2

Verificação de Modelos (*Model Checking*)

Em métodos formais, uma das abordagens que vem obtendo sucesso nos últimos anos é a de verificação de modelos (*model checking*), onde se verifica automaticamente a validade de propriedades acerca do comportamento de sistemas reativos[1]. Esta técnica verifica propriedades de um sistema através de enumeração exaustiva de todos os estados alcançáveis. Para realizar a validação das propriedades de sistemas reativos seguem-se os três passos abaixo:

1. Especificar quais são as propriedades que o sistema deverá ter para que seja considerado correto. Por exemplo, pode-se querer que o sistema nunca entre em *deadlock*, ou ainda, que ele sempre alcance um determinado estado.

2. O segundo passo é a construção do modelo formal do sistema. O modelo deve capturar todas as propriedades essenciais do sistema para verificar a correção do mesmo, contudo, também deverá possuir abstrações de detalhes do sistema que não afetem a correção das propriedades a serem verificadas. Por exemplo, em protocolos de comunicações se está interessado em testar propriedade de quando uma mensagem é recebida e não do conteúdo dela.

3. O terceiro e último passo é a própria execução do verificador de modelos para validar as propriedades especificadas do sistema. Neste passo, já têm-se as propriedades e o modelo. Assim, aplica-se o verificador e consegue-se garantir se o modelo do sistema possui ou não as propriedades desejadas. Caso todas as propriedades sejam verdadeiras, então o sistema está correto. Caso não obedeça a alguma propriedade, então

[1]Sistemas reativos têm como caracterização básica estados e transições. Estado é a descrição do sistema em um dado instante de tempo, ou seja, os valores associados as suas variáveis naquele instante. Transição é uma relação entre dois estados. Chama-se de computação uma seqüência infinita de estados onde cada estado é obtido através de um estado anterior e uma relação de transição entre eles.

é gerado um contra exemplo mostrando o porquê da não verificação da propriedade. Desta forma, pode-se detectar o erro e realizar a correção do modelo. Esse processo deve ser feito até que o sistema obedeça todas as propriedades. Deste modo, realiza-se um ajuste na especificação.

Atualmente, a verificação de modelos é muito empregada em informática na verificação formal de *software* e *hardware*[22][1] e existem duas abordagens para implementar verificação de modelos: a abordagem lógica e a abordagem que utiliza a teoria dos autômatos. Os verificadores de modelos SMV[10], PRISM[24], VIS[16], SVE[17] e o pacote MDG[19] utilizam a abordagem lógica, enquanto o SPIN[25], COSPAN[18] e FDR[23] usam a abordagem de teoria dos autômatos.

Como este trabalho utilizou o verificador de modelos SMV aqui estarão descritos a abordagem lógica (seção 2.1) e o verificador de modelos SMV (seção 2.2).

2.1 Abordagem Lógica

Na abordagem lógica, um sistema reativo será descrito através de um tipo de grafo de transição de estados, chamado de *estrutura de Kripke,* que captura a intuição do seu comportamento. Uma *estrutura de Kripke* é um conjunto de estados, um conjunto de transições entre estados e uma função que rotula cada estado com o conjunto de propriedades que são verdadeiras nele. Caminhos em *estrutura de Kripke* são computações em sistemas reativos. Deve-se ainda considerar que existe uma restrição na abordagem lógica: as relações de transições devem ser totais. As *estruturas de Kripke* são simples e suficientes para capturar os aspectos de comportamento dos sistemas reativos.

Definição 1 *Estrutura de Kripke para a Abordagem Lógica* $(\mu = (S, S_o, R, L))$:

- *um conjunto de estados* S;
- *um conjunto de estados iniciais* S_o*, onde* $S_o \subseteq S$;
- *uma relação de transição* $R \subseteq S \times S$*, onde* $\forall s \in S(\exists s'$ *tal que* $(s, s') \in R)$*, isto é,* R *é uma relação total;*
- *uma função de rótulos* $L : S \rightarrow \wp(\mathcal{V})$*, onde* $\mathcal{V}$ *é o conjunto de proposições atômicas.*

Definição 2 *Caminho em estrutura de Kripke para a Abordagem Lógica*

Um caminho em uma estrutura de Kripke μ *a partir de um estado* s *é uma seqüência infinita de estados* $\pi = s_0 s_1 s_2 ...$ *tal que* $s_0 = s$ *e* $\forall i \in \mathbb{N}[(s_i, s_{i+1}) \in R]$.

As propriedades a serem verificadas em um sistema reativo são dadas através de fórmulas que pertencem a uma linguagem lógica temporal que especificam os comportamentos desejados.

As lógicas podem ser dividas em lógica temporal de tempo ramificado e em lógica temporal de tempo linear. Tais lógicas se referem a noção de seqüências de estados que descrevem possíveis computações do sistema e não a valores de tempos ou a intervalos de tempos; uma vez que se deseja tratar de comportamentos de sistemas não-determinísticos que envolvem diferentes caminhos. Isto é, ou cada estado pode ter vários sucessores em termos de ramificação, ou o comportamento é dado por um conjunto de caminhos que são lineares.

Em lógica de tempo ramificado a multiplicidade de comportamentos pode ser especificada explicitamente por uma propriedade de todos os próximos estados ou por uma propriedade do próximo estado, enquanto em lógica de tempo linear, esta multiplicidade é expressada de forma implícita. Uma fórmula será satisfeita em uma *estrutura de Kripke* se todos os caminhos que são modelos da estrutura satisfazem a fórmula.

Sejam μ uma *estrutura de Kripke,* $s \in S$ em μ, e φ uma fórmula lógica temporal. Então uma formulação geral para o problema de verificação de modelos é introduzido como:

$$\mu, s \models \varphi$$

Se o estado s for um estado inicial ($s_o \in S_o$), então verifica-se se μ é um modelo da propriedade φ.

A verificação de modelos depende da lógica utilizada para expressar as propriedades. Assim, cada abordagem requer seu próprio algoritmo. Em lógica de tempo ramificado, as propriedades são relacionadas com um conjunto de estados. Como em verificação de modelos os estados são finitos, então a computação de ponto fixo pode ser utilizada de forma eficiente para explorar o conjunto de estados de sistema de transição finito. Em lógica de tempo linear, as fórmulas podem ser representada em termos de um *tableau*, que é uma *estrutura especial de Kripke*[25].

2.2 *Symbolic Model Verifier* (SMV)

O *Symbolic Model Verifier* (SMV)[11] é uma ferramenta para verificar propriedades em sistemas de estados finitos que trata o problema de verificação de modelos com a abordagem lógica. Assim, o SMV possui uma linguagem para especificação de modelos para descrever os sistemas de estados finitos, que será apresentado na seção 2.2.1, e possui uma lógica temporal CTL para expressar as propriedades a serem verificadas, que será mostrada na seção 2.2.2.

O SMV utiliza algoritmos simbólicos baseados em OBDD (*Ordered Binary Decision Diagram*) para determinar se uma especificação expressa em CTL é satisfeita ou não. Com estes algoritmos já se conseguiu verificar propriedades em alguns exemplos com mais de 10^{120} estados[22][12].

2.2.1 A Linguagem de Especificação de Modelos do SMV

Nesta seção, a linguagem de especificação de modelos será apresentada de forma resumida. Assim, serão descritos as principais características da linguagem e será mostrado um exemplo. A sintaxe e semântica da linguagem podem ser encontradas em [11].

As principais características da linguagem de especificação de modelos do SMV são:

- Módulos – O usuário pode decompor um sistema de estados finitos em módulos, que encapsulam uma coleção de declarações: "VAR" define as variáveis do módulo, que podem ser *booleanos*, conjunto enumerado de constantes ou instâncias de outros módulos; "INIT" inicializa as variáveis; "ASSIGN" define as relações de transições; "FAIRNESS" definem as *fairness constraints,* que são fórmulas em CTL[2]; "SPEC" especificações em CTL. Os módulos podem ser instanciados várias vezes, referenciar variáveis de outros módulos e possuem parâmetros que podem ser expressões, estados de outros módulos ou até mesmo instâncias de módulos.

- Sincronismo e *Interleaved* - módulos podem ser compostos de forma síncronas ou usando *interleaving*. Em composição síncrona, um passo corresponde a um passo de cada módulo. Em *interleaved*, cada passo corresponde a um passo de um único módulo. Se a palavra reservada "*process*" preceder uma instância de um módulo, *interleaving* é usado. Caso contrário, a composição síncrona é assumida. Cada processo tem uma variável *running* que indica se o processo está ativo ou não[3].

- Transições Não-Determinísticas – as transições de um estado em um modelo podem ser determinísticas ou não-determinísticas. Transição não-determinística é usada para descrever modelos mais abstratos onde certos detalhes são omitidos.

- Relações entre transições – as relações de transições de um módulo podem ser especificadas explicitamente em termos de relações binárias entre o atual e o próximo estado das variáveis, ou implicitamente como um conjunto de comandos de atribuições paralelas. Os comandos de atribuições paralelas definem o valor das variáveis no próximo estado em termos dos valores no estado atual e são definidos através da declaração "NEXT" para cada atribuição.

[2] *Fairness constraints* são condições que são inseridas para garantir justiça aos caminhos em CTL. Um exemplo simples é o de duas avenidas que se entroncam. Suponha que somente passem carros de uma das avenidas. Isto claramente não seria justo. Assim, deve-se garantir que os carros das duas avenidas possam passar no entroncamento.

[3] Se a condição de *running* for definido nas condições de *fairness* de um módulo significará que o módulo não poderá ficar indefinidamente sem ser executado.

Exemplo 1 *A figura 2.2.1.1 mostra como seria descrito na linguagem do SMV um programa que utiliza uma variável semáforo (semaforo) para implementar exclusão mútua entre dois processo assíncronos. Será definido um módulo usuário que terá uma variável estado que possui quatro estados: ocioso, o processo não quer entrar na região crítica; entrando, o processo quer entrar na região crítica; critica, o processo está utilizando a região crítica; e saindo, o processo não irá mais usar a região crítica. O módulo main terá uma variável semáforo, que será inicializada com 0, e os dois usuários. Como os processos são assíncronos os usuários serão definidos com a palavra "process".*

```
MODULE main
VAR
semaforo : boolean;
proc1 : process usuario;
proc2 : process usuario;
ASSIGN
init(semaforo) :=0;
SPEC
AG !(proc1.estado=critica  & proc2.estado=critica)

MODULE usuario
VAR
estado : {ocioso, entrando, critica, saindo};
ASSIGN
init(estado) := ocioso;
next(estado) :=
        case
        estado = ocioso : {ocioso,entrando};
        estado = entrando & !semaforo :critica;
        estado = critica : {critica, saindo};
        estado = saindo : ocioso;
        1 : estado;
        esac;
next(semaforo) :=
        case
        estado = entrando : 1;
        estado = saindo :0;
        1 : semaforo;
        esac;
FAIRNESS
running
```

Figura 2.2.1.1 - Exemplo da linguagem de especificação de modelos do SMV.

2.2.2 Computation Tree Logic (CTL)

Computation Tree Logic (CTL)[7] é a lógica temporal de tempo ramificado que o verificador de modelos SMV utiliza para verificar propriedades em sistemas de estados finitos. A sintaxe e semântica de CTL estarão apresentadas a seguir.

Definição 3 *Sintaxe de CTL:*
$\Phi :== true \mid false \mid \mathcal{V} \mid (\neg\Phi_1) \mid (\Phi_1 \wedge \Phi_2) \mid (\Phi_1 \vee \Phi_2) \mid (\Phi_1 \rightarrow \Phi_2) \mid [EX]\Phi_1 \mid [EG]\Phi_1 \mid [EF]\Phi_1 \mid E(\Phi_1 \mathcal{U}\ \Phi_2) \mid [AX]\Phi_1 \mid [AG]\Phi_1 \mid [AF]\Phi_1 \mid A(\Phi_1 \mathcal{U}\ \Phi_2)$, *onde* $\mathcal{V}$ *é uma proposição.*

A intuição de cada operador temporal de CTL é apresentado a seguir:

- EX g - Se existe um caminho tal que no próximo estado a fórmula g é verdade (figura 2.2.2.1);
- AX g - Se em todos os caminhos a fórmula g é verdadeira no próximo estado (figura 2.2.2.2);
- EF g - Se existe um estado em um caminho tal que a fórmula g é verdade (figura 2.2.2.3);
- AF g - Se existe um estado em todos os caminhos tal que a fórmula g é verdade (figura 2.2.2.4);
- EG g - Se existe um caminho tal que em todo estado deste a fórmula g é verdade (figura 2.2.2.5);
- AG g - Se em todo estado de todo caminho a fórmula g é verdade (figura 2.2.2.6).
- E($f\ \mathcal{U}\ g$) - Se existe um caminho tal que existe um estado k onde a fórmula g é verdadeira, então em todos os outros estados anteriores do caminho a fórmula f é verdadeira (figura 2.2.2.7);
- A($f\ \mathcal{U}\ g$) - Se em todo caminho tal que existe um estado k tal que a fórmula g é verdadeira, então em todos os outros estados anteriores do caminho a fórmula f é verdadeira (figura 2.2.2.8);

Os seguintes operadores podem ser definidos em termos de $[EX]$, $[EG]$, EU:

- $[AX]\Phi = \neg EX(\neg\Phi)$
- $[EF]\Phi = E(true \mathcal{U} \Phi)$
- $[AG]\Phi = \neg EF(\neg\Phi)$
- $[AF]\Phi = \neg EG(\neg\Phi)$
- $A(\Phi_1 \mathcal{U} \Phi_2) = \neg E((\neg\Phi_2)\mathcal{U}(\Phi_1 \wedge \Phi_2)) \wedge \neg EG(\neg\Phi_2)$

Definição 4 *Semântica de CTL:*

Seja uma estrutura de $\mu = (S, S_o, R, L)$ *e um estado* $s \in S$. *Define-se a noção de satisfação* ($\models$), *como:*

- $\mu, s \models P \iff P \in L(e)$;
- $\mu, s \models (\neg\alpha) \iff N\tilde{A}O\ \mu, s \models \alpha$;
- $\mu, s \models (\alpha \rightarrow \beta) \iff SE\ \mu, s \models \alpha\ ENT\tilde{A}O\ \mu, s \models \beta$.
- $\mu, s \models [EX]\alpha \iff \exists s' \in S$ *tal que* $(s, s') \in R$ *e* $\mu, s' \models \alpha$;
- $\mu, s \models [EG]\alpha \iff$ *existe um caminho* $\pi = s_0 s_1 s_2 ...$, *partindo de* s, *tal que* $\forall k \geq 0 [\mu, s_k \models \alpha]$;
- $\mu, s \models \exists(\alpha\, \mathcal{U}\, \beta) \iff$ *existe um caminho* $\pi = s_0 s_1 s_2 ...$, *partindo de* s, *tal que* $\exists k \geq 0 [\mu, s_k \models \beta, \forall j [0 \leq j < k, \mu, s_j \models \alpha]]$;

Exemplo 2 *A figura 2.2.1.1 mostra a fórmula CTL* $AG!(proc1.estado = critica\ \&\ proc2.estado = critica)$ *que expressa que nenhum dos dois processos estão utilizando a região crítica ao mesmo instante. A fórmula CTL* $AG(proc1.estado = entrando -> AF(proc1.estado = critica))$ *significa que se um processo deseja entrar na região crítica em algum instante no futuro ele utilizará a região crítica.*

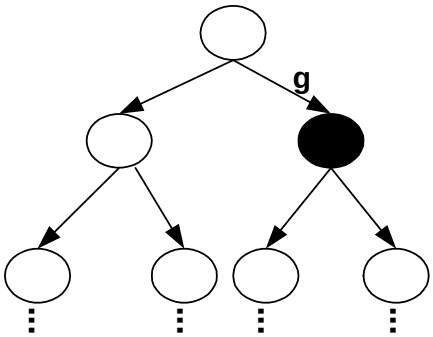

Figura 2.2.2.1 - EX g

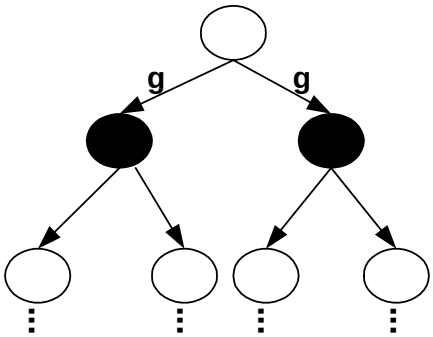

Figura 2.2.2.2 - AX g

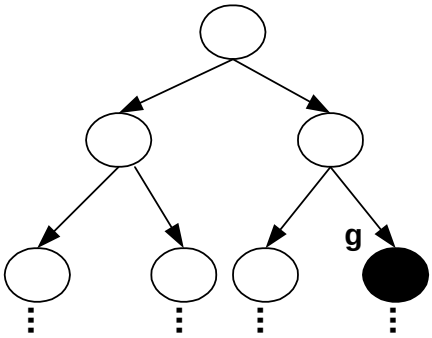

Figura 2.2.2.3 - EF g

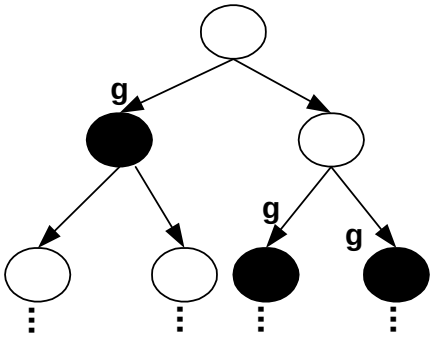

Figura 2.2.2.4 - AF g

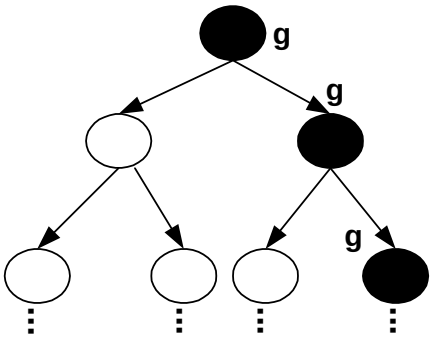

Figura 2.2.2.5 - EG g

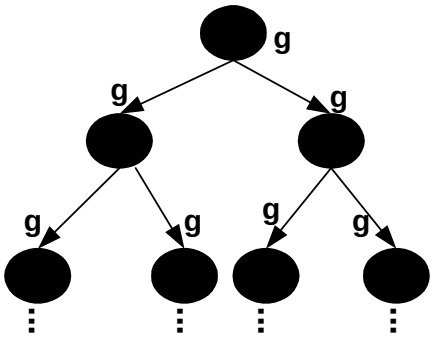

Figura 2.2.2.6 - AG g

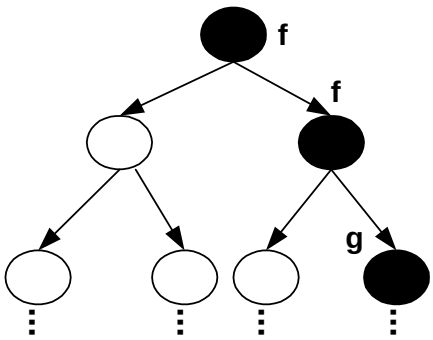

Figura 2.2.2.7 - E(fUg)

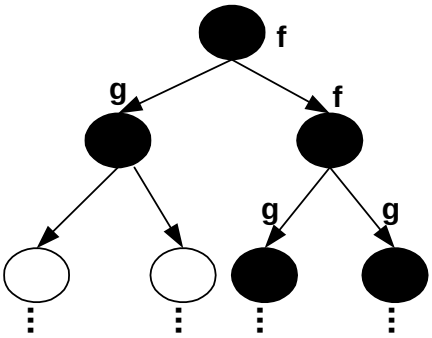

Figura 2.2.2.8 - A(fUg)