Geração de Gerenciadores de Sistemas Reativos*

Antonio G. Figueiredo Filho†

Hans K. E. Liesenberg‡

Departamento de Ciência da Computação
IMECC - UNICAMP
Caixa Postal 6065
13081 - Campinas, SP
Fone: (0192) 39-8442
Fax: (0192) 39-5808
e-Mail: antonio@dcc.unicamp.ansp.br
hans@dcc.unicamp.ansp.br

Sumário

Este artigo apresenta um gerador de programas adequados para implementar o controle de sistemas reativos complexos. A ferramenta utiliza *estadogramas*, uma extensão de diagramas de estados convencionais suportando conceitos de *hierarquia, concorrência* e *comunicação*. A sua entrada se constitue de uma descrição textual de um estadograma e produz como saída um programa funcionalmente equivalente em C. Em particular, trata-se de uma ferramenta útil para o desenvolvimento de gerenciadores de diálogos em interfaces homem-máquina. No final deste trabalho é descrita a utilização da ferramenta desenvolvida na construção de um editor topológico.

Abstract

This paper presents a program generator appropriate to implement complex reactive systems. The tool uses statecharts, an extension of conventional state diagrams supporting the notion of hierarchy, concurrency and communication. Its input consists of a textual description of a statechart and it outputs a functionally equivalent C program. The tool is useful for the implementation of man-machine interface manegement systems. At the end of this paper the construction of a topological editor aided by the developed tool is described.

*Este trabalho teve apoio financeiro do CNPq e Fapesp.

†Mestrando em Ciência da Computação (UNICAMP, Campinas-SP); Eng. Civil (UEMA, São Luís-MA); Áreas de Interesse: Interface com o usuário, computação gráfica e ambientes de programação.

‡Ph.D. (University of Newcastle upon Tyne, Inglaterra); Áreas de Interesse: Ambientes de programação, ferramentas de apoio a projeto de circuito integrados e impressos.

1 Introdução

O presente trabalho[1] consiste do desenvolvimento de um Gerador de Gerenciadores de Sistemas de Controle e baseia-se na notação de *"estadogramas"*[2] originalmente proposta por D. Harel [HAR 87] para representar especificações funcionais de sistemas de hardware ([DRU 85], [DRU 89]) em termos de diagramas de estados suportando definições hierárquicas. O gerenciador, descrito a seguir, recebe como entrada uma especificação definida através de estadogramas e gera um programa em C funcionalmente equivalente a esta especificação. O sistema completo será composto por um editor gráfico de *bolhas*[3], parte constituintes de um estadograma, que produz uma descrição textual intermediária. Esta descrição serve, então, de entrada ao gerador propriamente dito que, a partir da descrição fornecida, produz o programa em C equivalente ao estadograma originalmente definido via editor gráfico. O presente trabalho aborda a segunda parte do sistema conforme ilustrado na figura 1.

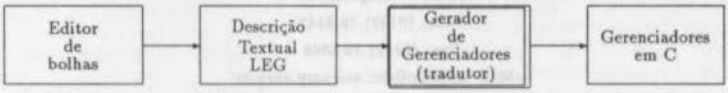

Figura 1: Descrição esquemática da ferramenta proposta

É possível observar na literatura que o enfoque estado/evento tem sido utilizado cada vez mais no contexto de Sistemas de Desenvolvimento de Interfaces Homem-Máquina adotando-se, geralmente, diagramas de transição de estados ou máquinas de estados finitos para a especificação de sistemas reativos ([HAT 89a], [HAT 89b], [MEY 89], [JAC 85], [JAC 83], [SUN 82], [TEN 81]). O comportamento de um diagrama de estado está associado, basicamente, 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, podem ser 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 (veja em [HAR 85] e [PNU 86] discussões sobre sistemas reativos e seus comportamentos), a representação deverá permitir a descrição de estruturas hierárquicas e modulares. O estadograma é uma representação bastante flexível 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 profundidade e ortogonalidade.

Na figura 2 é apresentado um estadograma que consiste de um encaixe sucessivo de bolhas com transições entre as mesmas representando o controle de um cronômetro em um relógio digital modelado por Harel em [HAR 87]. O estado global do estadograma é definido pela lista de bolhas no qual o sistema se encontra em um determinado instante.

No estadograma, a reatividade do sistema não é expressa simplesmente por uma mudança de seu estado, por uma ocorrência de eventos e pelo atendimento a possíveis condições associadas a estes. Também é possível associar às bolhas um *script* de ações e atividades. Uma ação pode ser executada ao se entrar e/ou sair de uma determinada bolha e uma atividade é caracterizada pela sua execução

[1] Uma versão preliminar deste trabalho é apresentada em [FIG 90]

[2] *statecharts*

[3] *blobs*

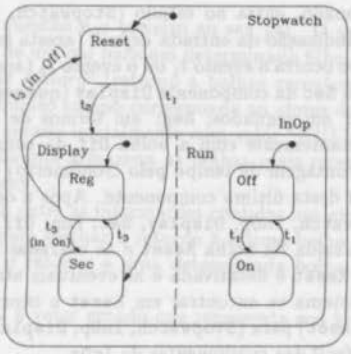

Figura 2: Estadograma de Cronômetro

enquanto o sistema permanecer na bolha associada.

Algumas extensões e aplicações de estadogramas são abordadas em [HAR 88], [HAR 85] e [DRU 86].

2 Estrutura dos Programas Gerados

Um gerenciador construído pelo gerador, 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 atividades e ações associadas às bolhas envolvidas nas mudanças efetuadas. A ferramenta é gerada na linguagem C. A seguir são introduzidas 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 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 programa construído pelo gerador 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]. Desta forma, uma bolha que contém outra é representada por um nó na árvore e a contida como filho deste nó.

A figura 2 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 ilustra bem diversas características de estadogramas.

O cronômetro, quando acionado, entra no estado (Stopwatch, Reset)[4] com a ação associada de zerar o cronômetro[5] dada a indicação da entrada *default* (aresta com origem em círculo cheio). O cronômetro sai deste estado, caso ocorra o evento t_1 ou o evento 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).

A estrutura da árvore que representa o encaixe de bolhas do estadograma da figura 2 é apresentado na figura 3. Os nós 1, 2, 3, 4, 5, 6, 7, 8 e 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.

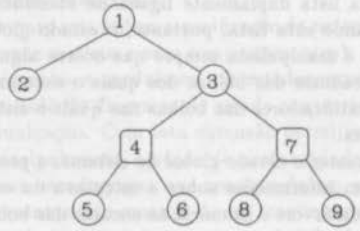

Figura 3: Estrutura da árvore representando a hierarquia do estadograma

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 composto por $n+1$ elementos, onde n representa o número de nós da árvore. A estrutura utilizada para representar os nós das árvores 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

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

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

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*[6] (retorno a bolhas recentemente visitadas e não retorno através de transições default). O quinto campo corresponde ao *status* dos *history* identificado através de um caracter. 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 da existência de history.

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 em profundidade na hierarquia e o sexto para efetuar o retorno a uma determinada bolha que seja irmã anteriormente visitada da bolha corrente.

Para o estadograma da figura 2 o vetor gerado que representa sua hierarquia é dado abaixo.

```
lista_global *hierarchy[10] =
  {
    { 0,  0,  0,  0, 'O', NULL},      /*  NULL  */
    { 2,  1,  0,  0, '*', NULL},      /*   1    */
    { 0,  3,  1,  0, 'O', NULL},      /*   2    */
    {-4,  2,  1,  0, 'O', NULL},      /*   3    */
    { 5, -7,  3,  0, 'O', NULL},      /*   4    */
    { 0,  6, -4,  0, 'O', NULL},      /*   5    */
    { 0,  5, -4,  0, 'O', NULL},      /*   6    */
    { 8, -4,  3,  0, 'O', NULL},      /*   7    */
    { 0,  9, -7,  0, 'O', NULL},      /*   8    */
    { 0,  8, -7,  0, 'O', NULL}       /*   9    */
  }
```

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 seja acionado e que, em seguida, ocorra o evento t_1, então a seqüência dos valores na quarta posição de cada elemento ficaria 0, 3, 0, -4, 6, 0, 0, 8, 0, 0 armazenando informações necessárias para efetuar um posterior retorno às mesmas bolhas, caso `Stopwatch` seja abandonado e posteriormente revisitado. A especificação de retorno a bolhas mais recentemente visitadas é feita através do mecanismo de *history*.

A seguir são apresentados alguns fragmentos do programa equivalente ao estadograma da figura 2 para ilustrar como as transições são especificadas no código gerado. O código abaixo corresponde ao programa principal.

```
main()
{
  current_state = InitState();
  do {
```

[6] O mecanismo de *history* proposto por Harel foi estendido para atender necessidades práticas detectadas no uso da ferramenta por terceiros na construção de interfaces homem-máquina.

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

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 um comando repetitivo onde, em cada iteração, é obtido o "próximo" evento e, em função do mesmo, é analisado o estado corrente para ver se este é afetado. Em caso afirmativo, a árvore é percorrida de tal forma a encontrar o primeiro "ancestral" comum[7] 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 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 2, 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 e os eventos t_1, t_2 e t_3 aos inteiros 1, 2 e 3.

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

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

```
}
```

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, é afetada pelo evento corrente. O percurso da lista é controlada 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 utualizado. Para efetuar esta implementação da função `traverse()` de uma maneira eficiente foi criada um vetor de funções, mostrado abaixo, onde `n` representa o número de bolhas do estadogramas em questão.

```
lista_global  *(*tab_case[])(int evento)
        =  {NULL,fcase1,fcase2,fcase3, .... ,fcasen}
```

Os índices deste vetor estão associados aos identificadores das 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 a 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. Nestas funções também são verificadas as condições estabelecidas para determinadas transições como, por exemplo, as transições que têm origem na bolha `Reg` (identificada no código pelo inteiro 5) da figura 2. 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()`.

A estrutura de cada uma das funções que especificam as transições entre bolhas é análoga à função abaixo que implementa as transições a partir da bolha `Reg`.

```
lista_global *fcase5(int event )

{
   lista_dupla *ptaux;
   ptaux = pt;

   switch ( event )
   {
     case 3 : if (in_blob(8))
                  return( fromBlobtoBlob(5,2) );
              if (in_blob(9))
                  return( fromBlobtoBlob(5,6) );
     case 4 : return( fromBlobtoBlob(5,6) );
   }
   return( ptaux );
}
```

3 Programando na Linguagem de EstadoGramas: Um exemplo em Interface Homem-Máquina

A linguagem descritiva de estadogramas, ou simplesmente Linguagem de EstadoGramas (LEG), permite a descrição, em forma textual, das características conhecidas dos estadogramas: hierarquia,

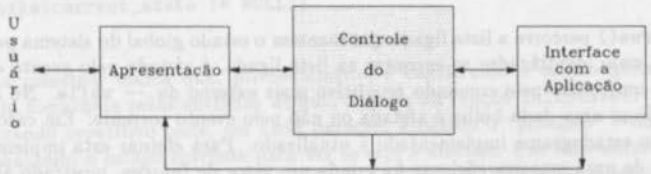

Figura 4: Modelo Lógico de Interface Homem-Máquina

concorrência e comunicação. Contudo, esta linguagem tem como propósito permitir que o processo de desenvolvimento de sistemas reativos se torne bastante facilitado e mais rápido a sua implementação. Uma aplicação típica desta linguagem é dar suporte ao controle do diálogo (i.e., a componente de controle do diálogo como visto na figura 4) de uma aplicação.

Um programa escrito em LEG associa código inerente da aplicação através de funções escritas em linguagem C e código próprio de LEG responsável pelo controle desejado da aplicação. A parte do código em C deverá estar em arquivos que posteriormente serão "ligados"[8] com o código gerado na segunda fase deste processo de tradução. Em síntese, os comandos desta linguagem são direcionados para o controle de aplicações, enquanto que as funções em C estarão associadas às ações possivelmente disparadas na entrada e/ou na saída de um bolha, ou simplesmente disparadas por uma transição entre bolhas causada por um determinado evento[9], definido assim a semântica do estadograma.

Para caracterizar a ferramenta apresentada, exemplifica-se o seu uso no contexto da geração de sistemas de gerenciamento de interface homem-máquina baseado em um modelo lógico, proposto por Seeheim (veja [GRE 85] e [GRE 86]) e esboçado na figura 4.

Este modelo permite identificar com clareza as principais componentes envolvidas em uma interface homem-máquina explorando a independência do diálogo. A aplicação do sistema gerado, neste contexto, concentra-se na componente de controle do diálogo, a qual define a estrutura do diálogo entre o usuário e o programa de aplicação. O controlador de diálogos deve manter um estado corrente da interface e ter controle sobre este, pois as ações executadas dependerão do contexto do diálogo. Para ilustrar o uso de estadograma neste contexto, exemplifica-se o fluxo de controle de um comando `Login` através da figura 5. A referência às ações no código LEG foi omitida, exceto na primeira bolha.

Neste exemplo, o comando `Login` especificado apresenta inicialmente as mensagens de apresentação do sistema (bolha `Start`) que dispara o evento `Login Prompted` e, em seguida, requer o nome do usuário na bolha `User Protocol`, através da bolha `Get User`. Se o sistema não reconhecer o nome do usuário (bolha `User Checker`), um novo pedido é apresentado até que seja fornecido um nome válido, disparando a transição rotulada por `User OK` que causa a saída de `User Protocol` e a entrada no estado `PW Protocol` e nos sub-estados `Get PW` e `Reset` das componentes `Password` e `Counting`, respectivamente. A seguir o sistema solicita a senha (bolha `Get PW`) e o contador de senhas inválidas é inicializado (bolha `Reset`). Segue-se o teste da senha (bolha `PW Checker`); se tiver sido fornecida uma senha incorreta, então o contador de senha inválida (bolha `Bad PW Counter`) na componente `Counting` é incrementado e o usuário é mais uma vez solicitado para reapresentar uma senha até que o limite

[8] *linked*

[9] Ações em estadogramas também podem ser associadas a transições.

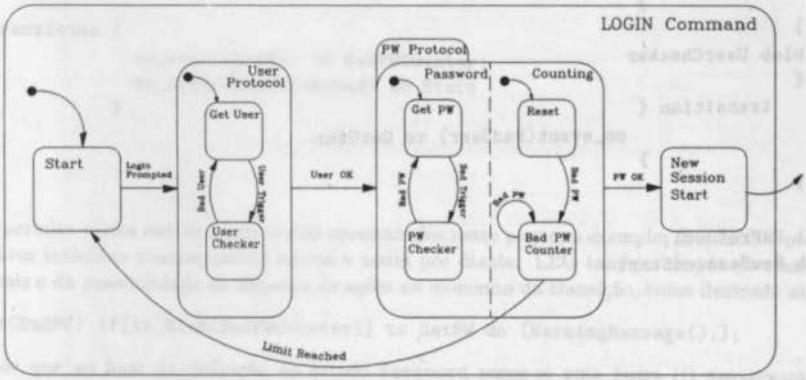

Figura 5: Especificação do Comando *LOGIN* em Estadograma

aceitável de entradas inválidas seja alcançado. Nesta última situação é disparado a transição **Limit Reached** fazendo com que o sistema reinicie o comando **LOGIN**; caso contrário, ou seja, a senha foi fornecida corretamente, a transição **PW OK** é disparada permitindo a entrada no estado **New Session Start**.

Apresenta-se a seguir um código em LEG referente ao exemplo apresentado na figura 5. Segue abaixo o programa principal deste exemplo.

```
# include "passwd.b"

main blob Login
{
   blob Start
   {
      on_entry : [Greet();];
      transition {
                  on_event(LoginPrompted) to UserProtocol
                 }
   }
   blob UserProtocol
   {
      transition {
                  on_event(UserOk) to  PWProtocol
                 }

   blob GetUser
   {
      transition {
```

```
                    on_event(UserTrigger) to UserChecker
                }
    }
    blob UserChecker
    {
        transition {
                    on_event(BadUser) to GetUser
                }
    }
  }
  call PWProtocol;
  blob NewSessionStart
  {
  }
}
```

O estado PWProtocol, contido no arquivo passwd.b incluído no início do programa principal, é definido abaixo.

```
blob PWProtocol
{
  transition {
              on_event(PWOk) to NewSessionStart
            }
  blob Password
  {
    blob GetPW
    {
      transition {
                  on_event(BadTrigger) to PWChecker
                }
    }
    blob PWChecker
    {
      transition {
                  on_event(BadPW) to GetPW
                }
    }
  } |
  blob Counting
  {
    blob Reset
    {
      transtion {
                 on_event(BadPW) to BadPWCounter
               }
    }
```

```
    blob BadPWCounter
    {
      transition {
                  on_event(BadPW) to BadPWCounter;
                  on_event(LimitReached) to Start
                 }
    }
  }
}
```

LEG permite vários outros recursos não apresentados neste pequeno exemplo. Em LEG é possível que arquivos incluídos possam incluir outros e assim por diante. LEG também dispõe de transições condicionais e da possibilidade de disparos de ações no momento da transição, como ilustrado abaixo.

```
on_event(BadPW) if[in_Blob(BadPWCounter)] to GetPW do [WarningMessage();];
```

Note-se que ao final da definição do estado `Password` segue-se uma barra (|) que separa esta componente da componente `Counting`.

4 Exemplo de Utilização do Tradutor

Foi desenvolvido um editor topológico, chamado Editor Topológico da LegoShell ou simplesmente Editor da LegoShell, por H. Piñón [PIÑ 90] baseado em estadogramas. Por ocasião da fase de desenvolvimento deste editor o tradutor LEG ainda não estava operacional. Assim sendo, as tabelas que descrevem a hierarquia de bolhas, a função `traverse()` e a função de disparos de ações foram geradas manualmente. Nesta fase foi utilizada a parte constante do código gerado pelo tradutor responsável pelo controle do estado do estadograma que especifica o controle do diálogo. Após a implementação do tradutor, escreveu-se o código do Editor da LegoShell em LEG e em seguida, utilizando-se do tradutor, foi feito a tradução para código em C obtendo-se um programa funcionalmente equivalente ao anterior.

O Editor da LegoShell é constituido basicamente de uma ferramenta capaz de manipular computações complexas. Estas computações são especificadas através de uma linguagem gráfica, chamada LegoShell, a qual consiste da generalização, através de extensões do conceito de *pipe* do UNIX. Esta ferramenta é uma das principais do ambiente A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquia de Abstração em Níveis Diferenciados) [DLI 87a] e [DLI 87b] que está sendo desenvolvido no Departamento de Ciência da Computação (DCC) da UNICAMP, o qual tem como objetivo principal oferecer suporte para o desenvolvimento de sistemas grandes, complexos e não convencionais.

A linguagem gráfica da LegoShell representa basicamente, como já foi dito, uma extensão do conceito unidimensional do *pipe* do UNIX para mais dimensões através da adição de conectores do tipo *mailbox* e *broadcast* às conexões entre programas, arquivos, dispositivos e computações (i.e., abstrações de *pipes* multidimensionais utilizados como blocos funcionais na construção de novos pipes). A LegoShell manipula objetos de interface padrão do tipo porta de entrada e saída ligadas por meio dos conectores que permitem a especificação do fluxo de dados entre os mesmos.

Assim, o Editor da LegoShell é uma ferramenta destinada a manipular estas computações e objetos básicos propostos pela LegoShell. Este editor, além de permitir a manipulação dos objetos servirá de interface para invocar funções inerentes ao ambiente.

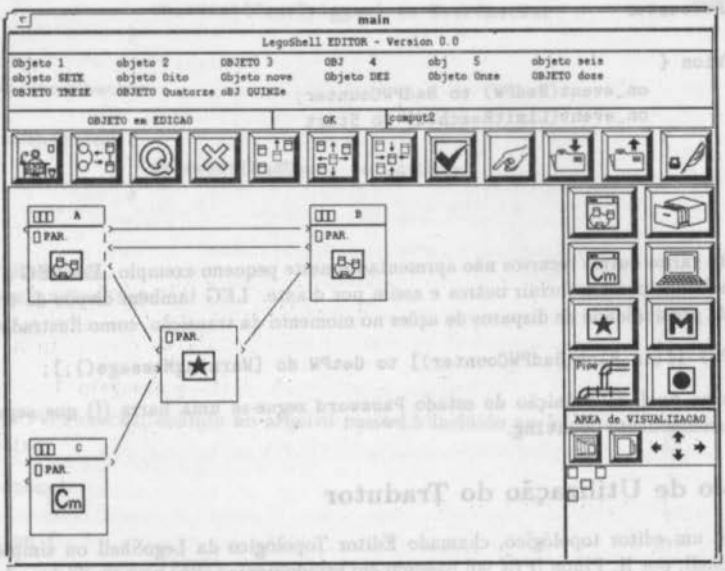

Figura 6: Interface homem máquina do Editor da LegoShell

A interface-usuário do Editor da LegoShell é apresentado na figura 6. Este editor teve como ambiente de desenvolvimento de sua implementação o sistema SunOS 4.1 que é uma versão UNIX para SPARCstations da Sun e na construção do módulo de apresentação da interface-usuário foi utilizada a biblioteca de *Widgets* para o sistema de janelas *X-Windows*.

5 Conclusão

O gerenciador desenvolvido pretende ser uma ferramenta que facilite a especificação de gerenciadores de sistemas complexos privilegiando o aspecto de controle. A definição das ações e atividades é da responsabilidade do usuário e são definidas em termos de funções e processos invocadas e acionadas pelos gerenciadores gerados. Provavelmente é interessante manter para cada domínio de aplicações bibliotecas de ações e atividades específicas. Espera-se que a ferramenta em questão seja particularmente útil para desenvolver gerenciadores de diálogos em interfaces homem-máquina.

Referências

[DLI 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, abr 1987, pp. 313-322.

[DLI 87b] Drummond, R. e Liesenberg, H. *Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS*, I Encontro IBM de Ciência e Tecnologia em Informática, nov 1987.

[DRU 85] Drunsinsky, D. & Harel, D., *Using Statecharts for Hardware Description*, Technical Report CS85-06, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, December 1985.

[DRU 86] Drunsinsky, D. & Harel, D., *Statecharts as an Abstract Model for Digital Control Units*, Technical Report CS86-12, Dept. of Applied Mathematics & Computer Science, The Weizmann Institute of Science, Rehovot, Israel, May 1986.

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

[FIG 90] Figueiredo Filho, A. G. e Liesenberg, H. K. E., *Geração de Gerenciadores de Sistemas Reativos*, Relatório Técnico, DCC-IMECC-UNICAMP, No. 20, maio 90.

[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.

[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 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 88] Harel, D., *On Visual Formalisms*, Comm. ACM, Vol. 31, No. 5, May 1987, pp. 514-530.

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

[HAT 89b] Hartson, R. and Hix, D., *Human-Computer Interface Development: Concepts and Systems for its Management*, ACM Computing Surveys, Vol. 21, No. 1, March 1989, pp. 5-92.

[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.

[MEY 89] Meyers, B. A., *User-Interface Tools: Introduction and Surveys*, IEEE Software, January 1989, pp. 15-23.

[PIÑ 90] Piñón A., H., *Editor Topológico para a Linguagem de Especificação de Computações LegoShell*, Tese de Mestrado, DCC-IMECC-UNICAMP, Campinas, SP, dez 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 al., Lecture Notes in Computer Sciences, Vol. 224, Springer-Verlag, New York, (1986), pp. 510-584.

[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.

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