Um Gerador de Aplicações para Sistemas Reativos

Carlos A. Alves Meira
Paulo Cesar Masiero

ICMSC-USP
C.P. 668
13560 - São Carlos - SP
e_mail: pcmasier@brusp.bitnet

RESUMO

A atividade de desenvolvimento de software pode se beneficiar em termos de qualidade e produtividade, quando utiliza programas gerais que conseguem automatizar parte substancial do trabalho rotineiro envolvido nessa atividade. Um programa desse tipo, denominado genericamente de Gerador de Aplicação, é apresentado, para a área de Sistemas Reativos. Dada uma especificação em Statechart, diversos produtos podem ser obtidos com o uso do gerador. Como exemplo, o trabalho mostra a geração de um programa simulador de Statechart.

1 Introdução

A sempre crescente demanda por sistemas de software cada vez mais abrangentes e complexos e a ausência de uma forma mais organizada e sistematizada no processo de desenvolvimento de software, resultaram, muitas vezes, na produção de sistemas de baixa qualidade. Além da baixa qualidade, vários outros problemas ocorreram, entre eles o alto custo de manutenção dos sistemas, dando origem ao termo "Crise de Software" [Pre87].

Na Engenharia de Software, os custos e a produtividade do desenvolvimento de software são dois aspectos importantes. Os custos atuais para o desenvolvimento são considerados excessivos e a produtividade, em geral, é considerada baixa, devido a fatores tais como: a habilidade individual e a complexidade do produto [Fai85]. Outro aspecto importante é a qualidade dos sistemas produzidos. O ideal, então, seria encontrar meios para reduzir os custos e ao mesmo tempo aumentar a produtividade, mantendo um bom nível de qualidade do software.

Os Geradores de Aplicação (GA) são ferramentas de apoio por computador ao desenvolvimento de software que, além de reduzir os custos e aumentar a produtividade, ajudam a melhorar a qualidade dos sistemas produzidos. Essencialmente, um gerador de aplicação é um utilitário que, a partir de uma especificação em alto nível de um problema implementável, transforma automaticamente essa especificação na implementação do problema. Os geradores de aplicação são mais conhecidos e têm obtido sucesso no domínio de Sistemas de Informação. Neste trabalho dá-se ênfase ao uso dessa técnica em outros domínios de aplicação.

O trabalho está organizado da seguinte forma: a seção dois faz uma introdução sucinta aos geradores de aplicação; a seção três apresenta o domínio de aplicação – sistemas

reativos – onde o gerador é utilizado; a seção quatro apresenta a geração de um programa simulador de Statecharts; a seção cinco compara a solução apresentada neste trabalho com outras abordagens; e, a última seção, a seis, apresenta as conclusões do trabalho.

2 Geradores de Aplicação

Tipicamente, durante a fase de análise de requisitos, o analista de sistemas deve interagir com o cliente para especificar "o que" o sistema deve fazer. Feito isso, o resultado da análise é passado para o projetista de sistemas, que é quem define "como" o sistema deve executar as funções especificadas.

Num ambiente com o suporte de geradores de aplicação surgem dois novos personagens: o analista de domínios e o projetista de domínios. A figura 1 [Cle88] mostra como ficam as várias funções num ambiente típico com o suporte de geradores de aplicação. O analista de domínios é quem especifica os requisitos do gerador, analisando as necessidades e os requisitos de uma série de problemas na área de interesse. O projetista de domínios toma essa especificação e a implementa no gerador. O projetista de sistemas, agora com o auxílio do gerador de aplicação, recebe a especificação do sistema e a transforma numa implementação.

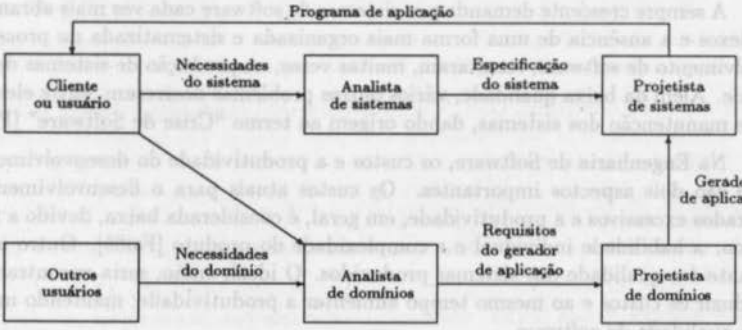

Figura 1: Relacionamento num ambiente com o suporte de geradores de aplicação

Seria ideal que o trabalho do projetista de sistemas se reduzisse apenas a fornecer corretamente a especificação para o gerador de aplicação, produzindo o sistema por completo. Entretanto, na prática, os geradores de aplicação geralmente criam apenas partes de um sistema, deixando o resto do trabalho para o projetista. Essas partes são chamadas de produtos da aplicação, que podem ser estruturas de dados, subrotinas ou segmentos de código.

A razão pela qual apenas partes de sistemas são criadas por geradores é que as aplicações podem não ser totalmente representadas por uma notação simples. Existem

aspectos da aplicação que podem não ser descritos eficientemente e é preferível ter-se uma representação simples da aplicação, mesmo que restrita, do que uma representação totalmente abrangente, mas complexa. Em [Lev86], Levy afirma que nem sempre a automação completa do processo de produção de software é a solução mais econômica e sugere critérios para avaliar percentualmente a porção do sistema a ser gerada.

O processo básico necessário para desenvolver um sistema usando um gerador de aplicação, mostrado na figura 2, começa com a especificação. Ela é fornecida para o gerador de aplicação, que cria o produto da aplicação na linguagem de programação alvo. Finalmente, o produto, juntamente com o código produzido manualmente pelo projetista de sistemas, é compilado para produzir o sistema executável.

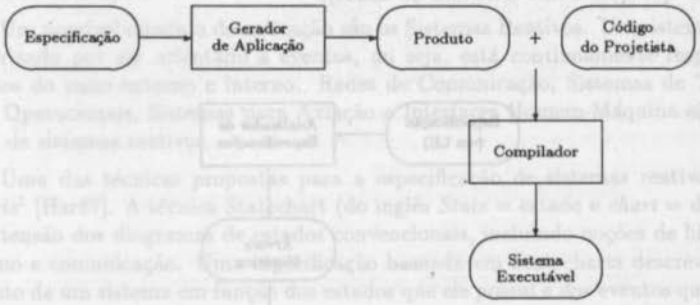

Figura 2: Processo de desenvolvimento usando um gerador

Com o uso de um gerador de aplicação a fase de manutenção fica facilitada, devido à modularidade e à uniformidade do código gerado e, ainda, para corrigir erros de especificação: ao invés de se ter que modificar diretamente o código, basta modificar a especificação de entrada e fornecê-la novamente ao gerador.

A especificação, por não conter detalhes de implementação, é mais fácil de ser escrita, lida e modificada. Sendo assim, permite-se que aplicações sejam geradas e mantidas por pessoas sem o conhecimento de linguagens de programação. Um outro ponto forte de geradores de aplicação é que eles, por acelerarem o processo de codificação, facilitam a construção de protótipos e o teste de especificações alternativas.

Por outro lado, os geradores de aplicação causam algumas dificuldades. Eles podem ser usados efetivamente apenas em poucas situações e são difíceis de ser construídos, pois requerem linguagens de especificação cuidadosamente projetadas, um conhecimento íntimo do domínio da aplicação e a habilidade de projetar unidades de software genéricas e confiáveis. Também, é difícil encaixá-los no processo de desenvolvimento de software: Quem deve considerá-los? Se não estão disponíveis, deve-se construí-los? Como afetam o cronograma? É difícil comparar os benefícios a longo prazo contra os custos a curto prazo de se construir geradores de aplicação.

Embora existam vários exemplos de geradores de aplicação em diferentes áreas, há pouca semelhança entre eles na forma como é realizada a tradução da especificação de entrada para o produto final. Isso faz com que o desenvolvimento de um gerador para uma nova área seja um novo esforço, sem a existência de meios para facilitar e agilizar esse desenvolvimento. É necessário, então, que se tenha uma forma mais organizada para a construção dos geradores de aplicação, e uma solução é tentar separá-los numa parte dependente e outra independente da área de aplicação.

Uma visão da arquitetura proposta para um gerador de aplicação em geral, pode ser vista na figura 3. O Analisador de Especificações é construído após identificar-se a área de aplicação e definir-se a Linguagem de Especificação (LE). Ele é a interface entre o meio externo e o gerador, que deve analisar a especificação de entrada e produzir as informações relevantes, que são armazenadas para posterior uso como uma Árvore Sintática (ou uma tabela, ou qualquer outra estrutura de dados).

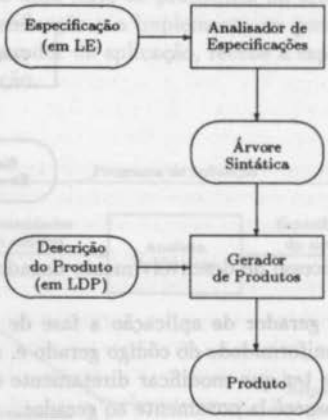

Figura 3: Arquitetura de um Gerador de Aplicação

O produto de um gerador de aplicação possui detalhes de código, na linguagem de programação alvo, que independem da aplicação e, sendo assim, não podem ser introduzidos na especificação de entrada. Como, para cada aplicação, os produtos gerados são diferentes, deve existir um meio para se fornecer esses detalhes referentes à linguagem.

Os detalhes referentes à linguagem de programação estão colocados na Descrição do Produto, que é parecida com o produto final, diferenciando-se nos pontos onde o código é dependente da especificação. Nesses pontos existem comandos de uma linguagem, chamada de Linguagem de Descrição de Produtos (LDP), que indicam como a árvore sintática deve ser manipulada para obter as informações desejadas. É importante observar que a descrição do produto é construída uma única vez para cada aplicação e, depois, é

utilizada na geração de todo produto semelhante.

3 Um Domínio de Aplicação: Sistemas Reativos

Apesar da construção de geradores de aplicação ser uma tarefa complicada, o passo mais difícil do desenvolvimento de geradores é reconhecer onde poder usá-los. Eles se adaptam bem a áreas onde as implementações das aplicações são semelhantes, tornando-se um processo repetitivo e monótono. Adicionalmente, é importante notar que a existência de métodos/técnicas formais para especificar aplicações dentro de um domínio alvo facilita o uso de geradores de aplicação, como é o caso do domínio apresentado neste trabalho. Quando isso não ocorre, há a necessidade prévia de se analisar e formalizar o domínio, o que não é tarefa simples.

Um possível domínio de aplicação são os Sistemas Reativos. Um sistema reativo é caracterizado por ser orientado a eventos, ou seja, está continuamente respondendo a estímulos do meio externo e interno. Redes de Comunicação, Sistemas de Telefonia, Sistemas Operacionais, Sistemas para Aviação e Interfaces Homem-Máquina são alguns exemplos de sistemas reativos.

Uma das técnicas propostas para a especificação de sistemas reativos são os *Statecharts*[1] [Har87]. A técnica Statechart (do inglês *State* = estado e *chart* = diagrama) é uma extensão dos diagramas de estados convencionais, incluindo noções de hierarquia, paralelismo e comunicação. Uma especificação baseada em Statecharts descreve o comportamento de um sistema em função dos estados que ele possui e dos eventos que causam as mudanças entre os estados, onde os estados são representados visualmente como bolhas conectadas por arcos, que representam as transições.

Para exemplificar a utilização dos Statecharts e introduzir informalmente os seus principais conceitos, é apresentada na figura 4 a especificação parcial do mostrador de um relógio de pulso digital. O mostrador é representado por onze estados em diferentes níveis hierárquicos, expressos pelo encapsulamento dos estados. Os estados *Normal* e *Volta* são duas formas de exibição do cronômetro e *Zero* é um estado no qual o cronômetro está desligado e o visor apresenta os dígitos com valor zero. Os eventos "a", "b" e "c" correspondem a pressionar os pinos externos do relógio.

O cronômetro ou está no seu estado inicial (*Zero*) – representado visualmente por uma seta com um círculo na origem, apontando para o estado – ou está em *Operação*, não podendo estar nos dois ao mesmo tempo. Diz-se, assim, que os estados *Zero* e *Operação* são uma decomposição OU (*XOR*) do estado *Cronômetro*, da mesma forma que *Normal* e *Volta* o são de *Display*.

Num Statechart dois subestados de um mesmo estado podem estar ativos ao mesmo tempo, caracterizando o paralelismo ou ortogonalidade. Uma decomposição E (*AND*) é aquela na qual estando num estado, o sistema deve estar em todos os seus componentes. No cronômetro *Display* e *Contador* são uma decomposição ortogonal de *Operação*, significando que, após o cronômetro ser acionado, o tempo passa a ser mostrado

[1]Também chamados de Estadogramas. No restante do texto será utilizado o termo original em inglês.

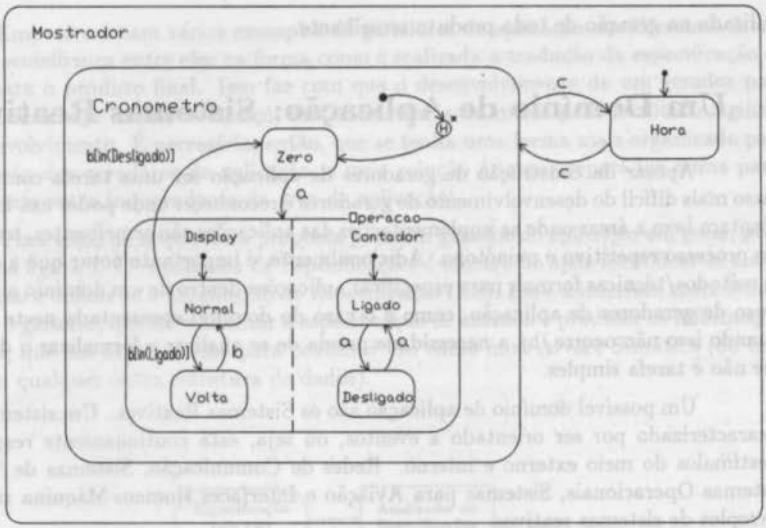

Figura 4: Statechart do Mostrador de um Relógio Digital

no visor e contado simultaneamente. O paralelismo é representado visualmente por uma bolha na qual os componentes estão separados por uma linha tracejada.

Na figura 4 notam-se, também, as transições que ocorrem entre os estados, representadas pelos arcos direcionados. Se o cronômetro estiver acionado, no estado *Ligado*, e pressionar-se o botão "a", vai-se para o estado *Desligado*, e vice-versa. Toda transição parte de um ou mais estados origens e chega a um ou mais estados destinos.

Uma outra maneira de se atingir um grupo de estados é através da "história" desse grupo, visualmente representada pela letra H inserida em um pequeno círculo. Ativar um determinado estado dentro de um grupo de estados que possui história significa considerar aquele estado que foi o mais recentemente visitado. A história se aplica ao nível no qual ela aparece ou, a partir dele, a todos os níveis mais internos do diagrama. Essa situação é representada visualmente pelo símbolo "*" associado ao símbolo de história.

O conceito de história pode estar, ainda, associado ao conceito de estado inicial (*default*), como mostra a figura 4, ativando-se o estado *default* (*Zero*), se for a primeira visita ao estado (*Cronômetro*), ou ativando-se o estado mais recentemente visitado em caso contrário.

Uma outra característica dos Statecharts é que certas condições podem estar associadas aos eventos. Por exemplo, na figura 4, a condição *in(Desligado)* obriga que a transição que leva do estado *Normal* para o estado *Zero*, só dispare se o estado *Desligado* estiver ativo no instante em que ocorrer o evento "b". Outros tipos de condições podem ser encontradas em [Har87].

Nos Statecharts a reatividade do sistema não é expressa simplesmente por mudanças nos estados internos do sistema. É possível associar ações e atividades aos eventos. Uma ação é executada instantaneamente ao se entrar e/ou sair de um determinado estado e uma atividade é caracterizada pela sua execução enquanto o sistema permanecer no estado associado.

4 Um Gerador de Aplicação para Sistemas Reativos

Nesta seção apresenta-se um gerador de aplicação para Sistemas Reativos especificados com Statecharts. O gerador recebe como entrada uma especificação em Statecharts de um sistema reativo e produz como saída um produto relacionado com o sistema especificado. Um possível produto a ser gerado (outros produtos podem ser considerados) é o programa que simula o sistema, realizando as mudanças de estados causadas pelos eventos e executando as ações e atividades associadas a esses eventos.

A seguir, é parcialmente apresentada uma possível implementação do simulador do Statechart do mostrador escrita na linguagem C, mostrando alguns aspectos da estrutura de dados e da estrutura de controle do programa. A hierarquia de estados, para maior rapidez de acesso, é implementada como uma tabela de tamanho fixo, com a última linha contendo apenas valores nulos, implementando uma estrutura de árvore com diversos ponteiros, onde cada nó representa um estado. O arranjo mostrado abaixo representa a hierarquia do mostrador, indicando que o sistema ainda não foi acionado (não existem estados ativos).

```
enum Estados{ MOSTRADOR, HORA, CRONOMETRO, ZERO, OPERACAO, DISPLAY,
              CONTADOR, NORMAL, VOLTA, LIGADO, DESLIGADO, NULO };

typedef struct Componentes{
                  enum Estados nome_est;
                  int prim_filho;
                  int pai;
                  int prox_irmao;
                  int ant_irmao;
                  int ligado;
                  int ult_lig;
                  int tipo;
                  int defo;
                  int nivel;
                }Estado;

Estado arvore_estado[] = {
  {MOSTRADOR,HORA,NULO,NULO,NULO, n_lig,n_lig,xor,n_def,0},
  {HORA,NULO,MOSTRADOR,CRONOMETRO,NULO,n_lig,n_lig,atomo,def,1},
  {CRONOMETRO,ZERO,MOSTRADOR,NULO,HORA,n_lig,n_lig,xor,n_def,1},
  {ZERO,NULO,CRONOMETRO,OPERACAO,NULO,n_lig,n_lig,atomo,def_hh,2},
```

```
  {OPERACAO,DISPLAY,CRONOMETRO,NULO,ZERO,n_lig,n_lig,and,n_def ,2},
  {DISPLAY,NORMAL,OPERACAO,CONTADOR,NULO,n_lig,n_lig,xor,n_def,3},
  {CONTADOR,LIGADO,OPERACAO,NULO,DISPLAY,n_lig,n_lig,xor,n_def,3},
  {NORMAL,NULO,DISPLAY,VOLTA,NULO,n_lig,n_lig,atomo,def,4},
  {VOLTA,NULO,DISPLAY,NULO,NORMAL,n_lig,n_lig,atomo,n_def,4},
  {LIGADO,NULO,CONTADOR,DESLIGADO,NULO,n_lig,n_lig,atomo,def,4},
  {DESLIGADO,NULO,CONTADOR,NULO,LIGADO,n_lig,n_lig,atomo,n_def,4},
  {NULO,NULO,NULO,NULO,NULO,n_lig,n_lig,atomo,n_def,0}
};
```

Dentre os campos de dados armazenados em cada linha da tabela estão os seguintes:

ligado: Indica se o estado faz parte da configuração, isto é, está ativo.
ult_lig: Armazena a história de ativações do estado.
tipo: Indica o tipo de decomposição: AND, OR ou ATOMO.
defo: Indica se o estado é *default* ou não.
nível: Indica o nível de profundidade (altura) do estado.

Na estrutura de controle do programa são realizadas as mudanças entre os estados. O programa principal indica que, primeiramente, é constituída a configuração inicial descendo-se na hierarquia pelas entradas *defaults*. A seguir é executado um comando repetitivo onde, a cada iteração, é obtido o próximo evento, e, em função do mesmo, é analisada a configuração para ver se esta é afetada. A configuração é salva em uma lista antes de cada passo da simulação. Durante a simulação, cada estado da lista é visitado.

```
main(){
    enum Eventos evento ;

    cabeca = (struct List *)malloc(sizeof(Lista));
    cabeca->proximo = NULL;
    cabeca->nome_estado = NULO;
    Ativa_Prole(MOSTRADOR);  /* Configuracao inicial,por default */
    while (evento != FIM){
        Exibe_Ativos(MOSTRADOR);
        evento = Ler_Eventos();
        Salvar_Conf_Ativos();
        Simular(evento);
     }
 }
```

No código da função `Simular`, apresentado abaixo, estão incorporadas as transições entre os estados em função dos eventos definidos na especificação. Um determinado estado ativo é selecionado e verifica-se se ele é afetado pelo evento corrente sendo analisado. Essa verificação é feita por uma estrutura de comando *switch* em dois níveis: no primeiro é selecionado o código correspondente ao estado e no segundo o código correspondente ao evento. No código também são incluídas as condições associadas aos eventos. A mudança

para a nova configuração e o controle da execução de ações e atividades são efetuadas pela função De_Para, cujo código não é apresentado neste artigo, mas pode ser encontrado em [Can91].

Para não tornar o exemplo muito complexo, o simulador não inclui toda a generalidade dos Statecharts, como a entrada de mais de uma transição simultaneamente, transições com várias origens ou destinos e outras condições. Um simulador completo pode ser encontrado em [For91].

```
Simular(nome_evento)
enum Eventos nome_evento;
{
  Lista *p_lista, *p_lista2;
  int i;

  p_lista2 = p_lista  = (struct List *)malloc(sizeof(Lista));
  p_lista = cabeca->proximo;
  while (p_lista !=  NULL){
    switch(p_lista->nome_estado){
       case HORA :
                  switch(nome_evento) {
                     case C : De_Para(HORA,CRONOMETRO);
                              break;
                  }
                  break;
       case CRONOMETRO :
                  switch(nome_evento) {
                     case C : De_Para(CRONOMETRO,HORA);
                              break;
                  }
                  break;
       case ZERO :
                  switch(nome_evento) {
                     case A : De_Para(ZERO,OPERACAO);
                              break;
                  }
                  break;
       case NORMAL :
                  switch(nome_evento) {
                     case B : if (In(LIGADO))
                                 De_Para(NORMAL,VOLTA);
                              if (In(DESLIGADO))
                                 De_Para(NORMAL,ZERO);
                              break;
                  }
                  break;
```

```
        case VOLTA :
                    switch(nome_evento) {
                        case B : De_Para(VOLTA,NORMAL);
                                 break;
                    }
                    break;
        case LIGADO :
                    switch(nome_evento) {
                        case A : De_Para(LIGADO,DESLIGADO);
                                 break;
                    }
                    break;
        case DESLIGADO :
                    switch(nome_evento) {
                        case A : De_Para(DESLIGADO,LIGADO);
                                 break;
                    }
                    break;
      }
     p_lista2 = cabeca->proximo;
     cabeca->proximo = p_lista2->proximo;
     p_lista = cabeca->proximo;
     free(p_lista2);
    }
}
```

É importante notar que o código da função **Simular** apresentado acima é particular ao exemplo específico do mostrador. Essa função, portanto, é reconfigurada para cada Statechart apresentado, pois nela estão incluídos os nomes dos estados, os nomes dos eventos e as condições referentes à especificação. Consequentemente, há a possibilidade de se ter o gerador de simuladores, gerando-se automaticamente as partes dependentes do Statechart dentro do programa simulador, como no procedimento **Simular**.

Apesar dos Statecharts serem uma linguagem de especificação gráfica, o gerador de aplicação receberá como entrada uma descrição equivalente aos diagramas, escrita numa linguagem textual. Essa linguagem, da mesma forma que os Statecharts, descreve o comportamento de um sistema em função dos estados e eventos. Existe também a possibilidade de utilizar um editor gráfico para Statecharts [Bat91] e depois produzir-se automaticamente a descrição textual equivalente, que servirá de entrada para o gerador de aplicação.

A Linguagem de Especificação de Statecharts (LES) está definida por uma gramática livre de contexto, contendo os símbolos terminais e as produções que identificam as construções sintáticas permitidas [Mei91]. Ela é composta de três seções principais, onde a primeira, opcional, permite a declaração de variáveis e seus tipos e a segunda permite a declaração de todos os estados e seus subestados, o tipo de decomposição, estados iniciais, etc. A terceira seção é utilizada para a especificação de transições, podendo-se especificar também as situações de história, condições, ações, etc.

Considerando o Statechart do mostrador (figura 4), a sua descrição textual equivalente é a seguinte:

```
ESTADO Mostrador OU
    SUBESTADOS Hora DEF, Cronometro
ESTADO Hora ATOMO
ESTADO Cronometro OU
    SUBESTADOS Zero DEF/H*, Operacao
ESTADO Zero ATOMO
ESTADO Operacao E
    SUBESTADOS Display, Contador
ESTADO Display OU
    SUBESTADOS Normal DEF, Volta
ESTADO Contador OU
    SUBESTADOS Ligado DEF, Desligado
ESTADO Normal ATOMO
ESTADO Volta ATOMO
ESTADO Ligado ATOMO
ESTADO Desligado ATOMO;
EVENTO c
    ORIGEM hora
    DESTINO cronometro
EVENTO c
    ORIGEM cronometro
    DESTINO hora
EVENTO a
    ORIGEM Zero
    DESTINO Operacao
EVENTO b[in(Ligado)]
    ORIGEM Normal
    DESTINO Volta
EVENTO b[in(Desligado)]
    ORIGEM Normal
    DESTINO Zero
EVENTO b
    ORIGEM Volta
    DESTINO Normal
EVENTO a
    ORIGEM Ligado
    DESTINO Desligado
EVENTO a
    ORIGEM Desligado
    DESTINO Ligado;
```

Tendo a especificação de entrada e o produto a ser gerado, resta definir como será feita a tradução. Seguindo a arquitetura proposta (figura 3), deve-se definir o produto a ser obtido, usando a Linguagem de Descrição de Produtos, cuja sintaxe completa está definida em [Mei91].

Como exemplo, mostra-se abaixo um trecho de programa em LDP para gerar o tipo enumerado `Estados`, visto anteriormente. Os trechos em itálico e começando com '%' correspondem aos comandos da LDP. Esses comandos manipulam os elementos sintáticos da Linguagem de Especificação armazenados na árvore sintática. Por exemplo, o comando *%impr(%simb(estado(e)))* coloca na saída o símbolo terminal que corresponde ao nome do estado "e", enquanto que o comando *%paratodo e:DESCRICAO_ESTADO %repita* percorre a árvore sintática visitando todos os nós do tipo *DESCRICAO_ESTADO*. Os trechos não enfatizados são copiados diretamente para o produto final.

```
%gabarito Tipo_enumerado
enum Estados{
  %paratodo e:DESCRICAO_ESTADO %repita
    %impr(%simb(estado(e))),
  %fimpt
    NULO };
%fimgab
```

No apêndice encontra-se a listagem da descrição em LDP, utilizada para gerar a função `Simular`. Esse exemplo é mais complexo e usa várias subrotinas (não incluídas no texto), como, por exemplo, a função *origem(e,t)*, que verifica se o estado *e* é o estado origem da transição *t*. A ordem dos estados dentro do comando `case` é determinada pela ordem das declarações de estados na especificação em LES do mostrador.

Outros comandos disponíveis na LDP são: comandos de atribuição e seleção; comandos para marcar e desmarcar estruturas sintáticas; e, comando que permite passagem de controle (*scape*) para a linguagem C, quando algum problema mais complexo não puder ser resolvido apenas com a LDP. Os tipos de dados manipulados pela linguagem são: estruturas sintáticas, inteiros e cadeias de caracteres.

5 Trabalhos Relacionados

Entre alguns exemplos do uso de geradores de aplicação em diferentes áreas, podem ser citados: **Synthesizer Generator**, que, a partir de uma especificação baseada na gramática de uma linguagem de programação, produz um editor orientado pela sintaxe para aquela linguagem [Rep84]; **IMX**, que é um gerador de programas para a extração de dados em bases de dados IMS [Bel85]; **Compilador Estelle-C**, que implementa, semi-automaticamente, protocolos de comunicação especificados na linguagem Estelle (*Extended State Transition Language*) [Vuo88]; e **TAGS**, que é um gerador de aplicação, baseado numa linguagem de especificação gráfica, para a produção de uma simulação executável de sistemas de informação e sistemas embutidos [Lew90].

O sistema STAGE, desenvolvido por Cleaveland [Cle88], é um gerador de geradores. Dessa forma, o gerador para Sistemas Reativos apresentado neste trabalho poderia ter sido desenvolvido com STAGE. As extensões citadas na seção seguinte deste trabalho vão permitir que se tenha um gerador de geradores com capacidade similar à de STAGE.

O sistema DRACO, desenvolvido por Neighbors [Nei84], também pode ser con-

siderado um gerador de geradores, com ênfase em reutilização e em refinamento de domínios para linguagens cada vez mais concretas, até a implementação. O gerador para Sistemas Reativos apresentado neste trabalho permite a geração apenas de um nível para outro, mas as extensões que estão sendo feitas poderão facilitar a tradução entre mais de dois níveis, encadeando-se vários geradores de aplicação, onde o domínio de saída de um é o domínio de entrada do seguinte.

6 Conclusões

O Gerador de Aplicação para Sistemas Reativos pode ser utilizado para tornar mais produtiva a passagem da especificação para a implementação de sistemas especificados com Statecharts. Estruturas de dados, rotinas de manipulação, etc., podem ser geradas automaticamente, em diferentes linguagens de programação, a partir de uma especificação de alto nível.

O gerador de aplicação relatado neste trabalho está em operação, tendo-se definido a Linguagem de Especificação para o domínio (Sistemas Reativos) e a Linguagem de Descrição de Produtos, bem como seus interpretadores. Um dos objetivos buscado com a construção deste GA foi o de identificar as partes que são genéricas aos geradores de aplicação, ou seja, aquelas partes que podem ser utilizadas por qualquer gerador, independentemente da área de aplicação, de forma a se poder com mais facilidade utilizar essa tecnologia para outras situações.

A Linguagem de Descrição de Produtos e o Gerador de Produtos podem ser utilizados por qualquer gerador. Na sintaxe da LDP, os objetos tratados são os elementos sintáticos da Linguagem de Especificação, que pode mudar, sem haver necessidade de mudar a LDP ou o Gerador de Produtos.

O Analisador de Especificações é a parte dependente da aplicação, isto é, para cada gerador que se deseje construir, deve ser construído um analisador de especificação próprio. Dessa forma, estamos desenvolvendo um conjunto de ferramentas que facilitem a definição e geração da Árvore Sintática para outras Linguagens de Especificação. Os utilitários LEX e YACC (gerador de analisador léxico e gerador de analisador sintático, respectivamente) do sistema UNIX, fazem parte desse conjunto de ferramentas.

Agradecimentos
À CAPES, pelo apoio financeiro dado ao autor principal do trabalho, e ao aluno João W. L. Cangussú, por ter programado o programa simulador de Statecharts em C.

Referências

[Bat91] BATISTA, J.E.S. - *Um Editor Gráfico para Statecharts.* Dissertação de Mestrado, ICMSC-USP, 1991.

[Bel85] BELANGER, D.G.; KINTALA, C.M.R. - Data-Extraction Tools. *AT&T Technical Journal*, 64(9): 2025-2035, 1985.

[Can91] CANGUSSÚ, J. W. L. - *Um Programa para Simulação de Statecharts*, Relatório Técnico, ICMSC-USP, 1991.

[Cle88] CLEAVELAND, J.C. - Building Application Generators. *IEEE Software*, 5(4): 25-33, 1988.

[Fai85] FAIRLEY, R.E. - *Software Engineering Concepts*. New York, McGraw-Hill, 1985.

[For91] FORTES, R.P.M. - *Uma Ferramenta de Apoio à Utilização de Statecharts para Especificação do Comportamento de Sistemas de Tempo Real Complexos*. Dissertação de Mestrado, ICMSC-USP, 1991.

[Har87] HAREL, D. et al. - On the Formal Semantics of Statecharts. In: Proceeding of the 2nd IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., 1987.

[Lev86] LEVY, L.S. - A Metaprogramming Method and Its Economic Justification. *IEEE Transactions on Software Engineering*, SE-12(2): 272-277, 1986.

[Lew90] LEWIS, T. - Code Generators. *IEEE Software*, 7(3): 67-70, 1990.

[Mei91] MEIRA, C.A.A. - *Geradores de Aplicação*, Dissertação de Mestrado (em preparação), ICMSC-USP, 1991.

[Nei84] NEIGHBORS, J.M. - The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5): 564-574, 1984.

[Pre87] PRESSMAN, R.S. - *Software Engineering: A Practitioner's Approach*. 2 ed. New York, McGraw-Hill, 1987.

[Rep84] REPS, T.; TEITELBAUM, T. - The Synthesizer Generator. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, 1984. *SIGPLAN Notices*, 19(5): 42-48, 1984.

[Vuo88] VUONG, S.T.; LAU, A.C.; CHAN, R.I. - Semiautomatic Implementation of Protocols Using an Estelle-C Compiler. *IEEE Transactions on Software Engineering*, 14(3): 384-393, 1988.

A Descrição em LDP da função Simular

```
%gabarito Simular
%NO ep, ep1, ev1;

Simular(nome_evento)
enum Eventos nome_evento;
{
  Lista *p_lista, *p_lista2;
  int i;

  p_lista2 = p_lista = (struct List *)malloc(sizeof(Lista));
  p_lista = cabeca->proximo;
  while (p_lista != NULL){
    switch(p_lista->nome_estado){
%paratodo e:DESCRICAO_ESTADO %repita
  %se tem_transicao(e) %entao
        case %impr(%simb(estado(e))):
                switch(nome_evento){
    %paratodo t:TRANSICAO %repita
      %se (! %estrutura_marcada(t)) %entao
        %se origem(e,t) %entao
          %ep = evento_primitivo(evento(t));
                  case %impr(%simb(ep)):
          %paratodo t1:TRANSICAO %repita
            %ev1 = evento(t1);
            %ep1 = evento_primitivo(ev1);
            %se ((%simb(ep1)==%simb(ep)) && origem(e,t1)) %entao
              %marca_estrutura(t1)
              %paratodo dest:ESTADO %em estados_destino(t1) %repita
                %se tem_condicao_in(ev1) %entao
                        if (In(%impr(estado_condicao(ev1))))
                %fimse
                        De_Para(%impr(%simb(estado(e))), %impr(%simb(dest)));
              %fimpt
            %fimse
          %fimpt
                        break;
        %fimse
      %fimse
    %fimpt
                }
                break;
  %fimse
%fimpt
    }
    p_lista2 = cabeca->proximo;
    cabeca->proximo = p_lista2->proximo;
    p_lista = cabeca->proximo;
    free(p_lista2);
  }
}
%fimgab
```