Um Processo de Síntese de Sistemas Reativos

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Antonio Gonçalves Figueiredo Filho e aprovada pela Comissão Julgadora.

Campinas, 18 de dezembro de 1991

Prof. Dr. Hans ~~Kurt Edmund~~ Liesenberg
K. E.

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

À minha família

Conteúdo

Lista de Figuras

Agradecimentos

Gostaria de agradecer a ajuda das pessoas que direta ou indiretamente contribuíram para a elaboração deste trabalho através de críticas, sugestões ou simplesmente estiveram ao meu lado dando um apoio. Agradeço o apoio financeiro do CNPq e FAPESP, também ao pessoal do projeto A_HAND, em especial ao Prof. Dr. Rogério Drummond, a H. Piñon, a C. A. Furuti e a C. A. Polanczyk. Assim como ao pessoal da Secretaria: D. Isabel, Cidinha e Luís.

A agradeço o apoio amigo de Carlos Mora, Alfonso Lobos e Henrique Schneider.

Um agradecimento especial ao meu orientador Hans Liesenberg e sua esposa, Maria Helena, pelo apoio recebido nos momentos mais difíceis.

Sumário

Esta dissertação apresenta um gerador de programas adequado para implementar o controle de sistemas reativos complexos. Este processo de geração consiste em uma síntese de sistemas reativos a partir de uma especificação baseada em estadogramas. Os estadogramas são diagramas de estados convencionais estendidos suportando conceitos de hierarquia, concorrência e comunicação. Esta ferramenta comporta-se como um tradutor que recebe como entrada uma descrição textual (i.e., um programa escrito em uma linguagem descritiva de estadogramas - LEG) e produz como saída um programa funcionalmente equivalente em C. Um programa escrito em LEG associa código inerente ao tratamento do controle da aplicação (código LEG propriamente dito) e código responsável pela aplicação (código escrito em C).

Para ilustrar uma aplicação desta ferramenta, descreve-se sua utilização na construção de um editor topológico.

Abstract

A Program generator appropriate to implement the complex reactive systems control is presented. This generation process consists in a synthesis of reactive systems from a statecharts-based specification. Statecharts are a broad extension of conventional state diagrams supporting the notion of hierarchy, concurrency and communication. This tool behaves like a translator that receives a textual description of a statechart(i.e., a program written in a statecharts language – LEG) in its input and it outputs a functionally equivalent C program. A program written in LEG combines code inherent to application control (represented by LEG code) and to application code (code written in C).

An application of this tool in a construction of a topological editor is described.

Capítulo 1

Introdução

Este capítulo trata inicialmente da motivação básica para realização do presente trabalho. Em seguida é feito um levantamento do atual panorama das atividades de pesquisas em interfaces-usuário introduzindo-se concomitantemente a terminologia da área e algumas classificações relevantes de sistemas de interface homem-máquina. Finalmente é apresentado o escopo deste trabalho e a organização dos capítulos restantes desta dissertação. Termos técnicos não muito enraizados em português têm seu equivalente em inglês relacionado em nota de rodapé na sua primeira ocorrência no texto.

1.1 Motivação

Atualmente, muitos esforços têm se concentrado em pesquisas com o objetivo de dotar os computadores de maior facilidade de uso. Anteriormente, os computadores eram usados por um número bastante restrito de pessoas que detinham conhecimento especializado. Portanto, os problemas quanto ao uso destes não eram considerados críticos. O advento da tecnologia em microeletrônica proporcionou um crescimento extremamente grande no universo de usuários de computadores, ocasionado pela decorrente queda de seus custos. Com isso os fabricantes começaram a persuadir mais pessoas, inclusive aquelas sem nenhum conhecimento específico em computação, a usarem computadores, apresentando cada vez mais novos modelos de computadores e ferramentas auxiliares com maior facilidade de uso. Assim, surgia uma nova área de pesquisa com o objetivo em melhorar a interação entre o usuário e o computador.

A criação de um bom sistema de interface entre o usuário e o computador, denominado aqui de *interface-usuário*, é uma tarefa difícil onde é preciso abordar vários aspectos desde aqueles relacionados com o usuário (conhecidos como fatores humanos) até os de projeto. Uma interface-usuário não somente é difícil de criar como também não existe nenhuma estratégia de projeto que possa garantir que a interface resultante seja "amigável"[1] e adequada ao usuário final. Portanto, uma prática bastante utilizada para atacar os problemas relacionados com o projeto de interface-usuário é a de obter diretrizes para nortear a especificação e o desenvolvimento de um sistema que tenha boa funcionalidade e que sobretudo seja considerado "amigável" pelo usuário final através de conceitos, necessidades e métodos extraídos de comentários do próprio usuário ([WEB 85], [GOO 84], [SWA 82]).

Motivado por este problema, apresenta-se neste trabalho um gerador de gerenciadores baseado em um diagrama de estados estendido ([HAR 87], [HAR 88]) adequado para a especificação e implementação de sistemas reativos complexos (veja em [HAR 85] e [PNU 86] discusões sobre sistemas reativos e seus comportamentos). Estes sistemas são caracterizados pela sua natureza de interação com o seu ambiente, dirigidos por eventos e continuamente estão sujeitos a mudanças de seus estado. Como exemplos, dentre outros, são citados alguns tipos de sistemas como redes de comunicação, sistemas de controle digital e interfaces homem-máquina. Adotou-se esse diagrama de estado pois permitem descrever estados e transições de maneira modular, habilitando, dentre outras características, abstrações de estados e ortogonalidade entre estados (i.e., concorrência). No Capítulo 2, será discutido com mais detalhe este tipo de diagrama de estado.

Para caracterizar a ferramenta aqui proposta, exemplifica-se o seu uso no contexto da geração de um sistema de gerenciamento de interface-usuário (SGIU) baseado em um modelo lógico de interface-usuário proposto por Seeheim ([GRE 85], [GRE 86]) cuja arquitetura do modelo em tempo de execução é esboçado na figura 1.1.

Vários modelos foram encontrados na literatura, como, dentre outros: *Dialogue Transaction Model* [HAH 89], o modelo apresentado por Smart [SMA 89] (semelhante ao modelo de Seeheim) e o *Dialogue Socket Model* [COU 87], mas adotou-se aqui o modelo proposto por Seeheim por permitir identificar com mais clareza as principais componentes envolvidas em uma interface-usuário, para descrever a seguir os principais SGIU's existentes e

[1] *user-friendly*

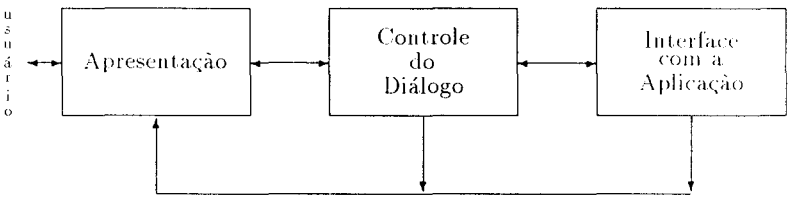

Figura 1.1: Modelo Lógico de um SGIU

por constituir-se de um modelo independente para um SGIU específico.

Os sistemas gerados pela ferramenta desenvolvida concentram-se na componente de controle do diálogo, que define a estrutura do diálogo entre o usuário e o programa de aplicação. Adotam-se daqui em diante os termos *gerenciador de diálogo* para indicar o sistema gerado e *tradutor* para a ferramenta geradora. A seguir, tem-se uma descrição de cada uma destas componentes da figura 1.1.

- **Componente de apresentação**
  Esta componente é responsável pela representação para efeitos externos dos símbolos manipulados via interface-usuário através de dispositivos de entrada e saída. Esta componente é dependente de máquina (dispositivo de *display*) e do estilo de interação adotado. Atualmente uma forma de interação muito utilizada baseia-se em sistemas denominados WIMP[2](janelas, ícones, menus e dispositivos de apontar/selecionar) [EDW 87]. A componente de apresentação pode ser vista como um nível léxico da interface-usuário, dado que a sua principal função é a conversão da representação de símbolos para efeitos externos em uma representação abstrata interna.

- **Componente de controle do diálogo**
  Como foi dito anteriormente, esta componente define a estrutura de como se processa o diálogo entre o usuário e o programa de aplicação. Esta recebe uma seqüência de símbolos[3] de entrada a partir da componente de apresentação, e uma seqüência de símbolos de saída da

[2] *Windows, Icons, Menus e Pointer system*
[3] *tokens*

aplicação. Baseado nestas duas seqüências esta componente determina a forma de como prosseguir o diálogo repassando informações relevantes para a aplicação e/ou usuário. O controlador de diálogos deve manter o estado corrente da interface e ter controle sobre este pois as ações executadas dependerão do contexto do diálogo. Esta componente pode ser vista com o nível sintático da interface-usuário. Gerenciadores de diálogos, gerados a partir do tradutor de sistemas de controle proposto neste trabalho, se constituem de uma componente típica desta natureza.

- **Componente da interface com a aplicação**
  Define a interface entre a interface-usuário e o resto do programa invocando os procedimentos da aplicação. Da ótica da interface-usuário esta componente representa funcionalmente a aplicação e pelo ponto de vista da aplicação representa a interface. Esta componente contém restrições quanto ao uso de funções da aplicação, permintindo verificar a validação semântica de entradas do usuário antes de chamar rotinas da aplicação. A interface com a aplicação representa, portanto, a semântica do diálogo. Dependendo da arquitetura do sistema, contudo, torna-se difícil identificar o que é inerente ao controle de diálogo e o que é inerente a aplicação.

As três componentes podem ser vistas como processos separados que se comunicam através de troca de símbolos assemelhando-se a execução de suas funções as tarefas efetuadas por um compilador [AHO 77].

1.2 Interface-Usuário: Terminologia e Ferramentas

Para melhor compreensão de alguns termos utilizados aqui, apresenta-se a seguir parte da terminologia introduzida por Hartson e Hix em [HAH 89].

Inicialmente, definem-se os termos associados à comunicação entre a componente de apresentação e a componente de controle do diálogo, que são "diálogo homem-máquina" e "interface homem-máquina".

- **Diálogo homem-máquina** – refere-se à comunicação biunívoca entre o usuário e o sistema computacional efetuada através da componente de controle do diálogo.

- **Interface homem-máquina** – já identificado neste texto como interface-usuário, refere-se ao meio pelo qual esta comunicação é estabelecida.

Assim, um *diálogo* refere-se à troca de símbolos e ações existente entre o homem e o computador, enquanto que a *interface* é o suporte de software e hardware através do qual ocorre a troca de símbolos.

Um importante critério para o desenvolvimento de uma interface-usuário é que esta possa ser facil e rapidamente modificável. Para que se alcance este critério, uma técnica explorada é a separação do código inerente à aplicação do código específico da controle do diálogo. Este procedimento é chamado de **independência do diálogo** . A independência do diálogo é um enfoque no qual as decisões (modificações) de projeto que afetam *somente* ao diálogo homem-máquina são isoladas das que afetam *somente* a estrutura do sistema de aplicação.

A interação entre o usuário e a componente de controle do diálogo é realizada através do que é chamado de *diálogo externo* – isto é, interface homem-máquina representada pela componente de apresentação. Com a separação da componente de controle do diálogo da componente de aplicação, na qual não existe nenhum mecanismo de comunicação direta com o usuário, surge o que é chamado de *diálogo interno*, que é representado pela componente de interface com a aplicação.

Obviamente, este diálogo interno não é inteligível pelo usuário em tempo de execução, mas sua representação formal em fase de projeto é a chave para a independência do diálogo. Assim, tanto a interface-usuário quanto o código da aplicação podem ser substituídos sem afetar um ao outro, contanto que ambos permaneçam consistente com a representação do diálogo interno.

Várias arquiteturas de interfaces-usuário foram encontrados na literatura que, de alguma forma, refletem essa independência entre componentes, como: Editor Topológico da LegoShell [PIÑ 90]; COUSIN (COoperative USer INterface) [HAY 81]; TAE Plus (Transportable Applications Environment Plus) [PER 87], [SZC 88] e [SZC 89]; DMS (Dialogue Management System) [HAH 89]; FLAIR II (Functional Language Articulated Interactive Resource) [WON 82]; RAPID/USE [WAS 85], State Diagram Specification Interpreter [JAC 83]; TIGER [KAS 82] e GWUIMS (George Washington User Interface Management System) [SIB 86].

Portanto, independência do diálogo é um ponto crucial para facilitar modificações da interface e consequentemente a sua manutenção.

A terminologia para ferramenta de desenvolvimento de interface ainda não está bem definida na literatura. Observa-se, então, que o termo "ferramenta"[4] tem sido utilizado para denotar desde uma simples biblioteca de interface até um completo ambiente de desenvolvimento de interface. Em [HAH 89], define-se Sistema de Gerenciamento de Interface-Usuário (SGIU) como sendo um conjunto de programas interativos de alto nível dedicado ao projeto, construção de protótipos, executação, avaliação e manutenção de interfaces-usuário, sendo tudo integrado sob uma interface voltada ao desenvolvimento de diálogo. Este conceito não tem sido utilizado rigosamente, chegando-se a referir-se a quase toda ferramenta relativa a interfaces homem-máquina como, por exemplo, a simples geradores de telas até construtores de protótipos de interfaces. Um outro termo bastante utilizado no contexto de interfaces-usuário é o de "conjunto de ferramentas"[5] que consiste de uma biblioteca de funções[6] para implementar características de interfaces de baixo nível (i.e., mostrar um objeto na tela, abrir janela, etc.) que podem ser chamadas por um SGIU ou por qualquer outro código de programa.

Myer ([MYE 89]) classifica as ferramentas envolvidas no desenvolvimento e gerenciamento de interface-usuário em duas grande classes:

- Conjunto de ferramentas de interface-usuário
- Sistema de desenvolvimento de interface-usuário

1.2.1 Conjunto de Ferramenta de Interface-Usuário

Um conjunto de ferramenta de interface-usuário (CFIU) é simplesmente uma biblioteca de ferramentas que implementam técnicas de interação, que permitem manipular os dispositivos físicos (tais como teclado, *mouse*, *tablet*, etc.) para efetuar entradas do tipo comandos, coordenadas, posicionamentos, etc. e facilitam a visualização de informações geradas por um programa de aplicação. Como exemplos de técnicas de interação temos "menus", barra de *scroll* gráfica, botões de tela para operação com *mouse*, etc.

A maioria dos sistemas de janelas[7] e SGIU's dispõem de um conjunto de bibliotecas de ferramentas que programas aplicativos podem utilizar. Estas

[4] *tool*
[5] *toolkit*
[6] rotinas de programas que podem ser invocados por outros programas
[7] *windows*

bibliotecas geralmente incluem vários tipos de conjuntos de ferramentas. O mais convencional é uma coleção de procedimentos (rotinas) que podem ser invocados pelo programa de aplicação. Um exemplo desse tipo é encontrado no Macintosh Toolbox [MAC 85]. Um outro tipo se utiliza do paradigma de programação orientada a objetos (com herança), o qual torna mais fácil a adequação de técnicas de interação para casos específicos. Tem-se como exemplo o conjunto de ferramentas chamado Widget [SWI 88] para o gerenciador de janelas *X Windows* [SCH 86]

Os CFIU's não oferecem muito suporte à fase de projeto de um controlador de diálogo, mas são úteis na construção de componentes de apresentação da interface.

1.2.2 Sistemas de Desenvolvimento de Interface-Usuário

Um sistema de desenvolvimento de interface-usuário (SDIU) consiste de um conjunto integrado de ferramentas que auxiliam programadores a criar e gerenciar vários aspectos associados ao desenvolvimento de interfaces. Alguns autores, como Kasik [KAS 82] (um dos primeiros a usar o termo "Sistema de Gerenciamento de Interface-Usuário" para sistemas deste tipo. Olsen [OLS 87] e Van der Boss [VAN 88], os chamam de Sistemas de Gerenciamento de Interface-Usuário (SGIU). Devido às limitações encontrados em CFIU's, foram criados SDIU's que permitiram a criação de interfaces mais abrangentes suportando uma maior quantidade de estilos de interação.

Um SDIU deverá manipular todos os aspectos da interface, incluindo a visualização (*look and feel*[8]) e todos os aspectos do diálogo entre o usuário e a aplicação. As funções básicas de um SDIU são:

- Manipular os dispositivos de entrada;
- Efetivar a validação de entradas do usuário;
- Separar as funções do programa de aplicação das funções de gerenciamento de tela (i.e., explorar a independência do diálogo);
- Permitir ao usuário adaptar a interface ao seu estilo de interação;
- Tratar de erros cometidos pelo usuário.

[8] Apresentação da interface para o usuário e a forma de interação do usuário com a interface-usuário

SDIU's podem ser classificados de acordo com a forma com que um projetista especifica uma interface:

- Interfaces baseadas em linguagens descritivas;
- Especificação gráfica de interfaces.

Interfaces baseada em linguagens

Esta classe integra a maioria dos SDIU's. Para definir um SDIU desta classe o projetista utiliza uma linguagem de propósito específico. Estas linguagens se apresentam de várias formas:

- Hierarquia de "menus";
- Diagramas de transição de estados;
- Gramáticas livres de contexto;
- Linguagens baseadas em eventos;
- Linguagens declarativas;
- Linguagens orientadas para objetos.

Na maioria dos casos, o projetista utiliza uma linguagem de propósito específico para especificar a sintaxe de uma interface, isto é, as seqüências válidas de símbolos constituindo comandos de entrada e ações de saída. Em [GRE 86] é apresentada uma extensiva comparação entre gramáticas livres de contexto, diagramas de transição de estado e linguagens baseadas em eventos.

- **Hierarquia de "menus"**
  Hierarquias de "menus" representam um tipo de SDIU dos mais simples. A navegação entre "menus" se dá através de seleções de opções em "menus" e o aparecimento de outros decorrentes destas seleções. Este tipo de interface é encontrado muito em sistemas de hipertexto, tal como o *Hypercard* da *Apple* [CON 87] e no sistema TIGER [KAS 82]. Não são muito adequadas para propósitos mais gerais.

- **Diagramas de transição de estados**
  O modelo de diagrama de transição de estados consiste basicamente de um conjunto de estados e um conjunto de transições que indicam a mudança de um estado para outro na ocorrência de um determinado evento. Assim, por exemplo, o diálogo moverá de um estado A para um estado B caso exista uma transição de A para B disparado por um dado evento. Segundo alguns autores, o primeiro SDIU foi implementado por von Newman em 1968, o qual usava máquina de estados finitos para um SGIU que somente manipulava entrada textual. Um problema quanto ao uso deste modelo é que não suporta interfaces que operam concorrentemente com objetos. Um outro problema que pode-se citar é o enorme crescimento do número de estados e transições quando cresce a especificação da interface. Em [JAC 85] é apresentado uma linguagem baseada em um diagrama de transição de estados; outro exemplo é o sistema RAPID/USE de Wasserman [WAS 85].

- **Gramáticas livres de contexto**
  Em SDIU's baseados neste modelo são utilizadas gramáticas livres de contexto para descrever uma linguagem empregada pelo usuário para comunicar-se com o aplicativo. Os símbolos terminais em uma gramática são os símbolos válidos de entrada que podem ser utilizados pelo usuário através da componente de apresentação. Seqüências desses símbolos válidas na gramática utilizada para descrever o diálogo em um dado SDIU representam ações do usuário. Segundo Green ([GRE 86]), o primeiro uso de gramática livres de contexto em projeto e implementação de interfaces-usuário não é de precisa identificação. Mas, em meados dos anos 70, vários grupos de pesquisa usaram geradores de analisadores sintáticos, tais como Yacc ([SCR 85], [JOH 75]), para criar interfaces-usuário para programas gráficos. SYNGRAPH [OLS 83] é uma ferramenta que se enquadra neste modelo.

- **Linguagens baseadas em eventos**
  As ações de SDIU's deste modelo são controladas diretamente através de eventos gerados via dispositivos de entrada. Cada dispositivo de entrada pode gerar um ou mais eventos em uma dada interação. Um evento é caracterizado pela natureza da interação acrescidos eventualmente de parâmetros. Por exemplo, suponha o evento de movimentação de um símbolo selecionado na tela que é efetuada cada vez que o cursor é movido. Assim os parâmetros associados a este evento são as coordenadas relativas do cursor em relação à sua posição an-

terior. Uma desvantagem deste modelo é que a criação de código torna-se, frequentemente, muito difícil porque o fluxo de controle não é localizado. Portanto, pequenas alterações em uma parte do código pode afetar muitas outras partes e também torna-se muito complicada a legibilidade do código quando este cresce. Squeak é um exemplo de uma linguagem baseada em eventos [CAR 85].

- **Linguagens declarativas**
  As interfaces baseadas em linguagens declarativas utilizam-se geralmente de pontos específicos de interação, tais como campos e "menus". O diálogo é descrito em função da manipulação destes pontos pelo usuário. Assim, interfaces suportada por linguagens declarativas são geralmente chamadas por alguns autores, como [OLS 87] e [MYE 89], de interfaces providas de diálogos baseados em formas: o usuário interage com a aplicação em áreas específicas onde insere textos ou seleciona apções através de menus, botões, etc. Uma vantagem deste tipo de linguagens é que elas liberam o projetista de problemas como o tratamento de uma seqüência de eventos podendo-se o mesmo concentrar mais na informação que é passada de um lado para outro via pontos de interação. A desvantagem principal destas linguagens é que estas suportam somente o desenvolvimento de interfaces baseadas em forma - eventualmente outros tipos desejados pelo usuário poderão ser implementados nas áreas gráficas já reservada para aplicação. Por exemplo, estes sistemas não suportam tarefas como: movimentação de objetos(dragging), *rubberbands* e desenhos de objetos gráficos. COUSIN [HAY 81] é um exemplo típico deste modelo.

- **Linguagens orientadas para objetos**
  Este modelo proporciona uma estrutura orientada a objetos na qual os projetistas programam a interface de forma incremental. Um SDIU desta classe é caracterizado pela especialização de SDIU's mais genéricos usando o mecanismo de herança suportado por linguagens orientadas para objetos. GWUIMS [SIB 86], que usa Lisp orientado para objetos, é um exemplo deste modelo.

Especificação Gráfica

Um sistema de especificação de interfaces-usuário gráfica permite definir uma interface por posicionamento de símbolos na tela. Estes sistemas se

preocupam mais com a visualização, isto é, com a componente de apresentação, e não tratam muito o controle de diálogo dando assim suporte a uma classe restrita de interfaces. Por outro lado, esta técnica de construção é de fácil uso não requerendo que o projetista seja um especialista em computação.

1.3 Estadogramas: O enfoque adotado

É possível observar na literatura que o enfoque estado/evento tem sido utilizado cada vez mais no contexto de SDIU's adotando-se geralmente diagramas de transição de estados ou máquinas de estados finitos para especificação de sistemas reativos ([JAC 83], [JAC 85], [SUN 82], [TEN 81]). O comportamento de um diagrama de estado está basicamente associado à mudança de um certo estado para outro após a ocorrência de um determinado evento ter sido disparado (ou acionado). Os diagramas de transição de estados, em geral, são representados por "grafos dirigidos", onde os nós representam os estados e as arestas representam as transições.

Para que o enfoque estado/evento seja útil na especificação de sistemas reativos, a representação deverá permitir a descrição de estruturas hierárquicas e modulares. D. Harel propõe, em [HAR 87], uma representação bastante flexível que denomina de *estadograma*[9] e que consiste de uma extensão do diagrama de estado convencional abrangendo os conceitos de: *hierarquia*, *concorrência* e *comunicação*. Com esta extensão permitiu-se ter novas características em diagramas de estados como, principalmente:

- A noção de abstração e refinamento - estados, em um estadograma podem ser repetidamente combinados em estados de níveis mais alto (ou, por outro lado, estados de níveis mais alto podem ser refinados em níveis mais baixos);
- As transições, em estadogramas, não estão restritas a um único nível, estas podem atuar de um estado em qualquer nível de abstração para qualquer outro;
- Apresenta a capacidade de ortogonalidade de estados, podendo assim ao mesmo tempo estar-se não só em um estado isolado, mas em um

[9] *Statecharts*

conjunto de estados em paralelo (permitindo, com isso, um certo mecanismo de concorrência);

- O estadograma é dotado de um mecanismo de *history* bastante poderoso que guarda a história dos estados já visitados.

No capítulo seguinte serão explorados os conceitos, características e aplicações dos estadogramas.

O tradutor proposto neste trabalho baseia-se na representação de diagramas de estados proposta por D. Harel.

1.4 Escopo da Tese

Este trabalho visa descrever uma ferramenta geradora de gerenciadores de sistemas reativos definidos nos próximos capítulos cujo funcionamento será exemplificado através da geração de um editor topológico. A geração se dá a partir de um arquivo de entrada contendo uma linguagem textual descritiva de estadogramas que, no caso particular do exemplo adotado, descreve o controle do diálogo a ser gerado e tem como resultado um programa escrito em linguagem C que corresponde à implementação da especificação do programa dado como entrada.

São apresentados no capítulo 2 conceitos, notações, características, funcionalidade e extensões dos "estadogramas". No capítulo 3, são abordados a arquitetura do gerador, a linguagem descritiva de estadogramas e a estrutura dos programas gerados pelo tradutor. Neste capítulo são tratados também detalhes particulares da implementação do tradutor. Para melhor entendimento da funcionalidade e aplicabilidade da ferramenta apresenta-se no capítulo 4 uma aplicação da utilização do tradutor na geração de um editor topológico e alguns de seus detalhes inerentes ao seu ambiente no qual este foi implementado. Finalmente, no capítulo 5 são apresentadas as conclusões e é feita uma análise crítica em relação à versão atual do projeto e algumas sugestões para futuras pesquisas.

Capítulo 2

O Estadograma

Este capítulo é dedicado a "estadogramas", conceitos no qual se baseia o presente trabalho, e a uma breve introdução a sistemas reativos.

2.1 Sistemas Reativos e Diagramas de Estados

A literatura sobre engenharia de software, linguagens de programação e projetos de software está repleta de artigos que descrevem métodos destinados a especificação e projeto de sistemas grandes e complexos. Estes sistemas computacionais podem ser vistos, basicamente, sob duas ópticas. A primeira considera programas como *funções* que mapeiam um estado inicial para um estado final; no caso não-determinístico, como *relações* entre estados inicial e final. Esta visão é particularmente apropriada para sistemas que aceitam todas suas entradas no início de sua operação e produzem suas saídas no fim de sua execução. Pnueli ([PNU 86], [HAR 85]) chama este tipo de sistema como *transformacional*, referindo-se à sua implementação como "transformadores de estados".

Por outro lado, existem sistemas que não podem ser tratados pela visão transformacional. Para exemplificar alguns destes sistemas, tem-se sistemas que idealmente nunca terminam sua execução, como: sistemas operacionais distribuídos, sistemas de controle de processo, redes de comunicação, bem como sistemas de interface homem-máquina. Além do mais, o propósito pelo qual estes sistemas encontram-se idealmente sempre em execução não é para obter um resultado final, mas para manter algum tipo de interação com o

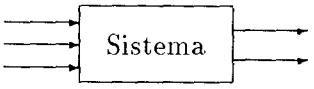

Figura 2.1: Sistema transformacional

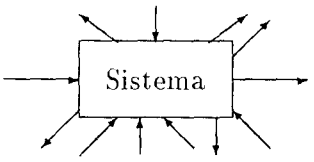

Figura 2.2: Sistema reativo

seu ambiente. Assim, refere-se a sistemas desta natureza como **sistemas reativos**.

Observa-se então que, sistemas reativos não podem ser adequadamente descritos através da especificação de um estado inicial e estado final. Uma descrição adequada deve contemplar o comportamento do sistema em um determinado instante através de seqüência (possivelmente infinita) de estados e eventos.

As figuras 2.1 e 2.2, apresentadas em [HAR 85], ilustram as definições de um sistema transformacional (como uma caixa preta) e sistema reativo (funcionando como um "cacto" preto, onde seus "espinhos", representados na figura pelas setas, identificam os elementos de sua interface que implementa a interação do sistema com o "mundo externo").

Outras tentativas para caracterizar sistemas desta última classe usam-se termos, tais como: sistema de tempo real, *embedded*, concorrente, distribuído, etc. Observa-se que em nenhuma destas caracterizações é possível

expressar a essência da **reatividade** do sistema. Reatividade caracteriza a natureza da interação entre o sistema e o seu ambiente. Isto mostra que esta interação não é restrita em transformar a entrada na saída quando o sistema termina a execução. Concorrência e distributividade, por outro lado, referem-se a organização interna do sistema. Um sistema reativo pode perfeitamente ser implementado por uma arquitetura seqüêncial ou por uma arquitetura concorrênte e/ou distribuída.

Assim, o problema em especificação e projeto de sistemas reativos reside na dificuldade de se descrever o seu comportamento de uma maneira clara, realista e que seja suficientemente adequada para uma análise computadorizada precisa.

Várias soluções foram propostas como tentativa para resolver este problema, como: redes de Petri [REI 85] , *Communicating Sequential Processes* (CSP) [HOA 78], *Calculus of Communicating System* (CCS) [MIL 80], ESTEREL [BER 85] e lógica temporal [PNU 86]. O enfoque mais recente para suportar este problema são os "estadogramas" [HAR 87], o qual revive o formalismo clássico de máquina de estados finitos (MEF's) e seu apêlo visual é esboçado por diagramas de transição de estados, tentando, com isso, torná-los apropriados para o desenvolvimento de sistemas grandes e complexos.

Os grupos que trabalham em sistemas desta natureza (i.e., sistemas reativos) tem deixado de usar MEF's convencionais e seus diagramas de estados, por diversas razões.

1. Diagramas de estados são originalmente "planos". Sendo assim, não proporcionam a noção natural de profundidade, hierarquia ou modularidade e, portanto, não suportam o desenvolvimento de sistemas usando metodologias descendentes, ascendentes ou passo-a-passo.

2. Diagramas de estados não são econômicos quanto ao número de transições necessárias para descrever sistemas. Tem-se geralmente um elevado no número de setas (representação gráfica das transições) quando se precisa representar transições em uma MEF's que tenha um grande número de estados. Este problema que poderiam ser aliviado se fosse possível representar estados em níveis diferentes de abstração.

3. Assim como nas transições, os diagramas de estados são também extremamente anti-econômicos em relação ao número de estados requeridos

na descrição de sistemas. Em um processo de desenvolvimento de sistemas, enquanto a especificação cresce linearmente, o número de estados cresce exponencialmente, quando da composição de sub-MEF's.

4. Finalmente, diagramas de estados são inerentemente sequenciais e não fornecem nenhum mecanismo natural para suportar concorrência.

A seguir é apresentado o conceito de "estadogramas" como uma maneira mais adequada para resolver estes problemas e suprir as deficiências encontradas em outros métodos.

O conceito de "estadograma" foi proposto originalmente por D. Harel, em [HAR 87], como um novo enfoque capaz de suportar essas deficiências enquanto preserva e estende o apêlo visual encontrado nos diagramas de estados convencionais. Estadogramas são uma extensão baseada em "hipergrafo"[1] dos diagramas de transição de estados padrão. Uma idéia básica de estadogramas está representada no quadro abaixo.

Estadogramas = diagramas de estados + hierarquia + ortogonalidade + comunicação

Nas subseções seguintes serão tratadas as principais características dos estadogramas. Outras características e extensões são apresentadas no apêndice A.

2.2 Abstração e Refinamento

A figura 2.3(a)[2], apresenta um exemplo de um estadograma, onde as arestas (ou setas) são rotuladas com eventos podendo estes estar associados a uma dada condição. As arestas representam as possíveis transições e os estados são representados por retângulos com cantos arredondados que D. Harel denomina de bolhas[3]. Nesta figura, são definidos três bolhas A, B e C e os

[1] *hypergraph*

[2] Serão utilizadas algumas figuras apresentadas em [HAR 87].

[3] *blobs*

eventos a, b, c e d. Caso o sistema representado pelo diagrama se encontre na bolha B, por exemplo, e ocorrer o evento a então o sistema muda da bolha B para a bolha A. A transição de A para C, por outro lado, só é efetuada se ocorrer o evento d e se for satisfeita a condição P concomitantemente. Com o propósito de padronizar uma terminologia neste texto e que se possa diferenciá-la em relação a diagramas de estados convencionais, decidiu-se chamar de bolha a um estado atômico (por exemplo, a bolha A na figura 2.3(a)) e de estado ao que Harel chama de configuração, que pode ser uma composição de várias bolhas, como mostrado mais a frente na figura 2.12.

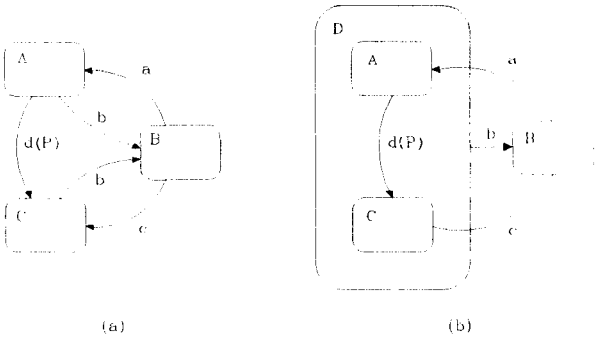

Figura 2.3: Abstração e Refinamento

A semântica associada a D é a de um *ou-exclusivo* (XOR) entre A e C, capturando a idéia de que se o sistema se encontrar na bolha D, o mesmo encontra-se em ou A ou em C, mas não em ambos. Desta forma D representa uma **abstração**[4] do subdiagrama composto por A e C. O estadograma da figura 2.3(b), portanto, é funcionalmente equivalente ao estadograma da figura 2.3(a). Na figura 2.3(b) existe um nível de abstração a mais representado por D. A figura 2.4 mostra uma outra forma mais abstrata de representar a figura 2.3(a). Em contrapartida a figura 2.3(a) representa um **refinamento** da figura 2.4. Reforçando, então, a transição rotulada com b na figura 2.3(b) permite sair dos sub-bolhas A ou C e, consequentemente, da bolha D, caso esta transição tenha sido acionada pela ocorrência do evento b.

Para identificar a bolha que representa o estado inicial em um dado

[4]Também chamada de **agrupamento**.

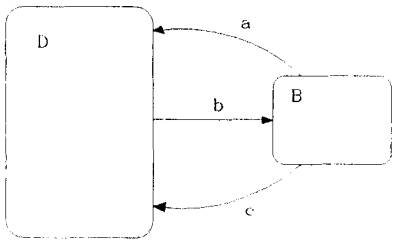

Figura 2.4: Abstração da figura 2.3(a)

contexto, ou seja a bolha *default*, com relação a A, B e C, são usadas notações especiais ilustradas a seguir. Para a figura 2.3(a) pode ser utilizada a notação da figura 2.5(a). Para figura 2.3(b), pode-se usar uma notação direta como na figura 2.5(b) ou em dois passos com na figura 2.5(c). Em todos os casos ilustrados na figura 2.5, o estado inicial à bolha A.

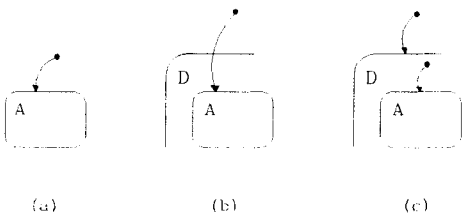

Figura 2.5: Representação de entrada *default*

2.3 History

Os estadogramas dispõem deste mecanismo bastante poderoso que permite a volta a bolhas já visitadas em passado recente. É apresentado a seguir este mecanismo e as extensões feitas à proposta de Harel.

2.3.1 History Hierárquico

Este tipo de *history* representa uma maneira bastante interessante e muito adequada para retornar a bolhas em níveis mais baixos de uma hierarquia nos quais o sistema se encontrava imediatamente antes da sua mais recente saída destes níveis. Refere-se a nível à relação hierárquica existente entre bolha e sub-bolhas. Assim, uma bolha sempre é contida em um nível imediatamente superior por apenas uma bolha, exceto a mais externa. Neste caso, o *history* pode ser representado por dois tipos de entradas: *entrada por history (H)* e *entrada por history estrela (H*)*. No caso de entrada simples por *history*, o sistema entra na bolha mais recentemente visitado no nível em que este modo foi especificado. A figura 2.6(a) ilustra este mecanismo: o sistema ao entrar na bolha F entra na mais recentemente visitada. Por exemplo, entraria em D se esta tivesse sido a último bolha visitada quando o sistema esteve em F pela última vez. Da mesma forma, podemos usar a notação equivalente da figura 2.6(b). Portanto, convém ressaltar que o *history (H)* só é aplicado no nível no qual este é especificado.

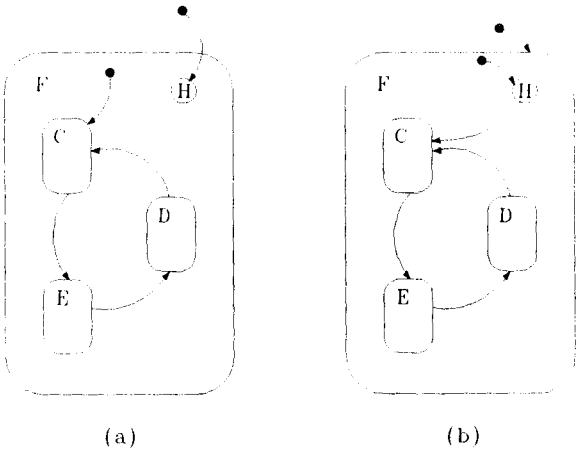

Figura 2.6: Entrada por *History*

A entrada *history estrela* é aplicada a todos os níveis inferiores da hierarquia do estadograma onde está especificado. A diferença entre as duas entradas será ilustrado através dos estadogramas da figura 2.7.

A entrada *history* na figura 2.7(a), seleciona somente entre G e F; isto é,

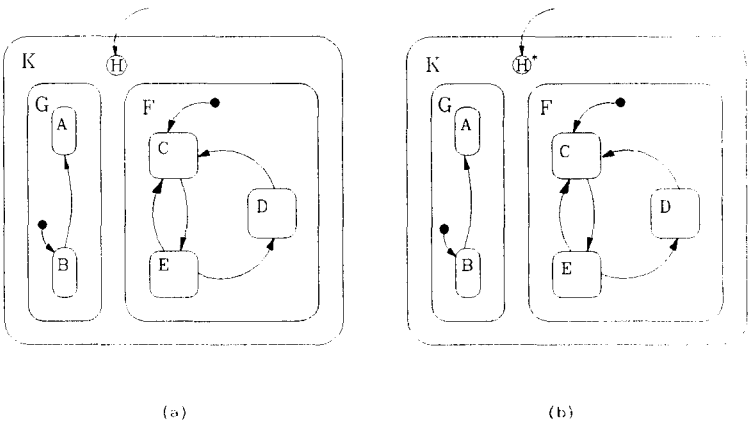

Figura 2.7: Representação do *History* estrela

o sistema entra em B independentemente se este estava anteriormente em A ou B ao deixar K, e por outro lado entraria em C se o sistema estivesse em C, D ou E no caso da última visita, pois B e C são as sub-bolhas *default* de G e F, respectivamente. Na figura 2.7(b), por outro lado, o sistema entraria no estado mais recentemente visitado dentre todos os níveis da hierarquia, desde A a E, ignorando portanto o estado *default*. Para reforçar o uso do *history*, na figura 2.8 o sistema entrará em B se o último visitado de K foi G, ou entrará no último estado visitado de F, podendo ser qualquer um dos últimos visitados entre C, D ou E. A primeira sub-bolha de F a ser visitada, contudo, é C por ser o estado *default* e esta é uma maneira de ignorar a condição de estado inicial (estado *default*).

2.3.2 History de Reentrância

Observa-se, na figura 2.10(a), que a saída c não corresponde ao refinamento da figura 2.9. Na figura 2.10(a), o evento c se aplica a todo o conjunto de sub-bolhas de D e só na sub-bolha L ocorre a transição para a bolha M. Portanto, não reflete a representação real da figura 2.9. Assim, uma abstração para este caso seria a representação da figura 2.10(b), a qual ilustra o fato que c se aplica a um subconjunto de sub-bolhas de D e não a todas sub-bolhas de D. A transição rotulada com b se aplica a todas as bolhas em todos os sub-níveis de D.

A seta associada a um *history* na figura 2.9, rotulada por d, é utilizada para representar este novo mecanismo, chamado aqui de *history de "reentrância"*. Suponha-se que, na figura 2.9, ocorra o evento d, estando o sistema em qualquer sub-bolha de D, causa a saída do subestado corrente (mas não a saída da bolha D) e entrada imediata para o mais recentemente visitada sub-bolha, (i.e., a bolha recém deixada). Por exemplo, suponha na bolha D, a sub-bolha corrente, em um determinado instante, seja a sub-bolha G e, então, seja disparado o evento d. Neste caso o sistema sai da sub-bolha G e volta a reingressar na mesma. Este mecanismo representado pela seta d, economiza no caso deste exemplo seis setas, uma para cada sub-bolha de D.

2.3.3 History de Retorno

Como será visto com mais detalhes no capítulo 4, onde é apresentado um exemplo de uma aplicação da ferramenta apresentada neste trabalho, onde precisou-se de uma facilidade a mais que o mecanismo de *history* descrito até aqui pudesse fornecer. O problema surgiu na implementação do Editor Topológico da LegoShell [PIÑ 90], quando precisava-se de um mecanismo com o qual fosse possível voltar às bolhas anteriormente visitadas "lateralmente", ou seja não na sua hierarquia, mas entre bolhas de mesmo nível (i.e., bolhas "irmãs"). Assim, de acordo com as necessidade e a especificação do problema de H. Piñon para a implementação do referido editor, foi criado um novo tipo de *history*, denominado aqui de *history de retorno*. Este *history*, representado pela seta (R), é exemplificado pela figura 2.11.

Inicialmente, o sistema ao entrar em T e em A, pelo mecanismo de entrada *default*, em seguida ocorre o evento a e o sistema vai para F. Se o evento a é novamente disparado, então o sistema muda para o estado N e assim por diante. Neste caso, quando existe um *history* de retorno, e é acionado o evento associado, no caso em função do disparo de k, o sistema retorna para o estado anteriormente visitado do mesmo nível de hierarquia em que o *history* de retorno está especificado. Por exemplo, suponha o que o sistema tivesse chegado à bolha N; assim, se ocorresse o evento k, o sistema *retornaria* para o vizinho (ou irmão) anterior de N que neste caso particular é o estado F e dentro deste o sistema entraria para a sua ex-sub-bolha corrente. Obviamente, se o sistema acaba de entrar em T e encontra-se na bolha A e ocorre o evento k nenhuma mudança de estado é realizada. No capítulo 3, é detalhado melhor este mecanismo e como foi implementado.

2.4 Ortogonalidade: Independência e Concorrência

Nesta seção introduz-se uma relevante propriedade de um estadograma em relação ao refinamento à decomposição em sub-bolha concorrentes, ou seja, estando em uma bolha desse tipo, significa estar em todas as suas sub-bolhas. A figura 2.12, mostra uma bolha X de componentes Y e Z, capturando a propriedade de que estando em X significa estar em um estado de Y (L, I ou S) e ao mesmo tempo em outro estado de Z (A ou F). Neste caso X é um produto ortogonal de Y e Z. Portanto, quando entra-se em X sem nenhuma informação adicional, entra-se na combinação (L, A), ou seja nas bolhas L e A por serem bolhas *default* das componente Y e Z, respectivamente. Logo, se o evento a ocorre, o sistema muda do estado L para I e de A para F concomitantemente, resultando em uma nova combinação (I, F). Neste caso acontece um certo tipo de *sincronização*, onde um simples evento causa acontecimentos simultâneos, no caso, mudança de bolhas. Mas, se ocorre o evento c, este afetaria somente a componente Y resultando na mudança de L para S, ou seja, teremos a combinação (S, A). Neste caso, existe um certo grau de *independência* entre as duas componentes de X, visto que a transição provocada pelo evento c afeta somente a componente Y, mas não a Z. Este exemplo denota o que é chamado de *ortogonalidade* entre componentes.

A figura 2.13[5] é uma forma convencional de se representar funcionalmente 2.12. Observa-se que na figura 2.13 precisou-se seis bolhas para representar a figura 2.12, porque a bolha X da figura 2.12 têm dois e três bolhas em suas componentes. É importante observar que, se estas componentes tivessem, ao invés de dois e três bolhas, mil bolhas em cada componente, então teria-se um milhão de bolhas na representação que não se utiliza de componentes.

Nota-se, portanto, um crescimento exponencial no número de bolhas, quando se usa automato de estados finitos, máquina de estados finitos, ou diagramas de estados. Assim, a ortogonalidade é um excelente mecanismo explorado pelo estadograma para evitar este problema.

Na figura 2.12, a transição rotulada com o evento b de I para L está associada uma condição *em* F que, na figura 2.13, é representada por uma seta de (I, F) para (L, F) rotulada com b. Como X foi dividido em duas compo-

[5] As figuras 2.13 e 2.12 consistem de uma adaptação as apresentadas em [HAR 87].

nentes ortogonais, este tipo de condição causa, portanto, dependência entre componentes. No caso particular, a condição *em F* causou uma dependência em relação ao comportamento de *Z*.

Em geral, em exemplo reais, bolhas providas de componentes encontram-se em um contexto maior. Assim, para representá-las graficamente usa-se as formas apresentadas nas figuras 2.14(a) e (b).

2.5 Ações e Atividades

Conforme descrição acima, o estadograma "puro" representa apenas o controle do sistema. Até agora a reatividade do sistema foi expressa somente por mudança de estado decorrente de transições entre bolhas, por seleção de eventos e o atendimento a possíveis condições associadas a estes. A seguir, são apresentadas outras importantes características incorporadas a estadogramas que facilitam associar semântica aos mesmos.

Para prover o estadograma com facilidades de execução de tarefas, denominadas **ações**, quando da transição entre bolhas, é adicionada aos rótulos das transições a notação ".../A", onde "A" indica a ação a ser disparada por ocasião da transição em questão. Ações também podem estar associadas a bolhas, isto é, pode-se especificar ações que podem ser executadas quando o sistema "entra" em uma determinada bolha, chamada de ações do tipo *on_entry* e ações que podem ser executadas quando o sistema "deixa" uma determinada bolha, chamada de ações do tipo *on_exit*.

Para inúmeras aplicações somente estes tipos de ações, não é suficiente. Desta forma é introduzido o conceito de **atividade**. Uma atividade é uma tarefa que é executada enquanto o sistema permanece em uma determinada bolha, i.e. um processo associado à atividade é disparado quando o sistema chega à bolha em questão e o processo é desativado por ocasião da saída desta bolha. Atividades são identificadas em estadogramas pelo termo *throughout*.

Estas novas características serão exemplificadas melhor através da figura 2.15. Supondo-se que o sistema inicialmente encontra-se no estado (C, D); supondo-se ainda que o evento a ocorra, então o novo estado será (B, D), sendo desta forma executadas as ações S e V ao entrar em A e B, respectivamente. Obviamente, se ocorre a transição de B para F, a ação S não será executada novamente. Mas, se ocorrer o evento d, o sistema deixa B e A, e executa as ações w e T e também causa a geração do evento b

associada a *on_exit* de A, que provoca transições na componente ortogonal de D para E, tornando a nova configuração (C,E) e executando a ação U na entrada de E. A atividade Z será executada enquanto o sistema estiver na bolha A.

2.6 Comentários Gerais

- Como já mostrado, os estadogramas são uma extensão do modelo de máquinas de estados finitos e estas são extremamente limitadas para especificação de sistemas, já que tendem a um crescimento exponencial do número de estados quando a complexidade do sistema modelado pela mesma cresce. Este problema, por outro lado, é facilmente resolvido pelo estadograma através da ortogonalidade entre bolhas componentes.

- Estadograma são extramente adequado para construção de sistemas reativos, no qual o sistema é caracterizado pelo grande poder de interação com o ambiente (figura 2.2).

- Observa-se que o enfoque por estadogramas permite separar a componente de controle da aplicação, sendo a sua semântica definida pelo *script* de ações e atividades associadas a transições e bolhas. A separação do código da aplicação do controle é um requisito cada vez mais pretendido pelos sistemas modernos. A esse procedimento, como visto, é chamado de independência do diálogo.

- Acredita-se que a descrição do controle usando estadogramas através de suas características de concorrência e hierarquia facilita a sua descrição e conseqüentemente o entendimento e a análise, tornando, o processo de desenvolvimento do sistema também mais fácil.

- O conceito de estadograma se constitue em um enfoque extremamente fácil de estendê-lo para suportar novas características necessárias para o desenvolvimento de sistemas reativos.

- Na literatura é encontrada a aplicabilidade de estadogramas em diversas áreas, desde para descrição de hardware ([DRU 89], [DRU 85]) até para especificação de sistemas reativos ([MEI 91], [HAR 90], [iLO 87]).

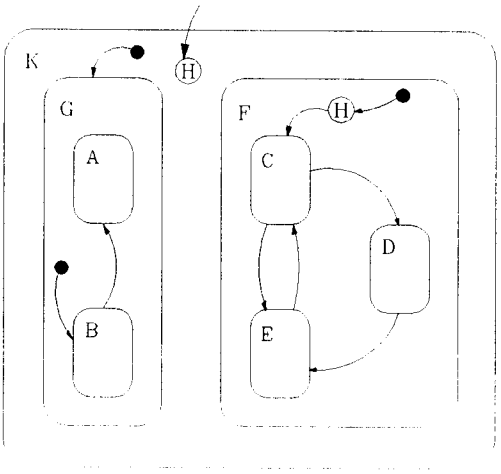

Figura 2.8: Aninhamento de *histories*

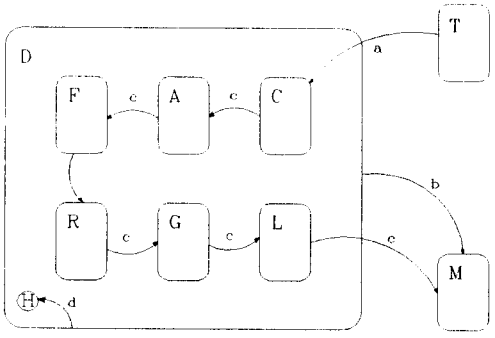

Figura 2.9: *History* de reentrância

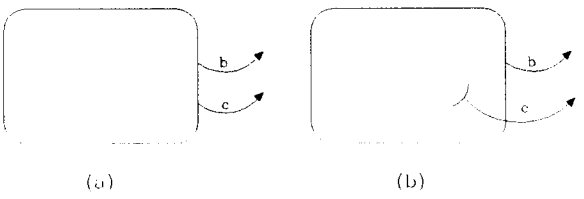

Figura 2.10: Abstração da figura 2.9

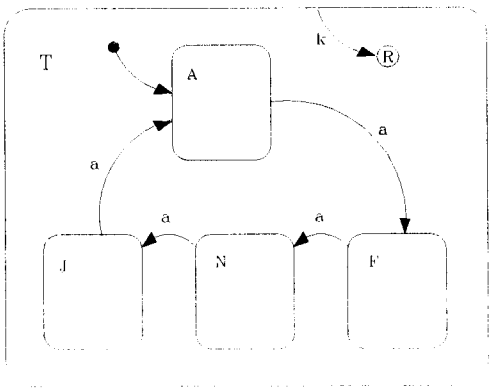

Figura 2.11: *History* de retorno

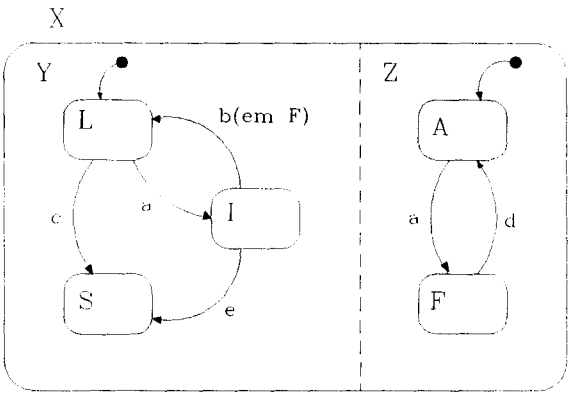

Figura 2.12: Decomposição concorrente de estadograma

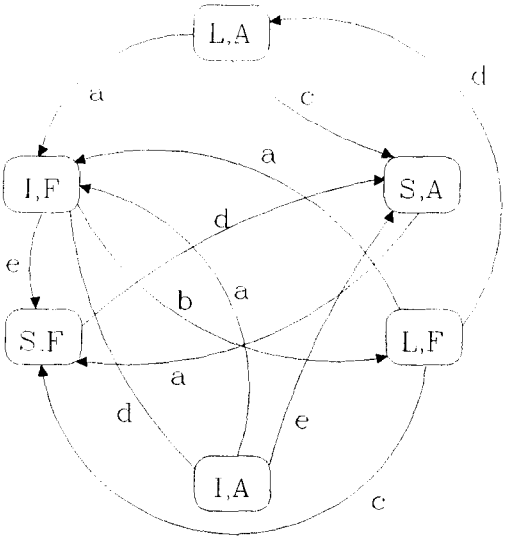

Figura 2.13: MEF's da figura 2.12

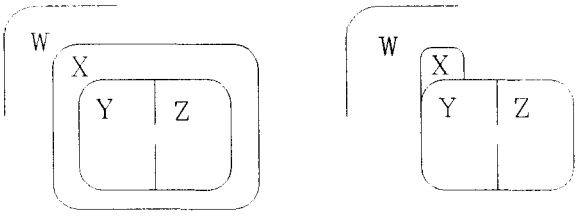

Figura 2.14: Notação da abstração de uma componente em estadograma

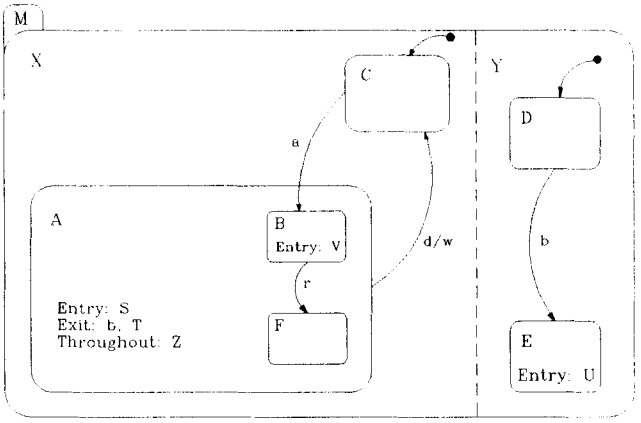

Figura 2.15: Ações e atividades

Capítulo 3

Arquitetura da Geração de Código e Implementação

Neste capítulo é apresentada a linguagem descritiva de estadogramas através de regras sintáticas acompanhadas de exemplos e através de programas ilustrativos simples. Em seguida descreve-se o processo de geração de código do tradutor e finalmente são tratados alguns detalhes de sua implementação.

3.1 Introdução

O presente trabalho consiste no desenvolvimento de um *Gerador de Gerenciadores de Sistemas de Controle*[1] baseado na notação de estadogramas. O gerenciador, descrito a seguir e apresentado em [FIG 90] e [FIG 91], recebe como entrada uma especificação definida através de estadogramas, ou seja, um programa escrito em uma linguagem descritiva de estadogramas e gera como saída um programa escrito na linguagem de programação C funcionalmente equivalente a esta especificação.

Este processo de tradução é realizado como esquematizado na figura 3.1. Por questões práticas, inicialmente é feita a tradução para linguagem C [KER 78], que se constitui o objetivo principal deste trabalho, e em seguida é feita a compilação do programa gerado em C. Portanto, o gerador em questão comporta-se como um tradutor, ou seja, um compilador no qual sua

[1] Em algumas situações, neste texto, associa-se frequentemente o termo **tradutor** ao Gerador de Gerenciadores de Sistemas de Controle.

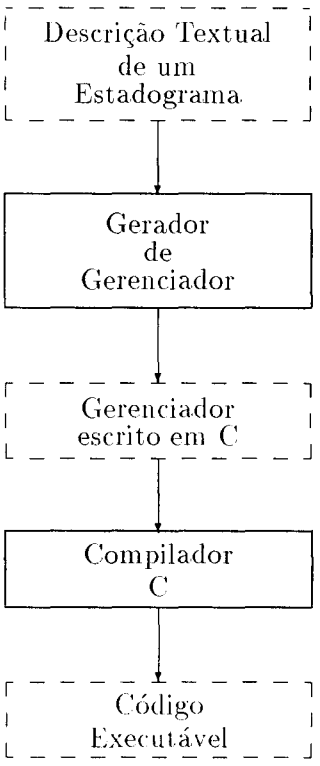

Figura 3.1: Esquema do processo de tradução

saída se constitui de um código-fonte em uma outra linguagem ao invés de um código executável. O sistema completo será composto por um editor de gráfico de bolhas[2] que produz como saída a descrição textual do estadograma especificado, como mostrado na figura 3.2.

Nas seções seguintes apresenta-se a definição da gramática da linguagem de descritiva de estadogramas e a estrutura dos programas gerados.

[2]Projeto que está sendo desenvolvido por V. Elias.

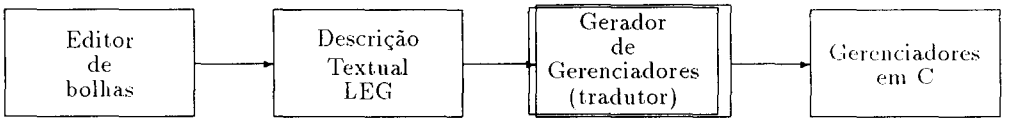

Figura 3.2: Descrição Esquemática da ferramenta proposta

3.2 A Linguagem Descritiva de Estadogramas

3.2.1 Estrutura da Linguagem

A Linguagem Descritiva de Estadogramas, ou simplesmente Linguagem de EstadoGramas (LEG), permite denotar através de uma representação textual, as características inerentes a estadogramas, como: hierarquia, concorrência e comunicação. Contudo, esta linguagem tem como propósito permitir que o processo de desenvolvimento de sistemas reativos complexos se torne bastante facilitado e, consequentemente tornando mais rápido a implementação de sistemas com controle complexo.

Uma aplicação importante desta linguagem é suporte o controle de diálogo (i.e., a componente de controle do diálogo como foi visto na figura 1.1) de aplicações interativas.

Um programa escrito em LEG associa código inerente a funcionalidade da aplicação através de funções escritas em linguagem C ao código próprio de LEG responsável pelo controle desejado para a aplicação. As partes do código em C, posteriormente, serão "ligadas"[3] com o código gerado na segunda fase deste processo de tradução. Em síntese, os comandos desta linguagem são direcionados para controle de aplicações, enquanto que as funções em C estarão associadas à possíveis ações disparadas por ocasião da entrada, da saída de uma bolha[4] ou simplesmente disparadas por um determinado evento quando da transição de uma bolha para outra. Ou seja, este código define de fato os aspectos funcionais da aplicação especificada. Esta separação permite ter-se um certo grau de independência de controle e funcionalidade.

[3] *linked*

[4] Como já dito na seção 2.5, na entrada e/ou saída de uma dada bolha podem ser acionadas ou desativadas ações e/ou atividades.

Os comandos de LEG são destinados basicamente a identificação de ações, transições e eventos, assim uma seqüência de comandos em geral é delimitada por { e } definindo a especificação de uma determinada bolha ou transição. LEG faz distinção entre letras maiúsculas e minúsculas como em C, ou seja, por exemplo: "EXIT" é diferente de "exit".

Utiliza-se, na seção seguinte, uma extensão do formalismo de Backus-Naur (BNF)[NAU 63] para descrever a sintaxe desta linguagem. BNF é um formalismo destinado a descrição de sintaxe de uma determinada linguagem. Assim, para construção envolvida por { e } significa que esta pode ter 0 ou repertidas ocorrencias e para construção envolvida por [e] significa que esta pode ser tratada opcionalmente.

3.2.2 Sintaxe de LEG

Para melhor compreensão da estrutura de LEG, define-se a seguir a sua sintaxe, tomando-se desde os símbolos mais básicos em LEG até definição de um possível programa. No apêndice B são apresentadas as produções da gramática de LEG e no apêndice C é apresentada esta gramática em Yacc, utilizada na implementação do tradutor. A definição da sintaxe de LEG, que segue abaixo, apresenta para cada produção exemplos e descreve a semântica associada. As restrições semânticas são tratadas pelo tradutor que emite as possíveis mensagens de erro, apresentadas no apêndice E.

- **Letra**

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

- **Dígitos**

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

  Letras e dígitos não têm nenhum significado inerente. Estes são utilizados exclusivamente para formar identificadores.

- **Delimitadores**

  <delimitadores> ::= { | } | |

  Os delimitadores { e } são utilizados para indicar o início e término da definição de uma bolha, no âmbito de um código LEG. O significado associado a | é para identificar que as bolhas separadas por este delimitador são componentes concorrentes.

Características tipográficas como espaço em branco, caracter de tabulação ou de mudança de linha não tem significado na linguagem. Portanto, estes podem ser utilizados livremente para facilitar a leitura do código.

- **Identificadores**

  - **Sintaxe**

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

  - **Exemplos**

    xy
    Agff
    La64

  - **Semântica**

    Os identificadores são símbolos definidos pelo programador utilizados para identificar nomes de bolhas e eventos. A atual versão do tradutor não permite a identificação de dois ou mais estados e/ou eventos em pontos diferentes de um estadograma com o mesmo nome. Em LEG alguns símbolos são "reservados", ou seja, estes não podem ser utilizados como identificadores de bolhas e/ou eventos, mas somente em declarações de comandos apropriados ou dentro de comentários. LEG faz distinção entre letras maiúsculas e minúsculas.

- **Expressões**

  Não existem expressões como construção LEG. Veja a definição do comando if de transition.

- **Comandos gerais**

  Os comandos em LEG são classificados em comandos de declarações, comandos de referência a ações e comandos relativos ao controle propriamente dito.

  - **Sintaxe**

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

  Os comandos <history> e <transições> estão associados a aspectos de controle do sistema enquanto que os comandos <ações> permitem a associação de código em C escrito pelo usuário a atributos de bolhas

relacionados a ações a serem disparadas por ocasião da entrada e saída destas bolhas.

- **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**

    Os tipos de history possíveis são assim definidos:

    * **deep** - usado para especificar a existência do *history estrela*, ou seja, o *history* cujo efeito é sentido em todos níveis inferiores da hierarquia (veja definição na seção 2.3.1);
    * **shallow** - define um *history* comum, isto é, o *history* aplicado somente ao nível da hierarquia no qual está especificado (definido na seção 2.3.1);
    * **pred**[5] - define a existência do *history* com retorno. Este tipo de *history*, como definido na seção 2.3.3, permite o retorno a bolha irmã da bolha corrente anteriormente visitada.

  - **Exemplos**

    history deep, pred;

    history shallow;

  - **Semântica**

    É bom salientar que os tipos *histories* `deep` e `shallow` não podem, evidentemente, ser especificados para uma mesma bolha.

  - **Nota**

    O *history* de reentrância ainda não encontra-se disponível na atual versão do tradutor, mas sua implementação foi realizada no modulo `engine.c`, que é parte integrante do código gerado a ser visto mais a diante.

- **Comandos de ações**

[5] *Pred* vem da palavra *Predecessor*

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

- **Sintaxe**

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

  O código em C associado às ações, como tratados na seção 2.5, são responsáveis pelos *script* executado pelo sistema, ao entrar, ao se encontrar e ao sair de uma determinada bolha, respectivamente.

- **Exemplo**

  on_entry : [circle( );] ;

  on_exit : [clear( );] ;

  throughout : [UpdateChart( );];

- **Semântica**

  As ações a serem executadas deverão estar encapsuladas por uma função, ou seja, assume-se que o código em C encontrado no programa fonte serão chamadas de função, como no exemplo dado as funções escritas em C `circle()`, `clear()` e `UpdateChart()`. Nenhuma verificação se o texto fornecido como sendo código C o é de fato. Para efeitos do tradutor o texto é manipulado como sendo uma cadeia de caracteres apenas a ser enxertada em lugar apropriado no código gerado ao final deste processo.

- **Comandos de controle**

  O comando transition permite a especificação da mudança do estado do sistema através de transições entre bolhas encapsulando comandos de decisão e de disparo de ações. Assim, opcionalmente é possível efetivar testes em relação às componentes concorrentes para transições condicionais em relação ao comportamento estado geral do sistema, através de expressões lógicas destinadas para este fim, como mostrado abaixo.

  - **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>

No comando **if** é permitido definir transições condicionais, podendo-se verificar se o sistema encontra-se concomitantemente em bolhas de outras componentes concorrentes. Este procedimento é feito através de invocações à função *in_Blob()* tendo como parâmetro o nome do estado em questão, neste teste pode-se utilizar os operadores lógicos da linguagem C, como &&, || e !. O comando **do** é utilizado opcionalmente, assim como o **if**, caso exista uma ação associada à transição propriamente dita, como exemplificado na figura 2.15 pela transição de A para C, que dispara a ação w na ocorrência do evento d.

- **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(); ];
}
```

- **Semântica**

A expressão que faz parte do comando **if** é simplesmente copiada para o código gerado, ou seja não são verificados eventuais erros sintáticos em C, mas é verificado se as bolhas especificadas como parâmetro de *in_Blob()* são bolhas válidas pertencentes a um conjunto de bolhas potencialmente correntes. Caso contrário, será emitida uma mensagem de erro pelo tradutor.

- **Definição de uma bolha**

A especificação de bolhas em LEG é feita de duas maneiras: uma onde o bolha é definida (expressada por um comando **blob**) e outra onde é feita uma referência a uma definição (expressada por um comando **call**) que pode estar definida no mesmo arquivo do programa principal ou em outro incluído pelo comando "INCLUDE" no programa principal ou no módulo que contém o comando **call**.

- **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> ;

- **Exemplo**

  Veja código de um programa escrito em LEG na seção 3.3 para o exemplo apresentado na figura reffig22.

- **O programa principal**

  O programa principal representa o corpo (ou a estrutura) da bolha mais externa do estadograma. Assim as construções sintáticas válidas para uma bolha também são válidas para o programa principal.

  - **Sintaxe**

    <bolha principal> ::= main <bolha em LEG>

  - **Exemplo**

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

  - **Nota**

    Obrigatoriamente todo programa em LEG, deverá ter um programa 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;
}
```

- **Nota**

  Os arquivos incluídos por um programa em LEG devem ter extensão **.b**. Caso contrário, serão ignorados. É permitido que os arquivos incluídos possam ter outros comandos de inclusão e assim por diante.

- **Bolha "default"**

A bolha "default", como ilustrado na figura 2.5 é caracterizada em um programa escrito em LEG como sendo a primeira bolha definida dentro da bolha que a contém.

- **Exemplo**

```
blob Worksheet
{
```

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

Assim, no exemplo acima a bolha *Processing* será reconhecida automaticamente pelo tradutor como sendo a bolha "default" da bolha *Worksheet*, uma vez que esta foi a primeira a ser definida.

- **Comentários em LEG**

  Os comentários servem meramente como documentação e auxílio ao programador e, portanto, são ignorados pelo tradutor. O comentário pode se estender por várias linhas, podendo ser utilizado para extensas descrições e torna-se útil quando deseja-se remover temporariamente trechos de códigos do programa em LEG. Os comentários, são delimitados pelos símbolos compostos /* e */. Dentro de um comentário poderá ocorrer qualquer seqüência de caracteres, exceto o símbolo composto */ que finaliza comentário.

3.3 Estrutura dos Programas Gerados

Um gerenciador construído pelo tradutor, objeto deste trabalho, constitui-se de uma ferramenta que efetua mudanças do estado global de um estadograma em função da ocorrência de eventos e executa as ações associadas às bolhas envolvidas nas mudanças efetuadas. São introduzidos, a seguir, alguns aspectos sobre a estrutura de dados do gerenciador e sua estrutura de controle.

O gerenciador mantém, uma lista duplamente ligada de identificadores de bolhas, nas quais o sistema se encontra, representando esta lista, portanto, o estado global do gerenciador em um determinado instante. Esta lista é manipulada sempre que ocorre alguma mudança no estado global retirando-se da lista os identificadores das bolhas dos quais o sistema "sai" em função

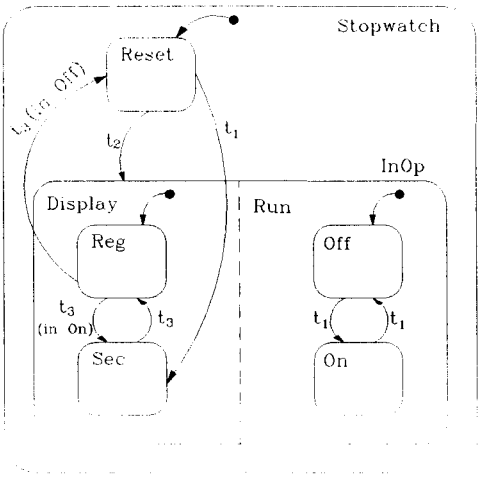

Figura 3.3: Estadograma de Cronômetro

de um dado evento e são adicionados os identificadores das bolhas nas quais o sistema "entra", sendo executado em cada caso as ações associadas.

Além da lista ligada para manter o estado global do sistema, o gerenciador mantém uma árvore que contém informações sobre a estrutura do estadograma a partir do qual o mesmo foi gerado. A estrutura da árvore é isomorfa ao encaixe das bolhas. Uma dada bolha sempre é contida em um nível imediatamente superior por apenas uma bolha, exceto a mais externa. O gerador implementado não suporta o compartilhamento de bolhas como sugerido em [HAR 89] e [HAR 88] e descrito no apêndice A. Desta forma, uma bolha que contém outra é representada por um nó na árvore e a contida como filho deste nó.

A figura 3.3 que ilustra o estadograma que descreve a operação de um cronômetro descrito funcionalmente por Harel [HAR 87]. Este cronômetro, no artigo original, faz parte da descrição de um relógio digital. Este exemplo está sendo utilizado por ser de fácil entendimento e por ilustrar bem algumas características de estadograma.

O cronômetro, quando acionado, entra no estado (`Stopwatch. Reset`)[6] com a ação associada de zerar o cronômetro[7] dada a indicação da entrada *default*. O cronômetro sai deste estado caso ocorra o evento t_1 ou o evento

[6] O estado global é representado no texto por uma lista de bolhas.

[7] As ações e atividades associadas a bolhas não são representadas na figura.

t_2 (aperto de uma tecla de função, por exemplo). No caso de t_1, a bolha `Sec` da componente `Display` (que representa a forma de visualização de tempo cronometrado – `Sec`: em segundos; `Reg`: em termos de horas, minutos e segundos) da bolha `InOp` é atingida concomitantemente com a bolha `Off` da componente `Run` (que representa o acionamento e a suspensão da contagem de tempo pelo cronômetro) da bolha `InOp` de acordo com a especificação da entrada `default` desta última componente. Após a ocorrência do evento t_1, o estado global transforma-se em (`Stopwatch, InOp, Display, Sec, Run, Off`). Na mudança são executadas as eventuais ações associadas à saída da bolha `Reset` e às entradas nas bolhas `InOp, Sec` e `Off`. A eventual atividade associada a `Reset` é desativada e as eventuais atividades associadas a `InOp, Sec` e `Off` são iniciadas. Caso o sistema se encontrar em `Reset` e ocorrer o evento t_2 então o estado global muda de (`Stopwatch, Reset`) para (`Stopwatch, InOp, Display, Reg, Run, Off`) de acordo com a especificação das entradas `default` das componentes de `InOp`.

A componente `Run` opera de modo independente da componente `Display`. A mudança de uma de suas bolhas para outra se dá na ocorrência do evento t_1. A componente `Display` tem o seu comportamento sujeito à ocorrência do evento t_3 e, quando em `Reg`, além da ocorrência de t_3, também depende do estado da componente `Run`. Caso a componente `Display` se encontre em `Reg` e ocorrer o evento t_3 então ocorre uma mudança para `Sec` se a componente `Run` se encontrar em `On`. Se, por outro lado, a componente `Run` estiver em `Off`, o sistema abandona `InOp` saindo de ambas as componentes e entra em `Reset` passando do estado (`Stopwatch, InOp, Display, Reg, Run, Off`) para (`Stopwatch, Reset`).

3.3.1 Programando em LEG

Programas em LEG têm, basicamente, uma estrutura aninhada que refletem a estrutura dos estadogramas quanto ao encaixe de bolhas. Para tornar mais clara a visão de programas escritos em LEG, apresenta-se um simples programa do exemplo mostrado na figura 3.2. Um programa principal para este exemplo seria.

```
#include "display.b"
#include "run.b"
```

```
main blob Stopwatch
{
  on_entry : [DrawWatch(); ];
  on_exit  : [ByeWatch(); ];

  blob Reset
  {
    on_entry : [ ResetTime();];
    transition {
                on_event(t1) to Sec;
                on_event(t2) to InOp
               }
  }
  blob InOp
  {
     call Display; |
     call Run;
  }
}
```

Os arquivos, display.b e run.b incluídos no início do programa principal são descritos a seguir:

Arquivo **display.b**

```
blob Display
{
  on_entry: [InOperationMode();];
  blob Reg
  {
    on_entry:[ShowTimeAsReg();];
    transition{
                on_entry(t3) if [in_Blob(On)] to Sec;
                on_entry(t3) if [in_Blob(Off)] to Reset
              }
  }
  blob Sec
  {
    on_entry: [ShowTimeAsSec();];
    transition {
```

```
                on_event(t3) to Reg;
              }
  }
}
```

Arquivo **run.b**

```
blob Run
{
  blob Off
  {
    transition { on_event(t1) to On }
  }
  blob On
  {
    on_entry : [SetTime();];
    throughout : [BlinkingMessage();]
    transition {
                on_event(t1) to Off
              }
  }
}
```

Segundo os comandos **on_entry**, **throughout** e **on_exit**, neste exemplo, tem-se chamadas de funções em C que deverão estar em outros arquivos definidos pelo usuário para serem posteriomente "ligados" no segundo passo do processo de compilação. A seguir, descreve-se o código gerado pelo tradutor a partir de uma especificação dada por um programa em LEG.

3.3.2 O Código Gerado

A estrutura da árvore que representa o encaixe de bolhas do estadograma da figura 3.3 é apresentado na figura 3.4. Os nós de 1 a 9 correspondem, respectivamente, às bolhas e aos componentes `Stopwatch`, `Reset`, `InOp`, `Display`, `Reg`, `Sec`, `Run`, `Off` e `On` no estadograma. Os nós 4 e 7 são representados de forma distinta dos demais pois representam componentes e não bolhas. No código gerado, as bolhas são definidas através de macros no arquivo `bstates.h`[8] como mostrado abaixo:

[8] Uma descrição dos arquivos gerado é apresentada no apêndice E.

```
#define Stopwatch  1;
#define Reset      2;
#define InOp       3;
#define Display    4;
#define Reg        5;
#define Sec        6;
#define Run        7;
#define Off        8;
#define On         9;
```

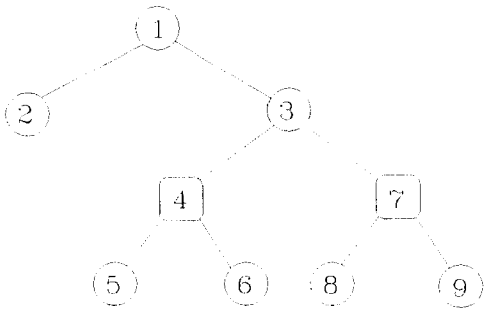

Figura 3.4: Estrutura da árvore representando a hierarquia do estadograma

A estrutura da árvore que representa uma hierarquia de bolhas não se altera durante a execução do programa gerado. Assim sendo uma árvore deste tipo é implementada por um vetor de estruturas composto por $n+1$ elementos, onde n representa o número de nós da árvore. A estrutura utilizada para representar os nós das arvores no vetor é composto de seis campos que contém informações que facilitam a navegação pela árvore. O primeiro campo corresponde ao primeiro filho do nó representado, o segundo aponta para o próximo irmão do nó em questão e o terceiro ao seu pai. O quarto campo é utilizado para memorizar a mais recente descida de um pai para um determinado filho. Esta informação é necessária para implementar o mecanismo de *history*. O quinto campo corresponde ao *status* dos *history* identificado através dos caracteres como "*" para *history estrela* , "h" para o *history* aplicado somente no primeiro nível, "b" corresponde ao *history* de retorno ao irmão, "#" indica a ocorrência de "b" e "*" e "^" indica "b" e

"h". O sexto campo, é gerado inicialmente com valor "NULL" e representa um apontador para uma lista simplesmente ligada que armazena as bolhas mais recentemente visitadas de mesmo nível quando na existência de *history* com retorno.

Para ficar mais clara, a diferença entre as informações contidas nos quarto e sexto campos, o quarto refere-se a informação necessária para efetuar o retorno à última bolha visitada na hierarquia e o sexto para efetuar o retorno a uma determinada bolha que seja "irmã" anteriormente visistada da bolha corrente.

Para o estadograma da figura 3.3 o vetor gerado que representa sua hierarquia é dado abaixo. Neste vetor foi adicionado ao segundo elemento, na representação da figura 3.3, no quinto campo um "*", identificador de *history estrela*, para ilustrar melhor o exemplo.

```
 lista_global *hierarchy[10] =

{
 {       0,          0,          0, 0, '0', NULL}, /* NULL     */
 {   Reset, Stopwatch,          0, 0, '*', NULL}, /*Stopwatch*/
 {       0,       InOp, Stopwatch, 0, '0', NULL}, /*Reset    */
 {-Display,      Reset, Stopwatch, 0, '0', NULL}, /*InOp     */
 {     Reg,       -Run,       InOp, 0, '0', NULL}, /*Display  */
 {       0,        Sec,  -Display, 0, '0', NULL}, /*Reg      */
 {       0,        Reg,  -Display, 0, '0', NULL}, /*Sec      */
 {     Off,  -Display,       InOp, 0, '0', NULL}, /*Run      */
 {       0,         On,       -Run, 0, '0', NULL}, /*Off      */
 {       0,        Off,       -Run, 0, '0', NULL}, /*On       */
}
```

O sinal "-" indica que o nó sendo apontado representa uma componente e não uma bolha. O primeiro elemento do vetor acima é composto por valores 0 e não é utilizado, pois em C todas estruturas indexadas iniciam com índice 0 e este valor representa neste vetor o apontador nulo. As listas de irmãos são circulares. Inicialmente o quarto valor de cada elemento, que representa os nós da árvore, é 0. Supondo-se que o cronômetro é acionado e que, em seguida, ocorra o evento t_1, então a seqüência dos valores na quarta posição de cada elemento ficaria 0, `InOp`, 0, `-Display`, `Sec`, 0, 0, `Off`, 0, 0

armazenando informações necessárias para efetuar um posterior retorno às mesmas bolhas, caso Stopwatch seja abandonado e posteriormente revisitado. O retorno ao mesmo estado seria necessário se tivessemos a nível de Stopwatch uma especificação (H*), por exemplo.

A seguir são apresentados alguns fragmentos do programa equivalente ao estadograma da figura 3.3 para ilustrar como as transições são especificadas no código gerado apresentando-se, então, a estrutura do código gerado. O código abaixo corresponde ao programa principal.

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

O programa principal indica que, primeiramente, é constituído o estado global inicial obtido descendo-se pela hierarquia pelas entradas *default*, através da função InitState(). A seguir é executado a função BlobMainLoop(), esta função é dependente da plataforma de *software* e *hardware*[9] no qual a aplicação está sendo gerada, no capítulo seguinte apresenta-se sua implementação para o sistema operacional SunOS com o sistema de janelas X Windows. Para sistema MS-DOS tem-se a seguinte estrutura:

```
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

[9] Especificamente a dependência de máquina da função BlobMainLoop() está associada a função que trata eventos propriamente dito, representada pela função GetEvent().

o estado corrente para ver se este é afetado. Em caso afirmativo a árvore é percorrida de tal forma a encontrar o primeiro "ancestral" comum[10] da bolha a ser abandonada e da bolha a ser atingida. Enquanto o percurso for na direção do ancestral comum, as bolhas representadas pelos nós deste percurso são abandonadas e serão executados ações referente ao comando "on_exit" correspondente a cada estado representado neste percurso. Uma vez alcançado o ancestral, o percurso é feito em direção ao nó que representa a bolha a ser atingida. O sistema entra nas bolhas na mesma ordem em que são visitados os nós correspondentes na árvore nesta segunda parte do percurso e portanto feito o papel inverso é executado a cada entrada em uma nova bolha a ação correspondente especificada pelo comando "on_entry". O estado global é alterado em função da direção do percurso, isto é, são removidos os identificadores de bolhas da lista ligada correspondentes aos nós da árvore, do percurso em direção ao ancestral comum, e são adicionados identificadores das bolhas representadas pelos nós visitados no percurso na outra direção.

No código da função `traverse()` são incorporadas as transições entre bolhas em função dos eventos e condições definidas no estadograma que serviu de especificação para o gerenciador gerado. A função, portanto, é reconfigurada para cada estadograma apresentado ao tradutor. No caso particular do estadograma da figura 3.3, o código da função *traverse()* corresponde ao apresentado abaixo. Os identificadores de bolhas, componentes e eventos são mapeados para inteiros pelo gerador. No caso particular abaixo as bolhas e componentes `Stopwatch`, `Reset`, `InOp`, `Display`, `Reg`, `Sec`, `Run`, `Off` e `On` são associados respectivamente aos inteiros 1, 2, 3, 4, 5, 6, 7, 8 e 9, como já mencionado na especificação do `bstates.h` e os eventos t_1, t_2 e t_3 aos inteiros 1, 2 e 3. Assim como as bolhas os eventos também são definidos através de macros no arquivo `bevents.h` como ilustrado abaixo.

```
#define t1      1
#define t2      2
#define t3      3
```

A seguir tem-se a função `traverse()` invocada no programa principal que controla a mudança de estados do aplicação.

[10] A construção do vetor representando a árvore facilita a busca do ancestral.

```
void traverse ( int evento )

{
   int nblob;
   lista_global *pt1 ; /* apontando para lista de
                          estado corrente */
               *pt = current_state;

   do {
        pt1 = pt;
        nblob = pt -> d;/* d - campo na definicao da
                           estrutura da lista_global
                           que armazena os identificadores
                           de bolhas */
        if ( tab_case[nblob] != NULL )
           pt = (*tab_case[nblob])(evento);
         if ( pt == pt1 )
            pt = pt -> next;
   }while (pt != NULL );
}
```

A função *traverse()* percorre a lista ligada que mantém o estado global do sistema para verificar se uma dada bolha, cujo identificador se encontra na lista ligada representando este estado, é afetada pelo evento corrente. O percurso da lista é controlado pelo comando repetitivo mais externo *do - while*. No corpo deste comando, é verificado se uma dada bolha é afetada ou não pelo evento corrente. Em caso afirmativo o estado corrente do estadograma implementado é atualizado. Para efetuar esta implementação da função *traverse()* de uma maneira mais eficiente possível foi criada um vetor de funções, mostrado abaixo, onde `n` representa o número de bolhas do estadogramas em questão.

```
lista_global  *(*tab_case[nblob])(int evento)
    =  {NULL,fcase2,fcase3, .... , fcasen}
```

Segue abaixo o vetor do exemplo em questão.

```
lista_global  *(*tab_case[nblob])(int evento) =
{
            NULL,
            NULL,
            fcase2,
            NULL,
            NULL,
            fcase5,
            fcase6,
            NULL,
            fcase6,
            NULL,
            fcase9
}
```

Os índices deste vetor estão associados aos identificadores de bolhas do diagrama e cada elemento deste vetor corresponde a uma chamada de função com o número de evento corrente como argumento. Estas funções implementam as transições entre bolha identificado pelo índice a uma nova bolha causada pela ocorrência do evento corrente e atualizam o estado corrente do estadograma implementado decorrente destas transições. Neste nível também são verificadas as condições estabelecidas para determinadas transições como, por exemplo, as transições que têm origem na bolha *Reg* da figura 3.3.

A estrutura de cada uma funções que especificam as transições entre bolhas é analoga à função abaixo que implementam as transições a apartir das bolhas *Reset* e *Reg*. No código gerado estas funções são geradas no arquivo `bfunct.c`.

A função fcase5()

```
lista_global *fcase5( event )

int event;

{
 lista_global *ptaux;
```

```
 ptaux = pt;

 switch ( event )
 {
    case t3 : if (in_blob(Off))
                 return( fromBlobtoBlob(Reg, Reset) );
              if (in_blob(On))
                 return( fromBlobtoBlob(Reg, Sec) );
 }
 return( ptaux );
}
```

A função fcase2()

```
  lista_global *fcase2( event )

  int event;

  {
   lista_global *ptaux;
   ptaux = pt;

   switch ( event )
   {
     case t1 : return( fromBlobtoBlob(Reset, Sec) );
     case t2 : return( fromBlobtoBlob(Reset, InOp) );
   }
   return( ptaux );
  }
```

O percurso da árvore, que representa a estrutura do estadograma em que se baseia o programa gerado, e o controle da execução das ações e atividades associadas às bolhas envolvidas neste percurso são efetuadas pela função *fromBlobtoBlob()*. Esta função representa a parte invariante do código gerado, isto é, função presente em qualquer aplicação, estando implementada

no arquivo `engine.c`. Ao ser invocada a função `fromBlobtoBlob()`[11] é iniciado o percurso na árvore a partir de uma bolha para outra. Assim, na subida até encontrar o ancestral comum a *from* e *to* são disparados as ações embutidas no código LEG pelo usuário através de chamadas a função `OnExit()` e quando na descida, isto é no caminho do ancestral até a bolha destino *to* são executados as possíveis ações ao entrar em cada bolha através da chamada a função `OnEntry()`.

Esta funções, `OnEntry()` e `OnExit()`, têm estruturas semelhantes. A seguir tem-se o código gerado no arquivo `bentry.c` responsável pelas ações *On_entry*.

```
void OnEntry( node )
int node;
{
  switch ( node )
  {
    case Stopwatch : DrawWatch();
                     break;
    case Reset     : ResetTime();
                     break;
    case Display   : InOperationMode();
                     break;
    case Reg       : ShowTimeAsReg();
                     break;
    case Sec       : ShowTimeAsSec();
                     break;
    case On        : SetTime();
                     break;
  }
}
```

As ações disparadas na saída de uma bolha pelo sistema são geradas no arquivo `bexit.c`, como ilustrado abaixo.

[11] Esta função tem dois parâmetros: *from* e *to* que representam a bolha de onde se inicia a mudança de estado e bolha para qual será efetuada a mudança, respectivamente.

```
void OnExit( node )
int node;
{
   switch ( node )
   {
     case Stopwatch : ByeWatch();
                      break;
   }
}
```

3.4 Detalhes de Implementação

Nesta seção serão abordados alguns detalhes de implementação concernentes à organização da estrutura de dados do tradutor tais como: a fases de análise e de síntese.

Por conveniências práticas, o tradutor está projetado, quanto a sua estrutura de dados para efetuar o processo de tradução, em dois passos. No primeiro passo correspondente a fase de análise, são realizadas a análise léxica e análise sintática e em seguida, à fase de síntese, é feita a geração de código propriamente dita correspondendo ao segundo passo do processo de tradução.

3.4.1 Análise Léxica

A fase de análise de um programa [KOW 83] é uma atividade realizada em duas partes: análise léxica que se responsabiliza em separar os símbolos terminais (identificadores, palavras reservadas, símbolos especiais, símbolos especiais compostos e constantes) convertendo-os para códigos convenientes; e a análise sintática que utiliza esses códigos sempre que encontra um símbolo terminal em uma determinada produção da gramática. Portanto, a análise léxica é a atividade de mais baixo nível realizada durante o processo de tradução, onde é transformada a seqüência de caracteres que compõem o programa fonte em uma seqüência de símbolos léxicos que servirão de entrada para o analisador sintático.

Utilizou-se nesta fase, como ferramenta de suporte o programa *Lex* [LES 75], para construir o analisador léxico do tradutor. O código fonte

para o *Lex* é constituído basicamente de uma tabela de expressões regulares e fragmentos de códigos em linguagem C, que correspondem às ações a ser executadas quando no reconhecimento de um determinado símbolo terminal[12]. Portanto, a cada símbolo terminal válido da linguagem está associado uma expressão regular para o seu reconhecimento.

O analisador léxico gerado pelo *Lex* consiste de um automato finito determinístico que executa o fragmento de código em C ao reconhecer a expressão regular associada.

3.4.2 Análise Sintática

Assim como o analisador léxico, o analisador sintático foi também gerado automaticamente utilizando-se para tal o programa *Yacc* ([JOH 75], [SCR 85]). O código fonte do *Yacc* descreve a sintaxe da gramática. Neste código, assim como no *Lex*, está associado a cada produção da gramática um fragmento de código que é executado após o seu reconhecimento.

O *Yacc* é uma ferramenta que a partir de uma linguagem de especificação, similar a BNF, descritiva da sintaxe de uma determinada linguagem, gera uma função capaz de reconhecer textos constituídos segundo essa gramática. Esta função, chamada de *Parser*, invoca uma rotina de nível mais baixo fornecida pelo analisador léxico para selecionar os símbolos terminais a serem analisados.

O esquema da combinação *Lex* e *Yacc* é apresentado na figura 3.5 (Figura retirada de [LES 75]).

Para a versão do tradutor no ambiente Unix, utilizou-se o *Bison* que se constitue de uma outra alternativa ao uso de *Yacc*. *Bison* é compatível com *Yacc* a nível de síntaxe e dispõe de facilidades adicionais. A entrada e saída de ambos são quase totalmente compatíveis entre si. Além dos arquivos de saída do *Bison* terem nomes diferentes do *Yacc* os algoritmos de análise implementados em *yyparser()* diferem-se, obtendo-se melhor desempenho no código gerado pelo *Bison*[13].

Apesar das dificuldades para o uso de *Lex/Yacc* para a construção dos analisadores (léxico e sintático), decidiu-se utilizá-los na implementação

[12] *token*

[13] Algumas características avançadas e outras facilidades proporcionada para um possível código gerado por *Bison* não foram aproveitadas na atual versão do tradutor para que se possa manter compatibilidade com *Yacc*.

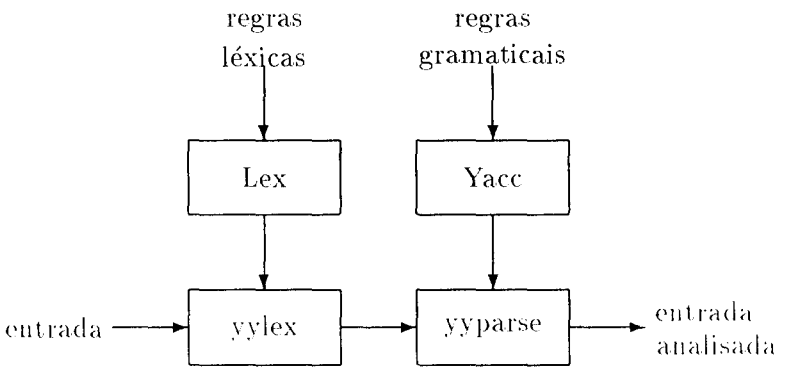

Figura 3.5: Combinação *Lex* e *Yacc*

devido a flexibilidade, especialmente do *Yacc*, de manutenção como pela universalidade: *Yacc* está disponível em qualquer sistema Unix que tenha um sistema de compilação C, havendo também programas compatíveis em domínio público.

3.4.3 Geração de Código

O tradutor com já mencionado opera em dois passos. No primeiro passo é efetuada a fase de análise sintática e, em seguida, vem a fase de síntese, onde é feita a geração de código propriamente dita. A escolha de uma geração em dois passos para a construção do tradutor se deve a algumas características já conhecidas, encontradas no programa escrito em LEG, como hierarquia e comunicação entre módulos (i.e., definição de uma bolha).

Durante o processo de tradução são criados algumas estruturas de dados intermediárias que permitem a transmissão de informações do primeiro para o segundo passo. Estas estruturas são tabelas de símbolos, tabelas de transições e uma árvore sintática que representa a estrutura do programa fonte. Este processo de tradução segue um procedimento padrão utilizado normalmente na construção de compiladores. Abaixo são descritas estas estruturas.

- **Tabela de Símbolos**

A medida que o tradutor processa o código fonte, este armazena algumas informações necessária para geração de código em uma tabela de símbolos, tais como: identificadores (podendo estes ser identificadores de bolhas, eventos ou chamadas de funções em C), categoria associada e, no caso de bolhas, um índice que servirá como seu identificador no programa fonte.

Adotou-se uma estrutura de lista, duplamente ligada que se comporta com uma pilha pelo ponto de vista de sua construção, para sua implementação. Abaixo tem-se uma a descrição dos campos significativos desta estrutura.

- **Identificador**: identificação interna (um número inteiro) de uma bolha, evento ou uma chamada de função em C;
- **Categoria**: tipo de identificador (bolha, evento ou chamada de função em C);
- **Índice hierárquico**: índice associado somente às bolhas, ordenado conforme sua hierarquia. Este índice indica o nível na hierarquia do estadogramas.

- **Tabela de transições**

Esta estrutura é semelhante a tabela de símbolos, obviamente com diferente campos. A tabela de transições permite associar informações necessárias para mudança de estados.

Esta tabela é montada a medida que forem encontrados comando transition. Os campos de seus elementos agregam as seguintes informações:

- **Identificador**: número da bolha corrente;
- **Destino**: número do bolha para a qual será efetuado a transição;
- **Evento**: apontador a um evento correspondente da transição na tabela de símbolos;
- **Condição**: apontador para a seqüência de caracteres representando o predicado do comando if associado. Esta seqüência, como já mencionado na seção 3.2.2, é uma expressão booleana formada de chamadas à função *in_Blob()* que tem como parâmetro um identificador de uma possível bolha concorrente.
- **Disparo**: apontador para caracter contendo a função escrita em C que será executada no momento da mudança de estados.

- **Árvore Descritiva da Hierarquia do Estadograma**

  Esta estrutura também é gerada à medida que o programa fonte é processado, ou seja, concomitantemente à geração da tabela de símbolos e tabela de transição. Os seguintes campos fazem parte da estrutura de cada nó desta árvore:

  - **Número da bolha**: este número corresponde a um apontador para o último campo de um elemento da tabela de símbolos, ou seja, o índice da hierarquia de bolhas;
  - **Bolha default**: apontador para o nó correspondente a bolha filha da bolha corrente;
  - **Irmão**: apontador para a bolha irmã da corrente;
  - **History**: um caracter que representa o tipo de *history* associado à bolha em questão;
  - **On_entry**, **Throughout** e **On_exit**: são apontadores para chamadas de possíveis ações (em forma de invocações a funções em linguagem C) na tabela de símbolos.

Capítulo 4

Uma Aplicação Utilizando Estadogramas

Neste capítulo apresenta-se um exemplo prático da utilização do tradutor para geração de um sistema reativo. Apresenta-se um editor topológico, sua plataforma seu ambiente operacional, características e alguns detalhes particulares de sua implementação com ênfase especial no tratamento de eventos, construído a partir de uma especificação de estadograma.

4.1 Considerações Gerais

O Editor Topológico da LegoShell foi o objeto da tese de mestrado de Hernán Piñon [PIÑ 90], que se constitui basicamente de uma ferramenta capaz de compor e manipular computações complexas. Estas computações são especificadas através de uma linguagem, chamada de LegoShell, a qual consiste em uma generalização do conceito de *pipe* do UNIX. Para melhor entendimento desta ferramenta faz-se, a seguir, uma breve abordagem do ambiente de desenvolvimento de software A_HAND para o qual este editor foi desenvolvido.

4.2 O Ambiente A_HAND

O A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquia de Abstração em Níveis Diferenciados) ([DRM 87a], [DRM 87b],

[DRM 89]) é destinado ao desenvolvimento de sistemas grandes e complexos que está sendo desenvolvido no Departamento de Ciência da Computação (DCC) da UNICAMP e tem como objetivo oferecer um ambiente que possa suportar o desenvolvimento de tais tipos de sistemas, i.e., sistemas grandes e não convencionais.

Uma das características mais relevantes do A_HAND é permitir que objetos manipulados no ambiente possam ser compostos de objetos mais simples, obtendo-se assim uma estrutura hierárquica de objetos.

Uma outra característica importante do A_HAND é a padronização da interface nos diversos níveis de abstração, com isso, as mudanças de níveis são suáves para o usuário, bem como a aprendizagem do uso de todo o ambiente é rápido. Assim, no A_HAND, interfaces gráficas e bastantes amigáveis são características inerentes ao ambiente.

Maiores detalhes sobre o ambiente A_HAND (requisitos, definições, características, etc.) poderão ser encontrada em [DRM 87a], [DRM 87b], [DRM 89], [POL 90], [PIÑ 90], [FUR 91] e [CAS 91].

4.3 A linguagem de Computações LegoShell

A LegoShell [DRM 89] é uma linguagem gráfica para configurar programas. A LegoShell permite construir *pipes* de UNIX estendidos para mais dimensões através de conectores do tipo "mailbox" e "broadcast" entre programas, arquivos, dispositivos e computações já especificadas. A LegoShell manipula objetos de interface padronizada do tipo porta de entrada e saída ligadas por meio dos conectores que permitem a especificação do fluxo de dados entre os mesmos.

4.4 Editor Topológico da LegoShell

O Editor Topológico da LegoShell [PIÑ 90], ou simplesmente Editor da LegoShell, é ferramenta destinada para compor computações e objetos básicos da LegoShell. Este editor, além de permitir a manipulação dos objetos, também desempenha o papel de interface de invocação das funções inerentes ao ambiente.

Este editor foi construído baseado em estadogramas. Por ocasião da fase de desenvolvimento deste editor, o tradutor não se encontrava ope-

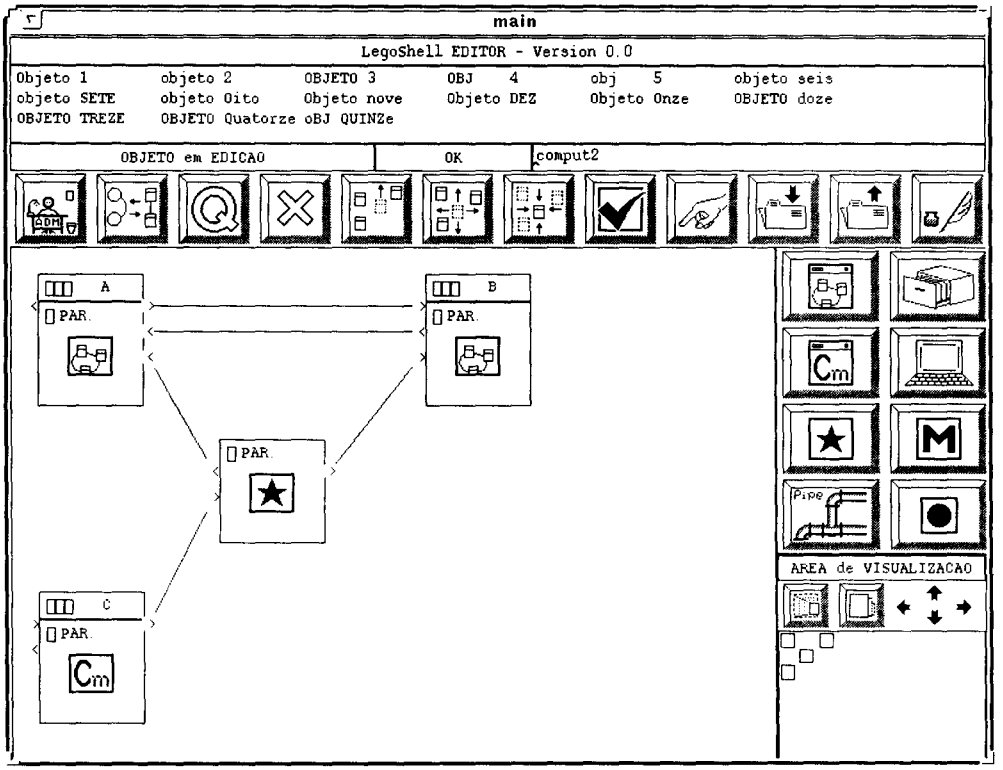

Figura 4.1: Interface-Usuário do Editor da LegoShell

racional. Assim sendo, as tabelas que decrevem a hierarquia de bolhas e a função que dispara as ações associadas a bolhas foram geradas manualmente. Nesta fase foi utilizada a parte invariante do código gerado pelo tradutor responsável pelo controle do diálogo (que encontra-se implementado no arquivo `engine.c`) que, nesta ocasião, já se encontrava implementada e testada. Após a implementação do tradutor, o código do Editor da LegoShell foi reescrito em LEG e em seguida, utilizando-se a ferramenta objeto deste trabalho, foi feito a tradução para código em C obtendo-se um programa funcionalmente equivalente ao anterior.

A interface-usuário do Editor da LegoShell, apresentada na figura 4.1, é composta das seguintes áreas:

- Cabeçalho: contém o nome dos objetos a serem editados. O processo de edição de objetos pode ser suspenso e, portanto, é possível editar outro objeto empilhando-se o objeto anterior;
- Menu de Barra: contém as principais funções do editor;
- Área de trabalho: área útil destinada à edição topológica;
- Menu de ícones: contém os ícones disponíveis ao ambiente que representam as facilidades providas pelo editor e podem ser selecionados e incluídos na área de trabalho durante o processo de edição;
- Área de Visualização: uma janela de visualização total do objeto corrente onde é indicado, através de um retângulo, a parte do objeto corrente que está sendo mostrada na área de trabalho.

4.5 Arquitetura do Editor

O Editor da LegoShell foi desenvolvido sobre o sistema de janelas *X Windows* ([NYE 90], [SWI 88], [SCH 86]) em SPARCstation da Sun com o sistema operacional UNIX utilizando-se para construção de interface-usuário a ferramenta *Athena X Toolkit* [NYE 90].

Esta ferramenta foi desenvolvida em várias camadas relacionadas abaixo cobrindo aspectos relacionados desde a estrutura topológica capaz de suportar os objetos complexos tratados pelo editor até os aspectos relacionados à interface com o usuário passando pela componente de controle suportada pelos estadogramas.

1. Funções topológicas;
2. Funções de edição topológica;
3. Funções geométricas;
4. Funções de representação;
5. Funções de edição gráficas;
6. Máquina de estadogramas;
7. Interface com o usuário.

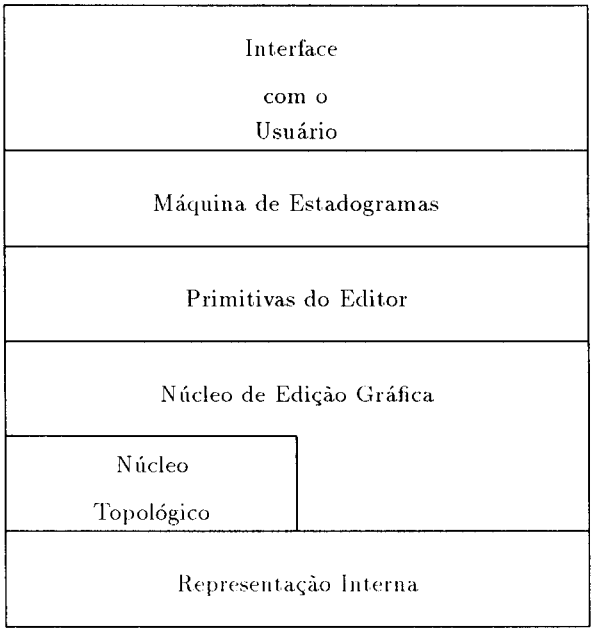

Figura 4.2: Arquitetura do Editor da LegoShell

A figura 4.2 apresenta os níveis de implementação do editor: o núcleo topológico agrupa as funções 1 e 2; o núcleo de edição gráfica agrupa as funções 3, 4 e 5; e seguida, tem-se a componente de controle gerada pelo tradutor que é o responsável, neste caso, pela semântica do editor e no último nível tem-se a interface com o usuário.

Nesta figura é apresentada uma visão mais abstrata do editor, podendo-se observar as três componentes separadamente, que estabelece, um paralelo com modelo apresentado na figura 1.1. Uma componente que está intrinsicamente relacionada a aplicação engloba as camadas da Representação Interna, Núcleo Topológico, Núcleo de Edição Gráfica e Primitivas do Editor. A componente correspondente ao controle é representada pela camada da Máquina de Estadogramas. Finalmente a componente de apresentação pode ser associada à camada responsável pela Interface com o Usuário.

4.6 O Ambiente de Desenvolvimento: X Window e Athenas X Toolkit

O sistema de janelas *X Window* tem-se tornado um padrão para estações de trabalho UNIX. *X Window* constitui-se de um ambiente bastante adequado para o desenvolvimento de *software*. Algumas de suas características são relacionadas abaixo:

- Permite trabalhar com múltiplos programas simultaneamente em diferentes janelas;
- Suporta hierarquia de janelas redimensionáveis e superposição de janelas, o que permite que uma grande variedade de aplicações sofisticadas e interfaces-usuário sejam construídas mais rapidamente;
- Permite execução distribuída das aplicações com independência de máquina e de sistema operacional;
- Permite o uso de diferentes níveis de abstração na definição de interfaces-usuário;
- É passível de ser estendido.

Para o suporte ao desenvolvimento de aplicações, o *X Window*[1] provê facilidades para uma ferramenta chamada *X Toolkit* (também denominada Xt), composta de duas classes de sub-rotinas (*Xt Intrinsics* e *Xaw Widget set*). Um *Widget* é um elemento de interface-usuário pré-construída. Xt é escrito em *Xlib*, que desempenha o papel de uma interface entre a linguagem C e o sistema *X Windows*. *Xt Intrinsics* e *Xlib* são requeridos pelo padrão X em qualquer sistema que permite programação de aplicações X em C. A biblioteca de *Xaw Widget* é baseada em Xt e proporcio na um pequeno número de *Widgets* que podem ser usados para escrever programas de aplicações simples.

O propósito de Xt é proporcionar uma camada orientada a objeto que suporta a abstração da interface-usuário, chamada de *Widget*. Um *Widget* é um fragmento de código reutilizável e reconfigurável que opera independentemente da aplicação exceto através de interações pré-estabelecidas. Assim, um *Widget set* é uma coleção de widgets que proporciona a construção de

[1] Neste texto em alguns momentos associa-se X a *X window*.

componentes de interfaces-usuário "amigáveis", chamada neste trabalho de componente de apresentação.

O uso de *widget* garante a separação do código da aplicação do código da componente de apresentação (i.e., interface-usuário) e proporciona elementos de interfaces-usuário pré-construídos[2] tais como *buttons*, *scrollbars*, *dialog boxes*, etc, ao programador que necessita somente integrá-los ao código da aplicação.

A figura 4.3 ilustra as camadas de *software* que usa *Xt Intrinsics* e um *widget set*. É importante que *Xt Intrinsics* seja baseado em *Xlib* que, por sua vez, estabelece um total acesso às facilidades do *X Protocol* responsável pela que manipula a interface entre uma aplicação e a rede.

4.6.1 Interação Widget – Aplicação

É tratado aqui inicialmente a independência operacional de um *widget* para depois entender melhor como é feita esta interação entre um *widget* e uma aplicação.

Como já mencionado, cada *widget* é altamente independente da aplicação. O Xt envia eventos para um *widget*, o qual executa as ações apropriadas de acordo com sua especificidade, sem a ajuda da aplicação. Por exemplo, *widgets* se redesenham automaticamente quando estes são expostos depois de terem sido cobertos por uma outra janela.

Os *widgets* são projetados para permitir que o usuário controle a aplicação através da interação com os mesmos. Portanto, *widgets* permitem a invocação de rotinas do código de aplicação através de funções escritas em Xt. chamadas a estas funções são invocadas em resposta a alguma ocorrência em um *widget*. Por exemplo, um widget "QUIT", quando ativado, pode invocar um código que desativa uma aplicação.

Para associar *widgets* a funções de alguma aplicação existem mecanismos entre os quais é destacado um chamado *callbacks*.

Genericamente, para que um *widget* venha a interagir com a aplicação serão declaradas uma ou mais listas de *callbacks*; a aplicação adiciona funções a estas listas de *callbacks*, as quais serão invocadas sempre que as condições de *callbacks* pré-definidas são satisfeitas.

Para melhor compreensão da estrutura básica de uma aplicação que uti-

[2] *ready-to-use*

liza X Toolkit, veja no apêndice D um pequeno exemplo.

4.7 Tratamento de Eventos pelo Editor da LegoShell

O enfoque adotado nesta seção será direcionado à parte da implementação do Editor da LegoShell dedicada ao tratamento de eventos no ambiente operacional sobre o qual este foi construído.

O tradutor é uma ferramenta que pode ser utilizada para geração de aplicações, como já mencionado, em diferentes plataformas de *software* e *hardware*. A seguir, são apresentados alguns detalhes relevantes da implementação do Editor relacionados com o tratamentos de eventos que também são relevantes para outras aplicações geradas para uma plataforma de mesma natureza.

Como apresentado no capítulo anterior a estrutura de um programa principal gerado pelo tradutor tem a seguinte estrutura:

```
main()

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

Nesta aplicação a função `BlobMainLoop()` foi substituída por `XtMainLoop()`[3] uma função do Xt, que executa um *loop* "infinito" que consome os eventos de um *buffer*, alimentado pelo *X Server*. `XtMainLoop()` envia, como mencionada no apêndice D, eventos para *widgets* e funções na ordem nos quais este ocorrem.

Tem-se a seguir alguns trechos de códigos do Editor da LegoShell que mostram como foi feito a conversão de, por exemplo, do acionamento do *mouse* em um determinado ícone do editor para um número inteiro que é solicitado

[3] Para melhor entendimento da rotinas mencionadas nesta seção, veja o apêndice D e para detalhes mais específicos consulte o capítulo 2 do volume 4 [NYE 90] ou [SWI 88].

pelo código gerado para efetivar uma possível mudança pelo gerado de estado no código gerado pelo tradutor.

Em um arquivo descrevem-se as definições dos *widgets* do editor onde tem-se o seguinte comando, por exemplo, toma-se aqui o *widget* "cpipe".

```
cpipe = XtCreateManagedWidget(
        "pipe",
        commandWidgetClass,
        micones,
        arg,
        n
        );
```

A chamada `XtCreateManagedWidget` cria neste caso o `widget Command`. Abaixo, descreve-se seus parâmetros:

- "`pipe`" - nome arbitrário para este *Widget*;
- `commandWidgetClasse` – especifica o nome da classe do *widget* criado, neste caso é um *Command widget*;
- `micones` este é o *widget* pai;
- `arg` lista de argumentos;
- `n` número de argumentos.

Em seguida a este comando neste mesmo trecho de código, vem a chamada `XtAddCallback`:

```
XtAddCallback(cpipe,XtNcallback,ActivateCpipe,NULL)
```

Esta chamada é usada para registra a função `ActivateCpipe` como se `callback` do `Command Widget`, seus parâmetros são:

- "`cpipe`" – é o *widget* que dispara a callback;
- `XtNcallback` – é um constante simbólica como detalhada no apêndice D;

- `ActivateCpipe` – esta é uma função que é chamada sempre que um *widget* é acionado, esta função funciona como um "disparador"[4] de ações e chamada de função **callback**. Como veremos mais a frente, dentro desta função é feita a invocação ao código que efetua a mudança de estado;

- `NULL` – é um parametro a ser passado para a função `ActivateCpipe`.

Em outro arquivo foram definidos as funções de *callback*, como mostrada abaixo pela funçãqo `ActivateCpipe` do Editor que foi tomanda como exemplo.

```
void ActivateCpipe(w, cliente_data, call_data)
widget w;
caddr_t  client_data;
caddr_t call_data;
{
     traverse(EINCLUDEPIPE)
     return;
}
```

Como se vê neste ponto, dentro do código da função `ActivateCpipe`, é invocado a função `traverse` que é parte integrante do código gerado pelo tradutor e que é responsável pelo controle da mudança de estados do Editor. O seu argumento `EINCLUDEPIPE` é uma macro pré-definida que representa um inteiro requerido para efetivar esta mudança. Neste caso foi gerado no arquivo `bevents.h`, como `# define EINCLUDEPIPE 11`.

[4] *dispatcher*

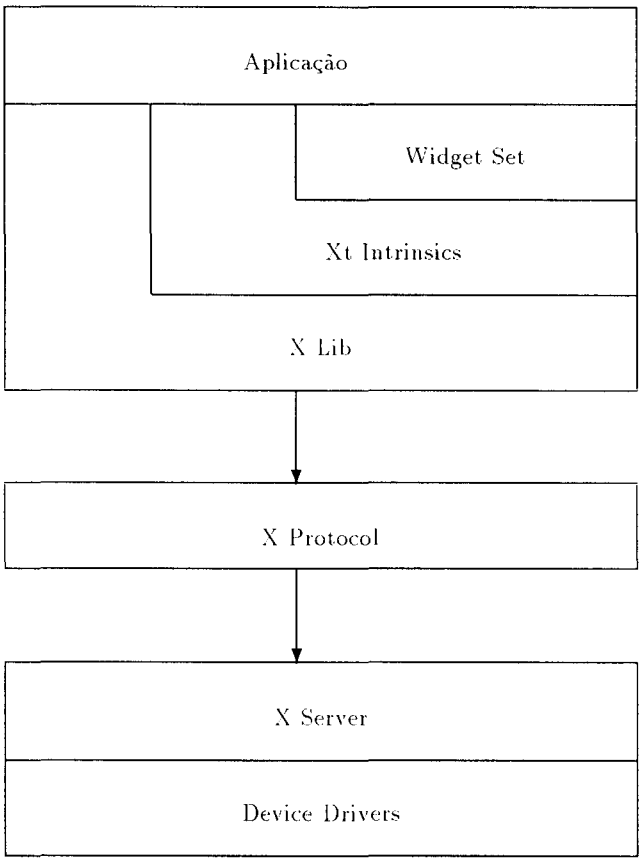

Figura 4.3: Arquitetura de software de aplicações baseada em *Xt Intrinsics*.

Capítulo 5

Conclusão

5.1 Considerações Gerais

O propósito geral deste trabalho consiste, basicamente, em apresentar uma ferramenta dedicada ao desenvolvimento de uma classe de sistemas em geral considerados grandes e complexos, chamados aqui de sistemas reativos, dando-se ênfase à componente de controle do diálogo. A ferramenta se comporta como um tradutor que a partir de uma descrição textual, i.e. um programa escrito em LEG (Linguagem de EstadoGrama), gera um programa escrito em linguagem C que representa o comportamento do sistema gerado.

LEG e, consequentemente, o tradutor (ambos tratados no capítulo 3) foram baseados em estadograma (no capítulo 2) extensão dos diagramas de estados convencionais com a incorporação de novas características como: concorrência, hierarquia e comunicação.

O tradutor foi escrito em linguagem C, utilizando-se as ferramentas *Lex* e *Yacc* para geração dos analisadores léxico e sintático, respectivamente. A atual versão encontra-se disponível para os sistemas operacionais UNIX e MS-DOS.

Historicamente, sistemas interativos têm sido projetados com a componente do diálogo e componente da aplicação combinados, isto é, sem uma clara identificação, em relação ao código da aplicação, do que é inerente a cada uma destas componentes. O problema com este enfoque tem-se tornado cada vez mais evidente, o que faz com que sistemas construídos sob este enfoque tornassem difíceis sua manutenção e modificações. Tais sistemas são

tratados como "sistemas monolíticos". Tem-se observado largamente na literatura que o propósito para o desenvolvimento de sistemas atuais está baseado na independência do diálogo, isto é, tem-se buscado ferramentas ou adotado metodologias que possam estabelecer de alguma maneira uma certa separação do código inerente da aplicação ao código relativo ao seu controle. Como pôde-se se observar, aplicações geradas pelo tradutor satisfazem estes requisitos. Um programa escrito em LEG denota perfeitamente esta separação, onde a parte relativa ao controle é expressado pelo código em LEG e a parte relativa a aplicação está representado pelas ações associadas escritas em C. Desta forma objetiva-se com esta ferramenta, uma melhora da produtividade em desenvolvimento de *software*, especialmente quando estes envolvem sistemas que mantém interações complexas com o seu "mundo externo".

5.2 Extensões e Perspectivas

Para que o tradutor se torne uma ferramenta mais genérica e mais flexível, pretende-se incorporar algumas características complementares a sua implementação para que o processo de desenvolvimento de sistemas de controle tenha uma produtividade maior. Algumas considerações se referem a implementações e outras em relação as suas características ou a gramática de LEG. O estadograma é um mecanismo, que devido a sua flexibilidade, pode ser facilmente estendido para atacar aplicações mais complexas e mais específicas e/ou genéricas. Assim, pretende-se incorporar mais funções a LEG e ao tradutor, como mencionado abaixo:

- A gramática, em *Yacc*, foi projetada para suportar várias definições de uma mesma bolha podendo-se ter em duas componentes, por exemplo, ou em outro nível hierárquico bolhas com nomes iguais. Na implementação do tradutor, este recurso somente é suportado na gramática reconhecida pelo analisador sintático gerado pelo *Yacc* e não encontra-se implementado no resto do sistema;

- Provavelmente em um ambiente de desenvolvimento de *software*, onde vários programadores estão desenvolvendo um projeto, utilizando-se LEG, no qual cada um esteja encarregado da implementação de determinados módulos e não somente do projeto como um todo, torna-se interessante que o tradutor possa dispor de uma opção que permita

a tradução em separado, isto é, a tradução somente seria feita para um determinado módulo ou conjunto de módulos. Com esta característica, justifica-se mais ainda a proposição do item anterior onde busca-se o uso de bolhas com nomes iguais em um mesmo projeto, já que um trabalho em grupo não seria exigido que seus participantes se preocupassem em dar nomes diferentes a suas bolhas;

- Alguns conceito originais dos estadogramas, como o que se chama aqui de *history de reentrância* e o comando *selection*, são suportados pela implementação da máquina de estadograma (`engine.c`), responsável pelo controle da aplicação, mas não estão sendo suportados pela gramática de LEG para a atual versão do tradutor;

- Como já mencionamos, o tradutor necessitará ser usado no desenvolvimento de diferentes aplicações onde se possa sentir eventuais restrições e com isso dotá-lo com facilidades adicionais como um conjunto de bibliotecas que tornem a geração de aplicações mais rápida e mais eficientes. Possíveis extensões na implementação nesse sentido poderiam ser feitas, como:

  - Um tratador de eventos para várias plataformas de *hardware* e *software*, como, por exemplo, para ambiente MS-DOS e/ou para mais ferramentas de *software* em ambiente UNIX – *X Window*, onde se possa dispor de mais rotinas para o tratamento de eventos;
  - Melhorar o processo de geração de aplicações para um ambiente análago no qual o Editor da LegoShell (usado com exemplo da aplicação do tradutor, apresentado no capítulo 4) foi gerado, em particular, em relação a características específicas de uma aplicação escrita para um ambiente *X Window*. Ou seja, prover ao tradutor rotinas que permitam a geração de aplicações mais automáticas para diferentes plataformas.

- Um interessante aspecto que deverá ser abordado em uma futura versão desta ferramenta e como futura pesquisa consiste na abordagem de uma semântica formal para descrição das características apresentadas dos estadogramas e de outras possíveis extensões como *sobreposição de bolhas*[1] (veja apêndice A) que não é suportada pela atual

[1] *Overlapping states*

versão do tradutor. Esta extensão permite especificações mais compactas. Este enfoque aponta para uma interessante direção para futuros trabalhos envolvendo aspectos semânticos e eficiência na implementação da superposição.

- Para que o tradutor possa ser de maior utilidade no processo de desenvolvimento de sistemas de controle, pretende-se compô-lo com o simulador de estadogramas, apresentado em [BAT 91], afim de que se possa ter, opcionalmente, associado à aplicação gerada a visualização do fluxo de controle no estadograma correspondente. Esta facilidade seria muito desejável para a depuração de aplicações geradas a partir de uma especificação em LEG.

O tradutor pretende ser uma ferramenta que facilite a especificação de gerenciadores de sistemas complexos privilegiando o aspecto de controle e pretende-se cada vez mais dotá-la com facilidades adicionais para que este processo se torne cada vez mais produtivo.

Bibliografia

[AHO 77] Aho, A. V., & Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, Reading Mass., 1977.

[BAT 91] Batista Neto, J. E. S., *Um Editor Gráfico Para Statecharts*, Dissertação de Mestrado, USP - São Carlos, (Abril 1991).

[BER 85] Berry, G. and Cosserat, I., *The ESTEREL: Synchronous Programming Language and its Mathematical Semantics*, in Seminar on Concurrency (S. Brookes and G. Winskel, eds.), Lecture Notes in Computer Science, vol. 197, Springer-Verlag, Berlim, 1985.

[BOR 87] Borland International, *Turbo C Reference Guide*, Borland International Inc., Scotts Valley, Ca., 1987.

[CAR 85] Cardelli, L. and Pike, R., *Squeak: a Language for Communicating with Mice*, SIGGRAPH'85, vol. 19, n. 3, july 1985, pp. 199-204.

[CAS 91] Castro, L. S., *Sistrac: Sistema de Suporte a Trabalho Cooperativo*, Dissertação de Mestrado, DCC - IMECC - UNICAMP, (Outubro 1991).

[CON 87] Conklin, J., *Hypertext: An Introduction and Survey*, IEEE Computer, Sept. 1987, pp. 17-41.

[COU 87] Contaz, J., *Abstraction for user interface*, IEEE Computer, march 1981, pp. 21-34.

[DRM 87a] Drummond, R. e Liesenberg, H., *A_HAND: Ambiente de Desenvolvimento de Software Baseado em Hierarquia de Abstração em Níveis Diferenciados*, IV Encontro do Projeto ETHOS, Petropópolis - RJ, (Abril 1987), pp. 313-322.

[DRM 87b] Drummond, R. e Liesenberg, H., *Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS*, I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro - RJ, (Novembro 1987).

[DRM 89] Drummond, R., *LegoShell: Linguagem de Computaçõcs*, Anais do III Simpósio Brasileiro de Engenharia de Software. 1989.

[DRU 85] Drunsinsky, D. and Harel, D., *Using Statecharts for Hardware Description*, Weizmann Institute of Science, CS85-06, December 1985.

[DRU 86] Drunsinsky, D. and Harel, D., *Statecharts as an Abstract Model for Digital Control Units*, Weizmann Institute of Science, CS86-12, may 1986.

[DRU 89] Drunsinsky, D. and Harel, D., *Using Statecharts for Hardware Description and Synthesis*, IEEE Trans. on Computer-Aided Design, vol. 8, no.7, july 1989, pp. 798-807.

[EDW 87] Edwards, A. D. N., *Visual programming languages: the next generation?*, SIGPlan Notices, April 1987, vol. 23, no. 4, pp. 43-50.

[FIG 90] Figueiredo Filho, A. G. & Liesenberg, H. K. E., *Geração de Gerenciadores de Sistemas Reativos*, Relatório Técnico n. 20/90, maio 1990.

[FIG 91] Figueiredo Filho, A. G. & Liesenberg, H. K. E., *Geração de Gerenciadores de Sistemas Reativos*, Anais do V SBES, Ouro Preto - MG, Outubro 1991.

[FUR 91] Furuti, C. A., *Um Compilador para uma Linguagem de Programação Orientada a Objetos*, Dissertaç ao de Mestrado, DCC IMECC - UNICAMP, agosto 1991.

[GOO 84] Good, M. D., Whiteside, J. A., Wixon, D. R. and Jones, S. J., *Building a User-Derived Interface*, Comm. ACM, October 1984, pp. 1032-1043.

[GRE 85] Green, M., *Report on Dialogue Specification Tools*, in User-Interface Management System, Gunther E. Pfaff, ed., Springer-Verlag, New York, 1985, pp. 9-20.

[GRE 86] Green, M., *A Survey of Three Dialogue Models*, ACM Trans. on Graphics, vol. 5, no. 3, July 1986, pp. 244-275.

[HAH 89] Hartson, H. R., *User-Interface Management Control and Communication*, IEEE Software, January 1989, pp. 62-70.

[HAH 89] Hartson, H. R. and Hix, D., *Human-Computer Interface Development: Concepts and Systems for its Management*, ACM Computing Surveys, vol. 21, n. 1, March 1989, pp. 5-92.

[HAR 85] Harel, D. & Pnueli, A., *On the Development of Reactive Systems*, in: K. R. Apt., Ed., Logics and Models of Concurrent Systems (Springer, New York, 1985), pp. 477-498.

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

[HAR 88] Harel, D., *On Visual Formalisms*, Comm. of the ACM, vol. 31, no. 5, May 1987, pp. 514-530.

[HAR 89] Harel, D. & Kahana, C.-A. *On Statecharts with Overlapping*, Technical Report CS89-05, Dept. of Applied Mathematics & Computer Science, The Weizmann Institute of Science, Rehovot, Israel, April 1989.

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

[HAY 81] Hayes, P. J., Ball, E. and Reddy, R., *Breaking the man-machine communication barrier*, IEEE computer, 14, March 1981, pp. 19-30.

[HOA 78] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, vol. 21, n. 8, August 1978, pp. 666-667.

[iLO 87] i-Logix Inc., *The Languages of STATEMATE*, Documentation for the STATEMATE System, Technical Report, i-Logix Inc., Burlington, MA., November 1987.

[JAC 83] Jacob, R. J. K., *Using Formal Specifications in the Design of a Human-Computer Interface*, Comm ACM, 26 (1983), pp. 259-264.

[JAC 85] Jacob, R. J. K., *A State Transition Diagram Language for Visual Programming* , Computer, August 1985, pp. 51-59.

[JAC 86] Jacob, R. J. K., *A Specification Language for Direct-Manipulation User Interfaces*, ACM Trans. on Graphics, vol. 5, no. 4, October 1986, pp. 283-317.

[JOH 75] Johnson, S. C., *YACC: Yet Another Compiler-Compiler*, Computing Science Tecnical Report, no. 32, octuber 1975, Bell Laboratories, Murray Hill, NJ.

[KAS 82] Kasik, D. J., *A user interface management system*, Comput. Graph., vol. 16, n. 3, pp. 99-106.

[KAS 89] Kasik, D. J., Lund, M. A. and Ramsey, H. W., *Reflections on Using a UIMS for Complex Application*, IEEE Software, january 1989, pp. 54-61.

[KER 78] Kerninghan, B. & Ritchie, D., *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

[KOW 83] Kowaltowski, T., *Implementação de Linguagens de Programção*, Ed. Guanabara Dois, 1983.

[LES 75] Lesk, M. E., *Lex - A Lexical Analyzer Generator*, Comp. Sci. Tech. Rep. no. 39, Bell Laboratories, Murray Hill, New Jersey, October, 1975.

[MAC 85] Macintosh Toolbox : Inside Macintosh, Addison-Wesley, Reading, Mass., 1985.

[MEI 91] Meira, C. A. A., *Sobre Geradores de Aplicação*, Dissertação de Mestrado, USP - São Carlos, (Setembro 1991).

[MYE 89] Myers, B. A., *User-Interface Tools: Introduction and Surveys*, IEEE Software, january 1989, pp. 15-23.

[MIL 80] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlim, 1980.

[NAU 63] Naur, P. (editor), *Revised Report on the Algorithmic Language ALGOL 60*, Comm. ACM, 6, (1963), pp. 1-17.

[NYE 90] Nye, A. and O'Reilly, T., *The definitive Guides to the X Windows System*, vol. 0 to 7, O'Reilly & Associates, Inc. Sebastopol CA, 1990.

[OLS 83] Olsen Jr., D. R. and Dempsey, E. P., *SYNGRAPH: A Graphical User-Interface Generator*, Computer Graphics, vol. 17, n. 3, July 1983, pp. 43-50.

[OLS 86] Olsen Jr., D. R. *Editing Templates: A User Interface Generation Tool*, IEEE CG & A, November 1986, pp. 40-45.

[OLS 87] Olsen Jr., D. R., *Large Issues in User-Interface Management*, Computer Graphics, vol. 21, no. 3, April 1987, pp. 71-147.

[PER 87] Perkins, D. C., Szczur, M. R., Moe, K. L. and Stephens, M. A., *TAE Plus: Evolution of a NASA User Interface Management System*, 26th Annual Technical Symposium of the DC Chapter of the ACM, june 1987.

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

[POL 90] Polanczyk, C. A., *Uma Ferramenta Baseada em Hipertexto para Desenvolvimento de Software*, Dissertação de Mestrado, DCC IMECC - UNICAMP, dezembro 1990.

[PNU 86] Pnueli, A., *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*, In Current Trends in Concurrency, Bakker, J. W. et alts., Lecture Notes in Computer Sciences, vol. 224, Springer-Verlag, New York, (1986), pp. 510-584.

[REI 85] Reisig, W., *Petri Nets: An Introduction*, Springer-Verlag, Berlim, 1985.

[SCH 86] Scheifler, R. W. and Gettys, J., *The X Window System* , ACM Trans. on Graphics, vol. 5, n. 2, april 1986, pp. 79-109.

[SIB 86] Sibert, J. L., Hurley, W. D. and Bleser, T. W., *An Object-Oriented User Interface Management Sysrtem*, SIGGRAPH'86, vol. 20, n. 4, august 1986, pp. 259-268.

[SCR 85] Schreiner, A. T. and Friedman Jr, H. G., *Introduction to Compiler Construction with UNIX*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1985.

[SMA 89] Smart, J., Leygues, F. and Lichtenhein, M., *ECMA TC 33 User Interface Technical Assessement Report*, ECMA (European Computer Manufacturers Association), march 19189.

[SUN 82] Sunshine, C. A., et al., *Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models*, IEEE Trans. Soft. Eng., 8 (1982), pp. 460-489.

[SWA 82] Swartout, W. and Balzer, R., *The Inevitable Intertwining of Specification and Implementation*, Comm. of the ACM, july 1982, pp. 438-440.

[SWI 88] Swick, R. and Weissman, T., *X Toolkit Athena Widget – C Language Interface*, X Window System, MIT, 1988.

[SZC 88] Szczur, M. R. and Miller, P., *Transportable Applications Environment (TAE) Plus: Experience in "Object"ively Modernizing a User Interface Environment*, OOPSLA'88 Proceedings, vol. 25, n. 30, september 1988, pp. 58-70.

[SZC 89] Szczur, M. R. *TAE Plus: Transportable Applications Environment Plus – Tool for Building Graphic-Oriented Applications*, Graphics Technology in Space Applications, Johnson Space Center, Texas, april 1989.

[TEN 81] Tenenbaum, A. S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[VAN 88] Van der Boss, J., *Abstract Interation Tools: A Language for User Interface Management Systems*, ACM Trans. on Programming Languages and Systems, vol. 10, no. 2, april 1988, pp. 215-247.

[WAS 85] Wasserman, A. I., *Extending transition diagrams for the specification of human-computer interaction*, IEEE Trans. Soft. Eng., SE-11, 8, august 1985.

[WEB 85] Weber, H. R., *Meditation on man-machine interfaces or our personal role in graphics dialogue programming*, Comput. & Graphics, vol. 9, no. 3, 1985, pp. 237-245.

[WON 82] Wong, P. C. S. and Reid, E. R., *FLAIR – User interface dialog design tool*, Comput. Graph., vol. 16, n. 3, july 1982, pp. 87-98.

Apêndice A

Outras Características dos Estadogramas

A.1 Entradas: *Condition* e *Seletion*

Foram especificados dois tipos de entrada para simplificar algumas entradas especiais para uma sub-bolha de uma bolha. Inicialmente, trata-se a entrada representada por (C) — *condition*, que é ilustrada na figura A.1. Note que a figura A.1(b) pode substituir a figura A.1. A figura A.1 é usada quando a topologia de setas e/ou condições são muito complexas, então omite-se os detalhes do diagrama com uma forma incompleta (resumida).

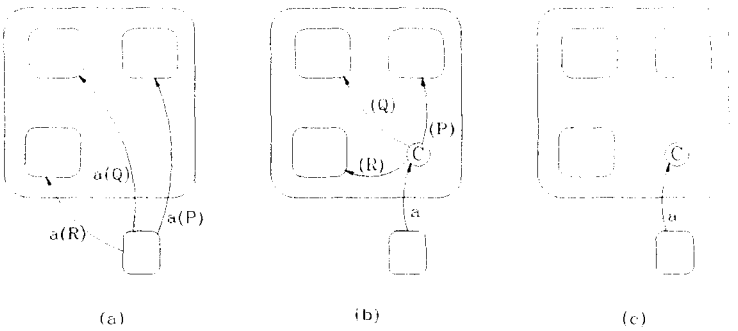

Figura A.1: Entrada por Condição

Um outro tipo de entrada é representado por S — *seletion*. Seleção ocorre

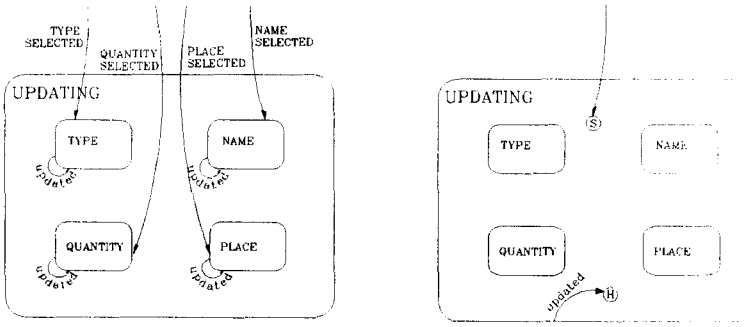

Figura A.2: Entrada por Seleção

quando o estado a ser visitado é determinado pelo "valor" de um evento genérico. Tal que, este evento corresponde a uma das opções pré-definidas e tem-se especificado para modelar cada opção disponível como estados.

Considere um sistema destinado para armazenamento de informações, apresentado na figura A.2, onde tem-se na tela disponível as teclas *type*, *name*, *quantity*, e *place* correspondendo, respectivamente ao tipo de objetos a ser armazenado, identificação do objeto, quantidade e posicionamento físico.

Um procedimento de atualização física permitiria o usuário a selecionar uma opção através da tecla apropriada e então fazer a atualização, possivelmente repetidamente. A figura A.2(a) modela a situação e figura A.2(b) mostra como a opção de "seleção" (unificado com este *history* simplifica várias tarefas. Assim, o usuário terá que especificar o evento *selection* somo sendo a disjunção dos quatro eventos de mais baixo nível, assim como a associação de cada um deles com o estado apropriado, tal que este procedimento (i.e., entrada via "S") torna bem definida.

A.2 Delays e Timeouts

Os estadogramas podem tratar restrições de tempo usando temporizadores implicitos como por saída de uma determinada bolha como mostrado na figura A.3. Formalmente, isto é feito pelo uso de expressões de evento (podendo fazer parte de uma biblioteca), o qual representa o evento que ocorre precisamente quando o número especificado (*number*) de uma certa unidade de tempo é completado a partir da ocorrência do evento (*event*)

especificado. A saída do estado *blink* como mostrado na figura A.3, seria especificado desta forma, como: *timeout(entered blink, 60)*.

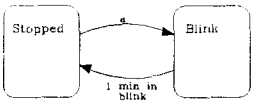

Figura A.3: Temporização associada a transições

Portanto, esta característica é inerentemente associada a bolhas foi, então, criado uma notação gráfica que indica que a bolha associada a restrição de tempo (figura A.4) e o evento genérico (*timeout(entered state, bound)*), onde *state* é a bolha fonte da transição e *bound* e limite máximo de tempo permitido nesta bolha.

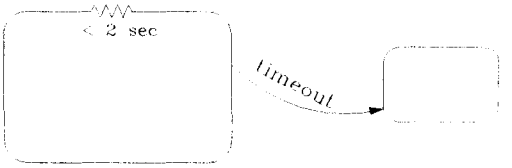

Figura A.4: Temporização associada ao estado

Vale observar que nenhuma transição poderá executar mudança de estados quando limite de tempo ainda não foi completado. Esta observação é válida principalmente quando se tem *delays* em níveis mais baixo. São encontrados exemplos e aplic, oes deste tipo de restrição de tempo nos estadogramas ilustrados em [DRU 89], [DRU 86] e [DRU 85].

A.3 Superposição de Bolhas

Bolhas em um estadogramas são organizadas como uma árvore AND/OR. Estas podem ser repetidamente decompostas, como visto no capítulo 2, em bolhas que contém sub-bolhas ou em componentes. Em estadogramas esta estrutura em árvore é uma consequência de suas características que torna mais natural para mente humana em detectar a estrutura hierárquica dessa

representação. A figura A.5 ilustra a situação na qual C tem dois ancestrais, talvés a mais obvia justificativa para superposição de estados seja a de se evitar a duplicação de subestadogramas[1] idênticos.

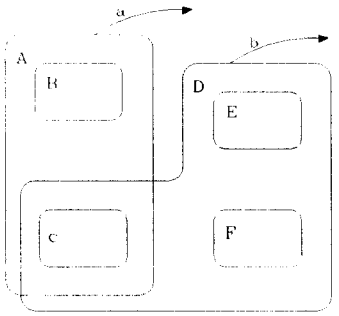

Figura A.5: Superposição de bolhas

Superposição de bolhas pode ser usado economicamente para descrever uma variedade primitivas de sincronização e para refletir situações natural em sistemas complexos. Em [DRU 85] [DRU 89] são apresentados vários exemplos. Superposição de bolha causa problemas semânticos, especialmente quando a superposição envolve componentes ortogonais. Sugestões para uma síntaxe e semântica formal de superposição de bolhas são apresentadas em [HAR 89].

[1] *Overlapping*

Apêndice B

LEG: Sintaxe em BNF

Apresenta-se a seguir a sintaxe de LEG discutida no capítulo 3.

- **Programa**

  <programa> ::= [<includes>] <bloco em LEG> <programa principal>

- **Includes**

  <includes> ::= #**include** "<arquivo>"

- **Identificador**

  <identificador> ::= <letras> <letras> | <digito>

- **Bloco**

  <estado> ::= <bloco em LEG> | <chamada>

  <bloco em LEG> ::= `blob` <identificador> <corpo>

  <corpo> ::= {[<comandos gerais>] [<estado/componente>] }

  <estado/componente> ::= <estado> | <componente>

  <componente> ::= <estado> | <estado> {<estado> | <estado>}

  <chamada> ::= `call` <identificador> ;

- **Comandos gerais**

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

- **History**

  <history> ::= history <tipos> ;

  <tipos> ::= <tipo> | <tipo> , <tipo>

  <tipo> ::= deep | shallow | pred

- **Ações**

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

- **Transições**

  <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>

- **Programa Principal**

  <programa principal> ::= main <bloco em LEG>

Apêndice C

Gramática Yacc para LEG

Na gramática apresentada abaixo foram eliminadas as ações em C associadas a cada regra.

```
program          : includes
                   blobprograms
                   mainprog
                 ;

blobprograms     : /* empty */
                 | blobprgs
                 ;

blobprgs         : blobfunction includes
                 | blobprgs blobfunction includes
                 ;

blobfunction     : BLOB IDENTIFIER
                   body
                 ;

mainprog         : MAIN
                   blobfunction
                   includes
                 ;
```

```
includes            : /* empty */
                    | incfiles
                    ;

incfiles            : INCFILE
                    | incfiles INCFILE
                    ;

body                : '{'
                        history
                        action
                        transitions
                        compstate
                      '}'
                    ;


history             : /* empty */
                    | HISTORY
                      ';'
                    ;


feature1            : /* empty */
                    | feature   ',' feature
                    | feature
                    ;

feature             : SHALLOW
                    | DEEP
                    | PRED
                    ;

actions             :  /* empty  */
                    | act
                    ;

act                 : act rest
```

```
                        | rest
                        ;

rest                    : action_command ':'
                          MENORSTAR
                          cfunction
                          STARMAIOR
  ';'
                        ;

cfunction               : CFUNCTION
                        | cfunction CFUNCTION
                        ;

action_command          : ON_ENTRY
                        | THROUGHOUT
                        | ON_EXIT
                        ;

transitions             : /*  empty  */
                        | transition_command
                        ;

transition_command      : TRANSITION
                          '{'
                             transition_body
                          '}'
                        ;

transition_body         :  transition_statements
                        |  transition_body
                           ';'
                           transition_statements
                        ;

transition_statements   : ON_EVENT
                          '('
                          IDENTIFIER
                          ')'
```

```
                        if_statement
                        TO
                        target
                        shooting
                      ;

shooting              : /* empty */
                      | DO
                        MENORSTAR
                        cfunction
                        STARMAIOR
                        ;

if_statement          :  /* empty */
                      |  IF
                         MENORSTAR
                         cfunction
                         STARMAIOR
                      ;

target                :  target '.'
                         IDENTIFIER
                      |  IDENTIFIER
                      ;

compstate             : /*  empty  */
                      | components
                      | states
                      ;

state                 : BLOB
                        IDENTIFIER
  body
                      | CALL
                        IDENTIFIER
                        ';'
                      ;

components            : components '|' state
```

```
                        |  state '|' state
                        ;

states                  :  state
                        |  states state
                        ;
```

Apêndice D

Aplicações em X Toolkit

Este apêndice apresenta a estrutura básica na qual toda aplicação em *X Toolkit* (Xt) está baseada. Assim, uma aplicação que utiliza Xt necessita efetuar a seguinte seqüência de passos padrões.

1. Incluir `<X11/Intrinsics.h>` em seu programa de aplicação, que são arquivos de inclusão padrão para Xt. Este arquivo de cabeçalho inclui automaticamente `<X11/Xlib.h>`, assim todas as funções Xlib são também definidas;

2. Incluir os arquivos de cabeçalhos específico para cada classe de *widget* que está sendo usado pela aplicação. Por exemplo, `<X11/Label.h>` e `<X11/Command.h>` (Cada classe de *widget* tem um arquivo de inclusão particular, no qual é usado no código do *widget*);

3. Iniciar o *Toolkit* com a função apropriada `XtInitialize`. (Aplicações complexa pode necessitar outras chamadas separadas, para `XtToolkit Initializa`, `XtCreateApplicationContext`, `XtDisplayInitiate` e `XtAppCreateShell`);

4. Criar *widgets* e associá-los a seus ancestrais. Este procedimento é feito através de chamada a função `XtCreateManagedWidget` para cada *widget*;

5. Se a um *widget* está associada uma função *callback*, a qual é geralmente definida usando o argumento `XtNcallback` ou pela função `XtAddCallback`, declara-se estas função dentro da aplicação;

6. incluir a chamada da `XtRealizeWidget` que é invocada somente uma vez na aplicação no *wigdet shell* (i.e., *widget* ancestral) retornado pelo `XtInitializa`. Neste ponto são criados as janelas para os *widgets* definidos;

7. Em seguida, começa o *loop* que processa eventos através da função `XtMainLoop`. Neste ponto, Xt toma o controle da aplicação e opera os *widgets*.

A seguir tem-se um exemplo apresentado em [NYE 90] (volume 1 capítulo 2) que espelha basicamente a estrutura da maioria das aplicações em Xt. Este exemplo é um simples programa que coloca uma mensagem e chamada uma função *callback* da aplicação.

```
#include <stdio.h>

/*  Include files required for all Toolkit programs */

#include <X11/Intrinsics.h>  /* Intrinsics Definitions */
#include <X11/StringsDefs.h> /* Standard Name-String
                                Definitions */

/* Public include files for widgets we actually use
   in this file  */

#ifdef X11R3
#include <X11/Command.h>     /* Athena Command Widget */
#else                        /* R4 or later  */
#include <X11/Xaw/Command.h> /* Athena Command Widget */
#endif                       /* X11R3 */


main(argc,argv)
int  argc;
char **argv;

{
   widget toplevel, goodbye;
```

```
    toplevel = XtInitialize (
               argv[0],     /* application name */
               "XGoodbye",  /* application class */
               NULL,        /* Resource Mgr. options */
               0,           /* Number of RM options */
               &argc,       /* number of args   */
               argv,        /* command line     */
               );

    goodbye = XtCreateManagedWidget (
              "goodbye",         /* arbitrary widget name */
              CommandWidgetClass, /* widget class from
                                     command.h */
              topLevel,           /* parent widget  */
              NULL,               /* argument list */
              0,
              );

    XtAddCallback (goodbye, XtNcallback, Quit, 0);

    /* Create windows for widgets and map them */

    XtRealizeWidget(topLevel);

    /* Loop for events */

    XtMainLoop();

}
```

A função *callback* associados ao *widget* criado vem a seguir.

```
/* Quit button callback function */

void Quit(w, client_data, call_data)
widget w;
caddr_t client_data, call_data;
```

```
{
  fprinf(stderr, "It was nice knowing you.\n");
  exit(0);
}
```

Apêndice E

Aspectos Gerais da Implementação

Apresenta-se neste apêndice alguns detalhes referentes ao ambiente operacional do tradutor, uma descrição suscinta de seus arquivos e as possíveis mensagens de erros.

E.1 Ambiente Operacional

Foram escolhidos dois sistemas operacionais como ambiente de desenvolvimento e execução do tradutor – Unix e MS-DOS. A atual versão encontra-se implementada nos sistemas SunOS (Sun Microsystems, BSD) e Clix (Intergraph, System V.3). No MS-DOS utilizou-se o compilador Turbo C [BOR 87].

A implementação, incluindo o código invariante gerado, está com aproximadamente 7000 linhas. Na seção seguinte são descritos os arquivos do tradutor.

E.2 Módulos do Tradutor

São descritos, resumidamente, a seguir os arquivos que compõem o tradutor:

- `main.c` – arquivo onde se encontra o programa principal;

- `yyinsert.c` - neste arquivo estão implementados as funções que criam a árvore descritiva de estadogramas, na qual é feita o segundo passo da tradução, e as tabelas de símbolos e de transições;
- `code.c` - O gerador de código, traduz as estruturas internas que descrevem o código LEG construções na linguagem C;
- `global.h` - define estruturas globais do tradutor;
- `blob.y` - arquivo como as produções da gramática de LEG para *Yacc*;
- `blob.l` - expressões regulares que reconhecem símbolos permitidos por LEG;
- `y.tab.c` - analisador sintático gerado pelo *Yacc*. No caso do *Bison*, o arquivo gerado é `blob.tab.c` e no MS-DOS este arquivo recebe o nome `ytab.c`;
- `lex.yy.c` - analisador léxico gerado pelo *Lex*;
- `error.c` - emite mensagens de erros;
- Existem outros arquivos onde são definidos os protótipos das funções do seu arquivo `.c` correspondente, por exemplo para `code.c` tem seus protótipos definidos em `code.h` e assim por diante.

A seguir tem-se a descrição dos arquivos que são gerados pelo tradutor:

- `bmain.c` - programa principal gerado;
- `engine.c` - parte invariante do código gerado, neste árquivo está implementada a função que realiza a mudança de estados;
- `traverse.c` - neste arquivo é implementado o percurso na lista de estados correntes para efetuar uma possivel mudança de estado;
- `bfunct.c` - arquivo de funções que especificam as transições entre bolhas. No final deste arquivo e também gerado o vetor de funções;
- `bstates.h` - definições de macros que associam identificadores de bolhas a inteiros;
- `bevents.h` - definição de macros para os eventos;

- `hierarc.h` – definição de estrutura hierarquica que espelha a árvore do estadogram;
- `bentry.c` – função que executa código C definido pelo usuário ao entrar em uma bolha;
- `bexit.c` – executa o código ao sair de uma determinada bolha.

E.3 Mensagens de Erro

A atual versão do tradutor encontra-se com um número reduzido de mensagens de erros. A medida que é iniciado o processo de tradução um diagnóstico é emitido pelo tradutor caso um possível erro seja encontrado no código fonte em LEG.

As mensagens de erros, em geral, informam o nome do arquivo onde o erro ocorreu, o número da linha e algum comentário adicional sobre o erro. As notações AAA, XXX, BBB e FFF representam o nome do arquivo, número da linha em que foi detectado o erro, nomes de bolhas e nome de funções escritas em LEG, respectivamente.

- **Error: (AAA) Line XXX: transition not supported, BBB is a component** – encontrada uma transição para a componente corrente, isto é, a componente que acaba de ser lida pelo tradutor. Não é permitida especificar uma transição em LEG para uma determinada componente, este procedimento é permitido especificando-a para a bolha (sua ancestral) na qual esta é envolvida;
- **Error: (AAA) Line XXX: Invalid transition between components, from BBB to BBB** – foi especificado transição de uma componente para outra, o que não é permitido por LEG.
- **Error: (AAA) Line XXX: Transition to undefined state, from BBB to BBB** – encontrada uma transição para uma bolha não definida;
- **Error: (AAA) Line XXX: State BBB does not exist** – a bolha BBB especificada na função `in_Blob` dentro do comando `if` não é uma bolha válida, isto é, esta bolha não existe no estadograma especificado;

- **Warning: (AAA) Line XXX: State BBB already defined** – bolha BBB definida mais de uma vez, assim fica valendo a última definição desta bolha;
- **Error: (AAA) Line XXX: Function FFF not defined** – A função chamada através do comando `call` em LEG não foi definida previamente;
- **Error: (AAA) Line XXX: Ambiguous definition on command history** – identificação imcompatível do comando `history`. Foi especificado no mesmo comando `history` os tipos de *histories* `deep` e `pred`;
- **Error: (AAA) Line XXX:More than one transition to BBB** – mais de uma transição para uma mesma bolha foi especificada em um mesmo comando `transition`;
- **Warning: (AAA) Line XXX: File "bfunct.c" was not generated, transtions not found** – nenhuma transição foi especificada no código fonte, portanto o arquivo efetua as transições, `bfunct.c`, não foi gerado.

Índice

M

O

P

R

S

T

U

W

X

Y