Verificação por Modelos

Manuel Alcino Cunha

Departamento de Informática
Universidade do Minho

2005/06

Introdução

- A área da verificação por modelos, ou *model checking*, tem por objectivo verificar automaticamente as propriedades lógicas de um modelo.
- Neste caso estamos interessados numa classe de modelos e numa lógica específicas: estruturas de Kripke e lógica CTL.
- Temos duas abordagens fudamentais ao problema da verificação:

  Directa Quando os algoritmos de verificação usam directamente a estrutura de Kripke.

  Simbólica Quando essa estrutura é representada implicitamente através de fórmulas proposicionais.

Verificação Directa

- Formalmente, dada uma estrutura de Kripke $M = (S, i, R, L)$ e uma fórmula CTL f, o objectivo da verificação por modelos directa é determinar todos os estados de M que satisfazem f.

$$[\![f]\!]_M \equiv \{s \in S \mid M, s \models f\}$$

- Um fórmula verifica-se num dado modelo sse for válida no estado inicial.

$$M \models f \Leftrightarrow i \in [\![f]\!]_M$$

Conectivas Mínimas

- Qualquer fórmula CTL pode ser expressa usando apenas as conectivas $\neg$, $\vee$, EX, EU e EG.

$$
\begin{aligned}
AX\ f &\equiv \neg EX\ \neg f \\
EF\ f &\equiv E[\text{true}\ U\ f] \\
AG\ f &\equiv \neg EF\ \neg f \\
AF\ f &\equiv \neg EG\ \neg f \\
A[f\ R\ g] &\equiv \neg E[\neg f\ U\ \neg g] \\
E[f\ R\ g] &\equiv EG\ g \vee E[g\ U\ (f \wedge g)] \\
A[f\ U\ g] &\equiv \neg E[\neg f\ R\ \neg g]
\end{aligned}
$$

- Sendo assim, um algoritmo de verificação de modelos apenas necessita de tratar estes cinco casos mais a validade das proposições atómicas.

Casos Triviais

- Dada uma estrutura de Kripke $M = (S, i, R, L)$ a validade de uma proposição atómica p define-se à custa da função de etiquetagem L.

$$[\![p]\!]_M = \{s \in S \mid p \in L(s)\}$$

- Para as conectivas proposicionais usam-se simples operações de conjuntos.

$$[\![\neg f]\!]_M = S - [\![f]\!]_M$$
$$[\![f \vee g]\!]_M = [\![f]\!]_M \cup [\![g]\!]_M$$

- Os estados que validam $EX\ f$ são aqueles com pelo menos um sucessor que valide f.

$$[\![EX\ f]\!]_M = R^{-1}([\![f]\!]_M) \equiv \{s \in S \mid \exists t \in S \cdot (s,t) \in R \wedge t \in [\![f]\!]_M\}$$

Operador Temporal *Sempre*

- A validade dos operadores *EG* e *EU* pode ser determinada recorrendo à sua definição recursiva (leis de expansão).
- Por exemplo, o operador temporal *sempre* pode ser definido como

$$EG\ f \equiv f \wedge EX\ (EG\ f)$$

- Esta lei implica que $[\![EG\ f]\!]_M$ corresponde à maior solução da seguinte equação recursiva no domínio $(\mathcal{P}(S), \subseteq)$.

$$X = [\![f]\!]_M \cap R^{-1}(X)$$

Cálculo de Maiores Pontos Fixos

- Dito de outra forma, $[\![EG\ f]\!]_M$ é o maior ponto fixo da função $\Pi : \mathcal{P}(S) \to \mathcal{P}(S)$.

$$\Pi(X) = [\![f]\!]_M \cap R^{-1}(X)$$

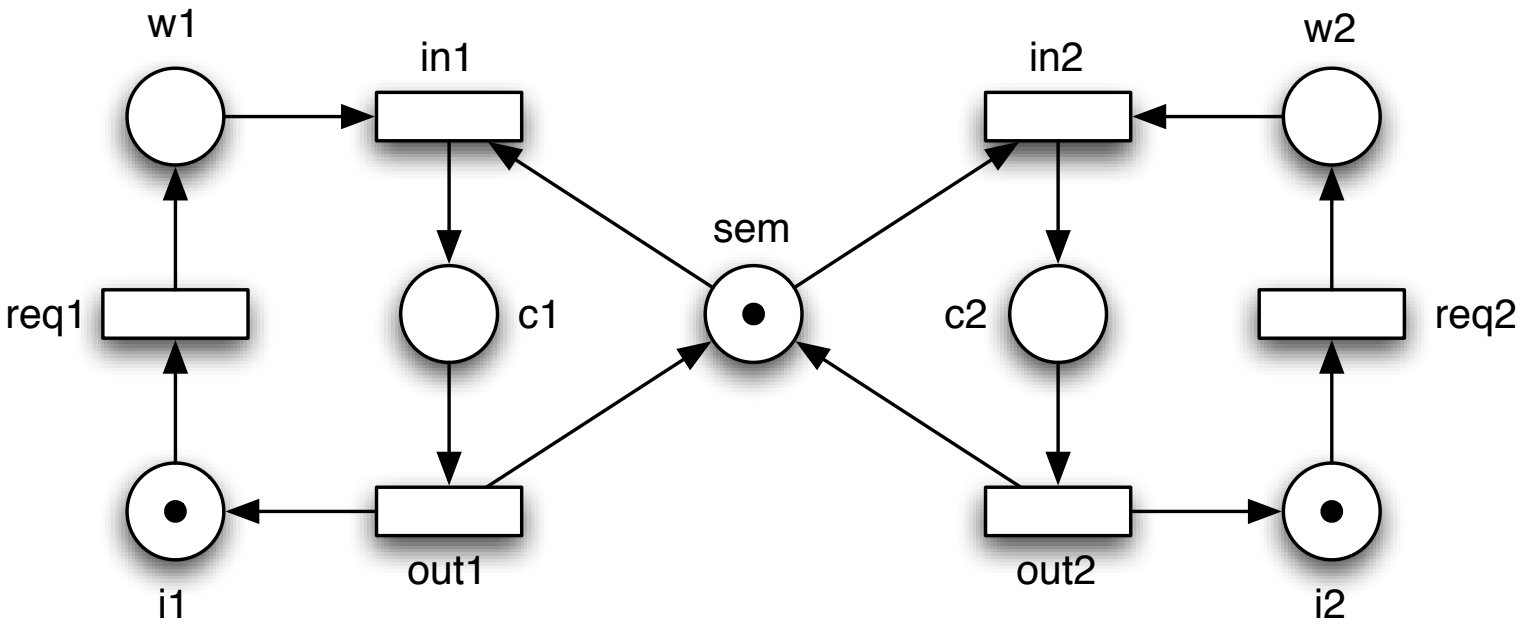
$$[\![EG\ f]\!]_M = \nu(\Pi)$$

- O teorema do ponto fixo de Kleene implica que $[\![EG\ f]\!]_M$ pode então ser calculado como o limite da série

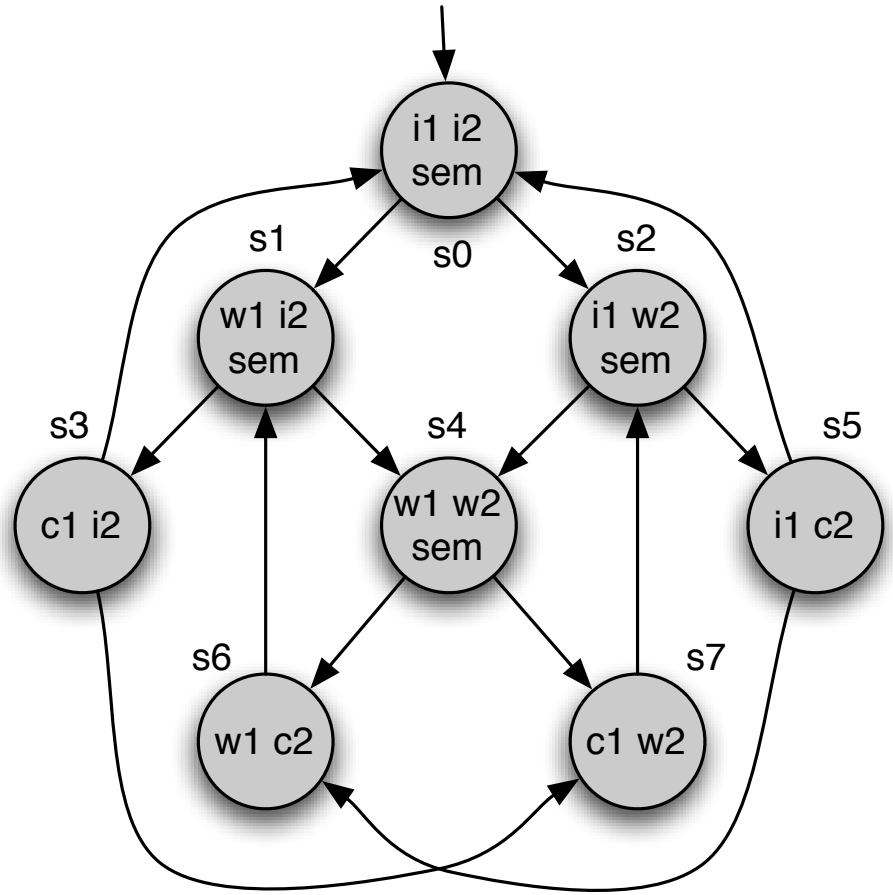
$$S, \Pi(S), \Pi(\Pi(S)), \Pi(\Pi(\Pi(S))), \ldots$$

- Obviamente a computação termina pois S é finito.

Exemplo: $[\![EG\ w_1]\!]$

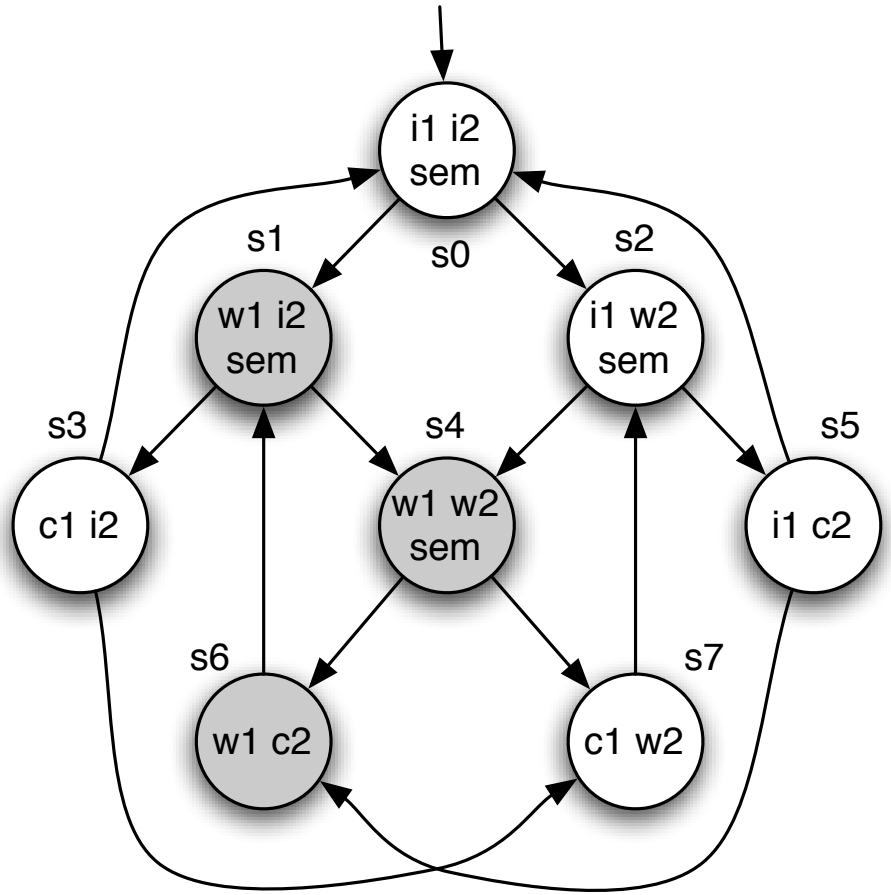


Exemplo: $[\![EG\ w_1]\!]$



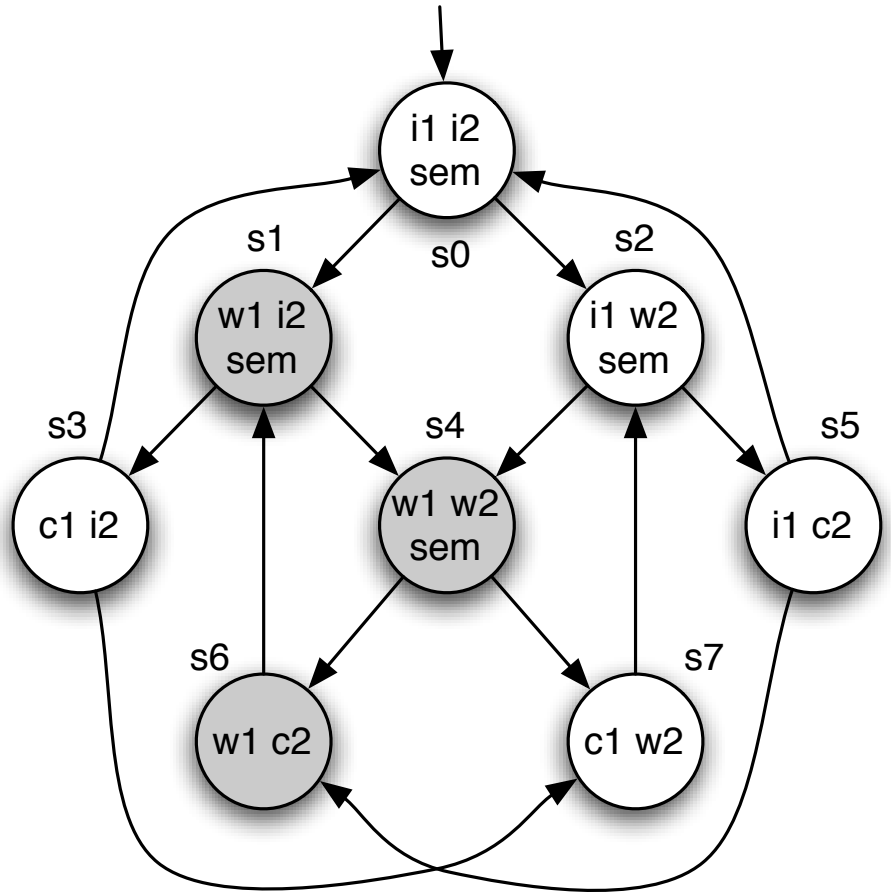
$$\Pi^0(S) = S$$

Exemplo: $[\![EG\ w_1]\!]$



$$\Pi^1(S) = [\![w_1]\!] \cap R^{-1}(\Pi^0(S)) = \{s_1, s_4, s_6\}$$

Exemplo: $[\![EG\ w_1]\!] = \{s_1, s_4, s_6\}$



$$\Pi^2(S) = [\![w_1]\!] \cap R^{-1}(\Pi^1(S)) = \{s_1, s_4, s_6\} \cap \{s_0, s_1, s_2, s_4, s_5, s_6\} = \Pi^1(S)$$

Operador Temporal *Até*

- De igual modo, o operador temporal *até* pode ser definido como

$$E[f\ U\ g] \equiv g \vee (f \wedge EX\ (E[f\ U\ g]))$$

- Esta lei implica que $[\![E[f\ U\ g]]\!]_M$ corresponde ao menor ponto fixo da função $\Pi : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$.

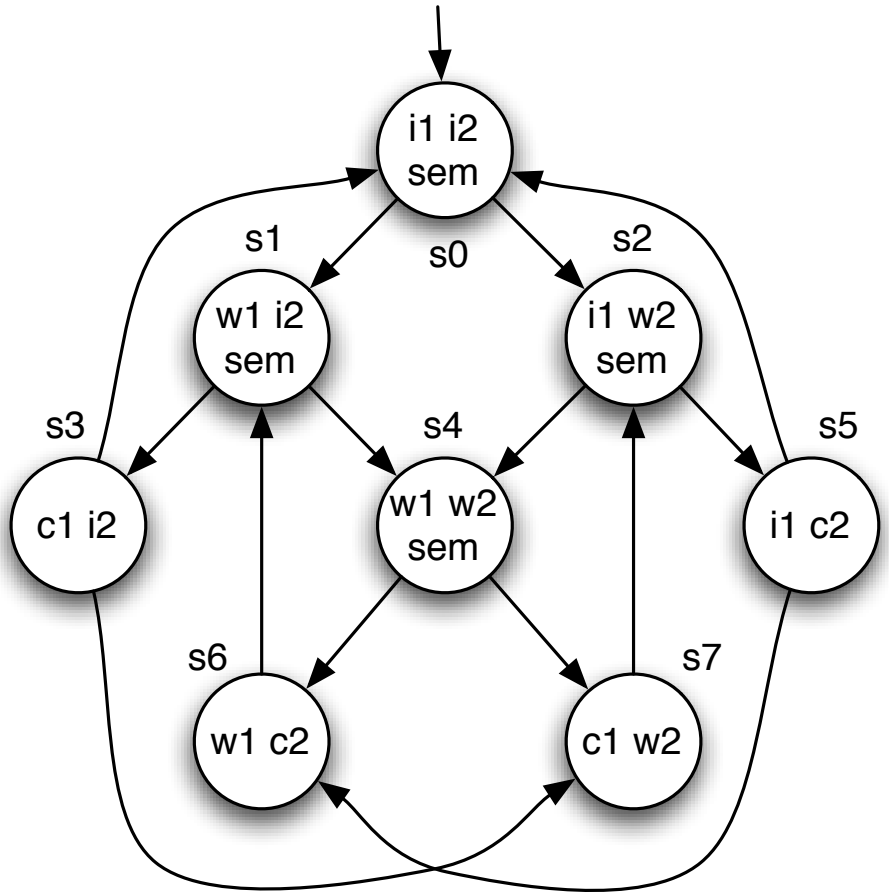
$$\Pi(X) = [\![g]\!]_M \cup ([\![f]\!]_M \cap R^{-1}(X))$$

$$[\![E[f\ U\ g]]\!]_M = \mu(\Pi)$$

- Sendo assim, este conjunto pode ser calculado como o limite da série

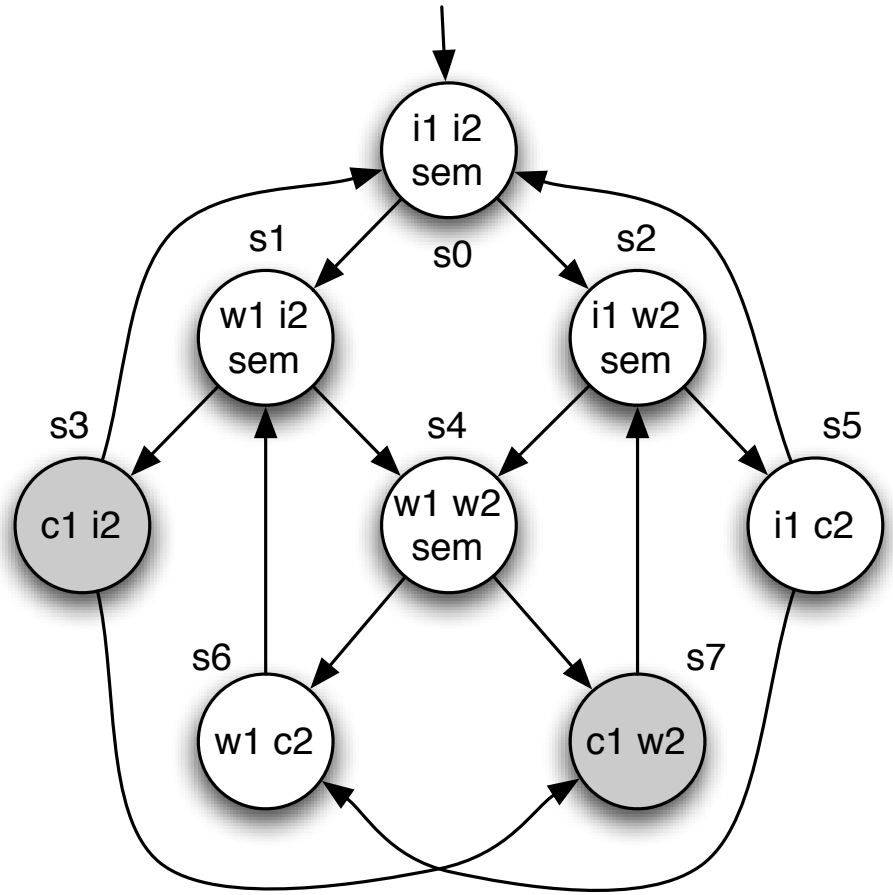
$$\emptyset, \Pi(\emptyset), \Pi(\Pi(\emptyset)), \Pi(\Pi(\Pi(\emptyset))), \ldots$$

Exemplo: $[\![E[w_1 \ U \ c_1]]\!]$



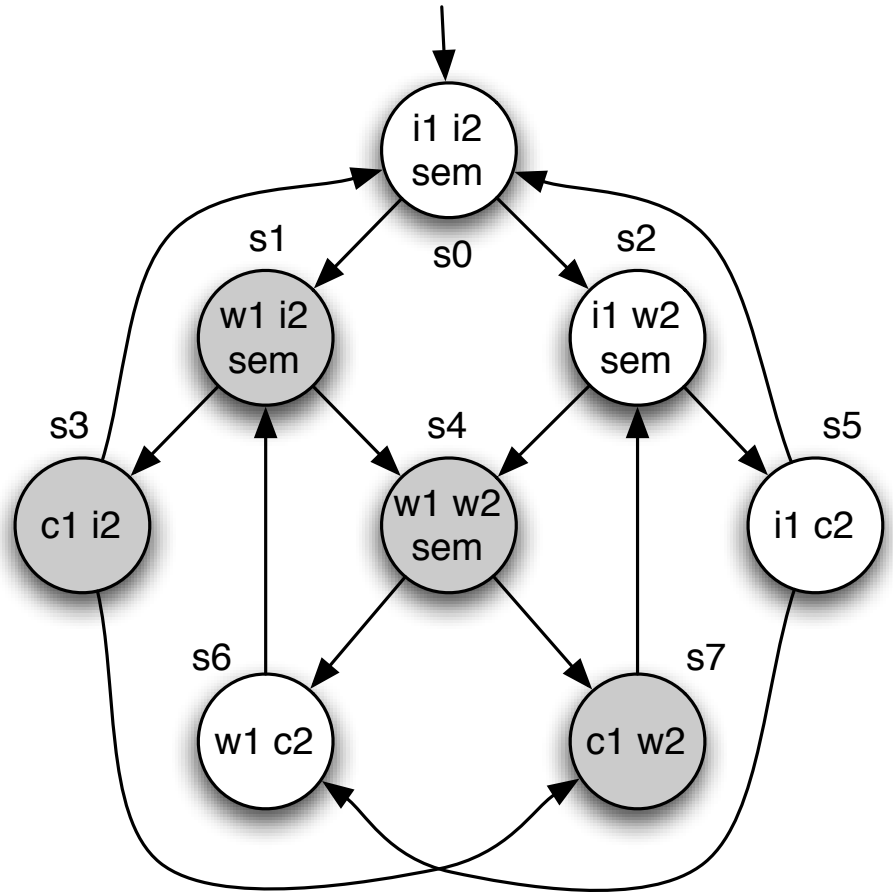
$$\Pi^0(\emptyset) = \emptyset$$

Exemplo: $[\![E[w_1 \ U \ c_1]]\!]$



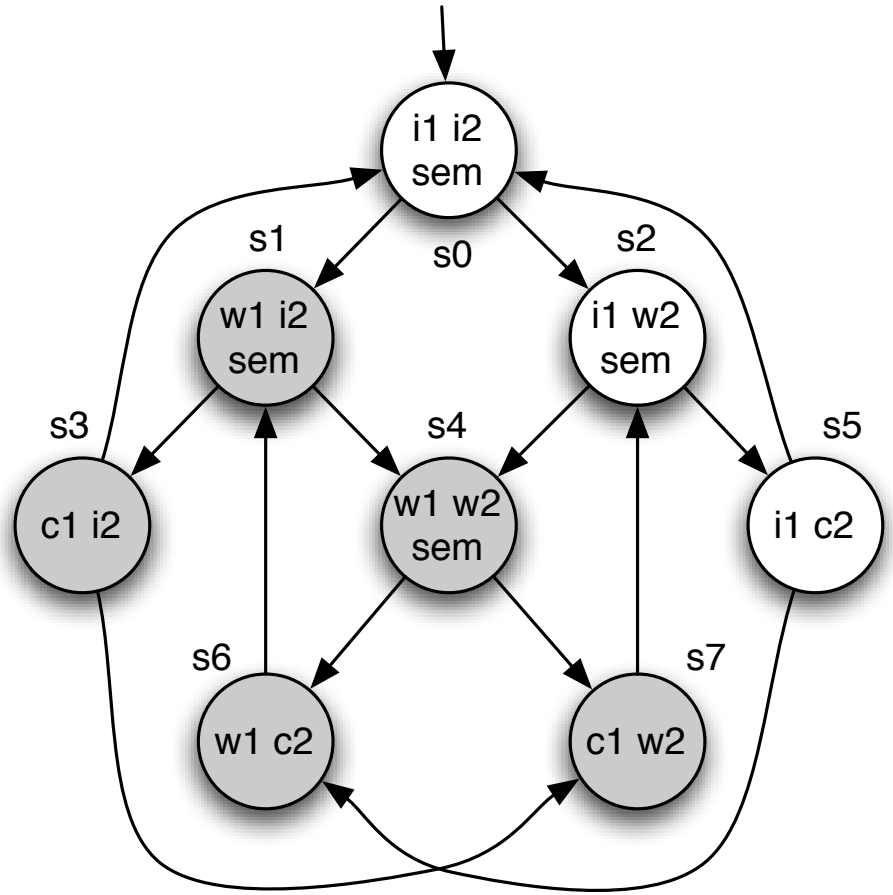
$$\Pi^1(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^0(\emptyset))) = \{s_3, s_7\}$$

Exemplo: $[\![E[w_1 \ U \ c_1]]\!]$



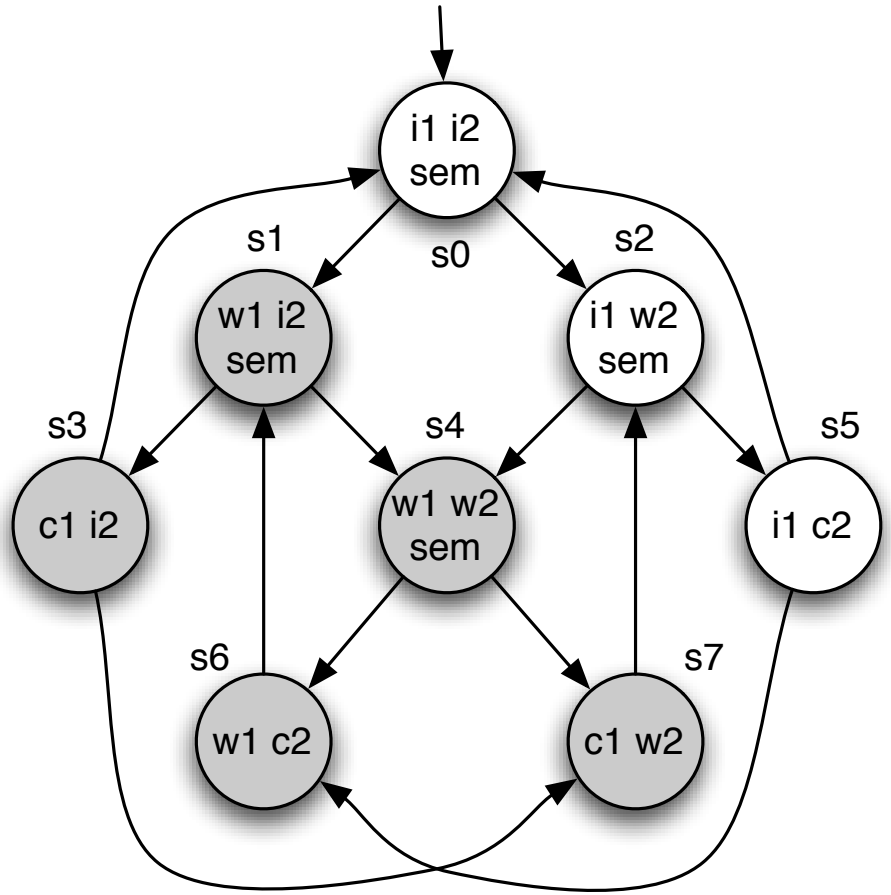
$$\Pi^2(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^1(\emptyset))) = \{s_1, s_3, s_4, s_7\}$$

Exemplo: $[\![E[w_1 \ U \ c_1]]\!]$



$$\Pi^3(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^2(\emptyset))) = \{s_1, s_3, s_4, s_6, s_7\}$$

Exemplo: $[\![E[w_1 \ U \ c_1]]\!] = \{s_1, s_3, s_4, s_6, s_7\}$



$$\Pi^4(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^3(\emptyset))) = \{s_1, s_3, s_4, s_6, s_7\}$$

Operador Temporal *Eventualmente*

- Embora não seja uma conectiva mínima, como o operador *EF* podemos extender esta técnica para verificar a sua validade.
- Este operador pode ser definido como

$$EF\ f \equiv f \vee EX\ (EF\ f)$$

- Esta lei implica que $[\![EFf]\!]_M$ corresponde ao menor ponto fixo da função $\Pi : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$.

$$\Pi(X) = [\![f]\!]_M \cup R^{-1}(X)$$

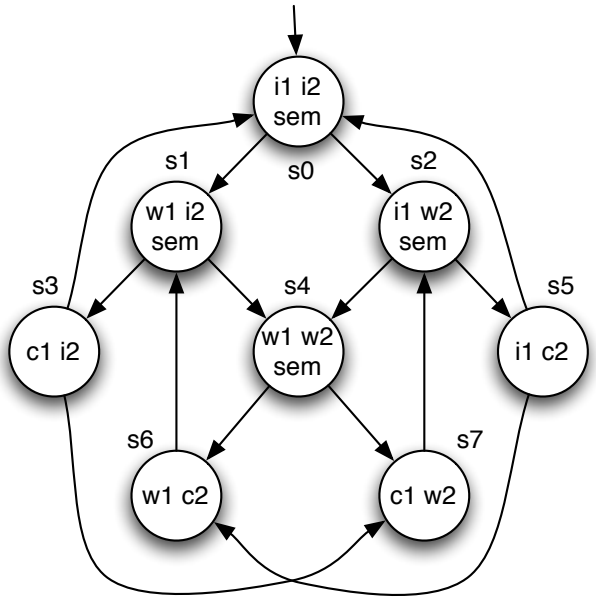
$$[\![EF\ f]\!]_M = \mu(\Pi)$$

Verificação de Fórmulas

- Antes de estabelecer a validade de uma fórmula f é necessário redefini-la usando o conjunto de conectivas mínimas (mais EF e $\wedge$ para facilitar).
- Depois usam-se os algoritmos definidos anteriormente para validar todas as sub-fórmulas de f, começando pelas mais pequenas e mais interiores, e prosseguindo de "baixo para cima" até à conectiva mais exterior.
- No final, a fórmula f será válida no modelo sse o estado inicial pertencer ao conjunto de estados que a validam.

Exemplo: $AG\neg(c_1 \wedge c_2)$

$$AG\neg(c_1 \wedge c_2) \equiv \neg EF\neg\neg(c_1 \wedge c_2) \equiv \neg EF\ (c_1 \wedge c_2)$$

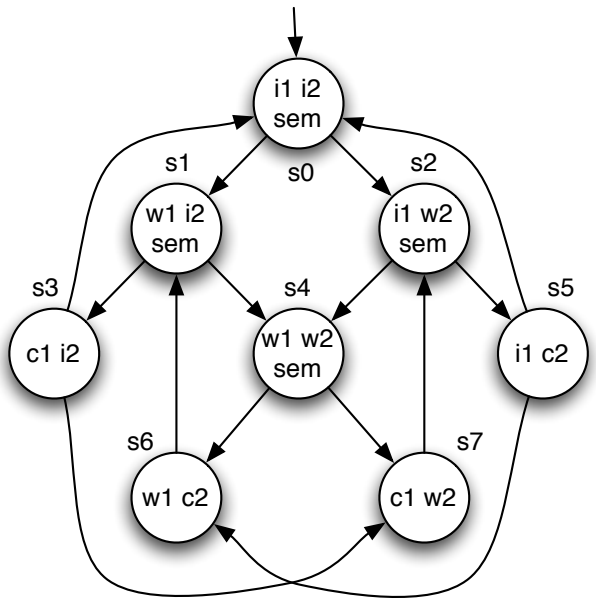


$$\begin{aligned}
[\![c_1]\!] &= \{s_3, s_7\} \\
[\![c_2]\!] &= \{s_5, s_6\} \\
[\![c_1 \wedge c_2]\!] &= \emptyset \\
[\![EF\ (c_1 \wedge c_2)]\!]^0 &= \emptyset \\
[\![EF\ (c_1 \wedge c_2)]\!]^1 &= [\![c_1 \wedge c_2]\!] \cup R^{-1}(\emptyset) = \emptyset \\
[\![\neg EF\ (c_1 \wedge c_2)]\!] &= S
\end{aligned}$$

$$M \models AG\neg(c_1 \wedge c_2)$$

Exemplo: $AG\,(w_1 \supset AF\,c_1)$

$$AG\,(w_1 \supset AF\,c_1) \equiv AG\,(\neg w_1 \vee AF\,c_1) \equiv \neg EF\neg(\neg w_1 \vee AF\,c_1) \equiv$$
$$\neg EF(w_1 \wedge \neg AF\,c_1) \equiv \neg EF(w_1 \wedge \neg\neg EG\neg c_1) \equiv \neg EF(w_1 \wedge EG\neg c_1)$$



$$\begin{aligned}
[\![w_1]\!] &= \{s_1, s_4, s_6\} \\
[\![c_1]\!] &= \{s_3, s_7\} \\
[\![\neg c_1]\!] &= \{s_0, s_1, s_2, s_4, s_5, s_6\} \\
[\![EG\neg c_1]\!]^0 &= S \\
[\![EG\neg c_1]\!]^1 &= [\![\neg c_1]\!] \cap R^{-1}(S) = [\![\neg c_1]\!] \\
[\![EG\neg c_1]\!]^2 &= \ldots = [\![\neg c_1]\!] \\
[\![w_1 \wedge EG\neg c_1]\!] &= [\![w_1]\!] \cap [\![\neg c_1]\!] = [\![w_1]\!] \\
[\![EF(w_1 \wedge EG\neg c_1)]\!]^0 &= \emptyset \\
[\![EF(w_1 \wedge EG\neg c_1)]\!]^1 &= \ldots = \{s_1, s_4, s_6\} \\
[\![EF(w_1 \wedge EG\neg c_1)]\!]^2 &= \ldots = \{s_0, s_1, s_2, s_4, s_5, s_6\} \\
[\![EF(w_1 \wedge EG\neg c_1)]\!]^4 &= \ldots = S \\
[\![\neg EF(w_1 \wedge EG\neg c_1)]\!] &= \emptyset
\end{aligned}$$

$$M \not\models AG\,(w_1 \supset AF\,c_1)$$

Reduzir a Complexidade

- Dada uma fórmula CTL f e uma estrutura de Kripke (S, i, R, L), o algoritmo de verificação baseado em pontos fixos tem complexidade

$$O(|f| \cdot |S| \cdot (|S| + |R|))$$

- Mas usando uma abordagem diferente é possível baixar a complexidade para

$$O(|f| \cdot (|S| + |R|))$$

- Para os casos triviais o algoritmo é indêntico.

Operador Temporal *Até*

- Para determinar $[\![E[f\ U\ g]]\!]$ basta começar por incluir todos os estados que validam g e, sucessivamente, acrescentar predecessores que validem f (evitando visitas repetidas).

checkEU $([\![f]\!], [\![g]\!]) \equiv$
  $T \leftarrow [\![g]\!]$;
  $[\![E[f\ U\ g]]\!] \leftarrow [\![g]\!]$;
  while $T \neq \emptyset$
    choose $s \in T$;
    $T \leftarrow T - \{s\}$;
    for $t \in R^{-1}(s)$
      if $t \notin [\![E[f\ U\ g]]\!] \wedge t \in [\![f]\!]$
        $[\![E[f\ U\ g]]\!] \leftarrow [\![E[f\ U\ g]]\!] \cup \{t\}$;
        $T \leftarrow T \cup \{t\}$;
  return $[\![E[f\ U\ g]]\!]$;

Operador Temporal *Sempre*

- Dada uma estrutura de Kripke $M = (S, i, R, L)$, para determinar a validade da fórmula $EG\ f$ basta considerar a restrição de M aos estados que validam f.
- Esta restrição será denominada M_f, e tem como estados $[\![f]\!]$ e como relação de acessibilidade $R_f = R \cap ([\![f]\!] \times [\![f]\!])$. Pode deixar de ser total.
- Depois é necessário determinar os *componentes fortemente ligados não triviais* de M_f.
  - Um subconjunto é um componente fortemente ligado se qualquer dos dos seus nodos é acessível a partir de todos os outros através de um caminho interno.
  - É não trivial se contém mais do que um nodo, ou contém apenas um nodo com um lacete.

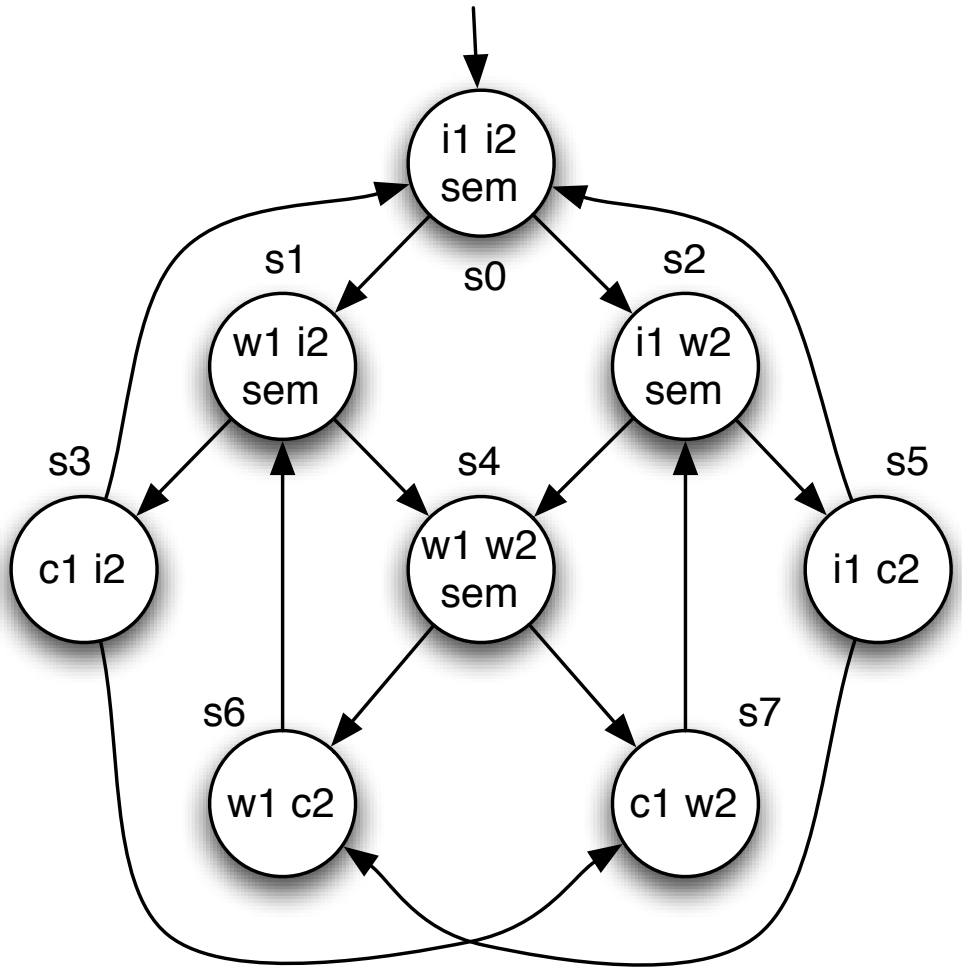
Operador Temporal *Sempre*

- O algoritmo para determinar a validade deste operador baseia-se no seguinte lema: $s \in [\![EG\ f]\!]$ sse $s \in [\![f]\!]$ e existe um caminho em M_f que liga s a um nodo num componente fortemente ligado não trivial de M_f.
- Para determinar $[\![EG\ f]\!]$ basta começar por incluir todos os estados pertencentes a um componente fortemente ligado não trivial e, sucessivamente, acrescentar predecessores que validem f (evitando visitas repetidas).
- O algoritmo seguinte utiliza um procedimento **scc** para determinar os componentes fortemente ligados, que utilizando o algoritmo de Tarjan pode ser implementado com complexidade $\mathcal{O}(|[\![f]\!]| + |R_f|)$.

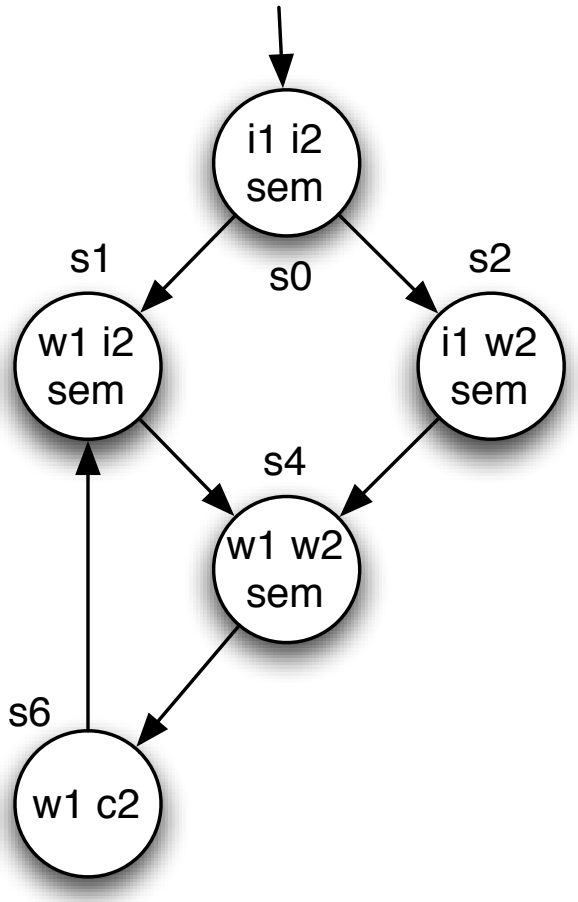
Operador Temporal *Sempre*

checkG $([\![f]\!]) \equiv$
$T \leftarrow \cup\{C \mid C \in \mathbf{scc}(M_f) \wedge \neg\mathbf{trivial}(C)\};$
$[\![EG\ f]\!] \leftarrow T;$
while $T \neq \emptyset$
choose $s \in T;$
$T \leftarrow T - \{s\};$
for $t \in R_f^{-1}(s)$
if $t \notin [\![EG\ f]\!]$
$[\![EG\ f]\!] \leftarrow [\![EG\ f]\!] \cup \{t\};$
$T \leftarrow T \cup \{t\};$
return $[\![EG\ f]\!];$

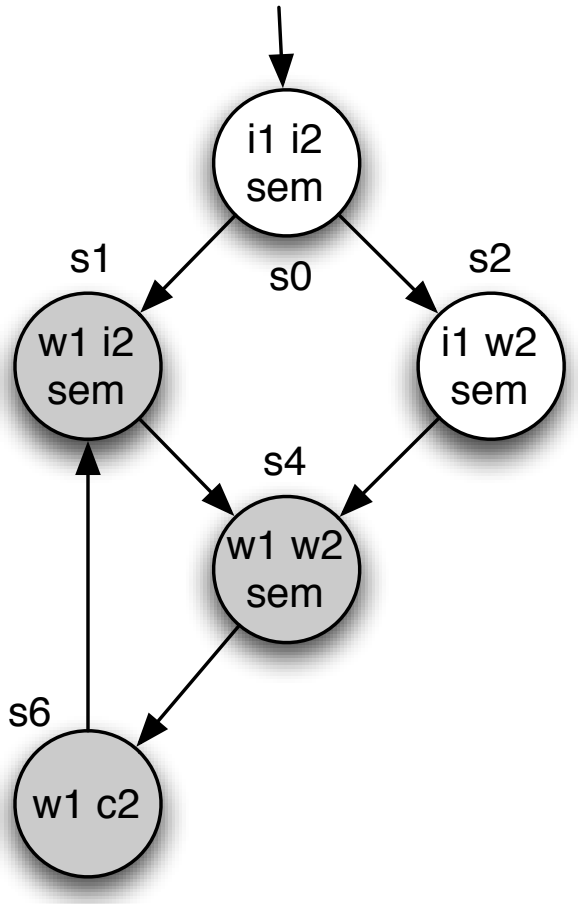
Exemplo: $EG\ (w_1 \vee sem)$



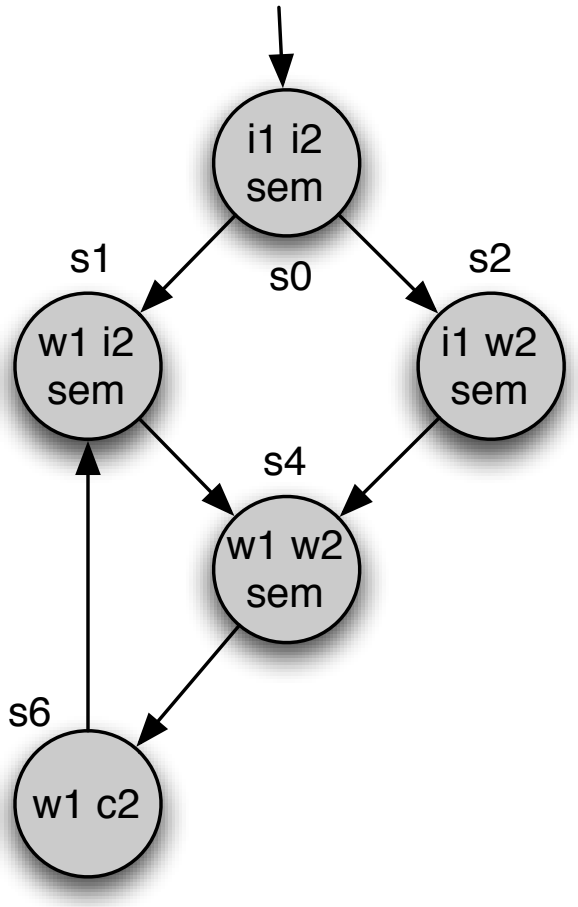
Exemplo: $EG\ (w_1 \vee sem)$



Exemplo: $EG\ (w_1 \vee sem)$



Exemplo: $EG\ (w_1 \vee sem)$



Justiça

- Algumas propriedades de animação só podem ser verificadas assumindo certas condições de justiça sobre o ambiente onde um determinado sistema executa:
  - Um *scheduler* normalmente garante que cada processo entra em execução um número ilimitado de vezes.
  - Um canal de comunicação pode perder mensagens, mas garantir que caso uma mensagem seja enviada infinitas vezes então também será recebida infinitas vezes.
- Normalmente distinguem-se dois tipos de justiça:

  Fraca Um traço é justo para uma determinada acção sse não for possível que esta esteja continuamente activa sem nunca executar.

  Forte Um traço é justo para uma determinada acção sse esta executar infinitas vezes caso esteja activa infinitas vezes.

Justiça em LTL vs CTL

- Para simplificar, vamos assumir que a condição de justiça pode ser expressa por um conjunto de fórmulas $f_1 \ldots f_n$ que devem ser válida infinitas vezes num caminho para que possa ser considerado justo.
- Para verificar uma fórmula LTL g na presença desta condição de justiça basta verificar a seguinte fórmula:

$$GF\ f_1 \wedge \ldots GF\ f_n \supset g$$

- No caso do CTL somos obrigados a mudar a semântica e adaptar os respectivos algoritmos de verificação.

$$M, s \models A_j\ h \Leftrightarrow \forall \pi \in M \cdot \text{ se } \pi_0 = s \text{ e } \mathbf{fair}(\pi) \text{ então } M, \pi \models h$$
$$M, s \models E_j\ h \Leftrightarrow \exists \pi \in M \cdot \text{ se } \pi_0 = s \text{ e } \mathbf{fair}(\pi) \text{ então } M, \pi \models h$$

- Sempre que se pretenda usar semântica justa será usado o subscrito j num operador temporal.

Verificação Directa com Justiça

- Para o operador EG_j basta adaptar o algoritmo por forma a excluir componentes que não sejam justos para com a condição de justiça.
- Um componente C é justo sse $\forall f_i \cdot C \cap [\![f_i]\!] \neq \emptyset$.
- Com esta adaptação, a fórmula fair $\equiv EG_j$ true será válida num estado s sse existir uma traço justo com início em s.
- Como um traço é justo sse qualquer seu sufixo for também justo então podemos reescrever os restantes operadores temporais da seguinte forma.

$$\begin{aligned} EX_j\ f &\equiv EX\ (f \wedge \text{fair}) \\ E[f\ U\ g]_j &\equiv E[f\ U\ (g \wedge \text{fair})] \end{aligned}$$

- Desta forma os algoritmos para determinar a validade destes operadores são idênticos.

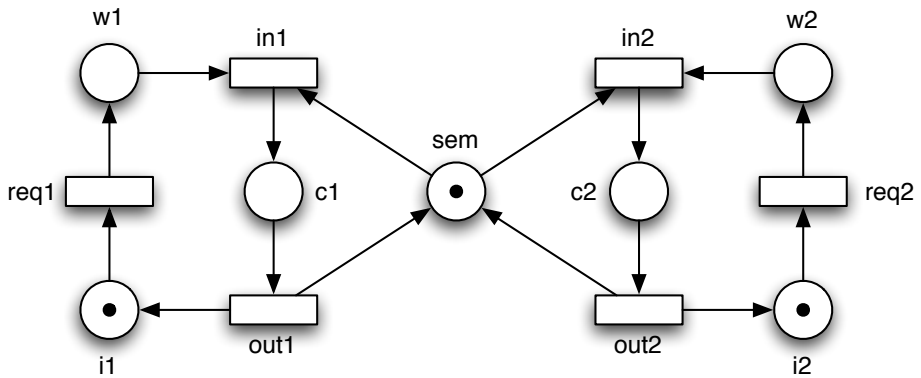
Introdução

- Na verificação simbólica os estados e a relação de acessibilidade da estrutura de Kripke são codificados através de fórmulas proposicionais.
- As fórmulas CTL são também codificadas através de fórmulas proposicionais, recorrendo à formulação dos operadores temporais como pontos fixos.
- Desta forma o problema da verificação da lógica CTL pode ser reduzido à verificação da lógica proposicional.
- Podem ser conseguidos ganhos de eficiência consideráveis pois não é necessário calcular *à priori* o conjunto de estados acessíveis.
- Recorrendo à codificação como *ordered binary decision diagrams* as fórmulas proposicionais podem ser verificadas de forma muito eficiente.

Codificação dos Estados

- Quando uma estrutura de Kripke resulta de uma rede elementar, os seus estados podem ser vistos como possíveis valorações (marcações) de variáveis boleanas (pertencentes ao conjunto L de lugares da rede).
- Para cada estado é possível definir uma fórmula que é válida apenas na respectiva marcação.

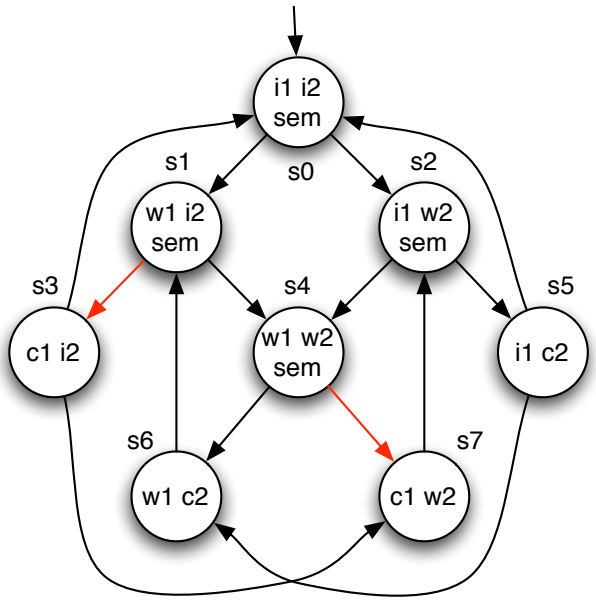
$$\phi_s \equiv (\wedge_{x \in s} x) \wedge (\wedge_{x \notin s} \neg x)$$



$$\phi_i \equiv i_1 \wedge \neg w_1 \wedge \neg c_1 \wedge sem \wedge i_2 \wedge \neg w_2 \wedge \neg c_2$$

Codificação das Transições

- Para poder representar uma transição (s, u) como uma fórmula proposicional é necessário criar um novo conjunto de variáveis L':
  - As variáveis de L são interpretadas no estado actual s.
  - As variáveis de L' são interpretadas no próximo estado u.
- Cada variável $x \in L$ tem uma correspondente $x' \in L'$.



$$\neg i_1 \wedge w_1 \wedge \neg c_1 \wedge sem \wedge i_2 \wedge \neg w_2 \wedge \neg c_2$$
$$\wedge$$
$$\neg i'_1 \wedge \neg w'_1 \wedge c'_1 \wedge \neg sem' \wedge i'_2 \wedge \neg w'_2 \wedge \neg c'_2$$

$$\neg i_1 \wedge w_1 \wedge \neg c_1 \wedge sem \wedge \neg i_2 \wedge w_2 \wedge \neg c_2$$
$$\wedge$$
$$\neg i'_1 \wedge \neg w'_1 \wedge c'_1 \wedge \neg sem' \wedge \neg i'_2 \wedge w'_2 \wedge \neg c'_2$$

Codificação das Transições

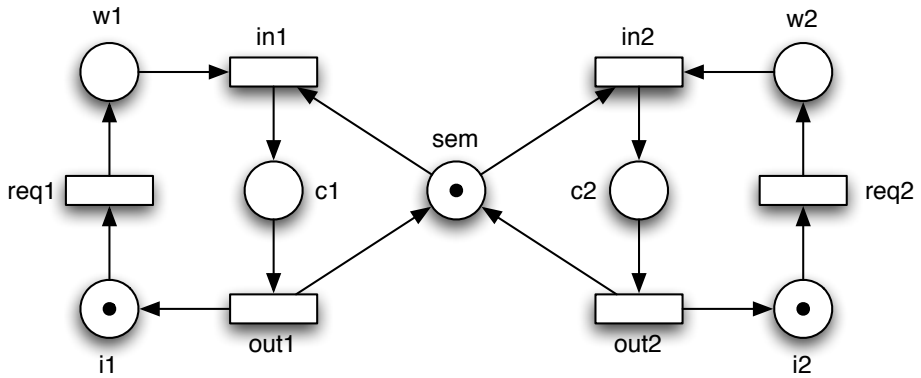
- A relação de acessibilidade é representada pela disjunção das fórmulas que codificam cada uma das transições.
- Usando lógica de primeira ordem como *açucar sintáctico* podemos ter uma codificação mais directa para cada uma das acções da rede: apenas são alteradas variáveis correspondentes a lugares na vizinhança, mantendo-se todas as restantes idênticas.

$$w_1 \wedge \neg c_1 \wedge sem \wedge \neg w_1' \wedge c_1' \wedge \neg sem'$$
$$\wedge$$
$$i_1' = i_1 \wedge i_2' = i_2 \wedge w_2' = w_2 \wedge c_2' = c_2$$

- Para facilitar a representação das variáveis que permanecem constantes vamos usar a seguinte abreviatura.

$$\overline{X} \equiv \bigwedge_{x \in X} (x' = x)$$

Exemplo



$$req_1 \equiv i_1 \wedge \neg w_1 \wedge \neg i'_1 \wedge w'_1 \wedge \overline{\{c_1, sem, i_2, w_2, c_2\}}$$

$$in_1 \equiv w_1 \wedge \neg c_1 \wedge sem \wedge \neg w'_1 \wedge c'_1 \wedge \neg sem' \wedge \overline{\{i_1, i_2, w_2, c_2\}}$$

$$out_1 \equiv c_1 \wedge \neg i_1 \wedge \neg sem \wedge \neg c'_1 \wedge i'_1 \wedge sem' \wedge \overline{\{w_1, i_2, w_2, c_2\}}$$

$$req_2 \equiv i_2 \wedge \neg w_2 \wedge \neg i'_2 \wedge w'_2 \wedge \overline{\{c_2, sem, i_1, w_1, c_1\}}$$

$$in_2 \equiv w_2 \wedge \neg c_2 \wedge sem \wedge \neg w'_2 \wedge c'_2 \wedge \neg sem' \wedge \overline{\{i_2, i_1, w_1, c_1\}}$$

$$out_2 \equiv c_2 \wedge \neg i_2 \wedge \neg sem \wedge \neg c'_2 \wedge i'_2 \wedge sem' \wedge \overline{\{w_2, i_1, w_1, c_1\}}$$

$$\mathcal{R} \equiv req_1 \vee in_1 \vee out_1 \vee req_2 \vee in_2 \vee out_2$$

Codificação das Transições

- Dada uma fórmula ϕ contendo apenas variáveis em L, a fórmula ϕ' obtém-se substituindo todas as variáveis $x \in L$ pelas correspondentes $x' \in L'$.
- É possível recuperar a relação de acessibilidade R a partir de $\mathcal{R}$ usando um dos seguintes métodos:
  - Dados $s, u \in S$ temos que $(s, u) \in R$ quando

$$\phi_s \wedge \phi'_u \supset \mathcal{R}$$

  - Usando s e u para definir o seguinte modelo $M : L \cup L' \to \mathbb{B}$ para a lógica proposicional

$$M(x) = (x \in L \wedge x \in s) \vee (x \in L' \wedge x \in u)$$

    então $(s, u) \in R$ quando

$$M \models \mathcal{R}$$

Codificação da Lógica CTL

- A verificação das conectivas clássicas da lógica CTL reduz-se à verificação da lógica proposicional.
- Para os operadores temporais *EG* e *EU* vamos mais uma vez usar a respectiva caracterização como maior e menor ponto fixo.

$$EG\ f \equiv \nu(\lambda z\,.\,f \wedge EX\ z)$$
$$E[f\ U\ g] \equiv \mu(\lambda z\,.\,g \vee (f \wedge EX\ z))$$

- Note que agora os pontos fixos são determinados sobre fórmulas proposicionais. Por exemplo, para determinar um maior ponto fixo começamos com a fórmula true e fazemos conjunções sucessivas até obter duas fórmula equivalentes em duas iterações consecutivas.

Codificação do Operador *EX*

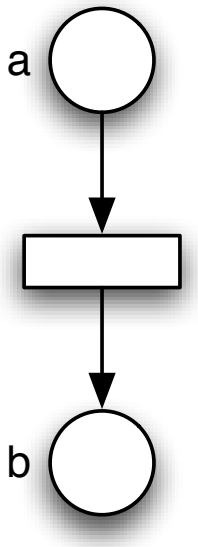
- A codificação do operador *EX* através de uma fórmula proposicional baseia-se na introdução (temporária) de um quantificador existencial sobre todas as variáveis $x' \in L'$.

$$[\![EX\ f]\!] = \exists \overline{x}' \cdot [\![f]\!]' \wedge \mathcal{R}$$

- Intuitivamente, esta fórmula será válida num modelo da lógica proposicional se existir alguma valoração para as variáveis $\overline{x}'$ acessível a partir dos actuais valores de $\overline{x}$ em que a fórmula f válida (quando interpretada no próximo estado).
- Para eliminar os quantificadores usamos a seguinte equivalência, em que $\phi|_{x \leftarrow v}$ é a fórmula que se obtém de ϕ substituindo x pelo valor v.

$$\exists x \cdot \phi \equiv \phi|_{x \leftarrow \text{true}} \vee \phi|_{x \leftarrow \text{false}}$$

Exemplo



$$\mathcal{R} \equiv (a \wedge \neg b \wedge \neg a' \wedge b') \vee (\neg a \wedge b \wedge \neg a' \wedge b')$$

$$\begin{aligned}
[\![EX\ b]\!] &\equiv \exists a', b' \cdot \mathcal{R} \wedge b' \\
&\equiv \exists a', b' \cdot \mathcal{R} \\
&\equiv \exists a' \cdot \mathcal{R}|_{b' \leftarrow \text{true}} \vee \mathcal{R}|_{b' \leftarrow \text{false}} \\
&\equiv \exists a' \cdot (a \wedge \neg b \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a') \\
&\equiv (a \wedge \neg b) \vee (\neg a \wedge b)
\end{aligned}$$

$$\begin{aligned}
[\![EX\ a]\!] &\equiv \exists a', b' \cdot \mathcal{R} \wedge a' \\
&\equiv \exists a', b' \cdot (a \wedge \neg b \wedge \neg a' \wedge b' \wedge a') \vee \ldots \\
&\equiv \text{false}
\end{aligned}$$

Motivação

- As técnicas de verificação simbólica assumem a existência de mecanismos eficientes para representar, manipular e validar fórmulas proposicionais.
- Os *Ordered Binary-Decision Diagrams* (OBDDs) são uma representação para fórmulas proposicionais onde essas operações podem ser implementadas de forma muito eficiente na maior parte dos casos.
- A ideia base desta representação consiste em impor restrições aos tradicionais *Binary-Decision Diagrams* (BDDs) por forma a conseguir representações canónicas para as fórmulas.

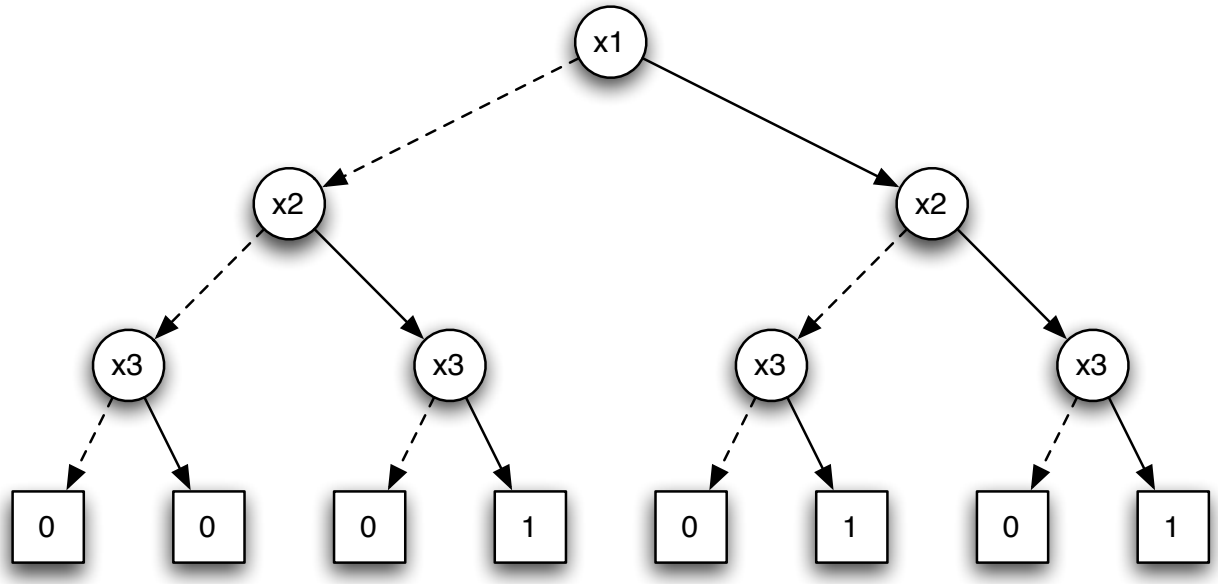
Binary-Decision Diagrams

- Um BDD representa uma expressão (ou função) booleana usando um *directed acyclic graph* (DAG) com uma única raíz.
- Se o DAG for uma árvore então temos uma *árvore de decisão*.
- Cada nodo não terminal v é etiquetado por uma variável booleana $var(v)$ e tem arcos orientados para dois filhos:
  - $lo(v)$ através de uma linha tracejada, correspondente ao caso em que a variável tem o valor 0; e
  - $hi(v)$ através de uma linha sólida, correspondente ao caso em que a variável tem o valor 1.
- Dada uma valoração para as variáveis, o valor da expressão é determinado seguindo o caminho respectivo desde a raíz até a um nodo terminal.

Exemplo

$$f \equiv (x_1 \vee x_2) \wedge x_3$$

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Ordered Binary-Decision Diagrams

- Num BDD *ordenado* é imposta uma ordenação total no conjunto de variáveis: qualquer caminho no grafo deve respeitar esta ordem.
- Qualquer ordenação é válida mas a eficiência depende da ordenação concreta escolhida.
- Para além de ordenado, também se assume que um OBDD é *reduzido*:
  - Unicidade: não podem existir dois nodos distintos com a mesma variável e os mesmos filhos.

$$var(u) = var(v) \wedge lo(u) = lo(v) \wedge hi(u) = hi(v) \Rightarrow u = v$$

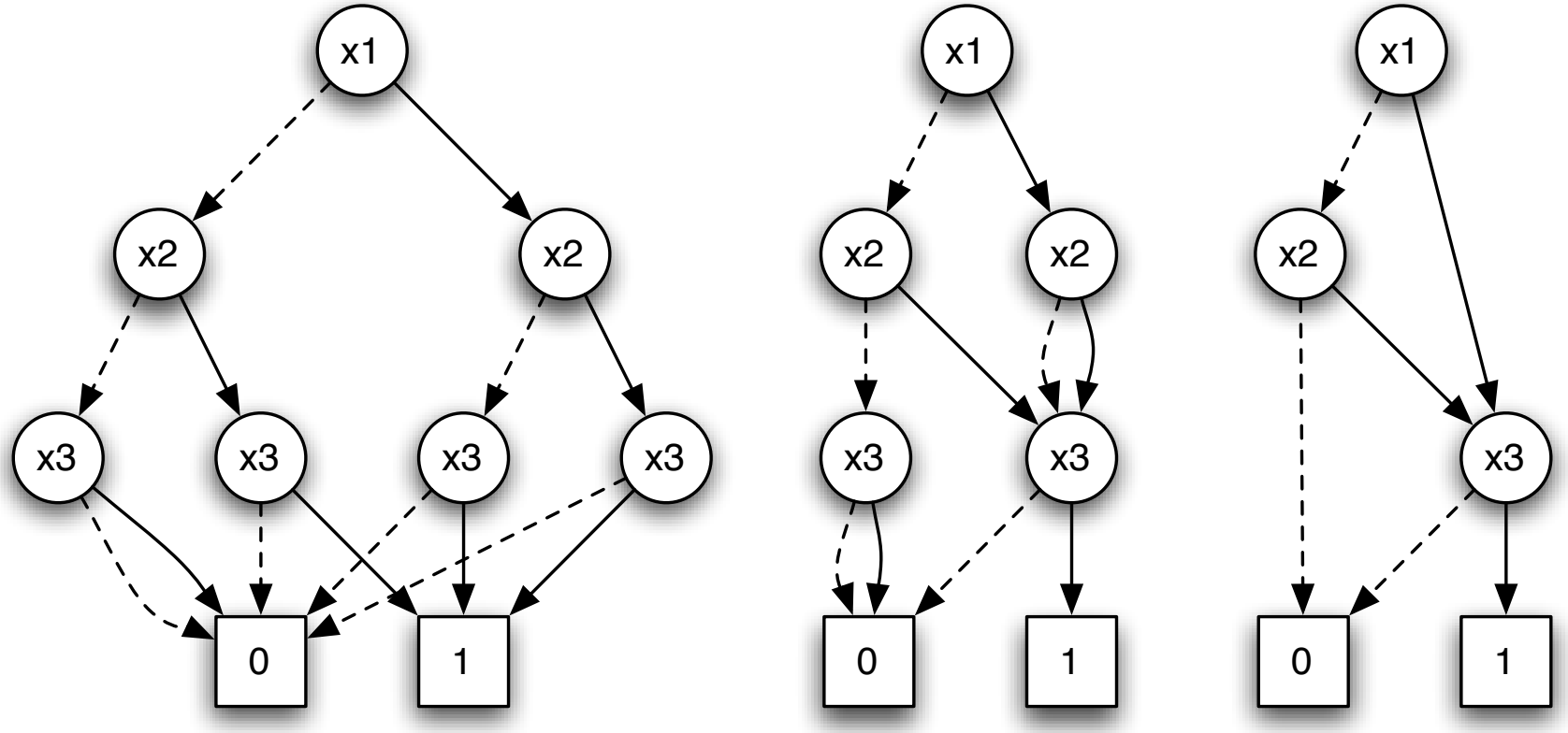
  - Não-redundância: nenhum nodo v pode ter filhos idênticos.

$$lo(v) \neq hi(v)$$

Redução de Binary-Decision Diagrams

- Dado um BDD *ordenado* é possível reduzi-lo a um OBDD através da aplicação sucessiva de uma das seguintes regras de transformação.
  - Remoção de terminais duplicados: deixar apenas um terminal para cada etiqueta e redireccionar todos arcos dos terminais eliminados para os restantes respeitando a etiquetagem.
  - Remoção de não-terminais duplicados: se dois nodos v e u verificam $var(v) = var(u)$, $lo(v) = lo(u)$ e $hi(v) = hi(u)$, então elimina-se um deles e redireccionam-se os seus arcos de entrada para o outro.
  - Remoção de testes redundantes: se um nodo v verifica $lo(v) = hi(v)$ então elimina-se v e redireccionam-se os seus arcos de entrada para $lo(v)$.
- Note que a aplicação de uma regra pode criar potencial para a aplicação de outras.

Exemplo de Redução



Representações Canónicas

- A representação de uma expressão booleana através de um OBDD é canónica: fixada uma ordenação das variáveis, dois OBDDs para a mesma função são necessariamente isomórficos.
- Este facto tem consequências importantes:
  - Testar a equivalência entre duas expressões é trivial.
  - Qualquer tautologia é representada através do OBDD com apenas um nodo terminal etiquetado com 1.
  - Uma expressão é satisfazível se não for representada através do OBDD com apenas um nodo terminal etiquetado com 0.
  - Se o valor the uma expressão é independente de uma variável x então o OBBD que a representa não pode conter x.

Efeito da Ordenação das Variáveis

- A forma e tamanho de um OBDD depende da ordenação imposta nas variáveis.
- A ordenação pode mudar a classe de complexidade da representação de forma significativa. Considere a seguinte expressão.

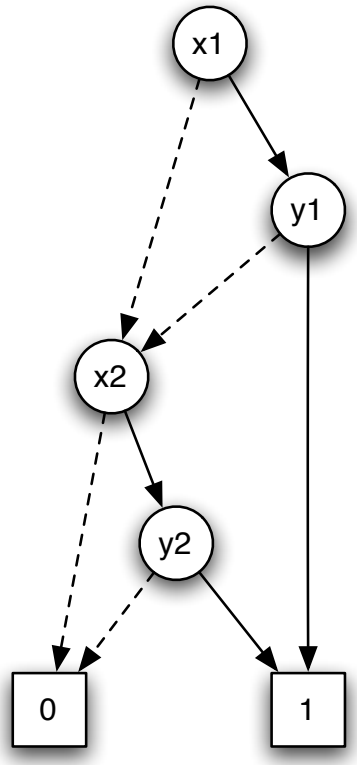
$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$$

  - Se ordenação das variáveis for $x_1 < y_1 < \ldots < x_n < y_n$ o número de nodos não terminais será $2n$.
  - Se a ordenação for $x_1 < \ldots < x_n < y_1 < \ldots < y_n$ o número de nodos é $2(2^n - 1)$.
- Para algumas classes de problemas existem métodos heurísticos que permitem escolher ordenações eficientes.

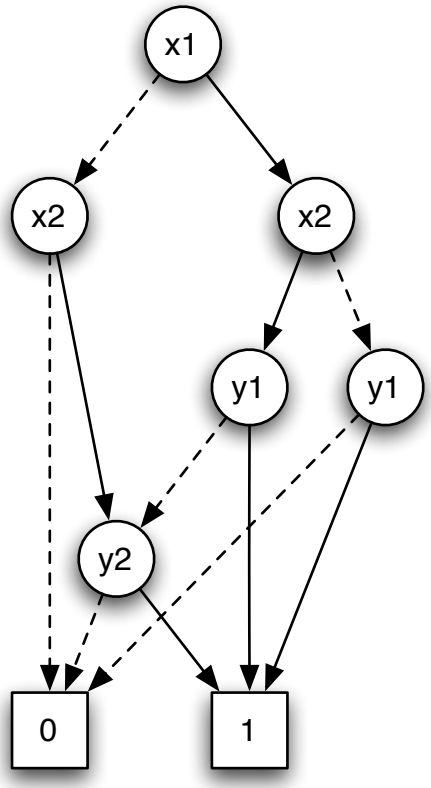
Exemplo

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2)$$

$x_1 < y_1 < x_2 < y_2$



$x_1 < x_2 < y_1 < y_2$



Expansão de Shannon

- Seja $x \to y, z$ o operador *if-then-else* definido como

$$x \to y, z \Leftrightarrow (x \wedge y) \vee (\neg x \wedge z)$$

- É possível redefinir qualquer expressão booleana usando apenas este operador e as constantes 0 e 1, garantindo adicionalmente que as variáveis aparecem não negadas e apenas nos testes.

$$\neg x \equiv x \to 0, 1$$
$$x \supset y \equiv x \to (y \to 1, 0), 1$$

- Esta forma normal corresponde a uma árvore de decisão e pode ser derivada usando a *expansão de Shannon*. Dada uma expressão f e uma variável x temos

$$f \equiv x \to f|_{x \leftarrow 1}, f|_{x \leftarrow 0}$$

Construção e Redução

- Fixando uma ordenação das variáveis e usando a expansão de Shannon temos um algoritmo recursivo para, dada uma expressão booleana, construir um BDD (uma árvore de decisão).
- Este BDD pode ser depois reduzido, usando as regras definidas anteriormente, por forma a obter um OBDD.
- Este método apenas pode ser usado para expressões com poucas variáveis, pois a árvore de decisão intermédia tem sempre um tamanho exponencial no número de variáveis.
- O ideal é ter um método que permita ir reduzindo o OBDD ao mesmo tempo que é construído.

Partilha de Representações

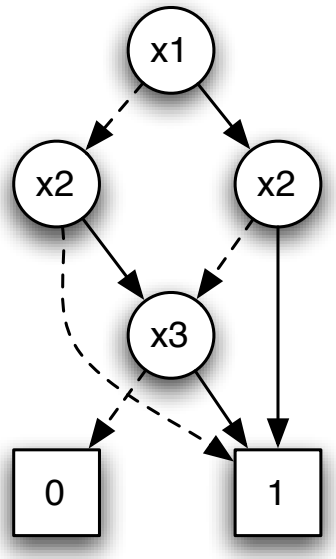
- Para conseguir uma representação eficiente de vários OBDDs simultaneamente usa-se um único grafo com vários pontos de entrada por forma a potenciar a partilha de subgrafos.
- Isto implica usar a mesma ordenação para as variáveis em todos eles.
- Adicionalmente, verificar a equivalência entre duas expressões equivale a testar se têm a mesma raiz.

Estruturas de Dados

- Os nodos de um OBDD serão identificados por um natural, estando os identificadores 0 e 1 reservados para os nodos terminais.
- Dada uma ordenação $x_1 < x_2 < \ldots < x_n$, as variáveis serão identificadas pelos seus índices.
- Um conjunto de OBDDs pode então ser armazenado numa tabela global T que associa cada identificador de nodo u a um triplo de naturais com $var(u)$, $lo(u)$ e $hi(u)$.
- Vamos assumir que esta tabela possui os seguintes métodos:
  - T.init() inicializa T com os nodos 0 e 1.
  - T.add(i, l, h) cria um novo nodo com os atributos respectivos e devolve o seu identificador.
  - T.var(u), T.lo(u) e T.hi(u) pesquisam os atributos do nodo identificado por u.

Estruturas de Dados

$$(x_1 \Leftrightarrow x_2) \vee x_3$$



u	*var*	*lo*	*hi*
0	4		
1	4		
2	3	0	1
3	2	1	2
4	2	2	1
5	1	3	4

- Esta tabela pode ser implementada com um *array* garantindo a execução das operações em tempo constante.
- Resta o problema de estimar o número de nodos que serão necessários.
- Para facilitar a implementação de alguns algoritmos assume-se que a variável de um nodo terminal é x_{n+1}.

Estruturas de Dados

- Para garantir que durante a sua construção um OBDD permanece reduzido, é necessário determinar a partir de um triplo (i, l, h) se existe algum nodo u em T tal que $var(u) = i$, $lo(u) = l$ e $hi(u) = h$.
- Para tal será mantida uma tabela global H que corresponde ao inverso de T, ou seja para cada nodo não terminal u temos

$$T(u) = (i, l, h) \quad \text{sse} \quad H(i, l, h) = u$$

- Nesta tabela são necessários os seguintes métodos:
  - $H.init()$ cria uma tabela vazia.
  - $H.member(i, l, h)$ verifica se o triplo dado pertence ao seu domínio.
  - $H.lookup(i, l, h)$ determina $H(i, l, h)$.
  - $H.insert(i, l, h, u)$ insere $(i, l, h) \mapsto u$.
- Usando uma tabela de hash como implementação estas funções podem ser implementadas em tempo constante.

Criação de Nodos

- A seguinte função apenas cria um nodo caso ele ainda não exista.
- Se os nodos apenas forem criados com esta função garante-se que o OBDD resultante é reduzido.
- O seu tempo de execução é constante.

mk$(i, l, h) \equiv$
  if $l = h$
    return l;
  if $H.member(i, l, h)$
    return $H.lookup(i, l, h)$;
  $u \leftarrow T.add(i, l, h)$;
  $H.insert(i, l, h, u)$;
  return u;

Construção Reduzida

- Usando a expansão de Shannon a seguinte função constroi um OBDD reduzido para uma expressão t.
- O uso de *mk* garante a partilha de subgrafos.
- O tempo de execução é exponencial no número de variáveis.

build $t \equiv$
    return aux$(t, 1)$;
aux$(t, i) \equiv$
    if $i > n$
        return $eval(t)$;
    $l \leftarrow aux(t|_{x_i \leftarrow 0}, i + 1)$;
    $h \leftarrow aux(t|_{x_i \leftarrow 1}, i + 1)$;
    return $mk(i, l, h)$;

Operações Binárias

- Dadas duas expressões booleanas f e g com nodos raiz u e v, respectivamente, o OBDD que representa $f \star g$ para um qualquer operador binário $\star$ será determinado pela função $apply(\star, u, v)$.
- Mais uma vez, este algoritmo baseia-se na expansão de Shannon. Se ambas as funções partilham uma variável x temos

$$f \star g \equiv x \rightarrow f|_{x \leftarrow 1} \star g|_{x \leftarrow 1}, f|_{x \leftarrow 0} \star g|_{x \leftarrow 0}$$

- Se g não depende de x então temos

$$f \star g \equiv x \rightarrow f|_{x \leftarrow 1} \star g, f|_{x \leftarrow 0} \star g$$

- Como o algoritmo é birecursivo, para evitar uma complexidade exponencial é usada uma tabela de hash G para fazer *memoization*.
- Note que a negação pode ser implementada como $\neg f \equiv f \oplus 1$.

Operações Binárias

apply$(\star, u, v) \equiv$
    $G.init()$; **return** aux(u, v);
aux$(u, v) \equiv$
    if $G.member(u, v)$
        return $G.lookup(u, v)$;
    if $u \in \{0, 1\} \wedge v \in \{0, 1\}$
        return $(u \star v)$;
    if $T.var(u) = T.var(v)$
        $w \leftarrow mk(T.var(u), aux(T.lo(u), T.lo(v)), aux(T.hi(u), T.hi(v)))$;
    if $T.var(u) < T.var(v)$
        $w \leftarrow mk(T.var(u), aux(T.lo(u), v), aux(T.hi(u), v))$;
    if $T.var(u) > T.var(v)$
        $w \leftarrow mk(T.var(v), aux(u, T.lo(v)), aux(u, T.hi(v)))$;
    $G.insert(u, v, w)$;
    return w;

Restringir

- Dada uma expressão f com raiz u, a função $restrict(u, i, b)$ determina $f|_{x_i \leftarrow b}$.
- Usando *memoization* a sua complexidade pode baixar para linear.

restrict$(u, i, b) \equiv$
    return aux u;
aux $u \equiv$
    if $T.var(u) > i$
        return u;
    if $T.var(u) < i$
        return $mk(T.var(u), res(T.lo(u)), res(T.hi(u)))$;
    if b
        return $T.hi(u)$;
    return $T.lo(u)$;