Lógicas Temporais e Verificação de Sistemas

Prof Mario Benevides
Davi R. Vasconcelos

9 de setembro de 2009

Outline

Definição

Verificação de Modelos é uma técnica automática para a verificação de sistemas concorrentes com número finito de estados.

A Necessidade de Métodos Formais

Sistemas de hardware e software são amplamente utilizados em aplicações onde falhas são inaceitáveis, como comércio eletrônico, redes de telefonia, sistemas de controle de tráfego aéreo, instrumentos cirúrgicos e muitos outros.

Em grande parte, nossas vidas dependem do funcionamento perfeito, contínuo e ininterrupto destas aplicações. A detecção de erros no curso de operação delas pode ser trágico. E mesmo em casos onde não haja risco de vida, desligá-las e substituir código de funcionamento crítico é muitas vezes economicamente inviável.

Por causa do sucesso da Internet e de Sistemas Embutidos (Embedded Systems) em automóveis, aviões e outros dispositivos que exigem total segurança, estaremos cada vez mais dependentes do perfeito funcionamento de dispositvos computacionais no futuro.Por causa do crescimento desta área tecnológica, será cada vez mais importante o desenvolvimento de métodos que validem nossa confiança na corretude de determinados sistemas.

Principais formas de verificação e validação

- Simulação
- Testes
- Verificação Dedutiva (Prova Matemática)
- Verificação de Modelos

Simulação

Apresentação das entradas a um modelo abstrato do sistema e posterior análise das saídas.

Testes

Apresentação de entradas ao sistema implementado e análise das saídas.

Desvantagens de Simulação e Testes:

Alto custo computacional e difuculdade da apresentação de todas as combinações de entradas ao modelo.

Verificação Dedutiva (Prova Matemática)

Princípios Básicos

Baseia-se na utilização de axiomas e regras matemáticas para provar a corretude do sistema.

Desvantagem da Verificação Dedutiva:

Requer um especialista, geralmente um matemático, para executá-la e possui enorme complexidade temporal.

Vantagem da Verificação Dedutiva:

Pode ser utilizada para verificar sistemas com número infinito de estados.

Verificação de Modelos

Busca Exaustiva

Utiliza uma busca exaustiva no espaço de estados para determinar se alguma afirmativa é verdadeira ou falsa.

Sempre termina

Sempre termina com uma resposta SIM ou NÃO.

Eficiente

Sua implementação é baseada em algoritmos de eficiência razoável, podendo ser executada em máquinas de potencial moderado.

Automática

Automatizada em quase toda sua totalidade.

Desvantagem:

Só é aplicável a sistemas com número finito de estados

Verificação de Modelos versus Prova Automática

Verificação de Modelos

Na verificacão de modelos, o sistema e a propriedade a ser validada são representados, respectivamente, como um modelo finito e uma fórmula da lógica.

Prova Automática

Na prova automática de teoremas (*automated theorem proving*), tanto o sistema como a propriedade são representados como fórmulas da lógica.

Outras Abordagens

- Lógica proposicional, lógica de primeira-ordem, lógicas dinâmicas, epstêmicas e temporais.
- Combinar Verificação de Modelos e Prova Automática.
- Combinar lógicas diferentes: Ex.: Conhecimento e Tempo

Processo de Verificação de Modelos

Modelagem

Nesta fase, gera-se um modelo a partir do design original do sistema, utilizando um formalismo aceito pela ferramenta de Verificação de Modelos. (Ex: Autômato)

Especificação

Nesta fase, determina-se quais perguntas serão feitas ao sistema. É comum a utilização de Lógica Temporal.

Verificação

- Nesta fase, são testadas as especificações, uma a uma, e a cada resposta negativa, indicando um erro, é possível computar todos os caminhos que levariam ao mesmo(contra-exemplos).
- É importante estar atento, pois um erro pode ocorrer em outras duas situações: uma especificação errada ou terminação anormal do programa por falta de recursos computacionais.

Lógica Temporal

A Lógica Temporal é útil na construção de modelos concorrentes, já que é capaz de expressar a ordem dos eventos no tempo incluí-lo de forma explícita e direta.

Estruturas de Kripke

Em nosso caso, o significado de uma fórmula estará relacionado a um autômato que representa os estados e as transições entre eless. Os autômatos geram as Estruturas de Kripke.

Outline

3 Lógicas

- CTL* *full Computation Tree Logic*
- CTL *full Computation Tree Logic* ★
- LTL *Linear Temporal Logic*

Verificação de Modelos para CTL*, CTL e LTL

- Dada uma fórmula α, que representa uma propriedade do sistema, verificar se ela é satisfeita ($\models$) na estrutura $\mathcal{G}$ que representa o sistema.
- $\mathcal{G} \models \alpha$

Complexidade da Verificação de Modelos para CTL*, CTL e LTL

- CTL *Linear no número de estados e no tamanho da fórmula*
- LTL *Linear no número de estados e Exponencial no tamanho da fórmula*
- CTL* *Linear no número de estados e Exponencial no tamanho da fórmula*
- Normalmente o tamanho da fórmula é pequeno

Verificação de Modelos - Implementações

- SMV: *Symbolic Model Verifier*
- SMVNu: software livre
- SPIN
- UPPAAL: trata de tempo-real
- HYTECH: autômatos híbridos
- PRISM: autômatos estocáticos

Complexidade da Prova Automática para CTL*, CTL e LTL

- CTL: *EXPTIME Completo*
- LTL: *PSPACE Completo*
- CTL*: *EXPTIME Completo*
- Geralmente pouco eficiente

Métodos de Prova

- Axiomático CTL, LTL, CTL*
- Tableaux CTL, LTL
- Dedução Natural CTL, LTL

Expressar Propriedades Temporais

- $F\varphi$ - φ vai ser verdadeira em algum instante no futuro
- $G\varphi$ - φ vai ser verdadeira sempre no futuro
- $P\varphi$ - φ foi verdadeira em algum instante no passado
- $H\varphi$ - φ sempre foi ser verdadeira no passado

Utilização CTL*, CTL e LTL

Validação de propriedades acerca do comportamento de sistemas reativos

Sistemas Reativos

- Estados e Transições
- Estado: descrição do sistema em um instante de tempo, i.e., os valores associados as suas variáveis naquele instante
- Transições: relação entre estados que expressam as mudanças de estados
- Computação: uma sequência infinita de estados, onde cada estado é obtido do anterior usando a relação de transição

Sistemas Reativos e Computações

- Sistemas Reativos são representados como **Autômatos**
- Todas as possíveis execuções do Autômato geram uma árvore chamada **Estrutura de Kripke**
- Computações são representadas como **Caminhos** numa Estrutura de Kripke

Estrutura de Kripe

- Conjunto de estados
- Relação de transições entre estados
- Função que rotula cada estado com o conjunto de propriedades verdadeiras nele

Definição

Uma estrutura de Kripke é definida por $\langle S, S_o, R, L\rangle$, onde:

- Um conjunto não-vazio de estados S.
- Um conjunto de estados iniciais $S_o \subseteq S$.
- Uma relação $R \subseteq S \times S$ **total**, i.e., para todo estado $s \in S$ existe um estado $s' \in S$ tal que $\langle s, s'\rangle \in R$.
- Uma função de rótulos $L : S \to 2^{\Pi}$, onde Π é o conjunto de proposições.

Exemplo

Exemplo de uma estrutura de Kripke $\mathcal{K} = \langle S, S_o, R, L \rangle$, onde

- $S = \{s,\ s_a,\ s_{ab}\}$.
- $S_o = \{s\}$.
- $\langle s, s_a \rangle \in R$, $\langle s_a, s \rangle \in R$, $\langle s, s_{ab} \rangle \in R$, $\langle s_a, s_{ab} \rangle \in R$, $\langle s_{ab}, s_{ab} \rangle \in R$.
- $L(s) = \emptyset$, $L(s_a) = \{a\}$, $L(s_{ab}) = \{a, b\}$.

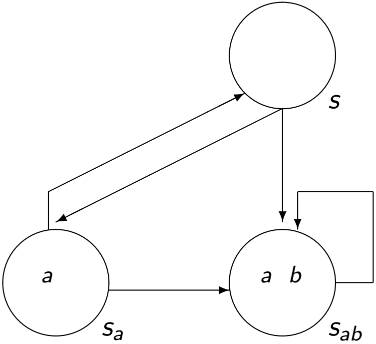


Figura: Estrutura de Kripke.

Caminhos num Estrutura de Kripke

- Um **caminho em uma estrutura de Kripke a partir de um estado** $s \in S$ é uma sequência $\pi = s_0, s_1, \ldots$ tal que $s = s_0$ e $\forall k \geq 0, \langle s_k, s_{k+1} \rangle \in R$
- Um s-caminho é um caminho contendo o estado s
- π_k = estado s_k do caminho π
- $\pi_{0,k}$ = prefixo $s_0, s_1, \ldots, s_k$ de π
- $\pi_{k,\infty}$ = sufixo $s_k, s_{k+1}, \ldots$ de π

Exemplo Anterior

Na estrutura $\mathcal{K}$ temos os seguintes caminhos a partir do estado s: $s, s_a, s, s_a, \ldots$; $s, s_a, s, s_a, \ldots, s_{ab}, s_{ab}, \ldots$; $s, s_{ab}, s_{ab}, \ldots$.

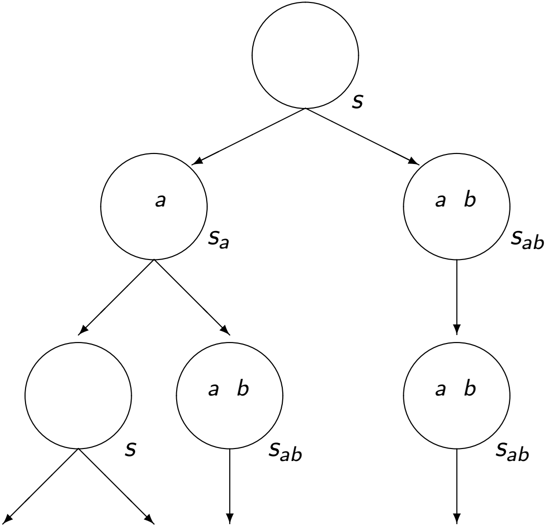


Figura: árvore de Computação Infinita.

Propriedades dos Sistemas Reativos

- Segurança (*safety*): nada ruim acontecerá.
  - O sistema nunca irá entrar em *deadlock*.
  - Nunca dois processos entram na região crítica ao mesmo tempo.
- Progresso (*liveness*): algo bom acontecerá.
  - Cada processo entrará na região crítica no futuro.
  - Toda requisição será atendida.

Exemplo Anterior

As propriedades a serem verificadas em um sistema reativo são expressas como fórmulas da linguagem da lógica temporal, que especificam os comportamentos desejados

Classificação das Lógicas Temporais

- Lógicas Temporais de Tempo Linear - LTL
- Lógicas Temporais de Tempo Ramificado - CTL e CTL*

Sequências de estados

Tais lógicas se referem a noção de sequências de estados que descrevem possíveis computações do sistema, e não a valores de tempos, ou a intervalos de tempos; tratam do comportamento de sistemas não-determinísticos que envolvem diferentes caminhos. Cada estado pode ter vários sucessores em termos de ramificação, seu comportamento é dado por um conjunto de caminhos lineares.

Full Computation Tree Logic - CTL*

Descreve propriedades de árvores de computações, que são obtidas a partir de um estado através do desdobramento de uma estrutura de Kripke em uma árvore de computação infinita. Ver exemplo

Linguagem de CTL*

- Fórmulas de Estados - Φ_s
- Fórmulas de Caminhos - Φ_p

Fórmulas de Estados - Φ_s

Descrevem propriedades que são avaliadas sobre estados:

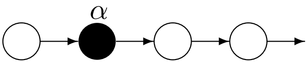
- $E\alpha$ - é verdadeira em um estado s se existe um caminho começando em s t.q.α é verdadeira neste caminho;
- $A\alpha$ - é verdadeira em um estado s se para todo um caminho começando em s, α é verdadeira neste caminho;

Fórmulas de Caminhos - Φ_p

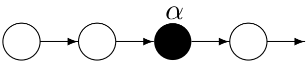
- Descrevem propriedades que são avaliadas sobre caminhos.
- Os operadores X ('*Next time*'), F ('*in the Future*'), G ('*Globally*') e $\mathcal{U}$ ('*Until*')

Fórmulas de Caminhos - Φ_p

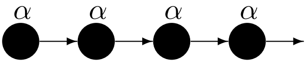
- $X\alpha$ - é verdadeira em um caminho π, se no próximo estado do caminho α é verdadeira.
- $F\alpha$ - é verdadeira em um caminho π, se em algum estado no caminho α é verdadeira.
- $G\alpha$ - é verdadeira em um caminho π, se em todo estado no caminho α é verdadeira.
- $\alpha\mathcal{U}\beta$ - é verdadeira em um caminho π, se α é verdadeira no caminho até que β seja verdadeira.



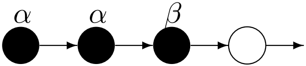
(a) - $X\alpha$.



(b) - $F\alpha$.



(c) - $G\alpha$.



(d) - $(\alpha\mathcal{U}\beta)$.

Figura: Operadores temporais X, F, G e $\mathcal{U}$.

Definição: Linguagem de CTL*

Sejam Π um conjunto de letras proposicionais e $P \in \Pi$ uma letra proposicional. A **linguagem de CTL*** é gerada pela seguinte BNF:

$$\Phi_s ::= P \mid (\neg \Phi_s) \mid (\Phi_s \wedge \Phi_s) \mid (\Phi_s \vee \Phi_s) \mid (\Phi_s \rightarrow \Phi_s) \mid$$
$$\mid (E\Phi_p) \mid (A\Phi_p)$$
$$\Phi_p ::= \Phi_s \mid (\neg \Phi_p) \mid (\Phi_p \wedge \Phi_p) \mid (\Phi_p \vee \Phi_p) \mid (\Phi_p \rightarrow \Phi_p) \mid$$
$$\mid (X\Phi_p) \mid (F\Phi_p) \mid (G\Phi_p) \mid (\Phi_p \,\mathcal{U}\, \Phi_p)$$

Definição: Semântica de CTL*

A definição de satisfação $\models$ de CTL* é dada em duas partes:

- Seja α uma fómula de estado da linguagem Φ_s, escrevemos $\mathcal{K} \models_s \alpha$ para dizer que a fórmula α é satisfeita na estrutura $\mathcal{K}$ no estado s.
- Seja α uma fómula de caminho da linguagem Φ_p, escrevemos $\mathcal{K} \models_\pi \alpha$ para dizer que a fórmula α é satisfeita na estrutura $\mathcal{K}$ no caminho π.

Satisfação em Estado: Semântica de CTL*

- $\mathcal{K} \models_s P \iff P \in L(s)$.
- $\mathcal{K} \models_s (\neg\alpha) \iff$ NOT $\mathcal{K} \models_s \alpha$.
- $\mathcal{K} \models_s (\alpha \wedge \beta) \iff \mathcal{K} \models_s \alpha$ E $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s (\alpha \vee \beta) \iff \mathcal{K} \models_s \alpha$ OU $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s (\alpha \rightarrow \beta) \iff$ SE $\mathcal{K} \models_s \alpha$ ENTAO $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s (E\alpha) \iff$ Existe um caminho π a partir de s tal que $\mathcal{K} \models_\pi \alpha$.
- $\mathcal{K} \models_s (A\alpha) \iff$ Para todo caminho π a partir de s vale que $\mathcal{K} \models_\pi \alpha$.

Satisfação em Caminho: Semântica de CTL*

- $\mathcal{K} \models_\pi \alpha \iff$ se α é uma fórmula da linguagem Φ_s, $\mathcal{K} \models_{\pi_0} \alpha$.
- $\mathcal{K} \models_\pi (\neg\alpha) \iff$ NOT $\mathcal{K} \models_\pi \alpha$.
- $\mathcal{K} \models_\pi (\alpha \wedge \beta) \iff \mathcal{K} \models_\pi \alpha$ E $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (\alpha \vee \beta) \iff \mathcal{K} \models_\pi \alpha$ OU $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (\alpha \rightarrow \beta) \iff$ SE $\mathcal{K} \models_\pi \alpha$ ENTAO $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (X\alpha) \iff \mathcal{K} \models_{\pi_{1,\infty}} \alpha$.
- $\mathcal{K} \models_\pi (F\alpha) \iff$ Existe um $k \geq 0$ tal que $\mathcal{K} \models_{\pi_{k,\infty}} \alpha$.
- $\mathcal{K} \models_\pi (G\alpha) \iff$ Para todo $k \geq 0$ vale que $\mathcal{K} \models_{\pi_{k,\infty}} \alpha$.
- $\mathcal{K} \models_\pi (\alpha \mathcal{U} \beta) \iff$ Existe $k \geq 0$ tal que $\mathcal{K} \models_{\pi_{k,\infty}} \beta$ e para todo $0 \leq l < k$ vale que $\mathcal{K} \models_{\pi_{l,\infty}} \alpha$.

Exemplo de Satisfação CTL*

- $\mathcal{K} \models_s (E(G(\neg b)))$ - 'existe um caminho a partir de s tal que sempre vale $\neg b$'.
- $\mathcal{K} \models_s (A(F\ a))$ - 'para todos os caminhos a partir de s no futuro vale a'.
- $\mathcal{K} \models_{s,s_{ab},s_{ab},\ldots} (F(G(a \wedge b)))$ - 'no caminho $s, s_{ab}, s_{ab}, \ldots$ vale no futuro que sempre vale a e b'.
- $\mathcal{K} \not\models_{s,s_a,s,s_a,\ldots} (G\ a)$ - não é o caso que vale no caminho $s, s_a, s, s_a, \ldots$ que sempre vale a'.
- $\mathcal{K} \not\models_s (A(G(a \vee b)))$ - não é o caso que vale para todo caminho a partir de s que sempre vale a ou b'.

Comparando CTL* com CTL e LTL

- CTL e LTL estão **Contidas** em CTL*
- CTL $\subseteq$ CTL*
- LTL $\subseteq$ CTL*
- LTL $\nsubseteq$ CTL
- LTL $\nsupseteq$ CTL

Computation Tree Logic - CTL

Vamos agrupar operadores que vão funcionar como um único operador:

Linguagem de CTL

- $[EX]\alpha$ - 'existe um caminho tal que no próximo estado vale α'.
- $[AX]\alpha$ - 'para todo caminho no próximo estado vale α'.
- $[EF]\alpha$ - 'existe um caminho tal que no futuro vale α'.
- $[AF]\alpha$ - 'para todo caminho no futuro vale α'.
- $[EG]\alpha$ - 'existe um caminho tal que sempre vale α'.
- $[AG]\alpha$ - 'para todo caminho vale sempre α'.
- $E(\alpha\mathcal{U}\ \beta)$ - 'existe um caminho tal que vale α até que vale β'.
- $A(\alpha\mathcal{U}\ \beta)$ - 'para todo caminho vale α até que vale β'.

Definição: Linguagem de CTL

Sejam Π um conjunto de letras proposicionais e $P \in \Pi$ uma letra proposicional. A **linguagem de CTL** é gerada pela seguinte BNF:

$$\Phi_s ::= P \mid (\neg\Phi_s) \mid (\Phi_s \wedge \Phi_s) \mid (\Phi_s \vee \Phi_s) \mid (\Phi_s \rightarrow \Phi_s) \mid$$

$$\mid ([EX]\Phi_s) \mid ([AX]\Phi_s) \mid ([EF]\Phi_s) \mid ([AF]\Phi_s) \mid$$

$$\mid ([EG]\Phi_s) \mid ([AG]\Phi_s) \mid (E(\Phi_s \,\mathcal{U}\, \Phi_s)) \mid (A(\Phi_s \,\mathcal{U}\, \Phi_s))$$

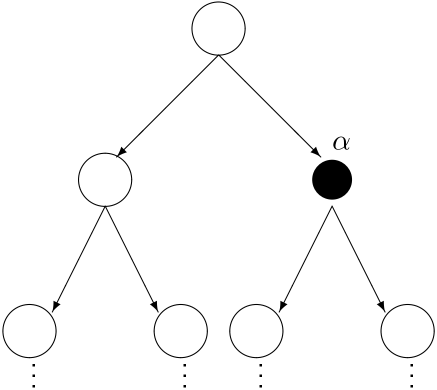
Definição: Semântica de CTL

Sejam Π um conjunto de letras proposicionais, P uma letra proposicional em Π, $\mathcal{K} = \langle S, S_o, R, L \rangle$ uma estrutura de Kripke para Π, $s \in S$ um estado da estrutura $\mathcal{K}$, e α uma fórmula de CTL. Escrevemos $\mathcal{K} \models_s \alpha$ para indicar que a fórmula α é satisfeita no estado s da estrutura $\mathcal{K}$. A definição de satisfação $\models$ segue.

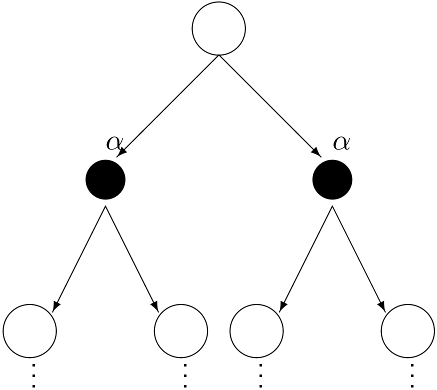
- $\mathcal{K} \models_s P \iff P \in L(s)$.
- $\mathcal{K} \models_s (\neg\alpha) \iff$ NOT $\mathcal{K} \models_s \alpha$.
- $\mathcal{K} \models_s (\alpha \wedge \beta) \iff \mathcal{K} \models_s \alpha$ E $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s (\alpha \vee \beta) \iff \mathcal{K} \models_s \alpha$ OU $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s (\alpha \rightarrow \beta) \iff$ SE $\mathcal{K} \models_s \alpha$ ENTAO $\mathcal{K} \models_s \beta$.
- $\mathcal{K} \models_s ([EX]\alpha) \iff$ Existe um caminho π a partir de s tal que $\mathcal{K} \models_{\pi_1} \alpha$.
- $\mathcal{K} \models_s ([AX]\alpha) \iff$ Para todo caminho π a partir de s vale que $\mathcal{K} \models_{\pi_1} \alpha$.

Definição: Semântica de CTL - Continuação

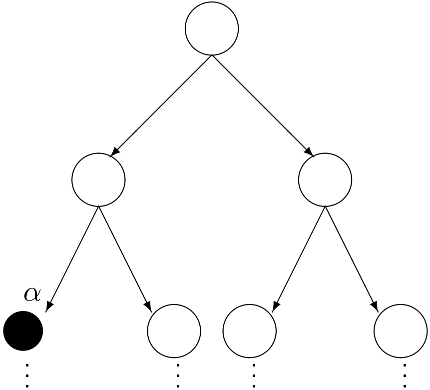
- $\mathcal{K} \models_s ([EF]\alpha) \iff$ Existe um caminho π a partir de s tal que existe um $k \geq 0$ tal que $\mathcal{K} \models_{\pi_k} \alpha$.
- $\mathcal{K} \models_s ([AF]\alpha) \iff$ Para todo caminho π a partir de s vale que existe um $k \geq 0$ tal que $\mathcal{K} \models_{\pi_k} \alpha$.
- $\mathcal{K} \models_s ([EG]\alpha) \iff$ Existe um caminho π a partir de s tal que para todo $k \geq 0$ vale que $\mathcal{K} \models_{\pi_k} \alpha$.
- $\mathcal{K} \models_s ([AG]\alpha) \iff$ Para todo caminho π a partir de s vale que para todo $k \geq 0$ vale que $\mathcal{K} \models_{\pi_k} \alpha$.
- $\mathcal{K} \models_s (E(\alpha \mathcal{U} \beta)) \iff$ Existe um caminho π a partir de s tal que existe $k \geq 0$ tal que $\mathcal{K} \models_{\pi_k} \beta$ e para todo $0 \leq l < k$ vale que $\mathcal{K} \models_{\pi_l} \alpha$.
- $\mathcal{K} \models_s (A(\alpha \mathcal{U} \beta)) \iff$ Para todo caminho π a partir de s vale que existe $k \geq 0$ tal que $\mathcal{K} \models_{\pi_k} \beta$ e para todo $0 \leq l < k$ vale que $\mathcal{K} \models_{\pi_l} \alpha$.



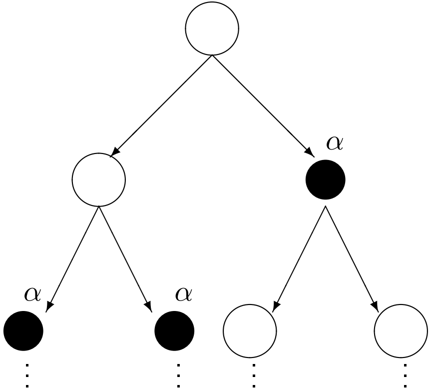
(a) - $[EX]\alpha$



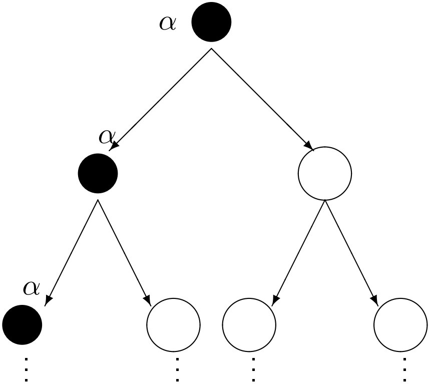
(b) - $[AX]\alpha$



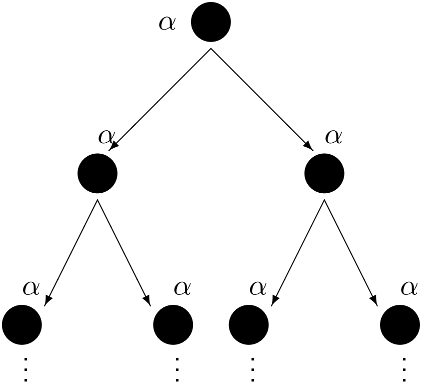
(c) - $[EF]\alpha$



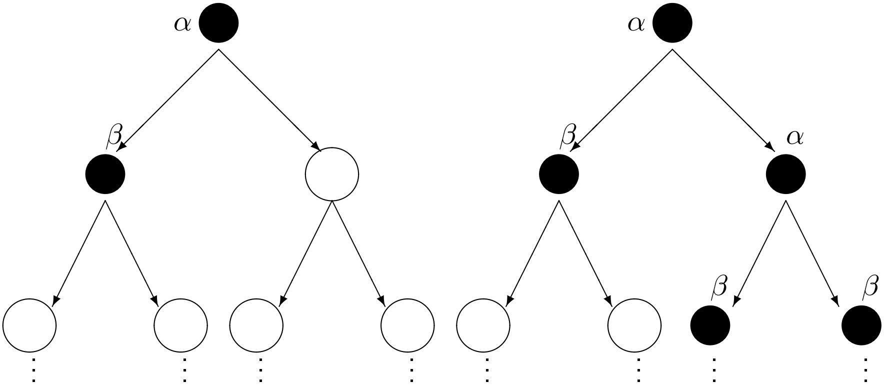
(d) - $[AF]\alpha$



(e) - $[EG]\alpha$



(f) - $[AG]\alpha$



(g) - $E(\alpha \mathcal{U} \beta)$ (h) - $A(\alpha \mathcal{U} \beta)$

Figura: Operadores de CTL.

Operadores Básicos CTL

- Básicos: $\neg, \rightarrow, [EX], \ [EG]$ e $[EU]$
- Definidos: $\wedge, \vee, [AX], [EF], [AF], [AG]$ e $[AU]$

- $\alpha \wedge \beta \iff \neg(\alpha \rightarrow \neg\beta)$
- $\alpha \vee \beta \iff (\neg\alpha) \rightarrow \beta$
- $[AF]\alpha \iff \neg[EG](\neg\alpha)$
- $[AX]\alpha \iff \neg[EX](\neg\alpha)$
- $[EF]\alpha \iff E(\top \ \mathcal{U} \ \alpha)$
- $[AG]\alpha \iff \neg[EF](\neg\alpha)$
- $A(\alpha\mathcal{U}\beta) \iff (\neg E((\neg\beta) \ \mathcal{U} \ (\neg\alpha \wedge \neg\beta))) \wedge (\neg[EG](\neg\beta))$

Exemplo de Satisfação CTL

- $\mathcal{K} \models_s ([EG](\neg b))$, 'existe um caminho a partir de s tal que sempre vale $\neg b$'.
- $\mathcal{K} \models_s ([AF]\ a)$, 'para todo caminho a partir de s no futuro vale a'.
- $\mathcal{K} \models_s ([EF]([AG](a \wedge b)))$, 'existe um caminho a partir de s tal que no futuro sempre vale a e b'.
- $\mathcal{K} \not\models_s ([EG]\ a)$, não é o caso que existe um caminho a partir de s tal que sempre vale a'.
- $\mathcal{K} \not\models_s ([AG](a \vee b))$, não é o caso que vale para todo caminho a partir de s que sempre vale a ou b'.

Linear Temporal Logic - LTL

A quantificação sobre caminhos é **Implicita**.

Definição: Linguagem de LTL

Sejam Π um conjunto de letras proposicionais e $P \in \Pi$ uma letra proposicional. A **linguagem de LTL** é gerada pela seguinte BNF:

$$\Phi_p ::= P \mid (\neg\Phi_p) \mid (\Phi_p \wedge \Phi_p) \mid (\Phi_p \vee \Phi_p) \mid (\Phi_p \rightarrow \Phi_p) \mid$$

$$\mid (X\Phi_p) \mid (F\Phi_p) \mid (G\Phi_p) \mid (\Phi_p \, \mathcal{U} \, \Phi_p)$$

Definição: Semântica de LTL

Sejam Π um conjunto de letras proposicionais, P uma letra proposicional em Π, $\mathcal{K} = \langle S, S_o, R, L \rangle$ uma estrutura de Kripke, π um caminho da estrutura $\mathcal{K}$, e α uma fórmula de LTL. $\mathcal{K} \models_\pi \alpha$ significa que α é satisfeita em $\mathcal{K}$ no caminho π.

- $\mathcal{K} \models_\pi P \iff P \in L(\pi_0)$.
- $\mathcal{K} \models_\pi (\neg\alpha) \iff$ NOT $\mathcal{K} \models_\pi \alpha$.
- $\mathcal{K} \models_\pi (\alpha \wedge \beta) \iff \mathcal{K} \models_\pi \alpha$ E $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (\alpha \vee \beta) \iff \mathcal{K} \models_\pi \alpha$ OU $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (\alpha \rightarrow \beta) \iff$ SE $\mathcal{K} \models_\pi \alpha$ ENTAO $\mathcal{K} \models_\pi \beta$.
- $\mathcal{K} \models_\pi (X\alpha) \iff \mathcal{K} \models_{\pi_{1,\infty}} \alpha$
- $\mathcal{K} \models_\pi (F\alpha) \iff$ Existe um $k \geq 0$ tal que $\mathcal{K} \models_{\pi_{k,\infty}} \alpha$
- $\mathcal{K} \models_\pi (G\alpha) \iff$ Para todo $k \geq 0$ vale $\mathcal{K} \models_{\pi_{k,\infty}} \alpha$
- $\mathcal{K} \models_\pi (\alpha \mathcal{U} \beta) \iff$ Existe $k \geq 0$ tal que $\mathcal{K} \models_{\pi_{k,\infty}} \alpha$ e para todo $0 \leq l < k$ vale $\mathcal{K} \models_{\pi_{l,\infty}} \alpha$

Exemplo de Satisfação LTL

- $\mathcal{K} \models_{s,s_{ab},s_{ab},\ldots} ((\neg(a \vee b))\mathcal{U}(a \wedge b))$: 'no caminho $s, s_{ab}, s_{ab}, \ldots$ não é o caso que vale a ou b até que valha a e b'.
- $\mathcal{K} \not\models_{s,s_a,s,s_a,\ldots} (G\ a)$: não é o caso que vale no caminho $s, s_a, s, s_a, \ldots$ que sempre vale a'.
- $\mathcal{K} \models_{s,s_{ab},s_{ab},\ldots} (F(G\ a))$: 'vale no caminho $s, s_{ab}, s_{ab}, \ldots$ que a partir de algum estado no futuro sempre vale a'.

Quantificação Implicita LTL

- A quantificação é universal sobre todos os caminhos de uma estrutura de Kripke.
- Um sistema é considerado correto em relação a uma propriedade se ela é verdadeira para todos os caminhos da estrutura a partir de todos os estados iniciais.

CTL* versus CTL versus LTL

- CTL e LTL estão **Contidas** em CTL*
- $A(FGp)$ - todo caminho tem um estado onde p vale - LTL mas NÃO CTL
- $[AG][EF]p$ - para todo caminho e todo estado existe um caminho onde p é verdadeiro - CTL mas NÃO LTL
- $A(FGp) \vee [AG][EF]p$ - CTL* mas NÃO CTL e NÃO LTL

CTL versus LTL

Propriedades usualmente utilizadas na verificação de sistemas reativos podem ser expressas em ambas as lógicas.

Outline

Estrutura de Kripke e Fórmula

- Sejam $\mathcal{K} = \langle S, S_o, R, L \rangle$ uma *estrutura de Kripke*, com estados finitos (S finito), e
- α uma fórmula de CTL, que expressa uma propriedade do sistema.

Problema da Verificação de Modelos

O problema da verificação de modelos é encontrar o conjunto dos estados que satisfazem a fórmula α:

$$\{s \in S \mid \mathcal{K} \models_s \alpha\}$$

Dizemos que o sistema satisfaz a propriedade se todos os estados iniciais $s \in S_o$ estão neste conjunto.

O Processo de Verificação de Um Sistema Reativo

Um verificador de modelos é uma ferramenta que automatiza a formulação acima. O processo de validação segue os três passos abaixo:

1. Especificar em CTL quais são as propriedades que o sistema deverá ter para que seja considerado correto. Por exemplo, podemos 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, que é definido, geralmente, em uma linguagem de alto nível (a linguagem do verificador).
3. O terceiro e último passo é a própria execução do verificador de modelos para validar as propriedades especificadas do sistema.

Verificação

- Tendo as propriedades e o modelo. Aplicamos o verificador e conseguimos 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 é gerado um contra exemplo mostrando o porquê da não verificação da propriedade.
- Desta forma, podemos detectar o erro e realizar a correção do modelo.
- Esse processo deve ser feito até que o sistema obedeça a todas as propriedades, realizando assim um ajuste na especificação.

Representação Explícita

O algoritmo descrito nesta seção utiliza uma representação explícita de uma estrutura de Kripke $\mathcal{K} = \langle S, R, L \rangle$, através de um grafo direcionado e rotulado.
Os nós do grafo representam os estados S, os arcos definem a relação de transição R e os rótulos associados com cada nó descrevem a função $L : S \rightarrow 2^{\Pi}$, onde Π é um conjunto de proposições.

Problema da Verificação de Modelos

O problema da verificação de modelos é encontrar o conjunto dos estados que satisfazem a fórmula α:

$$\{s \in S \mid \mathcal{K} \models_s \alpha\}$$

Dizemos que o sistema satisfaz a propriedade se todos estados os iniciais $s \in S_o$ estão neste conjunto.

Rotulando os Estados

- O algoritmo proposto opera rotulando cada estado s com o conjunto $label(s)$
- $label(s)$ contém as sub-fórmulas de α que são verdadeiras em s.
- Inicialmente, $label(s)$ é somente $L(s)$,
- O algoritmo é executado uma série de passos (o número de operadores em α). A cada passo sub-fórmulas com $k-1$ operadores são processadas.
- Quando uma fórmula é processada, a mesma é adicionada ao $label(s)$ do estado s se ela é verdadeira em s.
- $\mathcal{K} \models_s \alpha \iff \alpha \in label(s)$.

Algoritmos: Descrição Informal

Lembremos que as fórmulas de CTL podem ser expressas em termos de $\neg$, $\rightarrow$, $[EX]$, $[EG]$ e $E\mathcal{U}$. Desta forma, é necessário tratarmos apenas estes casos como segue.

- Para o caso $(\neg\alpha_1)$ (veja o algoritmo 2), rotulamos os estados que não são rotulados por α_1. A complexidade é $O(|S|)$.
- Para o caso $(\alpha_1 \rightarrow \alpha_2)$ (veja o algoritmo 3), rotulamos os estados que não são rotulados por α_1 ou são rotulados por α_2. A complexidade é $O(|S|)$.
- Para o caso $[EX]\alpha_1$ (veja o algoritmo 4), rotulamos os estados que tem algum sucessor rotulado por α_1. A complexidade é $O(|S| + |R|)$.

Algoritmos: Descrição Informal

Para o caso $E(\alpha_1 \mathcal{U} \alpha_2)$ (veja o algoritmo 5),

- adicionamos $E(\alpha_1 \mathcal{U} \alpha_2)$ a todos os estados rotulados com α_2,
- e então adicionamos $E(\alpha_1 \mathcal{U} \alpha_2)$ retroativamente, utilizando a inversa da relação R, a todos os estados rotulados com α_1 que podem ser alcançados por um caminho.
- A complexidade é $O(|S| + |R|)$.

Algoritmos: Descrição Informal

Para o caso $[EG]\alpha_1$ (veja o algoritmo 6) temos os passos:

1. Construímos $\mathcal{K}' = \langle S', R', L' \rangle$ a partir de $\mathcal{K} = \langle S, R, L \rangle$, removendo todos os estados de S que α_1 não vale e restringindo R e L.
2. Utilizamos o algoritmo do Tarjan para particionar o grafo $\langle S', R' \rangle$ em SCCs (*Strongly Connected Components*), onde um SCC é um subgrafo maximal C tal que todo nó em C é alcançável a partir de qualquer outro nó em C através de um caminho em C. Complexidade $O(|S'| + |R'|)$.
3. Adicionamos $[EG]\alpha_1$ aos estados que pertençam aos SCCs não-trivias (mais de um nó)
4. Por fim, adicionamos $[EG]\alpha_1$ retroativamente, utilizando a inversa da relação R, a todos os estados rotulados com α_1 que podem ser alcançados por um caminho.

Complexidade $[EG]\alpha_1$

- A complexidade do algoritmo $[EG]\alpha_1$ é $O(|S| + |R|)$;

Lemma

$\mathcal{K} \models_s [EG]\alpha_1 \iff s \in S'$ e existe um caminho em $\mathcal{K}'$ que leva o estado s a algum estado t em um SCC não-trivial de $\langle S', R' \rangle$.

Theorem

Existe um algoritmo para determinar se uma fórmula α de CTL é satisfeita em um estado s de $\mathcal{K} = \langle S, R, L \rangle$, cuja a complexidade algorítmica é $O(|\alpha| \times (|S| + |R|))$.

Algorithm 1 procedure Check(α)

while $|\alpha| \geq 1$ **do**
  if $\alpha = (\neg\alpha_1)$ **then**
    $Check(\alpha_1)$; $CheckNOT(\alpha_1)$
  else if $\alpha = (\alpha_1 \rightarrow \alpha_2)$ **then**
    $Check(\alpha_1)$; $Check(\alpha_2)$; $CheckIMP(\alpha_1, \alpha_2)$
  else if $\alpha = [EX]\alpha_1$ **then**
    $Check(\alpha_1)$
    $CheckEX(\alpha_1)$
  else if $\alpha = [EG]\alpha_1$ **then**
    $Check(\alpha_1)$
    $CheckEG(\alpha_1)$
  else if $\alpha = E(\alpha_1 \mathcal{U} \alpha_2)$ **then**
    $Check(\alpha_1)$; $Check(\alpha_2)$
    $CheckEU(\alpha_1, \alpha_2)$
  end if
end while

Algorithm 2 procedure CheckNOT(α)

for all $s \in S$ **do**
  if $\alpha \notin label(s)$ **then**
    $label(s) := label(s) \cup \{(\neg\alpha)\}$
  end if
end for

Algorithm 3 procedure CheckIMP($(\alpha_1 \rightarrow \alpha_2)$)

for all $s \in S$ **do**
  if $\alpha_1 \notin label(s)$ **or** $\alpha_2 \in label(s)$ **then**
    $label(s) := label(s) \cup \{(\alpha_1 \rightarrow \alpha_2)\}$
  end if
end for

Algorithm 4 procedure CheckEX(α_1)

$T := \{s \mid \alpha_1 \in label(s)\}$
while $T \neq \emptyset$ **do**
  choose $s \in T$
  $T := T \backslash \{s\}$
  for all t **such that** $\langle t, s \rangle \in R$ **do**
    if $[EX]\alpha_1 \notin label(t)$ **then**
      $label(t) := label(t) \cup \{[EX]\alpha_1\}$
    end if
  end for
end while

Algorithm 5 procedure CheckEU(α_1,α_2)

$T := \{s \mid \alpha_2 \in label(s)\}$
for all $s \in T$ **do**
  $label(s) := label(s) \cup \{E(\alpha_1 \mathcal{U} \alpha_2)\}$
end for
while $T \neq \emptyset$ **do**
  choose $s \in T$
  $T := T \backslash \{s\}$
  for all t **such that** $\langle t, s \rangle \in R$ **do**
    if $E(\alpha_1 \mathcal{U} \alpha_2) \notin label(t)$ **and** $\alpha_1 \in label(t)$ **then**
      $label(t) := label(t) \cup \{E(\alpha_1 \mathcal{U} \alpha_2)\}$
      $T := T \cup \{t\}$
    end if
  end for
end while

Algorithm 6 procedure CheckEG(α_1)

$S' := \{s \mid \alpha_1 \in label(s)\}$
$SCC := \{C \mid C \text{ is a nontrivial SCC of } S'\}$
$T := \cup_{C \in SCC}\{s \mid s \in C\}$
for all $s \in T$ **do**
  $label(s) := label(s) \cup \{[EG]\alpha_1\}$
end for
while $T \neq \emptyset$ **do**
  choose $s \in T$
  $T := T \setminus \{s\}$
  for all t **such that** $t \in S'$ **and** $\langle t, s \rangle \in R$ **do**
    if $[EG]\alpha_1 \notin label(t)$ **then**
      $label(t) := label(t) \cup \{[EG]\alpha_1\}$
      $T := T \cup \{t\}$
    end if
  end for
end while

Outline

Compactação e Eficiencia

- Aplicações reais onde número de estados é muito grande
- Usar OBDDs para representar as Estruturas de Kripke
- Representar uma estrutura de Kripke (um sistema) através de uma fórmula booleana
- OBDD - Ordered Binary Decision Diagrams
- Algoritmos eficientes para fazer Verificação de Modelos usando OBDD's
- Passou-se a tratar aplicações com número de estados $O(10^{20})$
- Hoje, com algumas otimizações, número de estados é $O(10^{120})$
- Todos os verificadores usam OBDDs

Dada uma estrutura de Kripke $K = \langle S, S_o, R, L \rangle$ sobre um conjunto de proposições $\Pi = \{p_1, \cdots, p_n\}$

- A função característica de um estado s, denotada por $[s]_\Pi$ é $[s]_\Pi = \left(\left(\bigwedge_{p_i \in L(s)} p_i \right) \wedge \left(\bigwedge_{p_i \notin L(s)} \neg p_i \right) \right)$
- Dado $\Pi' = \{p'_1, \cdots, p'_n\}$ e uma transição $t = \langle s_1, s_2 \rangle \in R$
- A função característica da transição t, denotada por $[t]_\Pi$ é $[t]_\Pi = [s_1]_\Pi \wedge [s_2]_{\Pi'}$

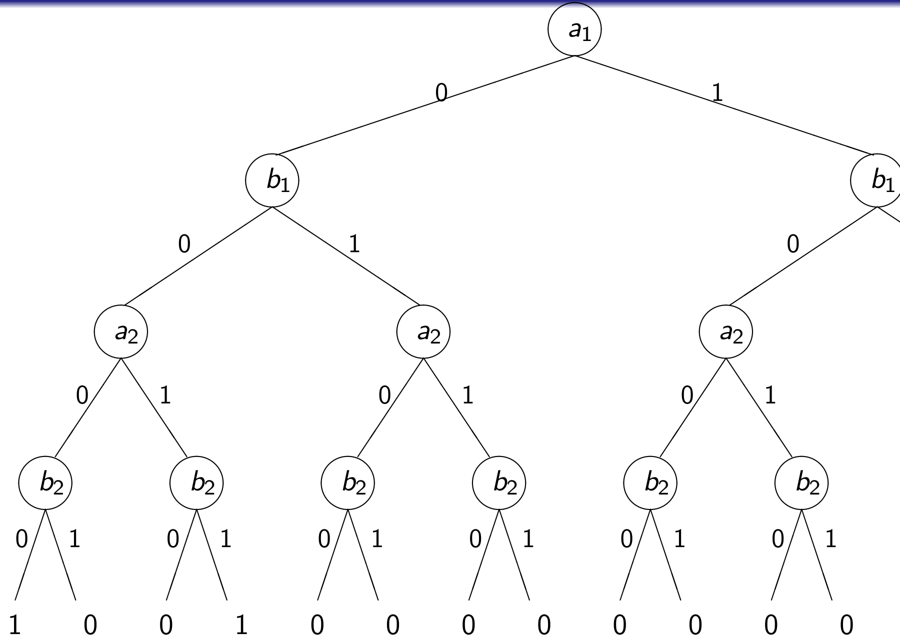
Exemplo Figura 2

- No exemplo: $\Pi = \{a, b\}$ e $\Pi' = \{a', b'\}$
- $[S_a]_\Pi = a \wedge \neg b$
- Transição $t_{sa} = \langle S, S_a \rangle \in R$
- A função característica da transição t_{sa} é $[t_{sa}]_\Pi = [S]_\Pi \wedge [S_a]_{\Pi'} = (\neg a \wedge \neg b) \wedge (a' \wedge \neg b')$
- O conjunto de todas as transições: $\{\langle s, s_a \rangle, \langle s_a, s \rangle, \langle s, s_{ab} \rangle, \langle s_a, s_{ab} \rangle, \langle s_{ab}, s_{ab} \rangle\}$.
- a estrutura da figura 2 é representada pela seguinte fórmula:

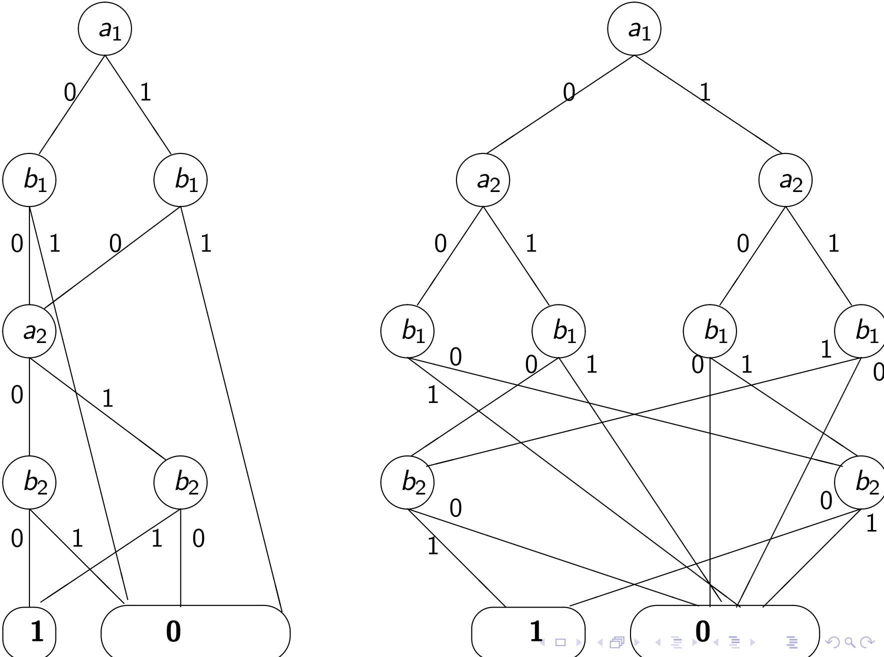
$$(\neg a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge \neg a' \wedge \neg b') \vee (\neg a \wedge \neg b \wedge a' \wedge b') \\ \vee (a \wedge \neg b \wedge a' \wedge b') \vee (a \wedge b \wedge a' \wedge b')$$

OBDD - Ordered Binary Decision Diagrams

- Construir um OBDD para a fórmula booleana que representa a estrutura de Kripke
- Uma representação concisa de uma árvore de decisão binária que utiliza uma ordenação nas variáveis Π
- A figura 8 abaixo representa uma árvore de decisão binária para a fórmula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$
- OBDD é construída unificando as sub-árvores que resultam no mesmo valor de verdade. Este processo é repetido até não haver nenhuma sub-árvore equivalente
- Na figura 9.a temos a OBDD obtida a partir da árvore 8 com ordenação $a_1 \succ b_1 \succ a_2 \succ b_2$
- Mais concisa do que a OBDD obtida com ordenação $a_1 \succ a_2 \succ b_1 \succ b_2$ apresentada na figura 9.b.
- O tamanho da OBDD gerada depende fortemente da ordenação das variáveis.


a_1
0
1
b_1
b_1
0
1
0
a_2
a_2
a_2
0
1
0
1
0
1
b_2
b_2
b_2
b_2
b_2
b_2
0
1
0
1
0
1
0
1
0
1
0
1
1
0
0
1
0
0
0
0
0
0
0
0

OBDD's - Ordered Binary Decision Diagrams



SMV

Iremos apresentar a linguagem do verificador de modelos *Symbolic Model Verifier* (SMV), que foi o primeiro a utilizar tal técnica para lógicas temporais.

É uma linguagem de especificação de isstemas de transição.

Usa como representação interna OBDD's, mas o usuário NÃO precisa saber nada sobre OBDD's.

SMV

a Seguir descrevemos as principais características da linguagem de especificação de modelos do SMV.

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. 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;
- 'SPEC' especificações em CTL.

Sincronismo e Assincronismo (*Interleaved*)

Módulos podem ser compostos de forma síncrona ou assíncrona.

- Em composição síncrona, um passo corresponde a um passo de cada módulo.
- Em composição assíncrona, cada passo corresponde a um passo de um único módulo.
- Se a palavra reservada '*process*' preceder uma instância de um módulo, assincronismo é usado. Caso contrário, a composição síncrona é assumida.
- Cada processo tem uma variável *running* que indica se o processo está ativo ou não. 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.

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 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.

Exemplo Especificação SMV: Semáforo

- Um programa que utiliza uma variável semáforo (*semaforo*) para implementar exclusão mútua entre dois processo assíncronos.
- 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.
- 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*'.

Exemplo Especificação SMV: Semáforo - Continuação

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

Exemplo Especificação SMV: Semáforo - Continuação

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