Propriedades Dinâmicas de Statecharts

lmes Aparecida Gasparotto Boaventura

Orientador: Dr. Paulo Cesar Masiero

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos - USP, como parte dos requisitos para obtenção do título de mestre em "Ciências de Computação e Matemática Computacional".

São Carlos

1992

*Aos meus pais que sempre apoiaram
meus estudos*

*Ao Maurílio pelo carinho, compreensão e
valiosa participação*

Ao Fabinho pela felicidade de sua existência.

Agradecimentos

Ao Professor Dr. Paulo Cesar Masiero pela orientação, apoio e atenção durante o desenvolvimento deste trabalho.

À Professora Renata P. de M. Fortes pelas discussões e ajuda durante o desenvolvimento do trabalho.

Aos Professores Alagacone Sri Ranga e Messias Menegethi Junior.

Ao Professor Odelar Leite Linhares pelo apoio e incentivo.

À CAPES pelo auxílio financeiro concedido.

Às secretárias de pós-graduação e do SCE pela eficiência e dedicação.

Ao Departamento de Ciências de Computação e Estatística do IBILCE - UNESP - São José do Rio Preto.

Aos professores, alunos e funcionários do Instituto de Ciências Matemáticas de São Carlos que, direta ou indiretamente colaboraram para o desenvolvimento deste trabalho.

Resumo

Os diagramas de estado, denominados Statecharts (ou estadogramas; em Português), surgiram recentemente e se constituíram rapidamente em uma alternativa valiosa para a especificação do aspecto comportamental de sistemas reativos. Outra alternativa, as Redes de Petri, existem há mais de vinte anos e têm incorporadas a si um conjunto volumoso de teoria, propriedades e ferramentas, desenvolvidos ao longo desses anos.

Inspirado em resultados teóricos e algoritmos existentes para Redes de Petri, este trabalho define uma árvore de alcançabilidade para Statecharts e com base nela foram desenvolvidos algoritmos para verificação das seguintes propriedades dinâmicas dos Statecharts: vivacidade, sequência de eventos, alcançabilidade, reiniciabilidade e uso de transições. Diversos exemplos são apresentados e é discutida a implementação desses algoritmos dentro de um ambiente onde já existiam um editor gráfico e um simulador de Statecharts.

Abstract

A new type of state diagrams, called Statecharts, were proposed recently and became an important alternative for specifying the behaviour of reactive systems. Another alternative, the Petri Nets, were proposed more than twenty years ago and a great body of theories, properties and tools associated is available to deal with dynamic properties.

Inspired on the theoretical results and algorithms existing for Petri Nets, this dissertation defines a reachability tree for Statecharts. Based on it, algorithms were developed for verification of the following dynamic properties: liveness, event sequence, reachability, reversibility and transition usage. Several examples are presented and the implementation of these algorithms is discussed in an environment which already consisted of a graphical editor and a simulator for statecharts.

Conteúdo

1	Introdução	1
1.1	Considerações Iniciais	1
1.2	Motivação do Trabalho	2
1.3	Organização da dissertação	2
2	Revisão da literatura e dos conceitos básicos	4
2.1	Considerações Iniciais	4
2.2	Statechart: Sintaxe e Semântica	5
2.2.1	A Sintaxe dos Statecharts	7
2.2.2	A Semântica dos Statecharts	12
2.3	Redes de Petri	13
2.3.1	Propriedades de Redes de Petri	15
2.3.2	Verificação das Propriedades de Redes de Petri	17
2.3.3	Ferramentas para Análise de Propriedades de Redes de Petri	19
2.4	Statecharts: situação atual em termos de verificação de propriedades	21
2.5	Comparações de Propriedades de Redes de Petri com as Propriedades de Statecharts	23
2.6	Considerações finais	27
3	Definição de uma Árvore de Alcançabilidade para Statecharts	29
3.1	Considerações Iniciais	29

3.2	Construção da Árvore de Alcançabilidade	29
3.3	Exemplos de Árvores de Alcançabilidade	33
3.4	Algoritmo	44
3.5	Considerações Finais	48
4	Propriedades Dinâmicas de Statecharts Baseadas na Árvore de Alcançabilidade	49
4.1	Considerações Iniciais	49
4.2	Alcançabilidade de uma Configuração	49
4.2.1	Alcançabilidade de uma Configuração a partir da Configuração Inicial	52
4.2.2	Alcançabilidade de uma Configuração a partir de uma Configuração qualquer	53
4.3	Reiniciabilidade	58
4.4	Impasse (Deadlock)	58
4.5	Uso de Transições	61
4.6	Seqüência Válida de Eventos	63
4.7	Considerações Finais	67
5	Ambiente STATSIM: Implementação da Árvore de Alcançabilidade e dos Algoritmos Verificadores de Propriedades Dinâmicas de Statecharts	69
5.1	Considerações Iniciais	69
5.2	O ambiente STATSIM	70
5.3	Implementação	73
5.4	Um exemplo de execução	75
5.5	Avaliação dos algoritmos desenvolvidos	88
5.6	Considerações Finais	89
6	Conclusões	91
6.1	Considerações Iniciais	91

6.2	Contribuições deste Trabalho	92
6.3	Sugestões para Futuras Pesquisas	93
	Bibliografia	94
A	Árvores de Alcançabilidade Geradas para os Exemplos da Dissertação	96
A.1	Árvore de Alcançabilidade gerada para o Statechart do Exemplo 1	96
A.2	Árvore de Alcançabilidade gerada para o Statechart do Exemplo 2	100
A.3	Árvore de Alcançabilidade gerada para o Statechart do Exemplo 3	104
A.4	Árvore de Alcançabilidade gerada para o Statechart do Exemplo do Mos- trador de um Relógio Digital	109

Capítulo 1

Introdução

1.1 Considerações Iniciais

Os Sistemas Computacionais vêm sendo utilizados como componentes passivos (monitoração) e ativos (controle) de sistemas reativos. Sistemas de controle de tráfego aéreo e urbano, aeronaves, máquinas industriais, sistemas de protocolos de comunicação, sistemas de monitoração de pacientes hospitalares e outras aplicações de Sistemas de Tempo Real são exemplos de sistemas reativos. Muitos desses sistemas referem-se a problemas altamente complexos e têm a característica de interagir com muitas máquinas e seres humanos ao mesmo tempo. Outros sistemas envolvem problemas complexos que requerem a obtenção de resultados em curtos períodos de tempo. Frequentemente, todos esses tipos de sistemas possuem requisitos severos com relação à confiabilidade, segurança e tolerância a falhas. Os problemas de segurança se tornam ainda mais importantes quando essas aplicações incluem sistemas onde as consequências de falhas são sérias e podem envolver graves consequências à vida humana.

O desenvolvimento de sistemas reativos, na maioria das vezes, requer alta tecnologia, projetistas altamente especializados, e um enorme esforço de projeto. Nesse contexto, é de extrema importância a existência de técnicas e ferramentas para modelagem, análise e avaliação desses sistemas, auxiliando os projetistas em suas tarefas, ajudando-os e até viabilizando suas atividades, no sentido de facilitar a elaboração de sistemas mais complexos. Isso é de grande interesse tanto para os projetistas quanto para os usuários, pois permite redução do tempo de desenvolvimento, e melhoria na qualidade, o que reflete positivamente na segurança e confiabilidade do produto final.

1.2 Motivação do Trabalho

Statecharts é uma técnica proposta recentemente por Harel e seu objetivo é descrever o aspecto comportamental de sistemas reativos. Essa técnica tem recebido especial atenção na literatura por dotar o formalismo de máquinas de estados finitos (e seus componentes visuais: os Diagramas de Transição de Estados e as Matrizes de Transição de Estados) de extensões que minimizam algumas de suas principais deficiências: dificuldades em se representar situações de concorrência e a tendência para o crescimento muito rápido do número de estados e transições ao se representar situações complexas, devido ao aspecto “plano” da técnica [Hare87a] e [Hare88b].

Os Statecharts possuem um grande poder de representação, permitindo que problemas complexos sejam especificados através de diagramas mais concisos. Esses diagramas criados podem se tornar difíceis de serem validados manualmente, e daí surgiu a motivação e necessidade para o desenvolvimento de ferramentas de suporte a essa técnica.

Existe atualmente no ICMSC um grupo de Engenharia de Software interessado em desenvolver ferramentas de apoio a Sistemas de Tempo Real, utilizando-se a técnica Statecharts. O ambiente STATSIM possui disponível algumas ferramentas de apoio à especificação e simulação de Statecharts: um editor gráfico, um analisador para entradas textuais e um simulador [Masi91].

Uma vez que se tem uma base de dados contendo a especificação interna de Statecharts, esse trabalho tem como objetivo o estudo de verificação de propriedades dos Statecharts e a implementação de algoritmos que realizem essa verificação a partir dessa base de dados. A solução proposta aqui, para resolver esse problema, é baseada em algumas extensões da árvore de alcançabilidade existente para Redes de Petri que tem essa mesma finalidade: a verificação das propriedades de sistemas modelados por essas redes.

Neste trabalho define-se e apresenta-se um algoritmo para se criar a árvore de alcançabilidade para Statechart e mostra-se como utilizar essa árvore para analisar as propriedades dinâmicas de Statecharts. As propriedades aqui analisadas são: alcançabilidade de configurações de estados, reiniciabilidade, impasse (deadlock), uso de transições e sequência válida de eventos.

1.3 Organização da dissertação

Após este capítulo introduzindo este trabalho, o Capítulo 2 apresenta uma introdução à teoria de Statecharts, seguida de um resumo sobre sua sintaxe e semântica. É apresentada também nesse capítulo uma introdução à teoria de Redes de Petri, mostrando

suas propriedades, como podem ser analisadas, e algumas ferramentas com essa finalidade.

Baseando-se nas teorias estudadas sobre Statecharts e Redes de Petri, é feita uma comparação entre suas propriedades de maneira a mostrar que Statecharts possui um comportamento semelhante ao comportamento de Redes de Petri, com relação as suas propriedades dinâmicas.

No Capítulo 3 define-se uma árvore de alcançabilidade para Statecharts de forma semelhante à árvore de alcançabilidade de Redes de Petri. São apresentados ainda os detalhes de como construir essa árvore de alcançabilidade através de quatro exemplos, os quais abrangem os diversos casos que devem ser considerados para obter-se tal árvore. No final desse capítulo é mostrado o algoritmo para a construção da árvore de alcançabilidade.

O Capítulo 4 mostra como utilizar a árvore de alcançabilidade para analisar as propriedades dinâmicas de Statecharts. Para cada propriedade dinâmica definida é discutido, através de exemplos, como analisá-las, e também é dado um algoritmo com essa finalidade.

O Capítulo 5 é dedicado à discussão dos principais aspectos de implementação dos algoritmos e a mostrar, através de um exemplo real, a execução desses algoritmos. No final desse capítulo é feita uma avaliação dos algoritmos desenvolvidos.

No Capítulo 6 encontram-se as conclusões finais sobre o estudo realizado, as contribuições deste trabalho, bem como sugestões para novas pesquisas.

Capítulo 2

Revisão da literatura e dos conceitos básicos

2.1 Considerações Iniciais

Neste capítulo faz-se uma introdução à teoria de Statecharts, e também à teoria de Redes de Petri. Ambas são técnicas que podem ser utilizadas para representar o aspecto comportamental de sistemas reativos. Estas duas técnicas oferecem um certo rigor formal ao desenvolvimento de sistemas. Além disso, elas possuem uma notação gráfica que facilita o entendimento do sistema modelado.

Essa correlação de Statechart com Redes de Petri é importante para mostrar que Statechart possui um comportamento semelhante ao de Redes de Petri com respeito às suas propriedades dinâmicas. Em Redes de Petri existem técnicas bem definidas para garantir a validação de suas propriedades e neste trabalho essas idéias são transportadas para Statecharts.

Existe uma grande quantidade de ferramentas a respeito de Redes de Petri que têm como objetivo a verificação de suas propriedades dinâmicas. Algumas dessas ferramentas são discutidas neste capítulo.

Este capítulo discute também alguns trabalhos relacionados com o tema central desta dissertação: a verificação de propriedades de Statecharts. No final é feita uma comparação entre as propriedades de Redes de Petri e as propriedades de Statecharts, mostrando-se através de uma tabela as suas semelhanças.

2.2 Statechart: Sintaxe e Semântica

Os statecharts (ou Estadogramas) foram introduzidos por Harel, em [Hare87a], [Hare88b], como um formalismo visual para especificar o comportamento de sistemas reativos. Statechart é uma extensão das Máquinas de Estados Finitos (MEF), incluindo nestas os conceitos de hierarquia, paralelismo e comunicação, apropriando-as para o uso em aplicações grandes e complexas.

Esquemáticamente tem-se:

STATECHART = Diagrama de Estados + hierarquia + concorrência + mecanismo de comunicação [Hare87b].

Em um diagrama Statecharts, os estados são representados por retângulos com bordas arredondadas, também chamados bolhas; e as transições por setas direcionadas, ligando os estados.

A hierarquia é representada por aninhamento de bolhas, como ilustrado na Figura 2.1. Nessa figura os símbolos e , f , g e h significam eventos que disparam as transições e o símbolo c entre parenteses representa uma condição. Assim $g(c)$ dispara a transição de A para C quando g ocorrer e a condição c for verdadeira nesse momento. A pequena seta iniciando num ponto significa o estado inicial ou default; isso significa que ao entrar no estado D, o estado A será ativado por default.

Figura 2.1

O fato de A e C serem disjuntos e estarem completamente dentro de D significa que D é um estado com decomposição do tipo OU - exclusivo (XOR). Assim

estar em D é equivalente a estar em A ou em C, mas não em ambos; e diz-se que A e C são subestados de D.

O paralelismo é representado pelos Statecharts através de linhas tracejadas delimitando os estados concorrentes, conforme mostra a Figura 2.2. O estado M consiste de três componentes ortogonais: N, O e P relacionados por um AND: a entrada no superestado M indica que o sistema estará num subestado de N, num subestado de O e num subestado de P, simultaneamente.

Figura 2.2

Em um estado paralelo, todas as transições que estão associadas a um determinado evento são disparadas simultaneamente quando o evento ocorrer. Por exemplo, na Figura 2.2 estando em (B,D,H), se o evento *e* ocorrer, o resultado é (C,F,G).

Uma característica interessante dos Statecharts é a capacidade de lembrar uma visita feita anteriormente a um estado. A Figura 2.3 mostra uma entrada por história em A. Tal entrada ignora a entrada default e entra naquele estado de A, o qual o sistema visitou da última vez que esteve em A.

Existe uma diferença entre os símbolos *H* e *H**. Ao representar-se uma

entrada por história com o símbolo H em um estado, tem-se a história apenas no nível desse estado, enquanto que H^* representa história de forma recursiva, para todos os subestados do estado com história.

Figura 2.3

A sintaxe e a semântica operacional que regem os conceitos de Statecharts foram definidas de maneira formal e rigorosa por Harel em [Hare87b]. Dessa forma, os Statecharts, além de facilitarem a manipulação e visualização de aplicações complexas, possuem uma definição formal, de maneira a possibilitar uma simulação detalhada do sistema e sua validação através de verificação de propriedades.

A seguir retratam-se os aspectos mais importantes da sintaxe e semântica formais de Statecharts. Mais detalhes podem ser encontrados em [Hare87b], de onde foram extraídas as definições abaixo.

2.2.1 A Sintaxe dos Statecharts

A sintaxe de Statecharts é definida sobre conjuntos básicos de elementos que são: estados, transições, eventos primitivos, condições primitivas e variáveis. Com base nesses conjuntos definem-se os conjuntos estendidos de: eventos, condições, expressões e rótulos, e as inter-relações entre eles.

Estados

O conjunto de estados, S , é definido juntamente com uma função hierárquica ρ , uma função tipo ψ , um conjunto de símbolos história H e uma função default δ .

A função hierárquica $\rho : S \rightarrow 2^S$, define para cada estado os seus sub-estados. Se $\rho(x) = \rho(y) \Rightarrow x = y$. Existe um único estado $r \in S$ tal que para $\forall s \in S$, $r \notin \rho(s)$; r é a raiz do Statechart. ρ^*, ρ^+ são extensões de ρ definidas por:

$\rho^*(s) = \cup \rho^i(s), i \geq 0$, $\rho^+(s) = \cup \rho^i(s), i \geq 1$, onde $\rho^i(s)$ representa o conjunto dos descendentes de s no nível i .

A função tipo $\psi : S \rightarrow \{AND, OR\}$ define para cada estado seu tipo. Se $\rho(s) \neq \emptyset$ e $\psi(s) = OR$ então $\rho(s)$ é uma decomposição XOR de s . Isto significa que quando o sistema estiver no estado s , ele deverá estar em apenas um dos sub-estados de s . Se $\rho(s) \neq \emptyset$ e $\psi(s) = AND$ então $\rho(s)$ é uma decomposição AND de s . Isto significa que quando o sistema estiver no estado s , ele deverá estar simultaneamente em todos os subestados de s .

O conjunto de símbolos história, H , está relacionado ao conjunto de estados por uma função $\gamma : H \rightarrow S$ tal que : $\gamma(h_1) = \gamma(h_2) \Rightarrow h_1 = h_2$ e $\gamma(H)$ é um subconjunto do conjunto de estados OR. Isto é, somente um estado OR pode ter um símbolo história associado a ele. Define-se $\omega : S \cup H \rightarrow S$ por $\omega(x) = x$ se $x \in S$ e $\omega(x) = \gamma(x)$ se $x \in H$.

A função default, $\delta : S \rightarrow 2^{S \cup H}$, define para um estado s um conjunto de estados e símbolos história que estão contidos no estado.

Se $x \in \delta(s) \Rightarrow$ para $x \in S, x \in \rho^+(s)$, e para $s \in H, \delta(x) \in \rho^*(s)$.

$\delta(s)$ é o conjunto default para S .

Expressões

O conjunto de variáveis é denotado por V_p . O conjunto de expressões, V , é definido por indução como segue:

1. Se k é um número então $k \in V$.
2. Se $v \in V_p$ então $v \in V$.
3. Se $v \in V$ então $current(v) \in V$.
4. Se $v_1, v_2 \in V$ e op é uma operação algébrica então $op(v_1, v_2) \in V$.

$current(v)$ é abreviado por $cr(v)$

Condições

O conjunto de condições primitivas é denotada por C_p . O conjunto de condições C é definido indutivamente como segue:

1. $T, F \in C$, onde T é true e F é false
2. Se $c \in C_p$ então $c \in C$.
3. Se $s \in S$ então $\text{in}(s) \in C$.
4. Se $e \in E$ então $\text{not yet}(e) \in C$.
5. Se $u, v \in V$, $R \in \{=, >, <, \neq, \leq, \geq\}$ então $u R v \in C$.
6. Se $c \in C$ então $\text{current}(c) \in C$.
7. Se $c_1, c_2 \in C$ então $c_1 \vee c_2, c_1 \wedge c_2, \neg c_1 \in C$.

$\text{current}(c)$ é abreviado $\text{cr}(c)$, $\text{not yet}(c)$ é abreviado $\text{ny}(c)$.

Eventos

O conjunto de eventos primitivos é denotado por E_p . O conjunto de eventos, E , é definido como segue:

1. $\lambda \in E$, λ é o evento nulo.
2. Se $e \in E_p$, então $e \in E$.
3. Se $c \in C$, então $\text{true}(c), \text{false}(c) \in E$.
4. Se $v \in V$, então $\text{changed}(v) \in E$.
5. Se $s \in S$, então $\text{exit}(s), \text{entered}(s) \in E$.
6. Se $e_1, e_2 \in E$ então $e_1 \vee e_2, e_1 \wedge e_2 \in E$.
7. Se $e \in E, c \in C$ então $e[c] \in E$.

Ações

O conjunto de ações A é definido como segue:

1. $\mu \in A$, μ é a ação nula.
2. Se $c \in C_p, d \in C$ então $c := d \in A$.
3. Se $v \in V_p, u \in V$ então $v := u \in A$.
4. Se $a_i \in A, i = 0, \dots, n$ então $a_0; \dots; a_n \in A$.

$a \in A$ é atômica se é da forma 1, 2 ou 3.

Rótulos

O conjunto de rótulos, L , é o conjunto de pares $E \times A$; e para $l = (e, a)$ escreve-se e/a . Informalmente, se e/a é um rótulo de uma transição t , então t é disparada por e e a é executada instantaneamente quando t é tomada.

Transições

O conjunto de transições T é definido como o conjunto de triplas $T \subset 2^S \times L \times 2^{S \cup H}$. Uma transição $t = (X, l, Y)$ é composta de: um conjunto origem X , um conjunto destino Y , denotados $origem(t)$, $destino(t)$, respectivamente, e um rótulo l . Informalmente, se $l = e/a$, o sistema está em X e e ocorre, então t está habilitada e pode ser tomada. Se t é tomada, a é executado e o sistema transiciona então para Y .

São mostradas a seguir, através de diagramas, algumas definições de termos técnicos usados nas próximas seções.

Um estado s é básico se $\rho(s) = \emptyset$. No diagrama da Figura 2.4, os estados $\{B_1, C, D, E\}$ são básicos.

Figura 2.4

Para um conjunto de estados X , definem-se o $LCA(X)$ e $LCA^+(X)$ como sendo o *Menor Ancestral Comum de X* e o *Menor Ancestral Comum estrito de X* , respectivamente. No diagrama da Figura 2.4, tomando-se $X = \{B, C, D\}$, o $LCA(X) = B$ e o $LCA^+(X) = A$.

Figura 2.5

Para uma transição $t \in T$, o $LCA(t)$ é definido como sendo o LCA^+ do conjunto de estados formado pelos seus estados origem e destino. Na Figura 2.4, o LCA da transição a é M , pois $X = \{B\}$ e $Y = \{E\}$ e $LCA^+(X \cup Y) = M$.

Dois estados x e y são ortogonais, denotado por $x \perp y$, se $x = y$ ou o $LCA(x, y)$ é um estado AND. Por exemplo, na Figura 2.5 os estados G e E são ortogonais, pois $LCA(G, E) = M$, que é um estado AND.

Um conjunto de estados X é um *conjunto ortogonal* se $\forall x, y \in X, x \perp y$. Na Figura 2.5 o conjunto $X = \{G, E\}$ é um *conjunto ortogonal*, e diz-se que X é um *conjunto ortogonal relativo ao estado M* .

Um conjunto de estados X é um *conjunto ortogonal maximal relativo a s* se X for um conjunto ortogonal relativo a s e para qualquer $y \in \rho^*(s), y \notin X \Rightarrow X \cup \{y\}$ não

é ortogonal. Na Figura 2.5 o conjunto $X = \{G, E, A\}$ é um *conjunto ortogonal maximal relativo a M*, pois $\rho^*(M) = \{M, N, A, B, C, O, E, F, P, G, H\}$ e para todo $y \in \rho^*(M)$ tal que $y \notin X \Rightarrow \{y\} \cup \{G, E, A\}$ não é ortogonal.

Uma *configuração de estado de s* é um conjunto ortogonal relativo a s onde todos os seus elementos são estados básicos.

Uma *configuração de estado maximal de s* é um conjunto ortogonal maximal relativo a s , onde todos os seus elementos são básicos. Na Figura 2.5, o conjunto $X = \{B, E, G\}$ é uma *configuração de estado maximal de M*.

Uma transição $t = (Y, l, Z)$ é *estruturalmente relevante* a uma configuração de estado X se para todo $y \in Y$ existe $x \in X$ tal que $x \in \rho^*(y)$. Se t é estruturalmente relevante em X e se o sistema está em X então t pode disparar. No diagrama da Figura 2.4 a transição f é estruturalmente relevante à configuração de estado $X = \{C\}$.

2.2.2 A Semântica dos Statecharts

A semântica operacional dos Statecharts é baseada em uma sequência de instantes de tempo $\{\sigma_i\}, i \geq 0$, que corresponde a porções do SSD (Sistema Sob Desenvolvimento). Os intervalos de tempo básicos são definidos por $L_i = [\sigma_i, \sigma_{i+1})$. Em σ_{i+1} o SSD reage ao estímulo externo ocorrido no intervalo L_i . A semântica de Statecharts define formalmente mudanças ocorridas no SSD como uma reação a estímulos externos. Um estímulo externo associado a σ_{i+1} é uma tripla (Π, Θ, ξ) onde:

- Π é o conjunto de eventos primitivos externos ocorridos no intervalo L_i .
- Θ é o conjunto de condições primitivas externas cujo valor é *true* em $[\sigma, \sigma_{i+1})$, para algum $\sigma \geq \sigma_i$.
- ξ é uma função determinada pelo ambiente externo, tal que para uma variável v , $\xi(v) = x$ se o valor de v é x em $[\sigma, \sigma_{i+1})$, para algum $\sigma \geq \sigma_i$.

Uma configuração do sistema associado a um instante σ_{i+1} é uma tupla (X, Π, Θ, ξ) onde X é uma configuração de estado maximal em relação ao estado raiz e (Π, Θ, ξ) é um estímulo externo associado a σ_{i+1} .

A reação do sistema em algum instante é composta pelo conjunto de transições tomadas naquele instante e pelo conjunto de eventos gerados quando as transições são tomadas. Assim, uma reação do sistema é um par (Υ, Π') onde Υ é o conjunto de transições, chamada de *um passo*, e Π' é o conjunto de eventos atômicos gerado por Υ .

Dada uma configuração do sistema $CS = (X, \Pi, \Theta, \xi)$ define-se o conjunto $\{RS = (\Upsilon, \Pi)\}$ de possíveis reações do sistema para a CS dada.

Intuitivamente, um passo é um conjunto de transições consistentes que são estruturalmente relevante a uma dada configuração do sistema e habilitadas em relação ao estímulo externo dado. O conjunto de eventos gerados é o conjunto de eventos ocorridos como resultado de tomar as transições e executar as ações associadas com essas transições. Todas as transições participantes em um passo Υ são tomadas simultaneamente. Na definição formal de um passo o conjunto de transições tomadas é sequenciado em *micro-passos*, onde cada um deles é um subconjunto de Υ . Assim um passo é definido como uma sequência de micro-passos. Embora os micro-passos sejam essenciais para a definição precisa e o bom entendimento da semântica dos Statecharts, são considerados como um mecanismo interno para o cálculo dos passos e não são visíveis aos usuários. Os usuários sofisticados podem utilizar esse mecanismo interno de micro-passos para definir sistemas mais complicados e não usuais. Os detalhes da semântica formal de Statecharts podem ser encontrados em [Hare87b].

2.3 Redes de Petri

Uma Rede de Petri (RP) é composta de um conjunto de lugares (Places) P , um conjunto de transições T , uma função de entrada (input), I , uma função de saída (output), O , e uma marcação inicial M_0 . A função de entrada I associa a cada transição os seus lugares de entrada; a função de saída O associa a cada transição os seus lugares de saída [Pete81], [Brag82].

Uma Rede de Petri é ilustrada por uma estrutura de grafo dirigido que possui dois tipos de nós: lugares (representados por círculos) e transições (representadas por arcos). Os arcos do grafo podem ser direcionados das transições para os lugares e vice-versa, representando as funções de entrada e de saída. A Figura 2.6 mostra uma Rede de Petri e o seu grafo correspondente.

Os lugares de entrada para uma transição t são os lugares dos quais saem os arcos que incidem em t . Por exemplo, na Figura 2.6, os lugares de entrada para t_1 são P_1 e P_4 ; para t_4 são P_5 e P_6 . Os lugares de saída de uma transição t são os lugares onde incidem os arcos vindos de t . Na Figura 2.6, os lugares de saída de t_3 são P_1 , P_4 e P_5 .

Os aspectos dinâmicos de uma Rede de Petri são denotados por marcações que são associações de “marcas” (ou fichas) aos seus lugares. Essas marcações se alteram durante a execução da Rede de Petri. A execução é controlada pelo número e distribuição de fichas segundo as seguintes regras:

$C = \{P, T, I, O\}$

$P = \{P1, P2, P3, P4, P5, P6, P7, P8\}$

$T = \{t1, t2, t3, t4, t5, t6\}$

$I(t1) = \{P1, P4\}$ $O(t1) = \{P2\}$

$I(t2) = \{P2, P5\}$ $O(t2) = \{P3\}$

$I(t3) = \{P3\}$ $O(t3) = \{P1, P4, P5\}$

$I(t4) = \{P5, P6\}$ $O(t4) = \{P7\}$

$I(t5) = \{P4, P7\}$ $O(t5) = \{P8\}$

$I(t6) = \{P8\}$ $O(t6) = \{P4, P5, P6\}$

Figura 2.6: Rede de Petri

- Uma transição t está habilitada se e somente se em cada um de seus lugares de entrada existe, pelo menos, uma ficha para cada arco que se origina nesses lugares.
- Qualquer transição habilitada pode disparar.
- Ao disparar uma transição t , é removido, de cada lugar de entrada L' de t , tantas fichas quantos forem os arcos que originam de L' , e são colocadas, em cada lugar de saída L de t , tantas fichas quantos forem os arcos incidentes em L .
- O procedimento de disparar transições pode continuar enquanto houver transições habilitadas, caso contrário a execução termina.

Cada mudança do número e posição das fichas, ocorrida durante a execução de uma Rede de Petri, define uma marcação (M). A marcação inicial (M_0) é a marcação do início da execução da Rede. A marcação é representada por uma n -upla $(L_1, L_2, \dots, L_n)$, onde n é o número de lugares da rede de Petri em questão.

Assim, na Figura 2.6, $n = 8$ e $M_0 = (1,0,0,1,1,1,0,0)$. As transições habilitadas são t_1 e t_4 . Se t_1 disparar resultará em $M_1 = (0,1,0,0,1,1,0,0)$, conforme mostra essa figura.

2.3.1 Propriedades de Redes de Petri

As principais propriedades que podem ser analisadas nas Redes de Petri são limitabilidade, segurança, conservação, vivacidade, alcançabilidade, sequência de disparos e reiniciabilidade [Pete81], [Mura84].

a. Limitabilidade e segurança

Uma Rede de Petri é K -limitada se para algum K natural fixo cada um de seus lugares é K -limitado, ou seja, não há mais que K fichas em qualquer lugar da rede. Isto significa que o espaço de marcações alcançáveis é finito e o número de marcações é $(K + 1)^n$, onde n é o número de lugares.

Segurança é um caso especial de limitabilidade: um lugar é seguro se é limitado por uma ficha, ou seja, é 1-limitado. Uma Rede de Petri é segura se cada um dos seus lugares é seguro, ou seja, não existe nenhum lugar com mais de uma ficha.

Se uma Rede de Petri não é limitada (possui infinitas marcações), então o sistema físico correspondente é impossível de ser implementado. A limitabilidade da rede representa os limites de recursos de "Hardware" e "software" do sistema.

b. Conservação

Redes de Petri podem ser usadas para modelar sistemas de alocação de recursos. Como os recursos não são criados nem destruídos, para esses tipos de Redes de Petri, as fichas não devem ser criadas nem destruídas. Uma Rede de Petri é conservativa se a soma de fichas em cada uma das marcações é sempre igual, isto é, o número de entradas em cada transição disparada é igual ao número de saídas. Em geral pode-se atribuir “pesos” a cada um dos pontos da rede permitindo que o número de pontos mude, mas a soma ponderada dos pontos fique constante.

c. Vivacidade

Outro problema que pode aparecer na alocação de recursos é a paralisação total (“deadlock”). Um deadlock em uma Rede de Petri é uma transição (ou conjunto de transições) que não pode disparar. A Rede de Petri da Figura 2.6 ilustra essa situação ao considerar o disparo da sequência t_1t_4 , pois nessa condição tem-se $M_3 = (0,1,0,0,0,1,0)$ e nenhuma transição está habilitada nessa marcação.

Uma transição é **viva** se ela não está paralisada. Isto não significa que a transição está habilitada, mas simplesmente que ela pode se tornar habilitada. Uma transição t_j de uma Rede de Petri é potencialmente disparável em uma marcação M se existe uma marcação M' , resultante do disparo de uma transição em M , na qual a transição t_j esteja habilitada. Uma transição está viva numa marcação M se ela for potencialmente disparável em toda marcação resultante de M . Uma transição está morta se nunca puder ser disparada.

Uma Rede de Petri é viva para uma marcação inicial M_0 , se todas as suas transições são potencialmente disparáveis e todas as marcações são alcançáveis a partir da marcação inicial M_0 .

d. Alcançabilidade

O problema de alcançabilidade é o seguinte: dada uma Rede de Petri marcada com uma marcação M e dada uma outra marcação M' , essa marcação M' é alcançável a partir de M ? Muitos outros problemas podem frequentemente ser reduzidos a este problema básico. Por exemplo, para a Rede de Petri da Figura 2.6 ocorre deadlock se o estado $(0,1,0,0,0,1,0)$ for alcançado.

e. Seqüência de disparos

Outro aspecto a ser analisado diz respeito à seqüência de transições disparadas, ao invés de estados. Isto está relacionado com “vida”, pois pode-se verificar se uma transição t pode ser disparada. Mais geralmente, pode-se verificar a possibilidade de disparo de uma seqüência específica de transições ou de algum subconjunto dessa seqüência.

f. Reiniciabilidade

Uma Rede de Petri é reiniciável para uma certa marcação inicial M_0 se, para qualquer marcação M decorrente e pertencente ao conjunto de marcações alcançáveis de M_0 , existe uma sequência de disparos que faça a rede voltar à marcação inicial M_0 .

Um sistema físico, representado por uma Rede de Petri reiniciável, retorna ao seu estado inicial após a execução de uma ou mais tarefas. Esta condição é, geralmente, necessária para o bom funcionamento de vários sistemas.

2.3.2 Verificação das Propriedades de Redes de Petri

Para análise de Redes de Petri duas técnicas vem sendo utilizadas: análise gráfica, através da árvore de alcançabilidade e máquina de fichas, e análise algébrica, envolvendo equações de matrizes [Mura84]. Apresenta-se a seguir a técnica gráfica através da árvore de alcançabilidade, onde foi inspirado este trabalho.

Uma árvore de alcançabilidade é uma estrutura cujos nós representam as marcações (ou estados) alcançáveis a partir de uma marcação inicial M_0 da Rede de Petri.

Cada nó na árvore de alcançabilidade é rotulado com uma marcação, e os arcos são rotulados com transições. A raiz da árvore de alcançabilidade é rotulada com a marcação inicial M_0 .

Freqüentemente, o conjunto de marcações alcançáveis de uma Rede de Petri é infinito. Para obter uma representação finita de um conjunto infinito, deve-se mapear várias marcações em um mesmo nó da árvore. Este mapeamento “muitos para um” é conseguido reunindo-se um conjunto de estados em um nó da árvore e ignorando-se o número de pontos nos lugares da rede quando este número for muito grande. Isto é representado usando-se um símbolo especial, w , para o número de pontos nesse lugar. O símbolo w representa um valor que pode ser arbitrariamente grande (w poderia ser pensado como um símbolo para infinito).

Dado um nó x na árvore, os nós adicionais são inseridos na árvore para todas as marcações que são imediatamente alcançáveis da marcação do nó x . Para cada transição t , que é habilitada pela marcação M do nó x , um novo nó com marcação M' é criado, e um arco rotulado t é dirigido do nó x , com marcação M , para o novo nó, com marcação M' , imediatamente alcançável de M . Este processo é repetido para todos os novos nós. Mais detalhes sobre a construção desta estrutura podem ser encontrados em [Pete81] e [Brag82].

A Figura 2.7 é um exemplo da árvore de alcançabilidade para a Rede de Petri da Figura 2.6.

Figura 2.7: Árvore de Alcançabilidade para a Rede de Petri da Figura 2.6

Análise da Rede de Petri usando a Árvore de Alcançabilidade

As propriedades de *segurança* e *limitabilidade* podem ser verificadas usando-se a árvore de alcançabilidade. Uma Rede de Petri é limitada se e somente se o símbolo w nunca aparece em sua árvore de alcançabilidade. A posição do símbolo w indica quais lugares são ilimitados. Se a Rede de Petri é limitada então ela representa um sistema de estados finitos e sua árvore de alcançabilidade é um grafo de estados. Um exame do conjunto finito de todas as marcações alcançáveis tornará possível a resposta a muitas questões. Por exemplo, para determinar o limite de um lugar particular, gera-se a árvore de alcançabilidade e procura-se na árvore o maior valor de determinado componente na

marcação, o qual corresponde ao lugar específico. Este é o limite do número de pontos do lugar específico. E se o limite de todos os lugares for 1, então a rede é segura.

Para verificar a propriedade de *conservação* basta considerar a soma ponderada de fichas para cada marcação existente na árvore de alcançabilidade. Se as somas são as mesmas para cada marcação alcançável, a rede é conservativa com relação a um dado vetor peso. Se as somas não forem iguais, a rede não é conservativa.

A verificação da propriedade de *reiniciabilidade* é facilmente realizada através da árvore de alcançabilidade. Basta olhar na árvore e verificar se para toda marcação M , decorrente de M_0 , existe um caminho que faça a rede retornar a M_0 .

A *vivacidade* será constatada se a árvore de alcançabilidade não possuir nós terminais. Os nós representados na árvore são todos reiniciáveis, ou seja, levam ao nó raiz.

Alcançabilidade e sequências de disparos são as propriedades mais simples de se verificar com a árvore de alcançabilidade. Para saber se uma marcação M é alcançável basta verificar se ela está representada como um nó da árvore. E para verificar uma sequência de disparos basta acompanhar os arcos da árvore.

Uma observação importante a ser feita com relação à árvore de alcançabilidade para Redes de Petri é que a análise das propriedades de vivacidade e alcançabilidade é limitada pela existência do símbolo w . A utilização do símbolo w é uma perda de informação, onde os números individuais são descartados e substituídos por w , e existem casos em que para Redes de Petri diferentes, a mesma árvore de alcançabilidade é gerada [Pete81].

2.3.3 Ferramentas para Análise de Propriedades de Redes de Petri

Um grande número de ferramentas tem sido desenvolvido utilizando o formalismo de Redes de Petri. São apresentadas, a seguir, várias ferramentas que incorporam o formalismo de Redes de Petri. Algumas dessas ferramentas estão voltadas para aplicações em microcomputadores e "workstations", enquanto que outras estão voltadas para aplicações em computadores de grande porte [Anac91].

. FARPE

A ferramenta FARPE utiliza Redes de Petri estocásticas e generalizadas (SPN e GSPN) para projeto e análise de sistemas de processamento distribuído e paralelo. É composta por dois módulos principais: editor e solucionador. O módulo editor permite a representação interna das redes a partir da descrição feita pelo usuário. O módulo

solucionador utiliza uma representação interna para verificar a consistência da rede, gerar o conjunto de alcance, e calcular a distribuição das probabilidades estacionárias para as marcações alcançáveis.

. GRASPIN

Projeto gráfico interativo de diagramas, grafos e redes, onde as classes de redes utilizadas são: Predicado/Transição e Predicado/Evento. As especificações são manipuladas por um "Laboratório" de Reescrita de Regras, que verifica propriedades de consistência, completeza e terminação. As operações são compiladas em LISP, há um simulador para sistemas Predicado/Transição - Predicado/Evento - Lugar/Transição - Evento/Numérico. É feita análise através de sistemas de grafo e Predicado/Evento. Há um tradutor de sistemas Lugar/Transição em especificações. A análise de alcançabilidade e vivacidade é feita utilizando técnicas de reescrita.

. GreatSPN

GreatSPN é uma ferramenta altamente interativa para construção, modificação, validação e solução de Redes de Petri estocásticas e temporizadas. Apresenta um editor gráfico para "workstation" SUN, com manipulação direta de objetos na tela. O modelo de validação é feito através de jogo de fichas interativo, e de cálculo de propriedades estruturais e exibição interativa (Lugar/Transição invariantes, conflitos de limitabilidade, exclusão mútua, confusão, etc.).

. NET-ANALYSER

O pacote consiste de dois programas que podem rodar independentemente, mas que se completam: "graphieditor" e "matrixeditor". A ferramenta analisa alcançabilidade, S e T invariantes, conflitos, bloqueio perpétuo, vivacidade, e o evento selecionado para parar a simulação.

. SISCOPE

SISCOPE - Sistemas CONcorrentes modelados com redes de PEtri é uma ferramenta, baseada em Redes de Petri voltada para a especificação de sistemas concorrentes. A ferramenta oferece facilidades para a verificação de propriedades dinâmicas, tais como: ausência de impasses, realização de progresso efetivo, minimidade, vivacidade, reiniciabilidade, limitabilidade e recuperabilidade.

2.4 Statecharts: situação atual em termos de verificação de propriedades

Harel, em [Hare88a], apresenta o sistema **Statemate**. Statemate é um ambiente gráfico automatizado, constituído por pacotes de consultas, testes e simulação para sistemas reativos, que permite a geração automática de código e prototipagem de sistemas. Ele realiza essas tarefas especificando e avaliando o Sistema Sob Desenvolvimento (SSD) a partir de três visões: estrutural, funcional e comportamental.

Para possibilitar a especificação e análise do SSD sob essas três visões, Statemate dispõe de três linguagens gráficas, que são os Diagramas de Módulos ("Module-charts"), Diagramas de Atividades ("Activity-charts") e Statecharts. A Figura 2.8 representa a estrutura geral do ambiente. O SSD é especificado e analisado, e suas informações ficam armazenadas em uma base de dados central, a partir da qual os outros módulos geram suas saídas.

O módulo de maior interesse, que levou à descrição do Statemate nesta seção, é o módulo de testes. Este módulo permite ao usuário depurar a especificação com relação a propriedades dinâmicas do sistema, tais como alcançabilidade, deadlocks e não determinismo.

Segundo Harel, o Statemate gera conjuntos de execuções de forma exaustiva ("força bruta") e, sobre eles, verifica o SSD com relação à alcançabilidade, não determinismo, deadlock e uso de transições. Para a verificação de alcançabilidade, por exemplo, o usuário entra com as condições finais e o Statemate procura seqüências de eventos externos e outras ocorrências que levam de um estado inicial a um que satisfaça aquelas condições impostas pelo usuário. Esses testes são feitos dinamicamente em tempo de execução.

Esse trabalho, embora tenha apresentado o módulo de testes como parte do Statemate, não entrou em detalhes dos aspectos teóricos e algoritmos envolvidos em sua implementação.

Melhart, em [Melh89] faz um estudo sobre a capacidade de análise para requisitos especificados em Statecharts. São apresentadas algumas idéias em desenvolver métodos de análise de completeza para sistemas de tempo real usando especificações em Statecharts. Essas idéias são discutidas nesse trabalho, mas na época não havia sido desenvolvida ainda a pesquisa e nada havia sido implementado. Dois tipos específicos de análise são discutidos:

Figura 2.8: Estrutura Geral do Ambiente STATEMATE

Análise de Segurança

Leveson e Stolzy, em [Leve87], desenvolveram técnicas de análise para Redes de Petri Temporizadas. Um modelo é analisado com relação a propriedades de segurança e tolerância a falhas. Essas técnicas necessitam basicamente do formalismo de Máquinas de Estados Finitos, a capacidade de derivar, no mínimo, um grafo de alcançabilidade parcial, e a especificação do intervalo máximo e mínimo para as transições.

Segundo Melhart, essas características fazem parte do formalismo de Statecharts, e os procedimentos e algoritmos descritos em [Leve87] podem ser transferidos para especificações em Statecharts, com algumas adições para este poder modelar falhas.

Leveson e Stolzy consideram cinco elementos básicos para que um modelo venha a falhar:

- um evento que não ocorre;
- um evento indesejado;
- uma sequência incorreta de eventos;
- dois eventos incompatíveis que ocorrem simultaneamente; e
- falhas no intervalo de tempo de sequências de eventos.

Algumas características adicionais foram sugeridas para especificar e analisar os requisitos de tempo em Statecharts, tais como: dar prioridades a determinadas transições como medida de segurança; expressar casos de eventos incompatíveis; e permitir falhas em transições.

Análise de completeza

Alguns critérios foram desenvolvidos para definir formalmente completeza em uma especificação de requisitos. Esses critérios requerem uma linguagem de especificação capaz de expressar tempo e valores de saídas, suposições de sincronia e intervalos de tempo para disparos ambientais e informações do estado interno. De acordo com Melhart o modelo ambiental em Statecharts e sua representação de estados possuem formas apropriadas e informações necessárias para realizar uma análise de completeza das suposições ambientais e requisitos que se preocupam com estados.

A conclusão desse estudo é que muitas características de Statecharts facilitam a análise dos requisitos de sistemas de tempo-real, enfatizando que a representação de estados permite avaliar as propriedades dinâmicas, através da análise de alcançabilidade. Melhart sugere também algumas adições na linguagem Statecharts para que um modelo possa ser avaliado com relação à segurança e completeza.

2.5 Comparações de Propriedades de Redes de Petri com as Propriedades de Statecharts

Para ilustrar as idéias trazidas de Redes de Petri para resolver o problema de verificação de propriedades de Statecharts, considera-se um exemplo mostrando um problema modelado em Redes de Petri, e esse mesmo problema especificado em Statecharts. Mostra-se também suas respectivas árvores de alcançabilidade.

O problema considerado é o problema de **Processos X Recursos**: considere um sistema com dois recursos diferentes (R1 e R2) e dois processos (P1 e P2). Se ambos os processos necessitam de ambos os recursos, haverá a necessidade de compartilhá-los. Para isso ser feito, cada processo deve requisitar, e posteriormente liberar, o recurso desejado. O processo P1 requisita primeiro o recurso R1 e então o recurso R2, e finalmente libera R1 e R2. O processo P2 é similar, mas primeiro requisita R2, e então requisita R1.

A modelagem desse problema usando Redes de Petri pode ser vista na Figura 2.6, e a árvore de alcançabilidade para essa Rede de Petri encontra-se na Figura 2.7, ambas na Seção 2.3.

Figura 2.9: Statechart do Problema de Processos X Recursos

Figura 2.10: Árvore de Alcançabilidade para o Statechart da Figura 2.9

O Statechart da Figura 2.9 representa a modelagem desse mesmo problema. Os processos P1 e P2 são representados pelo Statechart como estados paralelos, mostrando assim a concorrência entre os recursos R1 e R2. Cada um dos estados paralelos possui três subestados, os quais representam a forma de utilização dos recursos pelos processos. A configuração inicial (ou default) indica que os processos P1 e P2 estão ociosos, isto é, os recursos R1 e R2 disponíveis e os processos P1 e P2 prontos. A árvore de alcançabilidade para este statechart pode ser vista na Figura 2.10. A definição precisa da árvore de alcançabilidade de Statecharts será feita no próximo capítulo. Vários outros projetos para esse mesmo problema foram considerados, como por exemplo, a representação explícita dos recursos. A solução apresentada na Figura 2.9 é a mais concisa. Esse exemplo mostra que a seqüência de transições $\langle t1, t4 \rangle$ leva o sistema a uma situação de “deadlock”.

É importante notar que a árvore de alcançabilidade para o Statechart possui um comportamento semelhante ao da árvore da Rede de Petri. Um exemplo é que o “deadlock” pode ser constatado em ambas pela existência de nós terminais.

Mostra-se a seguir uma tabela onde são discutidas brevemente as propriedades de Redes de Petri e as propriedades de Statecharts, fazendo-se um relacionamento entre elas.

Propriedades	Redes de Petri	Statecharts
Limitabilidade	Uma RP é K-limitada se em cada um de seus lugares não há mais que K fichas	
Segurança	É um caso especial de limitabilidade, onde $k = 1$	
Conservação	Uma RP é conservativa se a soma das fichas, em cada uma de suas marcas, for sempre igual	
Vivacidade	Garante que o sistema nunca atingirá um estado de inatividade total, onde possa permanecer por tempo indefinido	Em Statecharts, com o nome de Impasse (deadlock) garante que a partir da configuração inicial, não existe nenhuma configuração alcançável na qual nenhuma transição está habilitada

Propriedades	Redes de Petri	Statecharts
Seqüência de Disparos	Pode-se verificar a possibilidade de disparo de uma seqüência específica de transições ou de algum subconjunto dessa seqüência.	Com o nome de Seqüência válida de eventos, é semelhante à definição de seqüência de disparos para Redes de Petri
Alcançabilidade	Garante, a partir de uma marcação inicial M_0 se uma determinada marcação M pode ser alcançada ou não	Define-se alcançabilidade de configurações de estados da mesma forma que alcançabilidade de marcações em Redes de Petri
Reiniciabilidade	Define-se reiniciabilidade em RP, quando o sistema retorna ao seu estado inicial após sua execução.	Um Statechart é reiniciável se existem meios de retornar à sua configuração inicial, a partir de qualquer configuração alcançável de CSo.
Uso de transição	A definição está relacionada com vivacidade. Uma transição está viva se pode ser disparada.	Pode-se definir da mesma forma que em Redes de Petri

Tabela 2.1

Como pode-se ver da tabela, as únicas propriedades de redes de Petri que não foram definidas para Statecharts são limitabilidade, segurança e conservação. O estudo dessas propriedades, a fim de buscar uma maneira de simular em Statecharts mais de uma ficha em cada lugar (talvez com o uso de variáveis), está previsto para uma próxima etapa, que complementará os estudos feitos neste trabalho.

2.6 Considerações finais

Este capítulo apresentou o formalismo de Redes de Petri e Statecharts, enfatizando a semelhança existente entre os dois métodos com relação às suas propriedades dinâmicas.

Da literatura revisada sobre Redes de Petri, constatou-se que existe uma teoria bastante sólida e um grande número de ferramentas implementadas com o objetivo

de verificar suas propriedades dinâmicas. Com relação a Statecharts, provavelmente devido ao seu aparecimento mais recente, esse é um assunto pouco explorado, e a literatura existente pouco revela sobre soluções dadas a este problema, que tem, entretanto, muita importância, na medida em que mais se difunde o uso dos Statecharts.

Aproveitando a semelhança entre Redes de Petri e Statecharts, este trabalho tem como objetivo trazer de Redes de Petri as idéias que estão bem fundamentadas para verificação de propriedades dinâmicas, e aplicá-las para solucionar o problema de verificação de propriedades dinâmicas em Statecharts.

Capítulo 3

Definição de uma Árvore de Alcançabilidade para Statecharts

3.1 Considerações Iniciais

Neste capítulo serão discutidos os aspectos de construção de uma árvore de alcançabilidade para Statecharts. A árvore de alcançabilidade definida é semelhante à árvore de Redes de Petri, com algumas características próprias, tais como: a notação utilizada; a maneira de representar a hierarquia dos estados; o tratamento de história; e a forma de utilizá-la. Essa árvore de alcançabilidade foi definida para ser utilizada pelos algoritmos verificadores de propriedades dinâmicas de Statecharts, que serão vistos no próximo capítulo.

Este capítulo está organizado da seguinte forma: na Seção 3.2 discute-se a construção da árvore de alcançabilidade de Statecharts, mostrando as suas características próprias. Na Seção 3.3 apresentam-se vários exemplos de Statecharts com suas respectivas árvores de alcançabilidade, procurando cobrir os diversos casos. E finalmente, na Seção 3.4, mostra-se o algoritmo para a construção da árvore de alcançabilidade.

3.2 Construção da Árvore de Alcançabilidade

(A árvore de alcançabilidade é construída considerando-se todas as possíveis seqüências de eventos do Sistema Sob Desenvolvimento (SSD). Em cada passo todos os eventos são considerados como verdadeiros, assim todas as transições habilitadas num passo levam a uma nova configuração do sistema. As condições são incluídas nas expressões dos eventos e também são consideradas quando uma transição é tomada.) A

execução do SSD é uma sequência representada pelo conjunto $\{(CS_i, \Upsilon_i, \Pi_i^*)\}, i \geq 0$, onde:

1. $CS_0 = (X_0, \Pi_0, \Theta_0, \xi_0)$, onde X_0 é a configuração de estado inicial e os outros símbolos são os estímulos externos que ocorrem no primeiro intervalo de tempo I_0 . Π_0 é o conjunto de eventos externos gerados pelo ambiente, Θ_0 é o conjunto de condições primitivas cujo valor é TRUE e ξ_0 é uma função que para cada variável x no ambiente, calcula o valor de x .
2. Υ_i é um passo tomado em CS_i , no instante σ_{i+1} , para $i \geq 0$.
3. Π_i^* é o conjunto de eventos gerados por Υ_i , para $i \geq 0$.

Dessa forma, tem-se uma sequência de configurações de estado onde CS_{i+1} é a configuração do sistema alcançada de CS_i , quando as transições em Υ_i , $i \geq 0$ são tomadas. É importante ressaltar que no momento da construção da árvore, cada uma dessas transições é considerada separadamente, ^{ou havendo} e não mais que uma transição ocorrendo ao mesmo tempo, como é a semântica dos Statecharts definida em [Hare87b].

Para cada um dos elementos dessa sequência toma-se o conjunto $RS = \{(\Upsilon_j, \Pi_j^*)\}$ de possíveis reações do sistema.

Na árvore, cada nó é associado a uma configuração de estados do SSD. Os nós são classificados como:

- Configuração nova.
- Configuração velha.
- Configuração terminal.
- Configuração história.

As *configurações novas* são aquelas configurações geradas e que ainda não foram inseridas na árvore.

As *configurações velhas* são as configurações que já foram processadas e inseridas na árvore de alcançabilidade.

As *configurações terminais* representam aquelas que possuem o conjunto $RS = \emptyset$, ou seja, não possuem nenhuma transição habilitada.

Os *nós história* representam todas as configurações alcançáveis a partir de transições ou estados associados ao símbolo de história. Os símbolos **H** e **h** aparecem nestes nós em substituição às configurações alcançáveis, naqueles estados que são ativados

por história H^* e H , respectivamente. Após a criação da árvore, ao percorrê-la, quando um nó história for encontrado, substituem-se os símbolos H e h pela última configuração alcançada por aquele estado. Este procedimento será visto com mais detalhes, por meios de exemplos, no capítulo seguinte.

A árvore de alcançabilidade é construída tomando-se a configuração de estados inicial (ou default) e fazendo-a raiz da árvore. A raiz da árvore é uma *configuração nova* a ser processada. Para essa configuração consideram-se todos os eventos que podem ocorrer. Para cada um deles uma nova configuração é gerada, classificada e inserida na árvore. Este procedimento se repete até que não existam mais *configurações novas* a serem processadas, ou seja, todas as configurações alcançáveis pelo SSD foram classificadas em configurações velhas ou terminais, dando o tratamento adequado quando aparece o símbolo de história.

A hierarquia de estados de um diagrama Statechart pode ser mostrada através de uma árvore E/OU. A Figura 3.1 mostra para um diagrama Statechart, sua árvore E/OU equivalente. Nessa figura, os semi-círculos que abrangem todos os arcos dos estados C e D representam a decomposição AND desses estados.

A notação utilizada para a representação de um nó da árvore de alcançabilidade é a seguinte:

- Parênteses “(” e “)” são utilizados para representar os superestados tipo XOR.
- Colchetes “[” e “]” são utilizados para representar os superestados tipo AND.
- Os estados átomos são representados por “1” ou “0”, conforme estejam ligados ou desligados. Podem ser representados, eventualmente, pelo seu próprio nome.

Para o Statechart da Figura 3.1, um nó da árvore de alcançabilidade teria a forma: $(0, [(0,0), (0,0)], [(0,0), (0, (0,0))])$, onde os parênteses mais externos representam o estado M, que é um estado XOR composto por três descendentes, o estado básico B, representado por “0” e os estados C e D que são do tipo AND e representados por colchetes. O estado C possui dois descendentes, do tipo XOR, C_1 e C_2 e cada um deles possui dois descendentes que são estados básicos: C_{11} , C_{12} , C_{21} e C_{22} , respectivamente. O estado D possui dois descendentes XOR, D_1 e D_2 , ambos representados por parênteses. O estado D_1 possui dois descendentes que são estados básicos: D_{11} e D_{12} . E finalmente, o estado D_2 possui como descendentes o estado básico F e o estado XOR E, o qual possui dois descendentes que são os estados básicos: E_1 e E_2 .

Figura 3.1: Diagrama Statechart e sua Árvore E/OU Equivalente

3.3 Exemplos de Árvores de Alcançabilidade

Nesta seção mostram-se vários exemplos de construção da árvore de alcançabilidade. Os exemplos considerados retratam os vários detalhes existentes em Statecharts e como foram tratados pelo algoritmo de construção da árvore. Para cada exemplo é dado um Statechart e sua respectiva árvore de alcançabilidade.

Exemplo 1:

Para o Statechart mostrado na Figura 3.2 a descrição de um nó da árvore é a seguinte:

$$(H,G,C,B,(E,[(F_{12},F_{11}), (F_{22},F_{21})]))$$

Aqui, ao invés de representar os estados básicos com “0” e “1” como foi dito anteriormente, mostram-se os nomes desses estados. Isto é feito com a finalidade de facilitar o entendimento da árvore de alcançabilidade, onde os nós são escritos normalmente, utilizando-se a notação com “0” e “1”. Este procedimento será aplicado a todos os exemplos mostrados nesta seção.

A árvore de alcançabilidade para o Statechart da Figura 3.2 é mostrada na Figura 3.3. Esse Statechart possui duas transições associadas a símbolos de história: a transição *f* e a transição *d*, e ambas têm como estado destino (ED), o estado D. O estado D é visitado por default, e essa configuração aparece como a raiz da árvore de alcançabilidade. Neste caso cria-se normalmente o *nó história*.

Um outro aspecto importante e que merece ser discutido é a forma de tratar as transições que têm como estado origem um superestado. No Statechart da Figura 3.2 têm-se as transições *a* e *d* saindo do superestado D. Essa organização hierárquica adotada pelos Statecharts significa que se o sistema estiver em qualquer um dos subestados de D e uma dessas duas transições ocorrer, por exemplo a transição *d*, o sistema irá para o estado B.

Ao construir a árvore de alcançabilidade, a maneira escolhida para organizar essa hierarquia foi a seguinte: para toda configuração de estados, onde são representados os subestados de D ligados, consideram-se as transições *a* e *d*. Outra solução seria considerar essas transições apenas no primeiro nó da árvore onde o estado D estivesse ligado, mas essa solução dificultaria bastante no momento de percorrer a árvore. Na árvore de alcançabilidade da Figura 3.3, pode-se observar a solução adotada para representar a hierarquia de estados. Nota-se que essa solução acaba por deixar “plana” a representação hierárquica dos Statecharts.

Figura 3.2

$(H, G, C, B, (E, [(F_{12}, F_{11}), (F_{22}, F_{21})]))$

Figura 3.3: Árvore de Alcançabilidade para o Statechart da Figura 3.2

O Statechart considerado nesse exemplo tem 14 estados e 11 transições, sendo duas delas associadas a símbolos de história; sua árvore de alcançabilidade correspondente possui 24 nós.

Exemplo 2:

A descrição de um nó para o Statechart da Figura 3.4 é:

$$(B,(T,U,(Z,[(L,R),(P,O)])))$$

Ao montar a árvore de alcançabilidade para esse exemplo, encontra-se, a partir dos estados D e T ligados, a transição *a* que está associada a um símbolo de história, cujo estado destino (ED) é o estado S. Dado que a transição *a* possui história, o procedimento seria criar um nó indicando história no estado S, como foi feito no exemplo 1. Neste caso, ao considerar simplesmente o nó história, a construção da árvore neste ramo pararia nesse ponto. Mas isto acarretaria perda de informação, e a árvore resultante não estaria correta, pois o estado S, que ainda não foi visitado até este momento, e todas as configurações alcançadas a partir dele não apareceriam na árvore.

A solução encontrada para casos como esses é, ao encontrar transições com símbolos de história associados, percorrer-se a árvore de alcançabilidade a partir desse ponto até o nó raiz, verificando se o sistema já visitou o estado destino dessa transição. Caso tenha visitado, constrói-se normalmente o nó história, como visto no exemplo 1; e caso contrário, quando esse estado ainda não foi visitado, considera-se a entrada por default e cria-se uma configuração onde esse estado é visitado pela primeira vez e, além disso, cria-se o nó história. A configuração resultante dessa entrada por default faz com que a árvore de alcançabilidade continue a ser construída e que contenha todas as configurações alcançadas a partir desse estado. No próximo capítulo, onde são mostrados exemplos de como percorrer a árvore, será visto que essa solução não causa não determinismo pela existência desses dois nós, o nó default e o nó história.

Neste exemplo, o Statechart com 14 estados e 11 transições, sendo duas associadas a símbolos de história, deu origem a uma árvore de alcançabilidade com 27 nós, que é mostrada na Figura 3.5

Exemplo 3:

Um nó da árvore de alcançabilidade para o Statechart da Figura 3.6 tem a forma:

$$((E,D),F,[(O,R),(P,Z)])$$

$(B, (T, U, (Z, [(L, R), (P, O)])))$

Figura 3.4

Figura 3.5: Árvore de Alcançabilidade para o Statechart da Figura 3.4

O Statechart da Figura 3.6 ilustra o caso em que o símbolo de história está associado ao estado e não à transição, como foi mostrado nos exemplos 1 e 2. Quando existe história associada a um estado, isso é equivalente a dizer que todas as transições que chegam no estado pai deste estado possuem história. Logo, ao tomar uma transição, além de verificar se está associada a um símbolo de história, deve-se verificar se o estado destino desta transição possui algum descendente associado a um símbolo de história; e caso possua, o tratamento a ser dado a essa transição é o mesmo dado às transições que possuem história. Este exemplo ilustra também o caso tratado pelo exemplo 2, e na árvore de alcançabilidade aparecem transições onde foram consideradas entradas por default, seguidas da criação de nós história.

Neste exemplo tem-se um Statechart com 12 estados, 11 transições e símbolo de história associado a um estado. A árvore de alcançabilidade gerada para esse Statechart possui 28 nós, e é mostrada na Figura 3.7.

Exemplo 4:

Um nó da árvore de alcançabilidade para o exemplo 4 tem a forma:

$$(B,[(G,F),(H,I),(L,J)])$$

No Statechart da Figura 3.8 vê-se uma transição cujo estado destino é um estado paralelo. A transição *b* tem como destino o estado D, existindo assim subtransições para cada um dos subestados de D. O exemplo mostra também que essas subtransições possuem história associada a uma delas e têm história associada ao estado P. Isto exige um tratamento diferenciado a cada uma dessas subtransições.

Para resolver casos como este, o algoritmo de construção da árvore de alcançabilidade verifica, para cada transição, se o seu estado destino é paralelo. Caso seja, cada uma das subtransições deve ser analisada. Essa análise é feita de forma análoga aos casos tratados nos exemplos 1, 2 e 3. A Figura 3.9 mostra a árvore de alcançabilidade para este exemplo. Pode-se notar que ele abrange todos os casos discutidos até aqui, ou seja, para a transição *b* foi necessário criar uma entrada por default e um nó história. Também, para a criação desse nó história, o tratamento paralelo, explicado neste exemplo, foi empregado.

O Statechart deste exemplo possui 14 estados e 8 transições, sendo que existe uma subtransição e um estado associados a símbolos de história. Sua árvore de alcançabilidade correspondente possui 33 nós.

$((E,D),F,[(O,R),(P,Z)])$

Figura 3.6

Figura 3.7: Árvore de Alcançabilidade para o Statechart da Figura 3.6

$(B, [(G, F), (H, I), (L, J)])$

Figura 3.8

Figura 3.9: Árvore de Alcançabilidade para o Statechart da Figura 3.8

3.4 Algoritmo

É mostrado abaixo o algoritmo para a construção da árvore de alcançabilidade para Statecharts (AAS).

```
1 Construir AAS
2 INÍCIO
3   Criar uma fila vazia  $FC$ ;
4   Fazer  $CS_0$  a raiz da árvore de alcançabilidade AAS;
5   Inserir  $CS_0$  na  $FC$ ;
6   ENQUANTO  $FC$  não for vazia FAÇA
7     Remover  $CS$  da  $FC$ ;
8     Obter  $RS$ , o conjunto de todas as possíveis transições de  $CS$ ;
9     SE  $RS$  não for vazio
10    ENTÃO
11      PARA CADA transição de  $RS$  FAÇA
12        Seja  $T$  uma transição de  $RS$ ;
13        Simular o passo para  $T$  obtendo  $CS'$ ;
14        SE algum estado destino  $ED^k$  de  $T$  tem um símbolo de história
15          ou seus estados defaults tem um símbolo de história
16        ENTÃO
17          SE  $ED^k$  ainda não foi ativado nesse caminho
18          ENTÃO
19            Obter  $ED_0^k$ , a configuração default para  $ED^k$ 
20            Criar um nó com  $CS'$  substituindo a subconfiguração
21              de  $ED^k$  por  $ED_0^k$ ;
22            SE  $CS'$  já existir na AAS
23            ENTÃO
24              Fazer o nó  $CS'$  uma configuração velha;
25            SENÃO
26              Fazer o nó  $CS'$  uma configuração nova;
27              Inserir  $CS'$  na  $FC$ ;
28            FIM-SE
29            Inserir o nó na AAS como um filho de  $CS$ ;
30            Rotular o arco de  $CS$  para  $CS'$  como "<exp-evento>/def;
31          FIM-SE
32          Criar um nó  $CS''$  de  $CS'$  substituindo a
33            subconfiguração  $ED^k$  por (H/h);
34          Fazer o nó  $CS''$  uma configuração história;
35          Inserir o nó  $CS''$  na AAS como um filho de  $CS$ ;
```

```

36         Rotular o arco de  $CS$  para  $CS''$  como "<exp-evento>";
37     SENÃO
38         Criar um nó com  $CS'$ ;
39         SE  $CS'$  já existir na AAS
40             ENTÃO
41                 Fazer o nó  $CS'$  uma configuração velha;
42             SENÃO
43                 Fazer o nó  $CS'$  uma configuração nova;
44                 Inserir  $CS'$  na  $FC$ ;
45             FIM-SE
46         Inserir o nó  $CS'$  na AAS como um filho de  $CS$ ;
47         Rotular o arco de  $CS$  para  $CS'$  como "<exp-evento>";
48     FIM-SE
49     Reinstanciar  $CS$  como sendo a configuração do Statechart;
50 FIM-PARA CADA
51 SENÃO
52     Fazer o nó  $CS$  uma configuração terminal;
53 FIM-SE
54 FIM-ENQUANTO
55 FIM Construir AAS

```

Alguns aspectos merecem uma análise mais detalhada no algoritmo proposto:

- A sentença da linha (13), onde se simula um passo da execução dos Statecharts, envolve todo o trabalho do Simulador de Especificações em Statecharts (SES), descrito em [Fort91], o qual foi desenvolvido com base na semântica formal desenvolvida por Harel [Hare87b].
- A condição " ED^k ainda não foi ativado nesse caminho", na linha 17, é verdadeira se no caminho que vai da configuração CS em questão até a raiz da árvore não existir nenhum estado básico ativo (representado pelo número 1) em alguma sub-configuração ED^k .
- Para avaliar a condição " CS' já existe na AAS" nas linhas 22 e 39, uma busca é feita na árvore em construção com o objetivo de encontrar a configuração CS' .
- As expressões dos eventos incluem também as condições; assim, foi usado o rótulo <exp-evento> para os arcos. As ações não são consideradas na árvore de alcançabilidade pois existe interesse apenas nos eventos externos. É suposto que as ações são calculadas em cada passo (linha 13).

- Na linha (49) do algoritmo, a sentença “Reinstanciar SC como sendo a configuração do Statechart” faz com que a árvore E/OU, usada pelo simulador, represente a configuração desejada naquele momento. Ao construir a árvore de alcançabilidade, toma-se uma configuração, e para cada transição relevante a essa configuração, através do simulador, obtém-se novas configurações, que ficarão representadas nessa árvore. Para que o algoritmo funcione corretamente é necessário fazer a cada nova configuração gerada, com que a árvore E/OU volte a representar aquela configuração tomada inicialmente.
- A notação ED^k indica o conjunto de estados destino associados a símbolos de história. A cardinalidade desse conjunto é $k \geq 1$. Se $k = 1$ então a transição tem somente um estado destino.

Para mostrar que o algoritmo desempenha seu papel corretamente, são feitas algumas afirmações sobre ele, seguidas de justificativas:

1) O nó raiz da árvore corresponde à configuração inicial.

Justificativa:

Ao executar a linha (4) do algoritmo, o simulador inicia a simulação considerando ligados os estados defaults do Statecharts em questão. A configuração CS_0 é trazida, então, da árvore E/OU e inserida como o primeiro nó da árvore.

2) Cada nó da árvore de alcançabilidade, com exceção dos nós história, é uma configuração do Statechart.

Justificativa:

Seja a raiz da árvore a configuração de estados inicial do Statechart. Para essa configuração de estados o algoritmo obtém todas as suas transições relevantes. Para que ele consiga obter todas essas transições, é necessário fazer com que todos os eventos da tabela de eventos sejam verdadeiros. Isso quer dizer que todos os eventos ocorreram e então todas as transições, cujo estado origem estão ligados, podem disparar. Para cada transição obtida, faz-se a simulação e o resultado é trazido da árvore E/OU e inserido na árvore de alcançabilidade como filho do nó raiz. Após a simulação de cada transição, a árvore E/OU é restaurada, como foi explicado anteriormente, e volta a representar a configuração de estados inicial. Toda configuração resultante dessa simulação, caso ainda não esteja representada na árvore, é inserida numa fila. Quando termina o processamento do nó raiz, novas configurações serão obtidas dessa fila, a árvore E/OU será restaurada, e esse procedimento se repete para os seus filhos e assim por diante.

Logo, considerando que o simulador funcione corretamente, todo nó da árvore, com exceção dos nós história, é uma configuração do Statechart.

3) O processo de construção da árvore de alcançabilidade é finito, pois a árvore de alcançabilidade de um Statecharts é finita.

Justificativa:

Um diagrama Statecharts possui um número finito de estados e um número finito de transições. A execução desse diagrama pode ser infinita, levando a uma sequência infinita de configurações de estado alcançados. Isso acontece quando em cada passo da execução sempre existem transições habilitadas. Por outro lado, dado que o número de estados e de transições são sempre finitos, o número de todas as configurações de estados possíveis para o diagrama é finito. Conclui-se então que a sequência infinita de configurações de estados, gerada pela execução infinita do Statechart, é constituída por repetição infinita das possíveis configurações de estados do diagrama.

Ao construir a árvore de alcançabilidade, quando aparecem configurações de estados repetidas, elas são consideradas em um único nó da árvore, e sempre que for inserida uma configuração já existente na árvore, esse nó apontará para aquele nó onde se encontra essa mesma configuração, e de onde saem os seus descendentes. O processo de construção da árvore termina quando todas as possíveis configurações de estados estiverem representadas na árvore.

Dessa forma, a árvore de alcançabilidade para Statecharts representa um grafo de estados, a menos dos arcos de história. O fato dos nós história também serem representados na árvore não influencia na sua natureza finita. Como esses nós representam as transições e estados com símbolos de história associados e esses são finitos, o número de nós história que aparecerá na árvore também será finito.

4) Ao percorrer a árvore de alcançabilidade, quando algum nó história é encontrado, existe uma configuração de estados pela qual esse nó pode ser substituído, de forma a representar corretamente a história do sistema.

Justificativa:

Considere o Statechart do Exemplo 2. Uma possível execução desse Statechart é: inicialização por default ativando os estados D e T, ocorrendo o evento a o sistema entra em S ativando o estado default Z, acontecendo o evento b o sistema transiciona para os estados paralelos R e O, ocorrendo o evento h o sistema transiciona para L e O, acontecendo o evento e_1 o sistema volta ao estado T, e se o evento a acontecer novamente, um nó história é encontrado o qual indica história no estado S.

Observando a árvore de alcançabilidade desse exemplo, para que esse nó história tenha sido alcançado, foi necessário percorrer um determinado caminho na árvore, e esse caminho, em algum momento, levou o sistema em S. Então a configuração de estados pela qual o nó história deve ser substituído se encontra em um dos nós pertencente ao caminho percorrido para se atingir esse nó história. Para que a substituição seja feita, é

só fazer o caminho inverso na árvore e verificar qual é o nó que representa a configuração de estados em S , onde o sistema esteve pela última vez. Nesse exemplo, o nó história deve ser substituído pelo nó C_8 .

O algoritmo de construção da árvore de alcançabilidade para Statecharts é bastante complexo pelo fato dos Statecharts possuírem uma semântica bastante rica. Todos os detalhes semânticos que devem ser analisados pelo algoritmo, tais como história associada a estados, história associada a transições e todos os casos gerados que foram discutidos pelos exemplos da seção 3.3 fazem com que seu tempo de execução seja elevado.

O número de estados e transições existentes no Statechart também contribui para o aumento do tempo de execução desse algoritmo. No momento de sua execução, quanto a árvore que está sendo criada se torna muito grande, muito tempo de processamento será gasto pelo algoritmo, pois a cada nova configuração gerada, ele faz uma busca na árvore para verificar se essa configuração já se encontra nela.

3.5 Considerações Finais

Este capítulo definiu e apresentou os principais aspectos da construção de uma árvore de alcançabilidade para Statecharts. Vários exemplos de Statecharts e suas árvores de alcançabilidade foram mostrados com o objetivo de ilustrar a construção da árvore. Foi apresentado um algoritmo para a construção da árvore de alcançabilidade, discutindo-se o seu funcionamento e sua complexidade.

A definição de uma árvore de alcançabilidade para Statecharts foi uma tarefa bastante trabalhosa. Embora sua definição tenha sido inspirada pela árvore de alcançabilidade existente em Redes de Petri, os aspectos particulares dos Statecharts levou à necessidade de se criar uma notação própria, resolver os problemas de história e o problema da hierarquia de estados. O empenho em definir e construir essa árvore, entretanto, justificou-se plenamente, uma vez que ela pode ser utilizada para verificar as propriedades dinâmicas de Statecharts, como será mostrado no próximo capítulo.

Capítulo 4

Propriedades Dinâmicas de Statecharts Baseadas na Árvore de Alcançabilidade

4.1 Considerações Iniciais

Este capítulo ilustra como utilizar a árvore de alcançabilidade para verificar as propriedades dinâmicas de Statecharts. Para aquelas propriedades dinâmicas consideradas na Tabela 2.1 do Capítulo 2, mostram-se também exemplos de como verificá-las, bem como seus algoritmos correspondentes. A complexidade de cada um desses algoritmos é igualmente discutida.

4.2 Alcançabilidade de uma Configuração

Como foi dito anteriormente, alcançabilidade em Statecharts é definida como sendo aquelas configurações de estados que o sistema pode alcançar.

Essa propriedade pode ser útil quando um sistema está sendo especificado, ao fornecer meios para verificar se todos os estados especificados são ativados em algum momento, ou, se sob certas condições, alcança uma determinada configuração. Por exemplo, num Sistema Avionístico deseja-se saber se os assentos podem ser lançados para fora quando a aeronave se encontra em alguma situação crítica. Nesse exemplo, uma análise de alcançabilidade pode ser feita para verificar se o sistema alcança ou não essa determinada situação, partindo de uma situação específica.

Figura 4.1

Figura 4.2: Árvore de Alcançabilidade para o Statechart da Figura 4.1

Dois tipos de alcançabilidade são considerados: a alcançabilidade de uma configuração a partir da configuração inicial e a alcançabilidade de uma configuração a partir de uma configuração qualquer.

4.2.1 Alcançabilidade de uma Configuração a partir da Configuração Inicial

Dada uma determinada configuração de estados, o objetivo é verificar, a partir da configuração inicial, se essa configuração pode ser alcançada. Uma configuração de estados é alcançável se for um nó da árvore. Portanto, o algoritmo que verifica essa propriedade, percorre a árvore em busca da configuração dada. Se ela estiver na árvore, é fornecida a sequência de eventos que leva da configuração de estados inicial até essa configuração de estados. Caso contrário, a configuração dada não é alcançável. Observe que pode existir mais do que uma sequência de eventos que leva à configuração desejada.

Exemplo:

Considere o Statechart e sua árvore de alcançabilidade nas Figuras 4.1 e 4.2. Fornecendo a configuração de estados onde os estados R e P estão ligados, o algoritmo percorrerá a árvore de alcançabilidade e a encontrará no nó C_9 , e fornecerá a seguinte sequência de eventos: a, b, k .

Como pode-se observar na árvore, essa sequência de eventos é o caminho mais curto para se chegar à configuração dada. Isto acontece pelo fato da busca ser feita em largura, e assim que a configuração é encontrada, a busca é encerrada. Para se obter outros caminhos possíveis basta percorrer a árvore ininterruptamente, e sempre que a configuração for encontrada, fornecer a sequência de eventos necessária para atingi-la. Por exemplo, a configuração $(0, (0, 0, (0, [(0, 1), (1, 0)])))$ também se encontra no nó filho de C_{10} , dada a transição i , e a sequência de eventos que leva a esse nó é: a, b, h, k, i . Na verdade, existem inúmeros caminhos para se chegar a essa configuração. Ao observar a árvore de alcançabilidade, pode-se notar que, de praticamente todos os nós folha, existe meios para se chegar a essa configuração.

Algoritmo para verificar alcançabilidade (VPDS/ALC):

```
1 Algoritmo VPDS/ALC
2 INÍCIO
3   Ler configuração fornecida,  $CS$ ;
4   Verificar se a configuração  $CS$  é consistente;
5   SE  $CS$  é inconsistente
6     ENTÃO
7       Imprimir (' A configuração dada é inconsistente');
8   SENÃO
9     Buscar configuração na árvore;
10    SE encontrou
11      ENTÃO
12        Fornecer a seqüência de eventos necessária para atingi -la, a partir da
13        configuração inicial
14      SENÃO
15        Imprimir ('A configuração dada não é alcançável');
16    FIM-SE;
17  FIM-SE;
18 FIM VPDS/ALC
```

O algoritmo, para verificar alcançabilidade, verifica primeiramente, se a configuração fornecida está consistente, isto é, ela deve realmente ser uma possível configuração do Statechart em questão, e além disso obedecer suas regras sintáticas e semânticas.

A complexidade desse algoritmo se resume na complexidade do algoritmo de busca na árvore, que na pior das hipóteses, caso a configuração não seja encontrada, é n , e em média é $n/2$, onde n representa o número de configurações de estado do Statechart. Esse número depende do número de estados e transições existentes no Statechart.

4.2.2 Alcançabilidade de uma Configuração a partir de uma Configuração qualquer

Nessa propriedade, dada duas configurações de estados CS_1 e CS_2 , verifica-se a existência de um caminho que leva de CS_1 a CS_2 , e caso positivo, mostra-se a seqüência de eventos necessária para o sistema alcançar CS_2 a partir de CS_1 .

Para verificar essa propriedade é necessário primeiro procurar na árvore as duas configurações fornecidas. Se as duas configurações estiverem na árvore então inicia-se a busca com base em CS_2 a partir de CS_1 . Essa busca é trivial caso CS_2 seja

um descendente de CS_1 . Caso contrário, um caminho dos ancestrais de CS_2 deve ser construído e uma busca é feita, a partir de CS_1 , verificando se existe algum descendente de CS_1 cujo ponteiro de retorno aponta para algum ancestral de CS_2 . Se for encontrado algum nó que satisfaça essa condição então CS_2 é alcançado a partir de CS_1 . Nesses dois casos mostra-se a seqüência de eventos que leva de CS_1 a CS_2 . Se as buscas acima não forem bem sucedidas, a última hipótese possível é que CS_2 pode ser alcançado a partir de CS_1 por história. O procedimento nesse caso é verificar se CS_1 possui algum descendente que seja um nó história. Em caso positivo deve-se verificar se a configuração CS_2 é alguma instância desse nó história encontrado, e se for, uma mensagem é emitida dizendo que CS_2 pode ser alcançado a partir de CS_1 por história, e nesse caso, nenhuma seqüência de eventos é mostrada.

Exemplos:

1) Nas Figuras 4.1 e 4.2 ao fornecer a primeira configuração (CS_1), onde o estado Z está ligado, e a segunda configuração (CS_2), onde os estados L e P estão ligados, o algoritmo localizará CS_1 e CS_2 nos nós C_3 e C_{10} , respectivamente. A resposta para esse exemplo é que a segunda configuração é alcançável a partir da primeira, e a seguinte seqüência de eventos será fornecida: b, h, k . Esse caso é o caso da busca trivial pois C_{10} é um descendente de C_3 .

2) Fornecendo a primeira configuração (CS_1) onde os estados L e P estão ligados, e a segunda configuração onde o estado U está ligado, os nós localizados pelo algoritmo são CS_1 no nó C_{10} e CS_2 no nó C_5 . Como C_5 não é um descendente de C_{10} , o caminho C_1, C_5 é construído pelo algoritmo, e uma busca é feita para verificar se existe um descendente de C_{10} que aponta para um dos elementos desse caminho. Existe um descendente de C_{10} que retorna para o elemento C_1 do caminho, portanto C_5 é alcançado a partir de C_{10} , através da seguinte seqüência de eventos: e_1, c .

3) Ao fornecer a primeira configuração (CS_1) como sendo o estado B ligado e a segunda configuração (CS_2), onde os estados R e O estão ligados, tem-se que CS_1 e CS_2 são localizados nos nós C_2 e C_7 , respectivamente. Como C_7 não é descendente de C_2 e não existe nenhum descendente de C_2 que aponta para algum ancestral de C_7 , então será feita uma busca nos descendentes de C_2 para verificar se existe algum nó história. Nesse caso o nó história C_6 é encontrado. Ao observar esse nó, nota-se que C_7 é uma instância dele. Logo, a resposta dada pelo algoritmo é que C_7 pode ser alcançado a partir de C_2 através de história, caso o evento b ocorra.

Algoritmo para verificar a alcançabilidade de uma configuração a partir de uma configuração qualquer (VPDS/ALC-QUALQUER)

1 Algoritmo VPDS/ALC-QUALQUER

2 INÍCIO

3 Ler a primeira configuração CS_1 ;

4 Ler a segunda configuração CS_2 ;

5 SE CS_1 ou CS_2 for inconsistente

6 ENTÃO

7 Imprimir ("Configuração dada é inconsistente");

8 SENÃO

9 Buscar a configuração CS_1 na árvore;

10 SE não encontrou

11 ENTÃO

12 Imprimir ("A configuração CS_1 não se encontra na árvore");

13 SENÃO

14 Buscar CS_2 na árvore;

15 SE não encontrou

16 ENTÃO

17 Imprimir ("A configuração CS_2 não se encontra na árvore");

18 SENÃO

19 Buscar CS_2 nos descendentes de CS_1 ;

20 SE encontrar

21 ENTÃO

22 Imprimir ("A configuração CS_2 é alcançável a partir de CS_1 ");

23 Mostrar seqüência de eventos que leva CS_1 à CS_2 ;

24 SENÃO

25 Construir caminho dos ancestrais de CS_2 , Cam_d ;

26 Buscar nos descententes de CS_1 algum nó cujo ponteiro
27 de retorno leva à um dos elementos de Cam_d .

28 SE encontrar

29 ENTÃO

30 Imprimir (" CS_2 é alcançável a partir de CS_1 ");

31 Mostrar seqüência de eventos que leva CS_1 à CS_2 ;

32 SENÃO

33 Buscar nó história, noh , que seja algum descendente de CS_1

34 SE encontrar e CS_2 for instância de noh

35 ENTÃO

36 Obter o nome do evento (ev) que leva CS_1 à noh ;

37 Imprimir ("Estando em CS_1 e o evento ev ocorrer,

```

38             o sistema pode alcançar  $CS_2$  por história");
39             SENÃO
40                 Imprimir (" $CS_2$  não é alcançável a partir de  $CS_1$ ");
41             FIM-SE
42         FIM-SE
43     FIM-SE
44 FIM-SE
45 FIM-SE
46 FIM-SE
47 FIM-SE
48 FIM VPDS/ALC-QUALQUER.

```

Esse algoritmo faz várias buscas na árvore de alcançabilidade. Em média faz três buscas, e em seu pior caso, cinco buscas são feitas. A complexidade desse algoritmo se resume em somar a complexidade do algoritmo de busca. Logo, em média, possui complexidade $3n/2$ (onde n é o número de configurações do Statechart).

Como pode ser visto pelo Exemplo 3, o algoritmo mostrado acima apenas diz que C_7 pode ser alcançado a partir de C_2 , mas não encontra um caminho de C_2 para C_7 . Em casos como este, não se sabe por qual configuração pode-se substituir o nó história.

Este problema fica completamente resolvido considerando-se uma outra maneira de resolver o caso de alcançabilidade a partir de uma configuração qualquer. Essa nova maneira é a seguinte: dado CS_1 e CS_2 , constrói-se uma nova árvore de alcançabilidade a partir da configuração CS_1 . Assim, o problema se resume em verificar a alcançabilidade de CS_2 a partir da configuração inicial, que é CS_1 .

A árvore de alcançabilidade para o Statechart da Figura 4.1, que foi construída considerando-se o estado B como default, ao invés do estado D, é mostrada na Figura 4.3. Observando essa nova árvore de alcançabilidade, nota-se que agora se consegue resolver o problema do Exemplo 3, isto é, mostrar um caminho que leva de C_2 à C_7 , que é: $b/def, a/def, b$.

Essa solução deve ser considerada como uma alternativa à solução dada anteriormente, quando esta não conseguir resolver de forma adequada algum caso mais complexo. Para os casos mais simples, como aqueles mostrados nos exemplos 1 e 2, a solução anterior parece mais eficiente, pois não existe a necessidade de, a cada dupla de configuração fornecida, sempre gerar uma nova árvore de alcançabilidade.

Figura 4.3: Árvore de Alcançabilidade Construída com a Configuração B como Default

4.3 Reiniciabilidade

[Para verificar se um Statechart é reiniciável, basta verificar se para toda configuração CS , alcançada a partir de CS_0 , existe uma sequência de eventos que a faça retornar à configuração CS_0 .]

O algoritmo discutido na seção anterior pode também ser utilizado para verificar a propriedade de reiniciabilidade. Para essa finalidade, é só fornecer ao algoritmo todas as configurações alcançáveis de CS_0 como sendo a primeira configuração, CS_1 e, a segunda configuração, CS_2 , a configuração CS_0 . Se para toda configuração CS_1 fornecida existir uma sequência de eventos que leva a CS_0 , então o Statechart é reiniciável. Por exemplo, o Statechart da Figura 4.1 é reiniciável, pois, observando-se sua árvore de alcançabilidade, nota-se que para cada configuração alcançável a partir do nó C_1 existe uma sequência de eventos que faz com que o Statechart retorne à configuração C_1 .

4.4 Impasse (Deadlock)

[Um sistema modelado em Statecharts poderá ter possíveis deadlocks se atingir configurações de estados das quais nenhuma transição pode disparar e, portanto, em sua árvore de alcançabilidade aparecerão nós terminais.] Aqui diz-se “possíveis deadlocks” pois existem sistemas que modelam algumas tarefas as quais possuem início e fim de processamento bem definido, e, neste caso a árvore de Alcançabilidade gerada possuirá nós terminais, que não são deadlocks, mas que indicam o final de processamento dessas tarefas. A Figura 4.4 pode ilustrar melhor esse caso; o estado b_5 representa final de processamento da tarefa modelada, e a árvore de alcançabilidade correspondente a esse Statechart, que é mostrada na Figura 4.5, possui dois nós terminais: C_6 e C_7 . A existência desses nós, neste caso, não representa um deadlock no sistema.

Já no exemplo de Processos X Recursos mostrados no Capítulo 2, a árvore de alcançabilidade da Figura 2.10 possui dois nós terminais: C_5 e C_6 . De acordo com o problema modelado esses nós terminais representam uma situação de impasse, onde o processo P_1 precisa do recurso R_2 para liberar R_1 e o processo P_2 precisa do recurso R_1 para liberar R_2 . Isso ocasiona uma paralização total no sistema.

[O algoritmo para verificação de “deadlocks” percorre a árvore toda em busca de configurações “terminais”. Ao encontrar alguma configuração “terminal” é emitido um aviso que existe um possível deadlock quando o sistema alcança essa configuração, e é exibida a sequência de eventos que leva, a partir da configuração inicial, a essa situação.]

Figura 4.4

Exemplos:

1) Considere a árvore de alcançabilidade do Problema de Processos X Recursos, mostrada na Figura 2.10. Para essa árvore serão obtidas as seguintes respostas:

Existe um possível deadlock quando o sistema atinge a configuração C_5 .

A sequência de eventos que leva o sistema a essa configuração é: $t_1[c_1]$, $t_4[c_4]$.

Existe um possível deadlock quando o sistema atinge a configuração C_6 .

A seqüência de eventos que leva o sistema a essa configuração é: $t_4[c_4]$, $t_1[c_1]$.

2) Para o exemplo mostrado nas Figuras 4.4 e 4.5 têm-se as respostas:

Existe um possível deadlock quando o sistema atinge a configuração C_6 .

A seqüência de eventos que leva o sistema a essa configuração é: e_2, e_3, e_4 .

Existe um possível deadlock quando o sistema atinge a configuração C_7 .

A seqüência de eventos que leva o sistema a essa configuração é: e_3, e_2, e_4 .

Figura 4.5: Árvore de Alcançabilidade para o Statechart da Figura 4.4

A diferença existente entre a análise feita pelo algoritmo, nesses dois exemplos ilustrados, é que no primeiro caso, quando o sistema atinge as configurações C_5 e C_6 , há realmente um deadlock, e no segundo caso, o fato do sistema alcançar as configurações C_6 e C_7 , significa o final de processamento do sistema.

Caberá à pessoa responsável pela especificação do sistema analisar as respostas dada pelo algoritmo e verificar se o deadlock realmente existe.

Algoritmo para verificar a existência de deadlock (VPDS/DEAD):

```
1 Algoritmo VPDS/DEAD
2 INÍCIO
3   encontrou deadlock = falso;
4   nó = raiz-árvore;
5   ENQUANTO existir nó faça
6     SE nó é terminal
7       ENTÃO
8         Imprimir ('Existe um possível deadlock quando o sistema atinge a
9           configuração ', nó);
10        Mostrar seqüência de eventos;
11        encontrou deadlock = verdade;
12      FIM-SE;
13      Selecionar outro nó;
14    FIM-ENQUANTO;
15  SE não encontrou deadlock
16    ENTÃO
17      Imprimir ('o Statechart em questão não possui situações de deadlock');
18  FIM-SE;
19 FIM VPDS/DEAD.
```

Como foi visto no capítulo anterior, no momento da construção de árvore, todos os nós podem ser do tipo “novo”, “velho”, “terminal” ou “história”. Dada essa classificação, o algoritmo para verificar a existência de deadlock é trivial, somente faz uma busca em todos os nós da árvore, procurando os nós do tipo terminal.

A eficiência desse algoritmo ficará comprometida caso a árvore analisada seja muito grande, pois a busca é feita em todos os seus nós. Esse algoritmo possui complexidade n , onde n representa o número de configurações do Statechart.

4.5 Uso de Transições

A propriedade de uso de transições garante que o sistema foi especificado corretamente, no sentido de não haver transições especificadas que nunca disparam, ou ainda, que nenhuma transição foi esquecida no diagrama Statechart.

Se uma transição nunca dispara, ou não foi especificada no modelo em Statecharts, ela não aparece como um arco da árvore de alcançabilidade. Dessa forma, dada uma transição, o algoritmo percorre a árvore recursivamente, em busca de arcos rotulados

com o nome dessa transição. Se for encontrada é porque não foi esquecida, e em algum momento pode disparar, e portanto são fornecidas todas as configurações de estados das quais pode disparar. Caso contrário, haverá um aviso de que essa transição não foi especificada ou não é disparável. }

Exemplo:

No caso das Figuras 4.1 e 4.2 suponha que seja fornecida a transição *b* ao algoritmo. A resposta será:

A transição b pode disparar a partir da configuração: (0,(0,0(1,[((0,0),(0,0))]))).

A transição b pode disparar a partir da configuração: (1,(0,0(0,[((0,0),(0,0))]))).

Dada uma transição com o nome *d*, a resposta será:

A transição d não foi especificada ou não é disparável.

Ao observar o Statechart da Figura 4.1, ve-se que a transição *d* realmente não aparece nessa figura, isto é, não foi especificada.

Ao fornecer a transição *m*, a resposta também será:

A transição m não foi especificada ou não é disparável.

Mas observando novamente a Figura 4.1, pode-se verificar que a transição *m* nunca disparará. A condição $in(B)$ nunca será satisfeita, pois quando *m* estiver habilitada, o sistema estará em R, não podendo estar também em B (*M* é um estado com decomposição do tipo XOR, logo o sistema deverá estar em apenas um de seus subestados). Erros como este, comum na especificação de um sistema, poderão ser detectados pela propriedade de uso de transições.

Algoritmo para verificar o uso de transições (VPDS/USO-TRANS)

```

1 Algoritmo VPDS/USO-TRANS
2 INÍCIO
3   Obter nome transição;
4   encontrou = FALSE;
5   nó = raiz árvore;
6   ENQUANTO existir nó faça
7     ENQUANTO existir arco faça
8       SE nome do arco = nome transição
9         ENTÃO
10          Imprimir('A transição nome transição pode disparar
11            da configuração: ', nó);

```

```

12         encontrou = verdade;
13         FIM-SE;
14         Obter outro arco;
15     FIM-ENQUANTO;
16     Obter outro nó;
17 FIM-ENQUANTO;
18 SE não encontrou
19 ENTÃO
20     Imprimir ('A transição nome transição não foi especificada
21             ou não é disparável');
22 FIM VPDS/USO-TRANS.

```

Esse algoritmo tem complexidade n (onde n representa o número de configurações de estado do Statechart), pois percorre a árvore inteira em busca da transição fornecida.

4.6 Seqüência Válida de Eventos

[Diz-se que uma seqüência de eventos é válida se cada um dos eventos dessa seqüência causar uma mudança na configuração de estados do sistema modelado.

Essa propriedade tem como objetivo validar uma seqüência de eventos, isto é, dada qualquer seqüência de eventos, existe um algoritmo que percorre a árvore de alcançabilidade em busca de cada elemento dessa seqüência. A busca é feita em duas etapas: na primeira etapa obtém-se o primeiro elemento da seqüência fazendo uma busca recursiva na árvore, e caso esse elemento seja encontrado, a segunda etapa procurará os outros elementos na árvore. A segunda etapa será bem sucedida se, a partir do nó onde incide o arco cujo rótulo é o nome do primeiro elemento da seqüência, existe um caminho onde os eventos correspondem aos elementos restantes da seqüência dada. Se a segunda etapa da busca falhar, continua-se a buscar recursivamente o primeiro elemento da seqüência, passando para a segunda etapa, se for encontrado um novo arco com o seu nome. O algoritmo termina se conseguir validar a seqüência de eventos ou se chegar ao final do processamento recursivo.]

Os nós história, criados pelo algoritmo de construção da árvore, serão úteis para verificar se uma seqüência de eventos é válida. O algoritmo que valida uma seqüência de eventos deve guardar o caminho percorrido numa lista, pois quando um nó história for encontrado, essa lista auxiliará no momento de restaurar a configuração de estados representada pelo nó história. Essa lista será também importante no momento da escolha entre um arco com um determinado nome e o arco com a extensão “/def” nesse nome.

Isso tem a ver com o nó história, pois o evento que aparece nesses dois arcos pode levar a um nó história ou a uma configuração de estados "default", respectivamente, dependendo da informação contida na lista onde se encontra o caminho percorrido. Os exemplos mostrados abaixo esclarecem essas idéias.

Exemplos

Suponha o Statechart e sua árvore de alcançabilidade mostrados nas Figuras 4.1 e 4.2, respectivamente, então:

a) Seja a seqüência a, b, h, e_1, c .

Considerando os rótulos dos nós, o caminho percorrido na árvore para validar essa seqüência é:

$$C_1 \xrightarrow{a/def} C_3 \xrightarrow{b} C_7 \xrightarrow{h} C_8 \xrightarrow{e_1} C_1 \xrightarrow{c} C_5$$

A resposta é: *A seqüência dada é válida.*

b) Seja a seqüência $c, b, a, b, t, b, h, e_1, a, k$.

O caminho percorrido é:

$$C_1 \xrightarrow{c} C_5 \xrightarrow{b} C_1 \xrightarrow{a/def} C_3 \xrightarrow{b} C_7 \xrightarrow{t} C_2 \xrightarrow{b} C_7 \xrightarrow{h} C_8 \xrightarrow{e_1} C_1 \xrightarrow{a} C_8 \xrightarrow{k} C_{10}$$

A resposta é: *A seqüência dada é válida.*

Nesses exemplos, nota-se como é que a lista do caminho percorrido auxilia a restauração do nó história. Toda vez que um nó história é encontrado, basta verificar qual é o estado em que ele indica história, e a partir do último elemento da lista volta-se procurando um nó no qual aquele estado esteve ligado; o fato da busca ser feita do final até o início da lista garante que essa é a última configuração de estados visitada dentro do estado com história. No exemplo do item b), observando a árvore, quando o sistema está em C_2 e acontece o evento b , o nó alcançado é o nó C_6 que é um nó história. Ao observar esse nó, vê-se que o estado D possui história, então volta-se na lista do caminho buscando-se a última vez em que o sistema entrou em D , encontra-se o nó C_7 . O nó C_7 é inserido então na lista caminho e a partir daí continua-se a buscar os novos elementos da seqüência de eventos.

Os exemplos ilustram também como utilizar a lista do caminho percorrido para optar entre dois ramos da árvore quando existe um arco rotulado com "nome-evento"/def. Na seqüência do item (a), dado o evento a , encontra-se os arcos rotulados a e a/def . O arco a leva a um nó com história em S e o arco a/def representa a primeira vez em que o sistema entra em S , e os seus subestados "defaults" é que são ativados. Ao observar a lista-caminho, que neste momento está vazia, conclui-se que o sistema nunca

esteve em S, logo o arco rotulado a/def é escolhido. No exemplo do item (b), a segunda vez em que ocorre o evento a , ao verificar a lista-caminho, nota-se que o sistema já esteve em S, então o arco rotulado com a é escolhido; este arco leva a um nó história e deve ser substituído pela configuração de estados representada pelo nó C_8

Vale observar que o arco a/def será utilizado apenas uma vez pelo algoritmo. Ao percorrer a árvore a fim de validar uma seqüência de eventos, o fato de passar pelo arco a/def quer dizer que o sistema está visitando pela primeira vez o estado S, portanto, nesse momento, essa configuração é inserida na lista do caminho percorrido e assim a história desse estado é ativada.

c) Considere a Sequência a, b, h, e_1, a, j .

A lista caminho é:

$$C_1 \xrightarrow{a/def} C_3 \xrightarrow{b} C_7 \xrightarrow{h} C_8 \xrightarrow{e_1} C_1 \xrightarrow{a} C_8 \xrightarrow{j}$$

A resposta para essa seqüência é: *Seqüência inválida*.

Observando a árvore de alcançabilidade da Figura 4.2, nota-se que do nó C_8 não sai nenhum arco rotulado com o evento j . Isto quer dizer que se o sistema estiver na configuração de estados representados pelo nó C_8 , e o evento j ocorrer, o sistema não se alterará, e, portanto, a seqüência de eventos dada não é válida.

Algoritmo para validar uma seqüência de eventos (VPDS/SEQ-EV):

```

1 Algoritmo VPDS/SEQ-EV
2 INÍCIO
3   Obter seqüência de eventos;
4   Inicializar lista-caminho;
5   Encontrou = FALSO;
6   SE existir elemento na seqüência de eventos
7     ENTÃO
8     Buscar seqüência eventos(raiz-árvore, encontrou, primeiro elemento
9       da seqüência);
10    SE não encontrou
11      ENTÃO
12        Imprimir('seqüência inválida');
13      FIM-SE;
14    FIM-SE;
15 FIM VPDS/SEQ-EV.
```

```

1  Buscar seqüência eventos(nó, encontrou, elemento da seqüência)
2  INÍCIO
3      SE existir arco a ser processado
4      ENTÃO
5          SE nesse nó existe arco rotulado com o nome = elemento da seqüência
6          ENTÃO
7              Encontrou = TRUE;
8              SE o arco possui a extensão “/def”
9              ENTÃO
10                 Verificar na lista-caminho se estado com história foi visitado;
11                 SE visitado
12                 ENTÃO
13                     Tomar o arco com rótulo “nome-evento”;
14                     Fazer nó-prossegue = elemento da lista onde estado com história
15                     foi visitado;
16                 SENÃO
17                     Tomar o arco com rótulo “nome-evento”/def;
18                     Fazer nó-prossegue = nó da árvore apontado por esse arco;
19                     SE nó prossegue = velho
20                     ENTÃO
21                         Fazer nó-prossegue = nó apontado pelo ponteiro de retorno;
22                     FIM-SE;
23                 FIM-SE;
24             SENÃO
25                 Tomar o arco com rótulo “nome-evento”;
26                 Fazer nó-prossegue = nó da árvore apontado por esse arco;
27                 SE nó prossegue = velho
28                 ENTÃO
29                     Fazer nó prossegue = nó apontado pelo ponteiro de retorno;
30                 FIM-SE;
31                 SE nó-prossegue = HISTÓRIA
32                 ENTÃO
33                     nó-prossegue = elemento da lista onde estado foi visitado;
34                 FIM-SE;
35             FIM-SE;
36         FIM-SE;
37     SE encontrou
38     ENTÃO
39         Inserir nó na lista-caminho;
40         Inserir nó-prossegue na lista caminho;

```

```

41      achou = VERDADE;
42      Buscar restante da seqüência de eventos(nó-prossegue, achou);
43      SE achou = VERDADE
44      ENTÃO
45          Imprimir ('A seqüência de eventos é válida');
46      SENÃO
47          encontrou = FALSO;
48          Inicializar-lista-caminho;
49      FIM-SE;
50  FIM-SE;
51  SE não encontrou
52  ENTÃO
53      Buscar seqüência de eventos (nó.prim-filho, encontrou, elemento
54      da seqüência);
55  FIM-SE;
56  SE não encontrou
57  ENTÃO
58      Buscar seqüência de eventos (nó.prox-irmão, encontrou, elemento da
59      seqüência);
60  FIM-SE;
61  FIM-SE;
62 FIM Buscar seqüência de eventos.

```

O tempo de execução desse algoritmo tende a aumentar consideravelmente com a complexidade do sistema e o tamanho da árvore de alcançabilidade gerada. Ao tentar validar uma seqüência de eventos, o algoritmo percorre a árvore de forma recursiva até encontrar o primeiro elemento da seqüência. Em seu pior caso, quando a seqüência é inválida, e esse primeiro elemento ocorre em vários arcos da árvore, toda vez que ele é encontrado, o algoritmo processa o restante da seqüência.

O algoritmo se torna ainda mais complexo pela necessidade de verificar, sempre, a extensão “/def” no nome dos arcos, e o fato de ter que restaurar os nós história fazendo uma busca na lista-caminho.

4.7 Considerações Finais

Este capítulo apresentou algumas propriedades dinâmicas de Statecharts, que podem ser analisadas através da árvore de alcançabilidade. Para cada uma das propriedades mostrou-se como utilizar a árvore para verificar essas propriedades, e como

foram desenvolvidos os seus respectivos algoritmos. Adicionalmente, discutiu-se a complexidade de cada um desses algoritmos.

Este capítulo mostrou também que a definição de uma árvore de alcançabilidade para Statecharts, e a sua utilização para verificar as propriedades dinâmicas de Statecharts se constituem em uma solução eficiente e elegante para resolver esse problema.

Capítulo 5

Ambiente STATSIM: Implementação da Árvore de Alcançabilidade e dos Algoritmos Verificadores de Propriedades Dinâmicas de Statecharts

5.1 Considerações Iniciais

Este capítulo tem como objetivo discutir os aspectos de implementação do algoritmo Construtor da Árvore de Alcançabilidade (CAA) e dos algoritmos Verificadores das Propriedades Dinâmicas de Statecharts (VPDS), e mostrar um exemplo de sua execução.

Inicialmente, na Seção 5.2, é apresentado o ambiente STATSIM, de modo a oferecer uma visão geral do sistema. Mostra-se também como os algoritmos desenvolvidos interagem com esse ambiente. A Seção 5.3 apresenta alguns aspectos da implementação dos algoritmos. Na Seção 5.4 são mostrados os resultados do uso dos algoritmos CCA e VPDS, utilizando-se como exemplo um modelo simplificado do mostrador de um relógio digital. Finalmente, na Seção 5.5, faz-se uma avaliação dos algoritmos desenvolvidos. Na seção final são apresentadas as conclusões gerais do capítulo.

5.2 O ambiente STATSIM

O ambiente STATSIM está dividido em duas partes principais: uma para receber a especificação de um Statechart e criar uma base de dados contendo a sua especificação interna, e a outra, para a simulação propriamente dita, conforme está esquematizado na Figura 5.1 [Masi91]. Esta figura mostra os componentes principais do ambiente STATSIM, que serão discutidos brevemente a seguir. Os detalhes desses componentes poderão ser encontrados em [Bati91] e [Fort91].

Figura 5.1: Arquitetura do Ambiente STATSIM

- **O Compilador LES**

Para a especificação dos Statecharts criaram-se uma Linguagem de Especificação de Statecharts (LES) e um analisador para ela. Como resultado da análise, quando não há erros, cria-se uma base de dados contendo a especificação interna do Statechart recebido como entrada.

- **O Editor Gráfico de Statecharts (EGS)**

O editor gráfico foi desenvolvido para oferecer uma alternativa visual para a criação dos Statecharts. Esse editor utiliza internamente o analisador da linguagem para reconhecer expressões e declarações. A base de dados gerada pelo editor gráfico é a mesma gerada pelo compilador da LES, a menos das informações gráficas. A sua característica principal é a de permitir a manipulação direta de objetos na tela, onde praticamente toda a edição é feita com a utilização do *mouse*, restringindo o uso do teclado apenas para declarações de informações textuais.

- **A Base de Dados**

Os estados, eventos e transições ficam armazenadas na memória principal durante a simulação. Existem opções para armazenar essas estruturas de dados em memória secundária e recuperá-las quando for conveniente.

- **O Simulador**

O simulador é um componente do sistema que aceita os eventos externos indicados pelo usuário do sistema e reage de acordo com a semântica dos Statecharts, reconfigurando-o a cada passo da simulação. A semântica utilizada para essa simulação foi definida formalmente por Harel em [Hare87b].

O conjunto de ferramentas que compõem o ambiente STATSIM foi desenvolvido em estações de trabalho SUN, utilizando-se a linguagem C e o *toolkit* para criação de interfaces Xview.

Os algoritmos Construtor da Árvore de Alcançabilidade (CAA) e Verificadores de Propriedades Dinâmicas de Statecharts (VPDS), embora ainda não estejam integrados ao ambiente STATSIM, utilizam seus módulos **Editor Gráfico de Statecharts (EGS)** e **Simulador de Especificações em Statecharts (SES)** para gerar a base de dados Statecharts e simular as especificações Statecharts. O módulo que faz a simulação é essencial para a construção da árvore de alcançabilidade, sem a qual seria impossível essa construção. A Figura 5.2 mostra como os algoritmos CAA e VPDS interagem com os demais módulos do ambiente STATSIM. Antes de executar o módulo CAA, deverá existir na base de dados a especificação de algum Statechart. Isso é feito usando-se o EGS, que ao terminar a edição de um Statechart, oferece a opção de armazená-lo em disco.

Figura 5.2: Interação dos Módulos CCA e VPDS com o Ambiente STATSIM

Terminada a especificação, ao ativar o módulo CAA, este carregará na memória as estruturas do Statechart escolhido para análise, e construirá a árvore de alcançabilidade, fazendo chamadas ao simulador, quando necessário. Após a árvore de alcançabilidade ter sido construída, pode-se ativar o módulo VPDS que iniciará o processo de análise das propriedades dinâmicas do Statechart.

5.3 Implementação

Assim como o ambiente STATSIM, o algoritmo construtor da árvore de alcançabilidade e os algoritmos verificadores de propriedades foram desenvolvidos em estações de trabalho SUN, utilizando-se a linguagem de programação C. Nessa primeira versão dos algoritmos, a interface usada é bastante simples, com menus de múltipla escolha e alguns diálogos sequenciais, mas o objetivo é integrar os algoritmos ao ambiente STATSIM, passando então a utilizar o padrão Xview de interfaces.

No que se refere ao volume de programas, os algoritmos foram desenvolvidos com um total de aproximadamente quarenta rotinas, somando em torno de duas mil linhas de código. Além disto, foram aproveitadas muitas das rotinas que estavam prontas no ambiente STATSIM.

Estrutura de dados da árvore de Alcançabilidade

A árvore de alcançabilidade para Statecharts, como pode ser vista em todos os exemplos fornecidos nos capítulos anteriores, possui um comportamento variável em relação aos seus nós e a quantidade de descendentes, ou seja, é uma árvore de n -vias, sendo n um número variável. Para ser possível sua implementação na linguagem C, onde é necessária a declaração de todas as estruturas de dados utilizadas, foi necessário transformar sua estrutura em uma árvore de 2-vias [Esak89]. A Figura 5.3 mostra como a árvore de alcançabilidade foi transformada em árvore de 2-vias.

Cada nó da árvore de alcançabilidade tem a seguinte estrutura de dados:

PAI	TIPO	PT-RETORNO
CONFIGURAÇÃO		
ARCO1		ARCO2
PRIM-FILHO		PROX-IRMÃO

onde:

- PAI é um ponteiro para o seu ancestral.
- TIPO é a classificação recebida pelo nó que pode ser NOVO, VELHO, HISTÓRIA ou TERMINAL.
- PT-RETORNO é um ponteiro para o nó da árvore onde aparece a mesma configuração existente no nó em questão, e de onde partem as próximas configurações alcançáveis a partir desse nó. Este campo é o responsável pelo mapeamento de várias configurações de estados em um único nó da árvore. Os nós com tipo VELHO como conteúdo possuem um PT-RETORNO diferente de nulo, que caracteriza configuração de estado do tipo VELHO, sendo nulo nos demais nós.
- CONFIGURAÇÃO é um campo formado por caracteres e descreve uma configuração de estados usando a notação apresentada no Capítulo 3.
- PRIM-FILHO e PROX-IRMÃO são ponteiros para o primeiro descendente e o próximo irmão do nó.
- ARCO1 e ARCO2 são os rótulos das transições associadas aos ponteiro PRIM-FILHO E PROX-IRMÃO, respectivamente. O rótulo de uma transição contém o nome do evento, condições e ações que fazem parte da transição. Em casos de transições ou estados associados a símbolos de história, a extensão "/def" aparece após o rótulo da transição, como foi discutido no Capítulo 3.

A interpretação dessa árvore de alcançabilidade de 2 vias é a seguinte: para obter os eventos que saem de um nó é necessário descer pelo seu ponteiro de primeiro filho e seguir a lista de próximos irmãos. Por exemplo, observando a árvore de alcançabilidade

representada na Figura 5.3, os eventos que saem do nó B são: c, d e e. No Apêndice A encontra-se a listagem de outras árvores de alcançabilidade de 2 vias, que foram geradas pelo algoritmo CCA.

A árvore de alcançabilidade é criada dinamicamente em memória principal pelo algoritmo de construção da árvore e todos os algoritmos que analisam as propriedades de Statecharts foram implementados sobre a estrutura da árvore de alcançabilidade.

Figura 5.3: Estrutura da Árvore de Alcançabilidade

5.4 Um exemplo de execução

Nesta seção apresenta-se a execução dos algoritmos CCA e VPDS, utilizando-se um exemplo real que é o mostrador de um relógio digital. A especificação desse exemplo em Statecharts aparece na Figura 5.4, extraído de [Hare88a]. A árvore de alcançabilidade gerada para esse exemplo será mostrada parcialmente nesta seção, ou seja, apenas as

configurações de estados será apresentada, a árvore de alcançabilidade completa poderá ser vista no Apêndice A. A forma de impressão da árvore, mostrada nesta seção, significa que os nós que estão abaixo de um determinado nó e identados mais à direita são filhos deste nó.

Figura 5.4: Statechart do Mostrador de um Relógio Digital

Alguns aspectos são interessantes de se observar no exemplo:

- as transições rotuladas **d [in(off)]**, **d [in(on)]** e **doismin [in(acertar)]**. A condição **[in(acertar)]** é sempre avaliada verdadeira para a transição **doismin [in(acertar)]**, pois sempre que essa transição estiver habilitada, o sistema estará em **acertar**. Logo essa transição aparecerá em seis arcos da árvore de alcançabilidade representando o fato de poder disparar a partir dos seis subestados do estado **acertar**. Já as transições **d [in(off)]** e **d [in(on)]** aparecerão em apenas um arco da árvore embora existam na árvore duas configurações onde os estados **reg** e **lap** estão ativos. Isto acontece porque as condições dessas transições são avaliadas verdadeiras em somente uma dessas configurações.
- a transição rotulada pelo evento **d** em **acerta-alarme**, que sai e retorna por história para esse mesmo estado. Essa transição, além de estar associada a um símbolo de história, tem como estado origem um superestado. Isso faz com que exista na árvore de alcançabilidade três arcos rotulados com essa transição, representando o seu disparo de cada um dos subestados de **acerta-alarme**, e cada um desses arcos chega em um nó história.
- a transição **a** com história associada ao estado **cronômetro**, que faz com que seja criado na árvore um arco com rótulo “a/def” e um nó história.

A seguir apresenta-se a execução comentada do mostrador do relógio digital. Os trechos abaixo, impressos com letra de impressora comum, são os originais impressos pelos algoritmos.

No primeiro passo da execução escolhe-se o arquivo onde contém a especificação do Statechart a ser analisado, que neste exemplo chama-se **mostrador**; em seguida a árvore de alcançabilidade é construída pelo algoritmo CCA.

Cada nó da árvore de alcançabilidade para esse exemplo tem a forma:

(data,hora,alarme,campinha,(hor,int-min,minut),([(lap,reg),(off,on)],zero),(ano,mes,dia,m,s,h))

Arvore de Alcancabilidade

```

(0100(000)([(00)(00)]0)(000000))
  (1000(000)([(00)(00)]0)(000000))
    (0100(000)([(00)(00)]0)(000000))
      (0010(000)([(00)(00)]0)(000000))
        (0001(000)([(00)(00)]0)(000000))
          (0000(000)([(00)(00)]1)(000000))
            (0100(000)([(00)(00)]0)(000000))
              (0000(000)([(01)(01)]0)(000000))
                (0000(000)([(10)(01)]0)(000000))
                  (0000(000)([(01)(01)]0)(000000))
                    (0100(000)([(00)(00)]0)(000000))
                      (0000(000)([(10)(10)]0)(000000))
                        (0100(000)([(00)(00)]0)(000000))
                          (0000(000)([(10)(01)]0)(000000))
                            (0100(000)([(00)(00)]0)(000000))
                              (0000(000)([(01)(10)]0)(000000))
                                (0000(000)([(00)(00)]1)(000000))
                                  (0000(000)([(10)(10)]0)(000000))
                                    (0100(000)([(00)(00)]0)(000000))
                                      (0000(000)([(01)(01)]0)(000000))
                                        (0000(000)([(HH)(HH)]H)(000000))
                                          (0000(001)([(00)(00)]0)(000000))
                                            (0000(hhh)([(00)(00)]0)(000000))
                                              (0000(010)([(00)(00)]0)(000000))
                                                (0000(hhh)([(00)(00)]0)(000000))
                                                  (0000(100)([(00)(00)]0)(000000))
                                                    (0000(hhh)([(00)(00)]0)(000000))
                                                      (0010(000)([(00)(00)]0)(000000))
                                                        (0010(000)([(00)(00)]0)(000000))
                                                          (0010(000)([(00)(00)]0)(000000))
                                                            (0010(000)([(00)(00)]0)(000000))
                                                              (0000(000)([(00)(00)]0)(000010))
                                                                (0000(000)([(00)(00)]0)(000100))
                                                                  (0000(000)([(00)(00)]0)(000001))
                                                                    (0000(000)([(00)(00)]0)(001000))

```

```

(0000(000)([(00)(00)]0)(010000))
(0000(000)([(00)(00)]0)(100000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))
(0100(000)([(00)(00)]0)(000000))

```

Após a árvore de alcançabilidade ter sido contruída, é apresentado ao usuário um menu de opções, onde se escolhe uma das propriedades dinâmicas a ser analisada. Mostra-se a seguir a escolha de cada uma das cinco opções apresentadas:

1) Alcançabilidade de uma configuração a partir da configuração inicial.

-
1. Verificar alcançabilidade de configuracoes de estados a partir da configuracao inicial
 2. Verificar alcançabilidade de configuracoes de estados a partir de uma configuracao qualquer
 3. Verificar se o Statechart possui situacoes de deadlock
 4. Verificar o disparo de transicoes
 5. Validar sequencias de eventos
 6. Sair

Entre com a sua escolha: 1

Entre com a configuracao:

ano

A configuracao <ano> e' alcancavel a partir da configuracao inicial
atraves da seguinte sequencia de eventos:

Evento: c

Evento: c
Evento: c
Evento: c
Evento: c
Evento: c

Entre com a configuracao:
minut

A configuracao <minut> e' alcancavel a partir da configuracao inicial
atraves da seguinte sequencia de eventos:

Evento: a
Evento: c

Entre com a configuracao:
lap,off

A configuracao <lap,off> e' alcancavel a partir da configuracao inicial
atraves da seguinte sequencia de eventos:

Evento: a
Evento: a
Evento: a/def
Evento: b
Evento: d
Evento: b

Entre com a configuracao:
reg,off

A configuracao <reg,off> e' alcancavel a partir da configuracao inicial
atraves da seguinte sequencia de eventos:

Evento: a
Evento: a
Evento: a/def
Evento: b
Evento: b

Entre com a configuracao:

data

A configuracao <data> e' alcancavel a partir da configuracao inicial
atraves da seguinte sequencia de eventos:

Evento: d

2) Alcancabilidade de uma configuração a partir de uma configuração qualquer

1. Verificar alcancabilidade de configuracoes de estados
a partir da configuracao inicial
2. Verificar alcancabilidade de configuracoes de estados
a partir de uma configuracao qualquer
3. Verificar se o Statechart possui situacoes de deadlock
4. Verificar o disparo de transicoes
5. Validar sequencias de eventos
6. Sair

Entre com a sua escolha: 2

Entre com a primeira configuracao:

hora

Entre com a segunda configuracao:

lap,off

A configuracao <lap,off> e alcancavel a partir de <hora>
atraves da seguinte sequencia de eventos:

Evento: a

Evento: a

Evento: a/def

Evento: b

Evento: d

Evento: b

Entre com a primeira configuracao:

reg,on

Entre com a segunda configuracao:

mes

A configuracao <mes> e alcancavel a partir de <reg,on>
atraves da seguinte sequencia de eventos:

Evento: d

Evento: a

Evento: c

Evento: c

Evento: c

Evento: c

Evento: c

Entre com a primeira configuracao:

int_min

Entre com a segunda configuracao:

alarme

A configuracao <alarme> e alcancavel a partir de <int_min>
atraves da seguinte sequencia de eventos:

Evento: c

Evento: c

Entre com a primeira configuracao:

alarme

Entre com a segunda configuracao:

ano

A configuracao <ano> e alcancavel a partir de <alarme>
atraves da seguinte sequencia de eventos:

Evento: a

Evento: a/def

Evento: a

Evento: c

Evento: c

Evento: c

Evento: c

Evento: c

Evento: c

3) Impasse (deadlock)

1. Verificar alcancabilidade de configuracoes de estados a partir da configuracao inicial
2. Verificar alcancabilidade de configuracoes de estados a partir de uma configuracao qualquer
3. Verificar se o Statechart possui situacoes de deadlock
4. Verificar o disparo de transicoes
5. Validar sequencias de eventos
6. Sair

Entre com a sua escolha: 3

O Statechart em questao nao possui situacoes de deadlock

4) Uso de transições

1. Verificar alcançabilidade de configurações de estados a partir da configuração inicial
2. Verificar alcançabilidade de configurações de estados a partir de uma configuração qualquer
3. Verificar se o Statechart possui situações de deadlock
4. Verificar o disparo de transições
5. Validar sequências de eventos
6. Sair

Entre com a sua escolha: 4

Entre com a transição a ser verificada:

d

A transição <d> pode disparar a partir da configuração <(0100(000)([(00)(00)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(1000(000)([(00)(00)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(0000(000)([(01)(01)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(0000(000)([(01)(10)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(0000(001)([(00)(00)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(0000(010)([(00)(00)]0)(000000))>

A transição <d> pode disparar a partir da configuração <(0000(100)([(00)(00)]0)(000000))>

Entre com a transição a ser verificada:

a

A transicao <a> pode disparar a partir da configuracao <(0100(000)([(00)(00)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0010(000)([(00)(00)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0001(000)([(00)(00)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0000(000)([(00)(00)]1)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0000(000)([(01)(01)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0000(000)([(10)(01)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0000(000)([(10)(10)]0)(000000))>

A transicao <a> pode disparar a partir da configuracao <(0000(000)([(01)(10)]0)(000000))>

Entre com a transicao a ser verificada:
c

A transicao <c> pode disparar a partir da configuracao <(0100(000)([(00)(00)]0)(000000))>

A transicao <c> pode disparar a partir da configuracao <(0010(000)([(00)(00)]0)(000000))>

A transicao <c> pode disparar a partir da configuracao <(0000(001)([(00)(00)]0)(000000))>

A transicao <c> pode disparar a partir da

configuracao <(0000(010)([(00)(00)]0)(000000))>

A transicao <c> pode disparar a partir da
configuracao <(0000(100)([(00)(00)]0)(000000))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(000010))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(000100))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(000001))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(001000))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(010000))>

A transicao <c> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(100000))>

Entre com a transicao a ser verificada:
d [in(off)]

A transicao <d [in(off)]> pode disparar a partir da
configuracao <(0000(000)([(01)(10)]0)(000000))>

Entre com a transicao a ser verificada:
doismin [in(acertar)]

A transicao <doismin [in(acertar)]> pode disparar a partir da
configuracao <(0000(000)([(00)(00)]0)(000010))>

A transicao <doismin [in(acertar)]> pode disparar a partir da

configuracao <(0000(000)([(00)(00)]0)(000100))>

A transicao <doismin [in(acertar)]> pode disparar a partir da configuracao <(0000(000)([(00)(00)]0)(000001))>

A transicao <doismin [in(acertar)]> pode disparar a partir da configuracao <(0000(000)([(00)(00)]0)(001000))>

A transicao <doismin [in(acertar)]> pode disparar a partir da configuracao <(0000(000)([(00)(00)]0)(010000))>

A transicao <doismin [in(acertar)]> pode disparar a partir da configuracao <(0000(000)([(00)(00)]0)(100000))>

5) Sequência válida de eventos

1. Verificar alcançabilidade de configuracoes de estados a partir da configuracao inicial
2. Verificar alcançabilidade de configuracoes de estados a partir de uma configuracao qualquer
3. Verificar se o Statechart possui situacoes de deadlock
4. Verificar o disparo de transicoes
5. Validar sequencias de eventos
6. Sair

Entre com a sua escolha: 5

entre com a sequencia de eventos:
a,a,a,b,a,c,d

Sequencia invalida

entre com a sequencia de eventos:
a,b,c

Sequencia invalida

entre com a sequencia de eventos:

a,b,d,b,a,a

A sequencia de eventos dada e' valida !

entre com a sequencia de eventos:

a,c,c,c,d,b

A sequencia de eventos dada e' valida !

5.5 Avaliação dos algoritmos desenvolvidos

Os algoritmos desenvolvidos foram avaliados em aproximadamente uma dezena de Statecharts, tendo sido testados todos os casos discutidos anteriormente, onde aparecem na árvore nós história, nós terminais, arcos com extensão “/def”, etc. Os exemplos que foram testados e analisados são relativamente pequenos, possuindo em média 15 estados, 20 transições e 2 transições associadas a símbolos de história. As árvores de alcançabilidade geradas para esses exemplos têm em média 25 nós. O tempo observado para construção da árvore para cada um dos exemplos testados é pouco significativo, e o tempo de resposta para os vários algoritmos verificadores de propriedades é bastante rápido.

O exemplo mostrado na seção anterior foi o maior dos exemplos testados. Esse exemplo possui 26 estados, 27 transições, sendo duas associadas a símbolos de história. A árvore de alcançabilidade gerada para esse exemplo possui 50 nós. Assim como nos casos anteriores, o tempo de execução observado para esse exemplo é pequeno.

As restrições da implementação dos algoritmos CAA e VPDS são as restrições atuais do ambiente STATSIM. Nesses ambiente, as estruturas de dados referentes aos estados e aos eventos foram implementadas com estruturas associadas a vetores de

ponteiros, que atualmente possuem alocação estática, limitados a certos valores constantes, que podem ser aumentados desde que o software seja recompilado. Atualmente, as constantes são:

- Nro. máximo de estados = 100;
- Nro. máximo de transições = 100;
- Nro. máximo de eventos = 100;

Uma questão a ser considerada é o crescimento exponencial de uma árvore de alcançabilidade, dependendo da complexidade do sistema envolvido. Em Redes de Petri esse aspecto tem sido discutido ao se utilizar as técnicas gráficas, baseadas na árvore de alcançabilidade, para a verificação de sistemas complexos. São sugeridos alguns métodos para gerenciar os limites dessa complexidade, que se resumem em fazer especificações e validações parciais do sistema [Boch80], [Leve87].

Em Statecharts existe também esse problema, pois para a construção da árvore de alcançabilidade todas as transições possíveis são consideradas, e para um sistema complexo, o número de possibilidades geradas pode ser extremamente grande. Um Statechart com muitos componentes ortogonais, cada um com muitos estados tem um grande número de configurações de estados, o que torna impraticável a construção de sua árvore de alcançabilidade. Uma maneira de resolver esse problema é limitar o escopo dos testes, isto é, construir a árvore de alcançabilidade para as partes críticas e isoladas do sistema [i-LO89]. Quando partes maiores do sistema necessitam ser testadas, uma possível solução seria eliminar algumas transições e assim a árvore de alcançabilidade gerada teria um número menor de configurações. É importante ressaltar que essas restrições de limites do sistema devem ser feitas com muito cuidado, a fim de garantirem que os testes sejam realizados sem perda de informações e de forma eficiente.

Para um sistema complexo que pode ser modelado em vários níveis hierárquicos de detalhes, acredita-se que esse problema pode ser contornado. O algoritmo de construção da árvore de alcançabilidade poderá ser usado para construir uma árvore de alcançabilidade para cada um desses níveis hierárquicos, e a verificação desse sistema poderá ser feita parcialmente.

5.6 Considerações Finais

Este capítulo apresentou o ambiente STATSIM e mostrou como os módulos CCA e VPDS interagem com esse ambiente. Discutiu-se também alguns aspectos de implementação do CCA e VPDS, e foi mostrado a execução desses módulos, utilizando-se um exemplo real.

Ficou claro que é fundamental a existência de uma rotina baseada na semântica dos Statecharts, para executar cada passo do statechart, e também de subsistemas de interface com o usuário, para receber do meio externo os statecharts e armazená-los internamente. Esses componentes foram fornecidos por ferramentas já disponíveis no ambiente STATSIM.

Capítulo 6

Conclusões

6.1 Considerações Iniciais

No Capítulo 2 deste trabalho foram apresentados os conceitos básicos de Statecharts, que é uma técnica adequada para o problema de especificação e validação do aspecto comportamental de Sistemas de Tempo-Real. Os Statecharts são uma forma de diagramas de transições de estados, e são uma alternativa vantajosa, em comparação com os DTEs convencionais, por terem várias extensões importantes, que facilitam e tornam mais concisas as especificações de sistemas complexos. Sua semântica formal faz com que seja possível a implementação de ferramentas com o objetivo de validar os sistemas modelados em Statecharts.

A revisão apresentada no Capítulo 2 mostrou também que o formalismo de Redes de Petri é adequado para a modelagem de Sistemas Reativos, apresentando muitos recursos para estudo de sistemas complexos, propiciando uma análise confiável desses sistemas. Nesse contexto, tem-se observado o aumento na utilização de Redes de Petri em ferramentas de apoio às tarefas de especificação, análise e testes desses sistemas.

Grande parte das ferramentas de testes que utiliza o formalismo de Redes de Petri é baseada em técnicas gráficas para a verificação de suas propriedades, em particular a análise gráfica feita através da árvore de alcançabilidade.

Através de estudos, verificou-se a possibilidade de se definir uma árvore de alcançabilidade para Statecharts, de forma semelhante à árvore de alcançabilidade de Redes de Petri. Assim, sobre essa árvore de alcançabilidade definida no Capítulo 3, foram desenvolvidos algoritmos para sua construção e para validação de especificações baseadas em Statecharts, verificando suas propriedades dinâmicas.

Os algoritmos desenvolvidos para a criação da árvore e verificação das pro-

priedades dinâmicas exigiram uma implementação trabalhosa, envolvendo situações ainda não discutidas na literatura, tais como o tratamento dado a história, hierarquia de estados, a forma de percorrer a árvore de alcançabilidade e a forma de restaurar os nós história. O uso dos conceitos e algoritmos apresentados foi ilustrado com um exemplo real do mostrador de um relógio digital, além de outros exemplos mostrados ao longo do trabalho.

Harel, em um artigo publicado recentemente [Hare92], analisa as técnicas, teorias e tecnologias que, em sua opinião, constituem o estado da arte atual no desenvolvimento de sistemas complexos. O trabalho comenta principalmente dois aspectos: a modelagem de sistemas e a análise dos modelos desenvolvidos. As técnicas de análise são classificadas de acordo com sua complexidade e também facilidade de obtenção:

1. Execução interativa e por processamento por lotes
2. Execução programada
3. Execução exaustiva
4. “Vigilantes” e verificação temporal
5. Geração de códigos

As duas primeiras possibilidades já estão disponíveis no ambiente STATSIM [Fort91] e a terceira encontra-se em fase inicial de desenvolvimento. Este trabalho dotou o ambiente de capacidade de execução exaustiva. “Vigilantes” são estados especiais acrescentados ao modelo para verificar situações indesejadas. Eles podem ser simulados por alcançabilidade. A disponibilidade de várias alternativas para análise de modelos, dentro do mesmo ambiente, é muito importante porque o projetista pode escolher a forma mais adequada e econômica para cada caso. Também, para maior segurança, pode ser conveniente fazer uso de mais de uma alternativa ao mesmo tempo.

6.2 Contribuições deste Trabalho

As soluções encontradas neste trabalho para a verificação das propriedades dinâmicas de Statecharts, embora relativamente simples, são de certa forma originais, pois não existe nada publicado até o momento na literatura mostrando efetivamente como validar sistemas especificados com Statecharts. O sistema STATEMATE faz isso, mas como já foi comentado, não há nada publicado a respeito dos algoritmos utilizados.

Este trabalho é um ponto de partida muito importante, porque muitos estudos podem ser feitos a partir de agora, aproveitando suas idéias e criando novos algoritmos para verificar outras propriedades que possam vir a ser definidas sobre a árvore de

alcançabilidade.

A existência desses algoritmos e sua integração ao ambiente STATSIM, contribuirá bastante para aumentar a funcionalidade desse ambiente, já que constituem uma nova alternativa para análise dos modelos de sistemas especificados em Statecharts, desenvolvidos dentro do ambiente.

6.3 Sugestões para Futuras Pesquisas

Em primeiro lugar pretende-se integrar esses algoritmos ao ambiente STATSIM, que se encontra em desenvolvimento no ICMSC-USP. Como foi dito anteriormente, esses algoritmos, estando incluído nesse ambiente, funcionarão como parte de uma ferramenta para validar as especificações de sistemas feitas em Statecharts. Para essa integração basta incluí-los no menu principal desse ambiente, criando-se uma nova opção que é a de "Verificação de Propriedades de Statecharts". Ao selecionar essa opção, escolhe-se um Statechart, e nesse momento sua árvore de alcançabilidade será gerada, e a partir de então pode-se começar a verificar suas propriedades dinâmicas.

Uma outra área de investigação a ser explorada seria o estudo detalhado da complexidade dos algoritmos desenvolvidos, e a ordem de crescimento da árvore de alcançabilidade gerada, em consequência da complexidade do sistema modelado.

Alguns estudos podem ser feitos a fim de definir outras propriedades sobre a árvore de alcançabilidade, como por exemplo as propriedades de limitabilidade, segurança e conservação; e ainda investigar se a árvore de alcançabilidade pode ser utilizada para se fazer análise de segurança/completeza, como quer Melhart [Melh89].

Finalmente, um estudo sugerido seria verificar a possibilidade de se evitar a criação do arco `/def` na árvore de alcançabilidade. Acredita-se que isso reduziria bastante o tempo de execução dos algoritmos.

O desenvolvimento do ambiente também deverá ser feito no sentido de desenvolver técnicas para geração de código a partir dos Statecharts (item 5 da classificação apresentada em 6.1). "Vigilantes" podem ser implementados ou simulados a partir da propriedade de alcançabilidade, mas outras propriedades temporais também merecem a atenção de novos estudos.

Bibliografia

- [Anac91] Anacleto, J. C. *Uma Ferramenta para Especificação e Análise de Sistemas Concorrentes*. Dissertação de Mestrado, ICMSC-USP, São Carlos, 1991.
- [Bati91] Batista Neto, J. E. S. *Um Editor Gráfico para Statecharts*. Dissertação de Mestrado, ICMSC-USP, São Carlos, 1991.
- [Boch80] Bochmann, G.; Sunshine, C.A. Formal Methods in Communication Protocol Design. *IEEE Trans. Comm.*, vol. COM-28(4), pp. 624-631, 1980.
- [Brag82] Braga Filho, L. J. *Redes de Petri em Especificação e Verificação de Protocolos de Redes de Computadores*. Notas Técnicas-Viçosa, CPD/UFV, pp. 14-41, julho-agosto 1982.
- [Esak89] Esakov, J.; Weiss, T. *Data Structures: an advanced approach using C*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1989.
- [Fort91] 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, São Carlos, 1991.
- [Hare87a] Harel, D. STATECHARTS: a visual formalism for complex systems. *Science of Comp. Programming*, 8, pp.231-274, 1987.
- [Hare87b] Harel, D. STATECHARTS: on the formal semantics of Statecharts. *IN Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*. Ithaca, New York, 1987.
- [Hare88a] Harel, D. et al STATEMATE: a working environment for the development of complex reactive systems. *IN Proceedings of the Tenth International Conference on Software Engineering*. (Singapore, April), Washington D.C., IEEE, 1988.
- [Hare88b] Harel, D. On visual formalisms. *CACM*, 31(5), pp. 514-530, 1988.

- [Hare92] Harel, D. Biting the Silver Bullet: toward a brighter future for system development. *IEEE Computer*, pp.08-20, January 1992.
- [i-LO89] i-LOGIX. *The STATEMATE Aproach to Complex Systems*. 1989.
- [Leve87] Leveson, N. G., Stolzy J. L. Safety Analysis using Petri nets. *IEEE Transactions on Software Engineering*, SE(13(3)), pp.386-397, 1987.
- [Masi91] Masiero, P.C.; Fortes, R.P. de M.; Batista Neto, J.E.S. Edição e Simulação do Aspecto Comportamental de Sistemas de Tempo-Real. *Anais do XVIII Seminário Integrado de Hardware e Software, SBC*, Santos-SP, pp.45-61, 1991.
- [Melh89] Melhart, B.E.; Leveson, N.G.; Jaffe, M.S. Analysis Capabilities for Requirements Specified in Statecharts. *Proceedings of fifth International Workshop on Software Specification and Design*. Pittsburg, Pennsylvania, USA, ACM Sigsoft Engineering Notes, 14(3), pp.100-102, 1989.
- [Mura84] Murata, T. Modeling and Analysis of Concurrent Systems. In: C.R. Vick, C.V. Ramamoorthy. *Handbook of Software Engineering*. Vam Nostrand Reinhold Company Inc., pp.39-63, 1984.
- [Pete81] Peterson, J. L. *Petri Net Theory and The Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

Apêndice A

Árvores de Alcançabilidade Geradas para os Exemplos da Dissertação

Aqui mostra-se a listagem completa das árvores de alcançabilidade que foram geradas pelo algoritmo CCA, e que aparecem como exemplos neste trabalho. As três primeiras árvores listadas se referem aos três primeiros exemplos mostrados no Capítulo 3, e a última representa a árvore de alcançabilidade completa do exemplo do mostrador de um relógio digital.

A.1 Árvore de Alcançabilidade gerada para o Statechart do Exemplo 1

Arvore de Alcançabilidade

Endereco: <108800> Configuracao <((0000(1[(00)(00)])))>

No retorno: <0> pai: <0>

Primeiro filho: <174576> proximo irmao: <0>

Arco 1: <d> Arco 2: <(null)>

Tipo no: <0>

Endereco: <174576> Configuracao <((0001(0[(00)(00)])))>

No retorno: <0> pai: <108800>

Primeiro filho: <175112> proximo irmao: <174728>

Arco 1: <e1> Arco 2: <a>
Tipo no: <0>

Endereco: <175112> Configuracao <((0010(0[(00)(00)])))>
No retorno: <0> pai: <174576>
Primeiro filho: <175920> proximo irmao: <0>
Arco 1: <f> Arco 2: <(null)>
Tipo no: <0>

Endereco: <175920> Configuracao <((0000(h[(hh)(hh)])))>
No retorno: <0> pai: <175112>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

Endereco: <174728> Configuracao <((0100(0[(00)(00)])))>
No retorno: <0> pai: <108800>
Primeiro filho: <225560> proximo irmao: <174992>
Arco 1: <c> Arco 2: <g>
Tipo no: <0>

Endereco: <225560> Configuracao <((1000(0[(00)(00)])))>
No retorno: <0> pai: <174728>
Primeiro filho: <176040> proximo irmao: <0>
Arco 1: <d> Arco 2: <(null)>
Tipo no: <0>

Endereco: <176040> Configuracao <((0000(H[(HH)(HH)])))>
No retorno: <0> pai: <225560>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

Endereco: <174992> Configuracao <((0000(0[(01)(01)])))>
No retorno: <0> pai: <108800>
Primeiro filho: <175296> proximo irmao: <0>
Arco 1: <d> Arco 2: <(null)>
Tipo no: <0>

Endereco: <175296> Configuracao <((0001(0[(00)(00)])))>
No retorno: <174576> pai: <174992>

Primeiro filho: <0> proximo irmao: <175496>
 Arco 1: <(null)> Arco 2: <a>
 Tipo no: <1>

Endereco: <175496> Configuracao <(0100(0[(00)(00)]))>
 No retorno: <174728> pai: <174992>
 Primeiro filho: <0> proximo irmao: <175648>
 Arco 1: <(null)> Arco 2: <h>
 Tipo no: <1>

Endereco: <175648> Configuracao <(0000(0[(10)(01)]))>
 No retorno: <0> pai: <174992>
 Primeiro filho: <158992> proximo irmao: <175784>
 Arco 1: <d> Arco 2: <k>
 Tipo no: <0>

Endereco: <158992> Configuracao <(0001(0[(00)(00)]))>
 No retorno: <174576> pai: <175648>
 Primeiro filho: <0> proximo irmao: <159192>
 Arco 1: <(null)> Arco 2: <a>
 Tipo no: <1>

Endereco: <159192> Configuracao <(0100(0[(00)(00)]))>
 No retorno: <174728> pai: <175648>
 Primeiro filho: <0> proximo irmao: <159344>
 Arco 1: <(null)> Arco 2: <i>
 Tipo no: <1>

Endereco: <159344> Configuracao <(0000(0[(01)(01)]))>
 No retorno: <174992> pai: <175648>
 Primeiro filho: <0> proximo irmao: <159480>
 Arco 1: <(null)> Arco 2: <k>
 Tipo no: <1>

Endereco: <159480> Configuracao <(0000(0[(10)(10)]))>
 No retorno: <0> pai: <175648>
 Primeiro filho: <299528> proximo irmao: <0>
 Arco 1: <d> Arco 2: <(null)>
 Tipo no: <0>

Endereco: <299528> Configuracao <(0001(0[(00)(00)]))>

No retorno: <174576> pai: <159480>
Primeiro filho: <0> proximo irmao: <160288>
Arco 1: <(null)> Arco 2: <a>
Tipo no: <1>

Endereco: <160288> Configuracao <(0100(0[(00)(00)]))>
No retorno: <174728> pai: <159480>
Primeiro filho: <0> proximo irmao: <160440>
Arco 1: <(null)> Arco 2: <i>
Tipo no: <1>

Endereco: <160440> Configuracao <(0000(0[(01)(10)]))>
No retorno: <175784> pai: <159480>
Primeiro filho: <0> proximo irmao: <160576>
Arco 1: <(null)> Arco 2: <j>
Tipo no: <1>

Endereco: <160576> Configuracao <(0000(0[(10)(01)]))>
No retorno: <175648> pai: <159480>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <175784> Configuracao <(0000(0[(01)(10)]))>
No retorno: <0> pai: <174992>
Primeiro filho: <159664> proximo irmao: <0>
Arco 1: <d> Arco 2: <(null)>
Tipo no: <0>

Endereco: <159664> Configuracao <(0001(0[(00)(00)]))>
No retorno: <174576> pai: <175784>
Primeiro filho: <0> proximo irmao: <159864>
Arco 1: <(null)> Arco 2: <a>
Tipo no: <1>

Endereco: <159864> Configuracao <(0100(0[(00)(00)]))>
No retorno: <174728> pai: <175784>
Primeiro filho: <0> proximo irmao: <160016>
Arco 1: <(null)> Arco 2: <h>
Tipo no: <1>

Endereco: <160016> Configuracao <(0000(0[(10)(10)]))>
No retorno: <159480> pai: <175784>
Primeiro filho: <0> proximo irmao: <160168>
Arco 1: <(null)> Arco 2: <j>
Tipo no: <1>

Endereco: <160168> Configuracao <(0000(0[(01)(01)]))>
No retorno: <174992> pai: <175784>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

A.2 Árvore de Alcançabilidade gerada para o Statechart do Exemplo 2

Arvore de Alcancabilidade

Endereco: <108800> Configuracao <(0(10(0[(00)(00)])))>
No retorno: <0> pai: <0>
Primeiro filho: <171032> proximo irmao: <0>
Arco 1: <a/def> Arco 2: <(null)>
Tipo no: <0>

Endereco: <171032> Configuracao <(0(00(1[(00)(00)])))>
No retorno: <0> pai: <108800>
Primeiro filho: <171640> proximo irmao: <171120>
Arco 1: Arco 2: <a>
Tipo no: <0>

Endereco: <171640> Configuracao <(0(00(0[(01)(01)])))>
No retorno: <0> pai: <171032>
Primeiro filho: <172504> proximo irmao: <171776>
Arco 1: <e1> Arco 2: <e1>
Tipo no: <0>

Endereco: <172504> Configuracao <(0(10(0[(00)(00)]))>
No retorno: <108800> pai: <171640>
Primeiro filho: <0> proximo irmao: <270624>
Arco 1: <(null)> Arco 2: <t>
Tipo no: <1>

Endereco: <270624> Configuracao <(1(00(0[(00)(00)]))>
No retorno: <171392> pai: <171640>
Primeiro filho: <0> proximo irmao: <270776>
Arco 1: <(null)> Arco 2: <h>
Tipo no: <1>

Endereco: <270776> Configuracao <(0(00(0[(10)(01)]))>
No retorno: <0> pai: <171640>
Primeiro filho: <159008> proximo irmao: <270912>
Arco 1: <e1> Arco 2: <k>
Tipo no: <0>

Endereco: <159008> Configuracao <(0(10(0[(00)(00)]))>
No retorno: <108800> pai: <270776>
Primeiro filho: <0> proximo irmao: <159224>
Arco 1: <(null)> Arco 2: <t>
Tipo no: <1>

Endereco: <159224> Configuracao <(1(00(0[(00)(00)]))>
No retorno: <171392> pai: <270776>
Primeiro filho: <0> proximo irmao: <159376>
Arco 1: <(null)> Arco 2: <i>
Tipo no: <1>

Endereco: <159376> Configuracao <(0(00(0[(01)(01)]))>
No retorno: <171640> pai: <270776>
Primeiro filho: <0> proximo irmao: <159512>
Arco 1: <(null)> Arco 2: <k>
Tipo no: <1>

Endereco: <159512> Configuracao <(0(00(0[(10)(10)]))>
No retorno: <0> pai: <270776>
Primeiro filho: <160384> proximo irmao: <0>
Arco 1: <e1> Arco 2: <(null)>

Tipo no: <0>

Endereco: <160384> Configuracao <(0(10(0[(00)(00)])))))>

No retorno: <108800> pai: <159512>

Primeiro filho: <0> proximo irmao: <160600>

Arco 1: <(null)> Arco 2: <t>

Tipo no: <1>

Endereco: <160600> Configuracao <(1(00(0[(00)(00)])))))>

No retorno: <171392> pai: <159512>

Primeiro filho: <0> proximo irmao: <335224>

Arco 1: <(null)> Arco 2: <i>

Tipo no: <1>

Endereco: <335224> Configuracao <(0(00(0[(01)(10)])))))>

No retorno: <270912> pai: <159512>

Primeiro filho: <0> proximo irmao: <335360>

Arco 1: <(null)> Arco 2: <j>

Tipo no: <1>

Endereco: <335360> Configuracao <(0(00(0[(10)(01)])))))>

No retorno: <270776> pai: <159512>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <270912> Configuracao <(0(00(0[(01)(10)])))))>

No retorno: <0> pai: <171640>

Primeiro filho: <159696> proximo irmao: <0>

Arco 1: <e1> Arco 2: <(null)>

Tipo no: <0>

Endereco: <159696> Configuracao <(0(10(0[(00)(00)])))))>

No retorno: <108800> pai: <270912>

Primeiro filho: <0> proximo irmao: <159912>

Arco 1: <(null)> Arco 2: <t>

Tipo no: <1>

Endereco: <159912> Configuracao <(1(00(0[(00)(00)])))))>

No retorno: <171392> pai: <270912>

Primeiro filho: <0> proximo irmao: <160064>

Arco 1: <(null)> Arco 2: <h>
Tipo no: <1>

Endereco: <160064> Configuracao <(0(00(0[(10)(10)])))))>
No retorno: <159512> pai: <270912>
Primeiro filho: <0> proximo irmao: <160200>
Arco 1: <(null)> Arco 2: <j>
Tipo no: <1>

Endereco: <160200> Configuracao <(0(00(0[(01)(01)])))))>
No retorno: <171640> pai: <270912>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <171776> Configuracao <(0(10(0[(00)(00)])))))>
No retorno: <108800> pai: <171032>
Primeiro filho: <0> proximo irmao: <171928>
Arco 1: <(null)> Arco 2: <t>
Tipo no: <1>

Endereco: <171928> Configuracao <(1(00(0[(00)(00)])))))>
No retorno: <171392> pai: <171032>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <171120> Configuracao <(0(00(H[(HH)(HH)])))))>
No retorno: <0> pai: <108800>
Primeiro filho: <0> proximo irmao: <171256>
Arco 1: <(null)> Arco 2: <c ^ e1>
Tipo no: <2>

Endereco: <171256> Configuracao <(0(01(0[(00)(00)])))))>
No retorno: <0> pai: <108800>
Primeiro filho: <172064> proximo irmao: <171392>
Arco 1: Arco 2: <t>
Tipo no: <0>

Endereco: <172064> Configuracao <(0(10(0[(00)(00)])))))>
No retorno: <108800> pai: <171256>

Primeiro filho: <0> proximo irmao: <172184>
Arco 1: <(null)> Arco 2: <t>
Tipo no: <1>

Endereco: <172184> Configuracao <(1(00(0[(00)(00)]))>
No retorno: <171392> pai: <171256>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <171392> Configuracao <(1(00(0[(00)(00)]))>
No retorno: <0> pai: <108800>
Primeiro filho: <172336> proximo irmao: <0>
Arco 1: Arco 2: <(null)>
Tipo no: <0>

Endereco: <172336> Configuracao <(0(HH(H[(HH)(HH)]))>
No retorno: <0> pai: <171392>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

A.3 Árvore de Alcançabilidade gerada para o Statechart do Exemplo 3

Arvore de Alcançabilidade

Endereco: <108800> Configuracao <((00)1[(00)(00)]>
No retorno: <0> pai: <0>
Primeiro filho: <220408> proximo irmao: <0>
Arco 1: <a> Arco 2: <(null)>
Tipo no: <0>

Endereco: <220408> Configuracao <((00)0[(10)(10)]>
No retorno: <0> pai: <108800>

Primeiro filho: <212920> proximo irmao: <220560>
Arco 1: <j> Arco 2: <c/def>
Tipo no: <0>

Endereco: <212920> Configuracao <((00)0[(01)(10)])>
No retorno: <0> pai: <220408>
Primeiro filho: <253544> proximo irmao: <213056>
Arco 1: <i> Arco 2: <k>
Tipo no: <0>

Endereco: <253544> Configuracao <((00)0[(10)(10)])>
No retorno: <220408> pai: <212920>
Primeiro filho: <0> proximo irmao: <253664>
Arco 1: <(null)> Arco 2: <k>
Tipo no: <1>

Endereco: <253664> Configuracao <((00)0[(01)(01)])>
No retorno: <0> pai: <212920>
Primeiro filho: <297008> proximo irmao: <253848>
Arco 1: <i> Arco 2:
Tipo no: <0>

Endereco: <297008> Configuracao <((00)0[(10)(01)])>
No retorno: <213056> pai: <253664>
Primeiro filho: <0> proximo irmao: <297192>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <297192> Configuracao <((00)1[(00)(00)])>
No retorno: <108800> pai: <253664>
Primeiro filho: <0> proximo irmao: <297392>
Arco 1: <(null)> Arco 2: <d/def>
Tipo no: <1>

Endereco: <297392> Configuracao <((01)0[(00)(00)])>
No retorno: <220560> pai: <253664>
Primeiro filho: <0> proximo irmao: <297496>
Arco 1: <(null)> Arco 2: <d>
Tipo no: <1>

Endereco: <297496> Configuracao <((hh)0[(00)(00)])>

No retorno: <0> pai: <253664>
 Primeiro filho: <0> proximo irmao: <297632>
 Arco 1: <(null)> Arco 2: <1>
 Tipo no: <2>

Endereco: <297632> Configuracao <((00)0[(01)(10)])>
 No retorno: <212920> pai: <253664>
 Primeiro filho: <0> proximo irmao: <0>
 Arco 1: <(null)> Arco 2: <(null)>
 Tipo no: <1>

Endereco: <253848> Configuracao <((00)1[(00)(00)])>
 No retorno: <108800> pai: <212920>
 Primeiro filho: <0> proximo irmao: <254048>
 Arco 1: <(null)> Arco 2: <d/def>
 Tipo no: <1>

Endereco: <254048> Configuracao <((01)0[(00)(00)])>
 No retorno: <220560> pai: <212920>
 Primeiro filho: <0> proximo irmao: <254152>
 Arco 1: <(null)> Arco 2: <d>
 Tipo no: <1>

Endereco: <254152> Configuracao <((hh)0[(00)(00)])>
 No retorno: <0> pai: <212920>
 Primeiro filho: <0> proximo irmao: <0>
 Arco 1: <(null)> Arco 2: <(null)>
 Tipo no: <2>

Endereco: <213056> Configuracao <((00)0[(10)(01)])>
 No retorno: <0> pai: <220408>
 Primeiro filho: <270312> proximo irmao: <227152>
 Arco 1: <j> Arco 2:
 Tipo no: <0>

Endereco: <270312> Configuracao <((00)0[(01)(01)])>
 No retorno: <253664> pai: <213056>
 Primeiro filho: <0> proximo irmao: <270480>
 Arco 1: <(null)> Arco 2:
 Tipo no: <1>

Endereco: <270480> Configuracao <((00)1[(00)(00)])>
No retorno: <108800> pai: <213056>
Primeiro filho: <0> proximo irmao: <270680>
Arco 1: <(null)> Arco 2: <d/def>
Tipo no: <1>

Endereco: <270680> Configuracao <((01)0[(00)(00)])>
No retorno: <220560> pai: <213056>
Primeiro filho: <0> proximo irmao: <270784>
Arco 1: <(null)> Arco 2: <d>
Tipo no: <1>

Endereco: <270784> Configuracao <((hh)0[(00)(00)])>
No retorno: <0> pai: <213056>
Primeiro filho: <0> proximo irmao: <270920>
Arco 1: <(null)> Arco 2: <l>
Tipo no: <2>

Endereco: <270920> Configuracao <((00)0[(10)(10)])>
No retorno: <220408> pai: <213056>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <227152> Configuracao <((00)1[(00)(00)])>
No retorno: <108800> pai: <220408>
Primeiro filho: <0> proximo irmao: <227336>
Arco 1: <(null)> Arco 2: <d/def>
Tipo no: <1>

Endereco: <227336> Configuracao <((01)0[(00)(00)])>
No retorno: <220560> pai: <220408>
Primeiro filho: <0> proximo irmao: <227440>
Arco 1: <(null)> Arco 2: <d>
Tipo no: <1>

Endereco: <227440> Configuracao <((hh)0[(00)(00)])>
No retorno: <0> pai: <220408>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

Endereco: <220560> Configuracao <((01)0[(00)(00)])>
No retorno: <0> pai: <108800>
Primeiro filho: <237184> proximo irmao: <220648>
Arco 1: <f> Arco 2: <c>
Tipo no: <0>

Endereco: <237184> Configuracao <((10)0[(00)(00)])>
No retorno: <0> pai: <220560>
Primeiro filho: <280648> proximo irmao: <237384>
Arco 1: <g> Arco 2: <e1>
Tipo no: <0>

Endereco: <280648> Configuracao <((01)0[(00)(00)])>
No retorno: <220560> pai: <237184>
Primeiro filho: <0> proximo irmao: <280848>
Arco 1: <(null)> Arco 2: <e1>
Tipo no: <1>

Endereco: <280848> Configuracao <((00)0[(10)(10)])>
No retorno: <220408> pai: <237184>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <237384> Configuracao <((00)0[(10)(10)])>
No retorno: <220408> pai: <220560>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <220648> Configuracao <((hh)0[(00)(00)])>
No retorno: <0> pai: <108800>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

A.4 Árvore de Alcançabilidade gerada para o State-chart do Exemplo do Mostrador de um Relógio Digital

Arvore de Alcançabilidade

Endereco: <108800> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <0> pai: <0>

Primeiro filho: <158856> proximo irmao: <0>

Arco 1: <d> Arco 2: <(null)>

Tipo no: <0>

Endereco: <158856> Configuracao <(1000(000)([(00)(00)]0)(000000))>

No retorno: <0> pai: <108800>

Primeiro filho: <159288> proximo irmao: <159000>

Arco 1: <d> Arco 2: <a>

Tipo no: <0>

Endereco: <159288> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <108800> pai: <158856>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <159000> Configuracao <(0010(000)([(00)(00)]0)(000000))>

No retorno: <0> pai: <108800>

Primeiro filho: <159400> proximo irmao: <159160>

Arco 1: <a> Arco 2: <c>

Tipo no: <0>

Endereco: <159400> Configuracao <(0001(000)([(00)(00)]0)(000000))>

No retorno: <0> pai: <159000>

Primeiro filho: <160136> proximo irmao: <159560>

Arco 1: <a/def> Arco 2: <c>

Tipo no: <0>

Endereco: <160136> Configuracao <(0000(000)([(00)(00)]1)(000000))>

No retorno: <0> pai: <159400>
Primeiro filho: <325200> proximo irmao: <160232>
Arco 1: <a> Arco 2: <a>
Tipo no: <0>

Endereco: <325200> Configuracao <(0100(000)((00)(00)]0)(000000))>
No retorno: <108800> pai: <160136>
Primeiro filho: <0> proximo irmao: <325456>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <325456> Configuracao <(0000(000)((01)(01)]0)(000000))>
No retorno: <0> pai: <160136>
Primeiro filho: <358512> proximo irmao: <0>
Arco 1: <d> Arco 2: <(null)>
Tipo no: <0>

Endereco: <358512> Configuracao <(0000(000)((10)(01)]0)(000000))>
No retorno: <0> pai: <325456>
Primeiro filho: <388728> proximo irmao: <358720>
Arco 1: <d [in(on)]> Arco 2: <a>
Tipo no: <0>

Endereco: <388728> Configuracao <(0000(000)((01)(01)]0)(000000))>
No retorno: <325456> pai: <358512>
Primeiro filho: <0> proximo irmao: <388944>
Arco 1: <(null)> Arco 2: <a>
Tipo no: <1>

Endereco: <388944> Configuracao <(0100(000)((00)(00)]0)(000000))>
No retorno: <108800> pai: <358512>
Primeiro filho: <0> proximo irmao: <389088>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <389088> Configuracao <(0000(000)((10)(10)]0)(000000))>
No retorno: <0> pai: <358512>
Primeiro filho: <422512> proximo irmao: <0>
Arco 1: <a> Arco 2: <(null)>
Tipo no: <0>

Endereco: <422512> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <389088>
Primeiro filho: <0> proximo irmao: <422656>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <422656> Configuracao <(0000(000)([(10)(01)]0)(000000))>
No retorno: <358512> pai: <389088>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <358720> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <325456>
Primeiro filho: <0> proximo irmao: <358864>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <358864> Configuracao <(0000(000)([(01)(10)]0)(000000))>
No retorno: <0> pai: <325456>
Primeiro filho: <402120> proximo irmao: <0>
Arco 1: <d [in(off)]> Arco 2: <(null)>
Tipo no: <0>

Endereco: <402120> Configuracao <(0000(000)([(00)(00)]1)(000000))>
No retorno: <160136> pai: <358864>
Primeiro filho: <0> proximo irmao: <402272>
Arco 1: <(null)> Arco 2: <d>
Tipo no: <1>

Endereco: <402272> Configuracao <(0000(000)([(10)(10)]0)(000000))>
No retorno: <389088> pai: <358864>
Primeiro filho: <0> proximo irmao: <402496>
Arco 1: <(null)> Arco 2: <a>
Tipo no: <1>

Endereco: <402496> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <358864>
Primeiro filho: <0> proximo irmao: <402640>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <402640> Configuracao <(0000(000)([(01)(01)]0)(000000))>
No retorno: <325456> pai: <358864>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <160232> Configuracao <(0000(000)([(HH)(HH)]H)(000000))>
No retorno: <0> pai: <159400>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <2>

Endereco: <159560> Configuracao <(0000(001)([(00)(00)]0)(000000))>
No retorno: <0> pai: <159000>
Primeiro filho: <160344> proximo irmao: <0>
Arco 1: <d> Arco 2: <(null)>
Tipo no: <0>

Endereco: <160344> Configuracao <(0000(hhh)([(00)(00)]0)(000000))>
No retorno: <0> pai: <159560>
Primeiro filho: <0> proximo irmao: <160488>
Arco 1: <(null)> Arco 2: <c>
Tipo no: <2>

Endereco: <160488> Configuracao <(0000(010)([(00)(00)]0)(000000))>
No retorno: <0> pai: <159560>
Primeiro filho: <335192> proximo irmao: <160632>
Arco 1: <d> Arco 2:
Tipo no: <0>

Endereco: <335192> Configuracao <(0000(hhh)([(00)(00)]0)(000000))>
No retorno: <0> pai: <160488>
Primeiro filho: <0> proximo irmao: <335336>
Arco 1: <(null)> Arco 2: <c>
Tipo no: <2>

Endereco: <335336> Configuracao <(0000(100)([(00)(00)]0)(000000))>
No retorno: <0> pai: <160488>
Primeiro filho: <368600> proximo irmao: <335480>
Arco 1: <d> Arco 2:

Tipo no: <0>

Endereco: <368600> Configuracao <(0000(hhh)([(00)(00)]0)(000000))>

No retorno: <0> pai: <335336>

Primeiro filho: <0> proximo irmao: <368760>

Arco 1: <(null)> Arco 2: <c>

Tipo no: <2>

Endereco: <368760> Configuracao <(0010(000)([(00)(00)]0)(000000))>

No retorno: <159000> pai: <335336>

Primeiro filho: <0> proximo irmao: <368904>

Arco 1: <(null)> Arco 2:

Tipo no: <1>

Endereco: <368904> Configuracao <(0010(000)([(00)(00)]0)(000000))>

No retorno: <159000> pai: <335336>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <335480> Configuracao <(0010(000)([(00)(00)]0)(000000))>

No retorno: <159000> pai: <160488>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <160632> Configuracao <(0010(000)([(00)(00)]0)(000000))>

No retorno: <159000> pai: <159560>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <159160> Configuracao <(0000(000)([(00)(00)]0)(000010))>

No retorno: <0> pai: <108800>

Primeiro filho: <159688> proximo irmao: <0>

Arco 1: <c> Arco 2: <(null)>

Tipo no: <0>

Endereco: <159688> Configuracao <(0000(000)([(00)(00)]0)(000100))>

No retorno: <0> pai: <159160>

Primeiro filho: <318352> proximo irmao: <159832>

Arco 1: <c> Arco 2:
Tipo no: <0>

Endereco: <318352> Configuracao <(0000(000)([(00)(00)]0)(000001))>
No retorno: <0> pai: <159688>
Primeiro filho: <345248> proximo irmao: <318496>
Arco 1: <c> Arco 2:
Tipo no: <0>

Endereco: <345248> Configuracao <(0000(000)([(00)(00)]0)(001000))>
No retorno: <0> pai: <318352>
Primeiro filho: <378672> proximo irmao: <345392>
Arco 1: <c> Arco 2:
Tipo no: <0>

Endereco: <378672> Configuracao <(0000(000)([(00)(00)]0)(010000))>
No retorno: <0> pai: <345248>
Primeiro filho: <412392> proximo irmao: <378816>
Arco 1: <c> Arco 2:
Tipo no: <0>

Endereco: <412392> Configuracao <(0000(000)([(00)(00)]0)(100000))>
No retorno: <0> pai: <378672>
Primeiro filho: <432424> proximo irmao: <412536>
Arco 1: <c> Arco 2:
Tipo no: <0>

Endereco: <432424> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <412392>
Primeiro filho: <0> proximo irmao: <432552>
Arco 1: <(null)> Arco 2:
Tipo no: <1>

Endereco: <432552> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <412392>
Primeiro filho: <0> proximo irmao: <432696>
Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>
Tipo no: <1>

Endereco: <432696> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <412392>

Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <412536> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <378672>
Primeiro filho: <0> proximo irmao: <412680>
Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>
Tipo no: <1>

Endereco: <412680> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <378672>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <378816> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <345248>
Primeiro filho: <0> proximo irmao: <378960>
Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>
Tipo no: <1>

Endereco: <378960> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <345248>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <345392> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <318352>
Primeiro filho: <0> proximo irmao: <345536>
Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>
Tipo no: <1>

Endereco: <345536> Configuracao <(0100(000)([(00)(00)]0)(000000))>
No retorno: <108800> pai: <318352>
Primeiro filho: <0> proximo irmao: <0>
Arco 1: <(null)> Arco 2: <(null)>
Tipo no: <1>

Endereco: <318496> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <108800> pai: <159688>

Primeiro filho: <0> proximo irmao: <318640>

Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>

Tipo no: <1>

Endereco: <318640> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <108800> pai: <159688>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>

Endereco: <159832> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <108800> pai: <159160>

Primeiro filho: <0> proximo irmao: <159976>

Arco 1: <(null)> Arco 2: <doismin [in(acertar)]>

Tipo no: <1>

Endereco: <159976> Configuracao <(0100(000)([(00)(00)]0)(000000))>

No retorno: <108800> pai: <159160>

Primeiro filho: <0> proximo irmao: <0>

Arco 1: <(null)> Arco 2: <(null)>

Tipo no: <1>