD. (x,y) REAL. Para tracar arco c/ SPLINE **********/
typedef struct tipo_ponto
        {
         float x;
         float y;
```

```
}COORD_REAL;

/****** INF. p/ Armazenar uma curva e traca-la c/ SPLINE ********/
typedef struct tipo_curva
        {
         int ordem;
         int nro_pontos;
         COORD_REAL malha[MAX_PTS];
}TIPO_CURVA;


*  typedef struct arc_exp
   {
TIPO_CURVA arc_exp;
COORDENADA pos_rot_exp;
        int nivel_visual_exp;
        int nivel_visual_fora;
        int ind_est_dest_exp;
        int ind_est_dest_fora;
}ARC_EXP;


*   ARC_EXP *ARC;


/*******************  Estrut. de um ARCO ***********************/
typedef struct arco
        {
          TIPO_CURVA arc;
*          int nro_exp;
*          ARC_EXP *arc_exp[11]; /* Um arco pode ter at'e 10 expansoes */
          COORDENADA hist,   /* posicao da historia(Se existir)*/
                     pos_rot;  /* Posicao do rotulo(nomeo) */
          int valor_hist;        /* Tipo da historia(H,H*, "") */
          int nivel_visual;    /* Nivel de visualizacao do arco */
          CADEIA rotulo;          /* Nome que aparece no video */
          CADEIA acao;   /* Expressao de acao p/ arco complexo */
          CADEIA evento;         /* Evento de um aarco complexo */
*          CADEIA condicao;
}ARCO;

/************ declaracao da estrutura RETANGULO  ****************/

typedef struct list  {
```

```
          COORDENADA c1,          /* Ponto inicial do poligono */
                     c2;            /* Ponto final do poligono */
          struct list *prox;
}RET;

/************ declaracao da estrutura GRAF  ********************/

typedef struct graf  {
        int x,                 /* Coord. do canto superior esquerdo */
            y,
            x_larg,   /* Coord. do canto inferior direito da bolha */
            y_alt,
            x_nome,   /* Coord. posicao inicial de escrita do nome */
            y_nome;
*            COORDENADA hist;
*            int valor_hist;
            COORDENADA cse; /* Canto superior da janela que contem a bolha */
            COORDENADA lh; /* Largura (l) e altura (h) da janela  */

            ARCO arc_hist;   /* Se houver historia, armazenar arco */
}GRAF;

/************  Estrutura  da pilha de janelas  ********************/

typedef struct pilha_jan  {
        PAR *fr_can;    /* Apontadores p/ canvas e frame da janela */
        int nivel;              /* nivel de visualizacao da janela */
        struct tab *bolha;                /* bolha a ser expandida */
        struct pilha_jan *prox;
}NO_PILHA_JANELA;


/*************Estrutura de Atividades dos estados ***********/

*  typedef struct atividade {
int fon_entry; /* flag */
CADEIA acao_on_entry;
int fthroughout;   /* flag  */
CADEIA ativ_throughout;
int fon_exit;  /* flag */
CADEIA acao_on_exit;
}ATIVIDADE;

typedef struct tab;
```

```
/******* ESTRUTURA INTERNA AUXILIAR PARA SER UTILIZADA NA
         EXECUCAO DO HISTORY DE RETORNO                          ***/


*   typedef struct lista {
struct tab *ant_irmao;
struct lista *prox_lista;
}LISTA;




/* -------------- TIPOS E ESTRUTURAS P/ SIMULACAO --------------- */

/*************** declaracao da estrutura TAB-EST ********************/

typedef struct tab  {
CADEIA nome_est;         /* nome do estado          */
        int ind_sub_est;         /* Valor do Indice do subest p/ este no' */
int tipo;           /* tipo final(and/xor/fim)
      ( 0 / 1 / 2 ) */
        int num_comp;
int defo;           /* default/nao =(1/0)
    defH/ defH* = (2/3)     */
        int nivel_visual;         /* Nivel visual da bolha */
*        int history;             /* history/nao (1/0)  */
int nivel;          /* altura-arvore (raiz=0)  */
int lig;  /* indica "ligado" (1/0)   */
int ult_lig;
     /* indica hist "ligado"    */
*        LISTA *prim_lista;
struct tab *pai ;
        /* pont. para o pai        */
struct tab *filho_defo;  /* pont. para filho default */
struct tab *prim_filho ; /* pont. para o 1o. filho  */
struct tab *prox_irmao ; /* pont. para prox irmao   */
struct tab *ant_irmao ;  /* pont. para irmao ant.   */
        GRAF gf;         /* Inf. graficas da bolha nao exp. */
        GRAF gf_exp;    /* Inf. graficas da bolha expandida */
int sentinela;      /* alerta = 1234           */
* ATIVIDADE activity;
}NO_STATE;

NO_STATE *subest[MAX_NRO_EST];   /* subest:vetor ponts p/ sub-estados*/
```

```
 /* estrut. que armazena os indices dos estados ativados/desativados */
typedef struct llig{
         int ind_sub_est;
         struct llig *prox;
           }Llig;

/*-----------------------------------------------------------------*/
/*** Estr. temporaria que guarda os sub arcos tracados que depois
      serao guardados nas sub transicoes(Usado em arcos M pa N) ****/

typedef struct lista_estados  {
            NO_STATE *bolha;  /* Ponteiro para a bolha corresp.   */
            struct lista_estados *prox; /* pont. para proximo no da lista */
            ARCO arco;                /* Inf. Graficas do sub arco */
}LISTA_ESTADOS;


/*-----------------------------------------------------------------*/
/*************** declaracao da estrutura da Pilha Polonesa ************/

typedef struct polon {
int tipo_item;   /* tipo do item a ser avaliado */
int ind_item;  /* indice do item              */
}PILHA_POL[TAM_POL];


/*-----------------------------------------------------------------*/
/*************** declaracao da estrutura da LISTA DE TRANSICOES *******/

typedef struct trans {
PILHA_POL *ev_aval;     /* evento a ser avaliado
                                   para efetivacao da transicao */
        PILHA_POL *ac_aval;  /* Pilha polonesa p/ avaliacao de um acao */
* PILHA_POL *co_aval;
int ind_est_fonte;      /* indice estado fonte      */
int ind_est_destino;   /* indice estado dest       */
char h[4];               /* indica historia: H / H* */
  struct trans *prox_tr;   /* pont. para a prox trans */
struct sub_trans *est_f; /* pont. para est_fonte da
        trans paralela      */
struct sub_trans *est_d; /* pont. para est_dest da
      trans paralela        */
int ind_gravacao;        /* indice de gravacao      */
int sentinela;           /* alerta = 4567           */
```

```
        ARCO arco;                  /* Inf. graficas do arco */

}TRANSICAO;

TRANSICAO *tr_ok[MAX_NRO_TR];     /* vetor de ponts. p/ trans habilit */


/*----------------------------------------------------------------------*/
/*************** declaracao da estrutura DE TRANSICOES PARALELAS ******/

typedef struct sub_trans {
int ind_est;                  /* indice estado           */
char h[4];                        /* indica historia: H / H* */
struct sub_trans *prox_sub_tr;  /* pont. p/ prox tr          */
        ARCO arco;                    /* Inf. graficas do sub arco */
int sentinela;                  /* alerta = 4567             */
}SUB_TRANSICAO;


/*----------------------------------------------------------------------*/
/****** declaracao da estrutura de TRANS relativa ao EVento ***********/

typedef struct  trans_ev {
        TRANSICAO *trans;               /* pont p/ trans
    relativa ao evento */
        struct trans_ev *prox_trans_ev; /* pont p/ proxima trans
                                           relativa ao evento */
} TRANS_EV;


/*----------------------------------------------------------------------*/
/********** declaracao da estrutura de  TAB-EV  **********************/

typedef struct evento {
CADEIA nome_ev;               /* nome do evento        */
int valor_ev;                 /* valor do evento       */
                TRANS_EV *trans;            /* pont. p/ inic lista de
                                               transicoes deste evento */
                int sentinela;              /* alerta = 9876           */
}NO_EVENTO;

NO_EVENTO *event[MAX_NRO_EV]; /* vetor de ponts. p/ eventos STTCHART */
```

```
/*-----------------------------------------------------------------*/
/******** declaracao estruturas LISTA de caminho de estados ***********/

typedef struct lista_est {
NO_STATE *ptr_est;        /* ptr para estado */
struct lista_est *prox_lista; /* ptr p/ prox est */
}LISTA_EST;

/*-----------------------------------------------------------------*/
/**** declaracao estrutura dos estados sintaticamente declarados ******/

typedef struct st {
CADEIA n_st;  /* nome estado     */
CADEIA n_pai;  /* nome estado-pai */
int tipo; /* tipo do estado  */
int defo; /* default         */
char marca;             /* marca para montar a arvEOU */
}ST;

/*-----------------------------------------------------------------*/
/**** declaracao estrutura TABELA de Estados Temporaria (TET)
      para a analise semantica para criacao da arvore EOU *************/

ST *subst[100];         /* tabela intermediaria de subestados criada
   na analise sintatica    */


/*-----------------------------------------------------------------
   TIPOS E ESTRUTURAS DOS ARQUIVOS
-------------------------------------------------------------------*/
/******************  Estrut. de um  ARCO ***********************/
typedef struct regis_arco
         {
           TIPO_CURVA arc;                /* Os pts. que geram o arco */
*           int nro_exp;
*           ARC_EXP arc_exp[11];
           COORDENADA hist,         /* posicao da historia(Se existir)*/
                      pos_rot;

                                          /* Posicao do rotulo(nomeo) */
           int valor_hist;             /* tipo do hist */
           int nivel_visual;         /* Nive de visualizacao do arco */

           char rotulo[40];             /* Nome que aparece no video */
```

```
          char acao[500];      /* Expressao de acao p/ arco complexo */
          char evento[500];           /* Evento de um aarco complexo */
*          char condicao[500];
}REGISTRO_ARCO;


/*************Registro de Atividades dos estados ************/

*   typedef struct regis_atividade {
int fon_entry; /* flag */
char acao_on_entry[100];
int fthroughout;   /* flag  */
char ativ_throughout[100];
int fon_exit;  /* flag */
char acao_on_exit[100];
}REG_ATIVIDADE;


/*********** registro a ser gravado para TABELA DE ESTADOS ************/

typedef struct regis_estado {
                int ind_est;        /* indice do estado  */
char nome_est[30];  /* nome do estado  */
int tipo;
int num_comp;            /* tipo final(fim/and/xor)  */
int defo;            /* default/ nao default
   / defH / defH*        */
int nivel;           /* altura da arvore (raiz=0)*/
*                int history;        /* history/nao */
                int nivel_visual;
int lig;             /* indica estado ligado  */
int ult_lig;         /* indica historia ligado  */
                GRAF info_grf;       /* Inform. graficas de uma bolha */
                GRAF info_exp;
*                REG_ATIVIDADE info_ativ;
}REGISTRO_ST;


/*--------------------------------------------------------------------*/
/***** registro a ser gravado para TABELA DE EVENTOS *****************/

typedef struct regis_ev {
int ind_ev;     /* indice do evento  */
char nome_ev[30];     /* nome do evento  */
```

```
int qtdd_trans;          /* quantas trans relati-
   vas ao evento         */
}REGISTRO_EV;


/*-----------------------------------------------------------------*/
/** registro a ser gravado para TABELA DE TRANSICOES *****************/

typedef struct regis_tr {
int ind_tr;              /* indice da transicao  */
int qt_pol_ev;        /* quantas linhas-pol eventos  */
               int qt_pol_ac;       /* quantas linhas-pol acao */
int ind_estf;            /* ind do estado-fonte */
int ind_estd;            /* ind do estado-dest  */

char h[4];               /* historia: H / H*  */
  int nro_sub_tr_f; /* nro de est_fontes paralelos
   na sub_trans    */
int nro_sub_tr_d; /* nro de est_dest paralelos
   na sub_trans    */
                REGISTRO_ARCO arc;          /* Inf. graficas do arco */
}REGISTRO_TR;


/*-----------------------------------------------------------------*/
/**registro a ser gravado p/ as TABELAS DE SUB-TRANSICOES *************/

typedef struct regis_sub_tr {
char ind_est;                /* indice  do estado */
char h[4];                        /* historia: H / H*  */
                REGISTRO_ARCO arc;       /* Inf. graficas do sub arco */
}REGISTRO_SUB_TR;

/*-----------------------------------------------------------------*/
/*** registro a ser gravado para as tabelas PILHAS-POLONESAS **********/

typedef struct regis_polon {
int tipo_item;         /* tipo do item a ser avaliado */
int ind_item;                      /* indice do item */
}REGISTRO_POL;

/*-----------------------------------------------------------------*/
/****** registro a ser gravado para as TRANS_EVENTOS *****************/
```

```
typedef struct  regis_trans_ev {
int ind_ev;     /* indice do evento  */
int ind_tr;             /* indice da transicao  */
}REGISTRO_TRANS_EV;

/*------------------------------------------------------------------*/
/****** registro a ser gravado para a TABELA DE VARIAVEIS *************/

typedef struct  regis_vars {
int ind_var;    /* indice da variavel */
int tipo_var;           /* tipo da variavel     */
                union {
                    int valor_i;
                    float valor_f;
                    char valor_byte;
                    char *valor_s;
                } u;                 /* valor da variavel    */
                char nome_var[30];      /* nome da variavel     */
}REGISTRO_VARS;

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