

# A Perspective on Model Checking

Manuel Alcino Cunha

Departamento de Informática  
Universidade do Minho

November 2007

# Motivation

- Safety-critical software is pervasive.
- Typically verified using simulation and testing.
- But how can we cover all possible interactions?
- Deductive verification (e.g. using theorem provers) is not cost-effective and requires a lot of expertise.
- For finite state systems model checking is a viable alternative: exhaustive search of the state space to check if a specification is valid.
- But how to deal with the state space explosion problem?

# The Process of Model Checking

**Modeling** Convert the design into a formalism accepted by the verification tool, abstracting irrelevant details. For reactive systems formalisms like *Petri nets* are popular.

**Specification** State the properties that the design must specify in some logical formalism. When we need to describe how the behavior of the system evolves over time *temporal logic* is a viable formalism.

**Verification** Check that the specified properties hold in the model. Ideally this process should be automatic. To tackle the state explosion problem, most tools rely on *symbolic model checking*.

# Modeling Reactive Systems

- Reactive systems interact frequently with their environment and usually do not terminate.
- Cannot be modeled by their input-output behavior.
- Key ingredients:
  - **States** Snapshots of the system variables.
  - **Transitions** Describe the effect of actions.
  - **Computations** Infinite sequences of states.
- *Kripke structures* provide the desirable level of abstraction to capture these ingredients.
- Other higher-level modeling languages can be compiled to Kripke structures.

# Kripke Structures

## Definition

Let  $A$  be a set of atomic propositions. A *Kripke structure* is a tuple:

$$(S, I, R, L)$$

where

- $S$  is a finite set of states.
- $I \subseteq S$  is the set of initial states.
- $R \subseteq S \times S$  is a total transition relation:

$$\forall s \in S \cdot R(s) \neq \emptyset, \text{ where } R(s) = \{s' \mid sRs'\}$$

- $L : S \rightarrow 2^A$  is a function that labels each state with the set of atomic propositions true in that state.

# Kripke Structures

- A path in a structure  $(S, I, R, L)$  starting in a state  $s$  is an infinite sequence of states  $\pi = s_0 s_1 s_2 \dots$ , such that  $s_0 = s$  and  $\forall i \geq 0 \cdot s_{i+1} \in R(s_i)$ .
- Given a path  $\pi$  its  $i$ -th state will be denoted by  $\pi_i$ .
- The suffix of  $\pi$  starting at its  $i$ -th state will be denoted by  $\pi^i$ .

# Petri Nets

- Modeling reactive systems directly in terms of *Kripke structures* is impractical.
- Even if an action is local to a component we must prescribe its effect in the global state.
- *Petri nets* allow us to model each component independently, leaving concurrency implicit.
- Unlike transition systems, where the modeling emphasis is on states, and algebraic methods, that focus on actions, Petri nets capture both.

## Key Ingredients of Petri Nets

- Places** Denote passive elements (e.g. variables, conditions, resources, channels) and are represented by circles.
- Transitions** Denote active elements (e.g. actions, events, instructions) and are represented by boxes.
- Arcs** Capture causality and are represented by arrows connecting places and transitions.



# Static Nets

## Definition

A *net* is a tuple

$$(P, T, F)$$

where

- $P$  is a set of places.
  - $T$  is a set of transitions, such that  $T \cap P = \emptyset$ .
  - $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation that captures arcs.
- 
- If  $P$  and  $T$  are finite the net is said to be finite.
  - Given  $a \in P \cup T$ ,  $\bullet a = F^{-1}(a)$  denotes its pre-set and  $a^\bullet = F(a)$  its pos-set.
  - This definitions trivially extend to sets.

# A Taxonomy of Petri Nets

**Elementary** Each place contains at most one token. A transition is enabled if all pre-conditions hold (i.e. are marked by a token) and no post-condition holds.

**Place/Transition** Places can have multiple tokens (optionally limited to a given capacity). Arcs can “carry” several tokens at once.

**Colored** Tokens may have different types. Arcs can restrict the type of tokens they carry.

# Elementary Nets

## Definition

An *elementary net* is a tuple

$$(P, T, F, M_0)$$

where

- $(P, T, F)$  is net.
- $M_0 \subseteq P$  is the initial marking.

# Dynamics of Elementary Nets

Given an elementary net  $(P, T, F, M_0)$ :

- Any subset  $M \subseteq P$  is a global state of  $\Sigma$ .
- A transition  $t$  is enabled in a given state  $M$  iff  $\bullet t \subseteq M$  and  $t^\bullet \cap (M - \bullet t) = \emptyset$ . This fact will be denoted by  $M \xrightarrow{t}$ .
- When a transition fires all tokens from the pre-conditions are consumed and tokens for all post-conditions are produced. Firing is atomic.
- The firing of  $t$  in state  $M$  leads to state  $M' = (M - \bullet t) \cup t^\bullet$ . This fact will be denoted by  $M \xrightarrow{t} M'$ .

# From Elementary Nets to Kripke Structures

- Given an elementary net  $(P, T, F, M_0)$  we can determine its semantics using Kripke structures as follows:

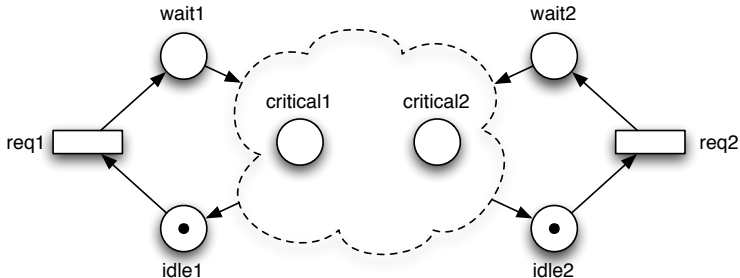
$$S = 2^P$$

$$I = \{M_0\}$$

$$R = \{(M, M') \mid \exists t \in T \cdot M \xrightarrow{t} M'\} \cup \{(M, M) \mid \nexists t \in T \cdot M \xrightarrow{t}\}$$

$$L = \text{id}$$

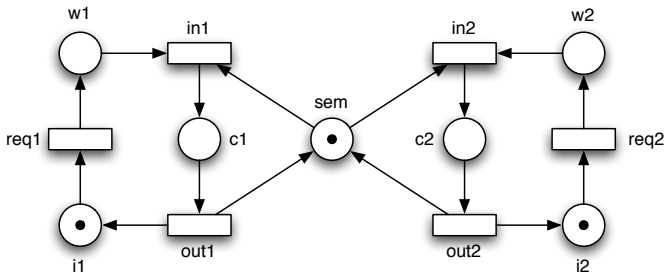
# Mutual Exclusion



**Mutual exclusion** Both agents cannot be simultaneously at their critical sections.

**Evolution** An agent cannot wait indefinitely to enter the critical section.

# Mutual Exclusion With Semaphores



- Mutual exclusion holds trivially.
- Evolution only holds if some external fairness restriction is imposed on the system.

# Temporal Logic

- Properties of reactive systems usually fall under two categories:
  - **Safety** A safety property states that “bad things” do not happen.
  - **Liveness** A liveness property states that “good things” do happen (eventually).
- Most safety properties can be easily stated directly on Kripke structures. For example, mutual exclusion:

$$\{c_1, c_2\} \notin R^*(M_0)$$

- But how to express safety properties like “an agent cannot be in its critical section without requesting it before”?



# Temporal Logic

- We can also state some animation properties directly on Kripke structures. For example, reversibility:

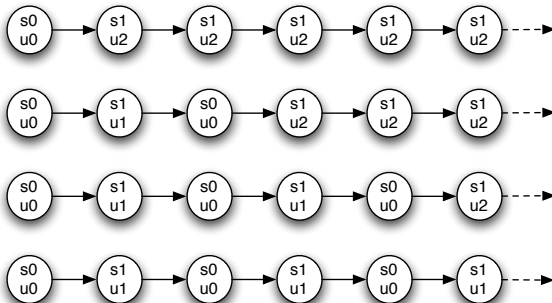
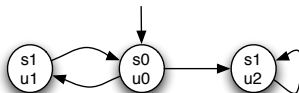
$$\forall s \in R^*(M_0) \cdot M_0 \in R^*(s)$$

- But how to express properties like evolution in mutual exclusion problems?
- We need a richer formalism in which to express properties that restrict the valid computations of the system.
- Temporal logic can be such formalism: although time is not mentioned explicitly, modal operators allow us to express rich causal orders within computations.
- Standard temporal logic is state oriented: the particular sequence of actions that lead to a computation is irrelevant.

# Models of Time

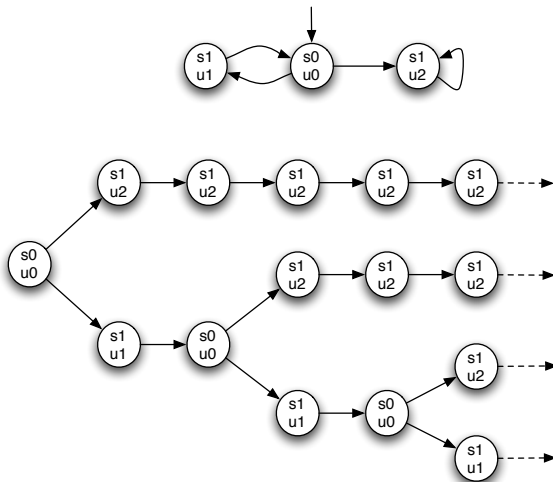
- There are two basic models of time in temporal logic:
  - Linear Time** The behavior of the system is the set of all infinite paths starting in initial states.
  - Branching Time** The behavior of the system is the set of all infinite computation trees unrolled from initial states.
- Both can be determined from a Kripke structure.

# Linear Time



■ ■ ■

# Branching Time



## CTL\*

- CTL\* is a branching time temporal logic:

*Full Computation Tree Logic*

- Besides classical operators, CTL\* has:

**Path quantifiers** Used to describe the branching structure in the computation tree.

**Temporal operators** Used to describe properties of a path through the tree.

- There are two type of formulas in CTL\*:

**State formulas** Which are true in a specific state.

**Path formulas** Which are true along a specific path.

# Path Quantifiers and Temporal Operators

- Path quantifiers:

$A f$   $f$  holds for all computation paths.

$E f$   $f$  holds for some computation path.

- Temporal operators:

$X f$   $f$  holds in the next state.

$F f$  Eventually (or in the future)  $f$  holds.

$G f$   $f$  always (or globally) holds.

$f U g$   $g$  eventually holds and until then  $f$  always holds.

$g R f$   $f$  holds up to a state where  $g$  holds, although  $g$  is not required to hold eventually.

- Temporal operators  $X$ ,  $F$ , and  $G$  are sometimes denoted using  $\bigcirc$ ,  $\diamond$ , and  $\square$ , respectively.

# Syntax

- Let  $A$  be the set of atomic propositions. State formulas are built from the following rules:
  - If  $p \in A$ , then  $p$  is a state formula.
  - If  $f$  and  $g$  are state formulas, then  $\neg f$ ,  $f \vee g$ ,  $f \wedge g$ , and  $f \supset g$  are state formulas.
  - If  $f$  is a path formula, then  $E f$ , and  $A f$  are state formulas.
- The syntax of path formulas is given by the following rules:
  - If  $f$  is a state formula, then  $f$  is also a path formula.
  - If  $f$  and  $g$  are path formulas, then  $\neg f$ ,  $f \vee g$ ,  $f \wedge g$ ,  $f \supset g$ ,  $X f$ ,  $F f$ ,  $G f$ ,  $f U g$ , and  $g R f$  are path formulas.

# Semantics of State Formulas

- We will define the semantics of CTL\* with respect to a Kripke structure  $M = (S, I, R, L)$ .
- If  $f$  is a state formula,  $M, s \models f$  means that  $f$  holds at state  $s$  in  $M$ . The relation  $\models$  is defined inductively as follows ( $p$  is an atomic proposition,  $f$  and  $g$  are state formulas, and  $h$  is a path formula):

$$M, s \models p \quad \Leftrightarrow \quad p \in L(s)$$

$$M, s \models \neg f \quad \Leftrightarrow \quad M, s \not\models f$$

$$M, s \models f \vee g \quad \Leftrightarrow \quad M, s \models f \text{ or } M, s \models g$$

$$M, s \models f \wedge g \quad \Leftrightarrow \quad M, s \models f \text{ and } M, s \models g$$

$$M, s \models f \supset g \quad \Leftrightarrow \quad M, s \not\models f \text{ or } M, s \models g$$

$$M, s \models \mathbf{A} h \quad \Leftrightarrow \quad \forall \pi \in M, \pi_0 = s \cdot M, \pi \models h$$

$$M, s \models \mathbf{E} h \quad \Leftrightarrow \quad \exists \pi \in M, \pi_0 = s \cdot M, \pi \models h$$



# Semantics of Path Formulas

- If  $f$  is a path formula,  $M, \pi \models f$  means that  $f$  holds along path  $\pi$  in  $M$ . The relation  $\models$  is defined inductively as follows ( $f$  and  $g$  are path formulas, and  $h$  is a state formula):

$$M, \pi \models h \quad \Leftrightarrow \quad M, \pi_0 \models h$$

$$M, \pi \models \neg f \quad \Leftrightarrow \quad M, \pi \not\models f$$

$$M, \pi \models f \vee g \quad \Leftrightarrow \quad M, \pi \models f \text{ or } M, \pi \models g$$

$$M, \pi \models f \wedge g \quad \Leftrightarrow \quad M, \pi \models f \text{ and } M, \pi \models g$$

$$M, \pi \models f \supset g \quad \Leftrightarrow \quad M, \pi \not\models f \text{ or } M, \pi \models g$$

$$M, \pi \models Xf \quad \Leftrightarrow \quad M, \pi^1 \models f$$

$$M, \pi \models Ff \quad \Leftrightarrow \quad \exists i \geq 0 \cdot M, \pi^i \models f$$

$$M, \pi \models Gf \quad \Leftrightarrow \quad \forall i \geq 0 \cdot M, \pi^i \models f$$

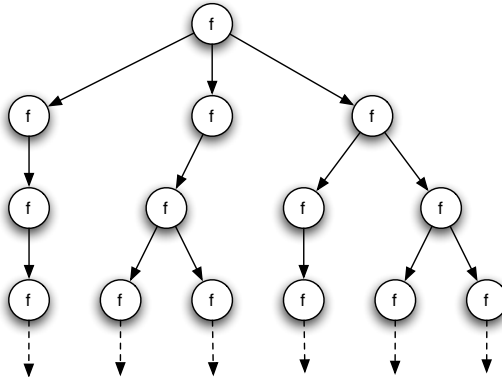
$$M, \pi \models f U g \quad \Leftrightarrow \quad \exists i \geq 0 \cdot M, \pi^i \models g \text{ and } \forall 0 \leq j < i \cdot M, \pi^j \models f$$

$$M, \pi \models g R f \quad \Leftrightarrow \quad \forall i \geq 0 \cdot (\exists 0 \leq j < i \cdot M, \pi^j \models g) \text{ or } M, \pi^i \models f$$

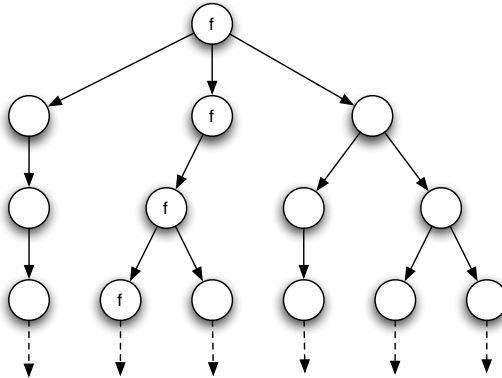
# LTL and CTL

- LTL is a linear time sublogic of CTL\* in which all formulas are of the form  $A f$ , where  $f$  is a path formula whose syntax is given by the following rules:
  - If  $p \in A$ , then  $p$  is a path formula.
  - If  $f$  and  $g$  are path formulas, then  $\neg f$ ,  $f \vee g$ ,  $f \wedge g$ ,  $f \supset g$ ,  $X f$ ,  $F f$ ,  $G f$ ,  $f U g$ , and  $g R f$  are path formulas.
- CTL is a branching time sublogic of CTL\* in which temporal operators must be immediately preceded by a path quantifier. Path formulas are restricted using the following rule:
  - If  $f$  and  $g$  are state formulas, then  $X f$ ,  $F f$ ,  $G f$ ,  $f U g$ , and  $g R f$  are path formulas.

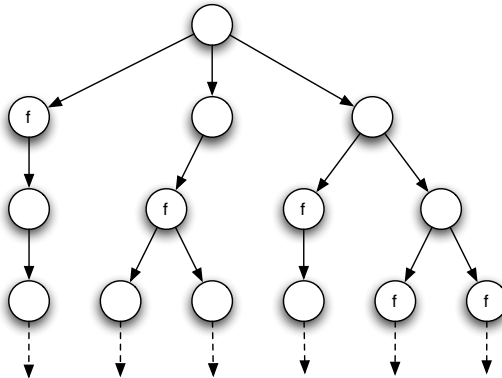
# Basic CTL operators: $AG f$



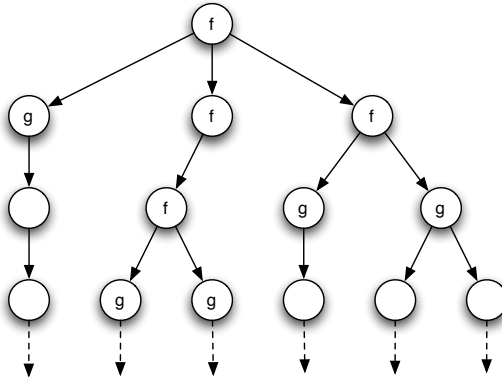
# Basic CTL operators: EG f



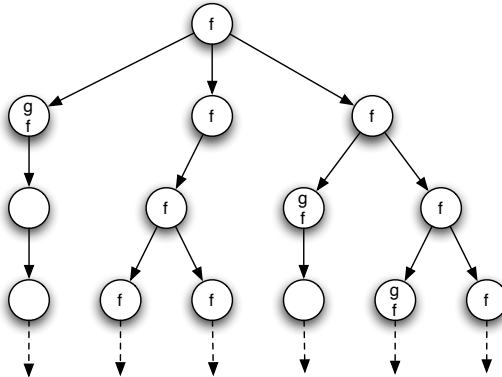
# Basic CTL operators: $AF f$



# Basic CTL operators: $f$ AU $g$



# Basic CTL operators: $g$ $AR$ $f$



## Minimal Set of CTL Operators

- All CTL formulas can be expressed using five operators:  $\neg$ ,  $\vee$ , EX, EU e EG.

$$f \wedge g \equiv \neg(\neg f \vee \neg g)$$

$$f \supset g \equiv \neg f \vee g$$

$$AX f \equiv \neg EX \neg f$$

$$EF f \equiv true \ EU f$$

$$AG f \equiv \neg EF \neg f$$

$$AF f \equiv \neg EG \neg f$$

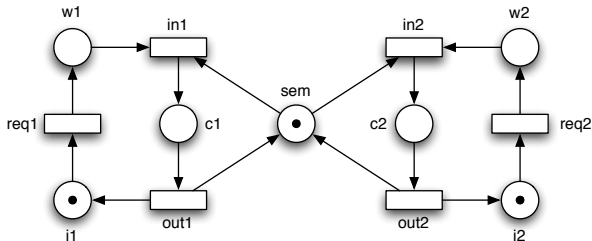
$$f AR g \equiv \neg(\neg f EU \neg g)$$

$$f ER g \equiv EG g \vee g EU (f \wedge g)$$

$$f AU g \equiv \neg(\neg f ER \neg g)$$



## Examples of CTL formulas



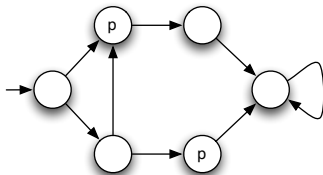
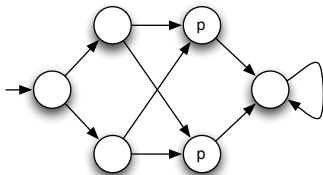
- Mutual exclusion:  $AG \neg(c_1 \wedge c_2)$
- Evolution:  $AG(w_1 \supset AF c_1) \wedge AG(w_2 \supset AF c_2)$
- Reversibility:  $AG EF(i_1 \wedge i_2 \wedge sem \wedge \dots)$
- No takeover:  $AG((w_1 \wedge i_2) \supset (c_1 AR \neg c_2)) \wedge \dots$

# LTL vs CTL

- Most properties can be expressed both in LTL and CTL, but the expressive power of both logics is incomparable.
- For example, reversibility cannot be expressed in LTL:

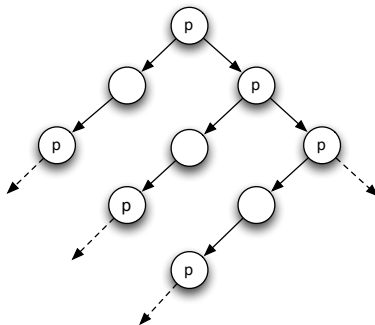
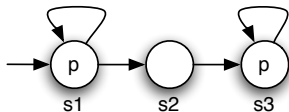
$AG\ EF\ init$

- LTL formulas are also not equivalent to the CTL formulas obtained by preceding each temporal operator by A. For example,  $AF\ AX\ p$  and  $FX\ p$  have different semantics.



# LTL vs CTL

- Although a computation tree is more expressive than a set of computations, there are properties that can only be expressed in LTL.
- For example,  $FG p$  cannot be expressed in CTL. Namely, its not equivalent to  $AF AG p$ .



# Model Checking

- We will focus on model checking techniques for CTL.
- Given a Kripke structure  $M = (S, I, R, L)$  and a CTL formula  $f$ , the goal of model checking is to find the set of all states in  $M$  that satisfy  $f$ :

$$\llbracket f \rrbracket_M \equiv \{s \in S \mid M, s \models f\}$$

- Formula  $f$  holds in a model  $M$  iff it holds in its initial states:

$$M \models f \Leftrightarrow I \subseteq \llbracket f \rrbracket_M$$

- Two different approaches to model checking:
  - Explicit** Based on an explicit enumeration and traversal of the Kripke structure.
  - Symbolic** When the Kripke structure is implicitly modeled by propositional formulas.

# Explicit Model Checking

- It suffices to handle six cases: atomic propositions and operators  $\neg$ ,  $\vee$ , EX, EG, and EU.
- Given a Kripke structure  $M = (S, I, R, L)$ , an atomic proposition  $p$ , and state formulas  $f$  and  $g$  we have:

$$\llbracket p \rrbracket_M = L^{-1}(p) = \{s \in S \mid p \in L(s)\}$$

$$\llbracket \neg f \rrbracket_M = S - \llbracket f \rrbracket_M$$

$$\llbracket f \vee g \rrbracket_M = \llbracket f \rrbracket_M \cup \llbracket g \rrbracket_M$$

- The states that satisfy EX  $f$  are the predecessors of states that satisfy  $f$ :

$$\llbracket \text{EX } f \rrbracket_M = R^{-1}(\llbracket f \rrbracket_M) \equiv \{s \in S \mid \exists t \in S \cdot (s, t) \in R \wedge t \in \llbracket f \rrbracket_M\}$$

## Model Checking EG: A Naive Approach

- A recursive algorithm to model check EG  $f$  can be derived from its expansion law:

$$\text{EG } f \equiv f \wedge \text{EX EG } f$$

- $\llbracket \text{EG } f \rrbracket_M$  is the largest solution to the following recursive equation in the domain  $(2^S, \subseteq)$ .

$$X = \llbracket f \rrbracket_M \cap R^{-1}(X)$$

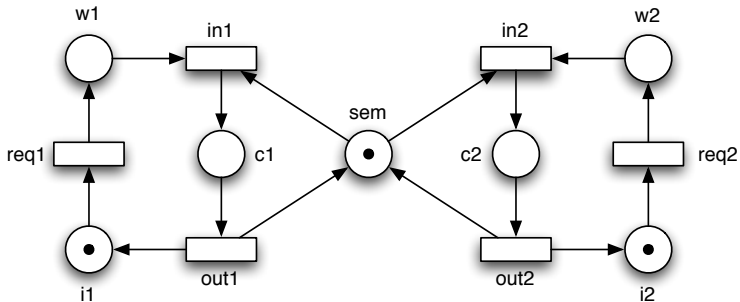
- Alternatively,  $\llbracket \text{EG } f \rrbracket_M$  is the largest fixpoint of  $\Pi : 2^S \rightarrow 2^S$ :

$$\llbracket \text{EG } f \rrbracket_M = \nu(\Pi), \text{ where } \Pi(X) = \llbracket f \rrbracket_M \cap R^{-1}(X)$$

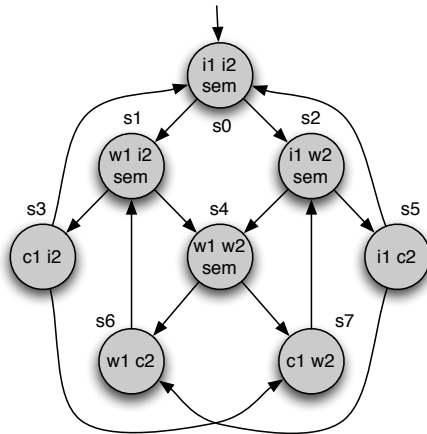
- This set can be computed as the limit of the following series:

$$S, \Pi(S), \Pi(\Pi(S)), \Pi(\Pi(\Pi(S))), \dots$$

# Example: $\llbracket EG w_1 \rrbracket$



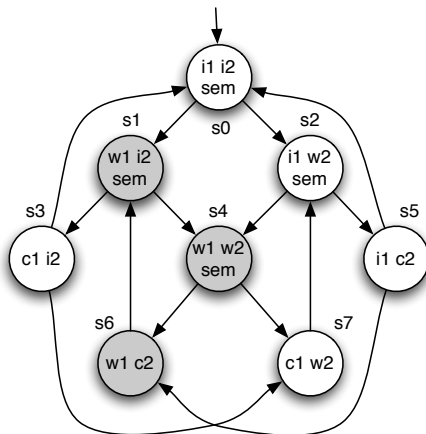
# Example: $\llbracket EG w_1 \rrbracket$



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

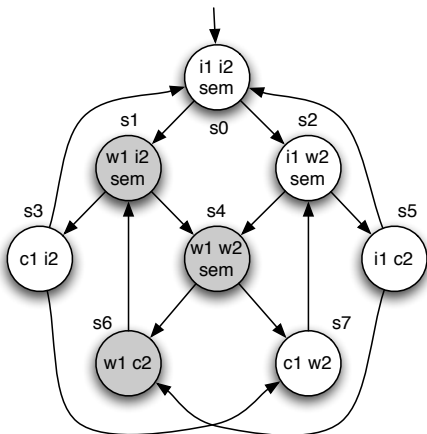


# Example: $\llbracket EG w_1 \rrbracket$



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

Example:  $\llbracket EG w_1 \rrbracket = \{s_1, s_4, s_6\}$



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

## Model Checking EU: A Naive Approach

- Similarly, we can derive a recursive algorithm to model check  $f \text{ EU } g$  from its expansion law:

$$f \text{ EU } g \equiv g \vee (f \wedge \text{EX}(f \text{ EU } g))$$

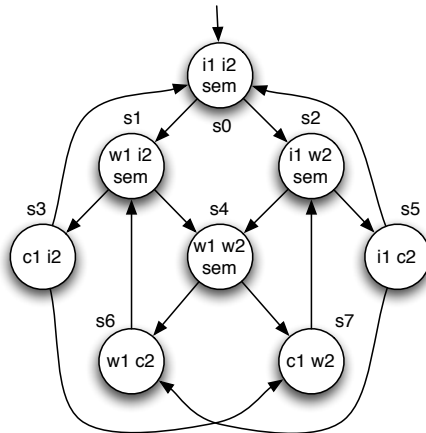
- $\llbracket f \text{ EU } g \rrbracket_m$  is the least fixpoint of  $\Pi : 2^S \rightarrow 2^S$ .

$$\llbracket f \text{ EU } g \rrbracket_M = \mu(\Pi), \text{ where } \Pi(X) = \llbracket g \rrbracket_M \cup (\llbracket f \rrbracket_M \cap R^{-1}(X))$$

- This set can be computed as the limit of the following series:

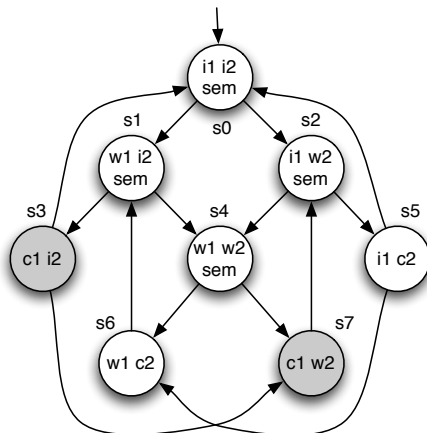
$$\emptyset, \Pi(\emptyset), \Pi(\Pi(\emptyset)), \Pi(\Pi(\Pi(\emptyset))), \dots$$

# Example: $\llbracket w_1 \text{ EU } c_1 \rrbracket$



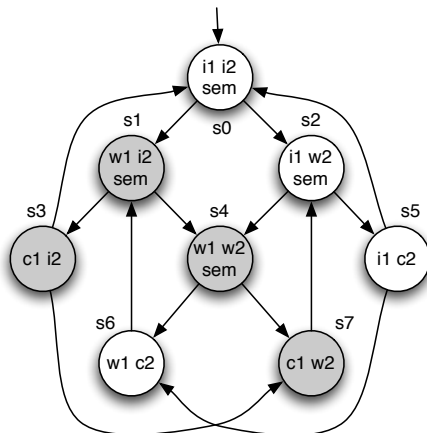
$$\Pi^0(\emptyset) = \emptyset$$

# Example: $\llbracket w_1 \text{ EU } c_1 \rrbracket$



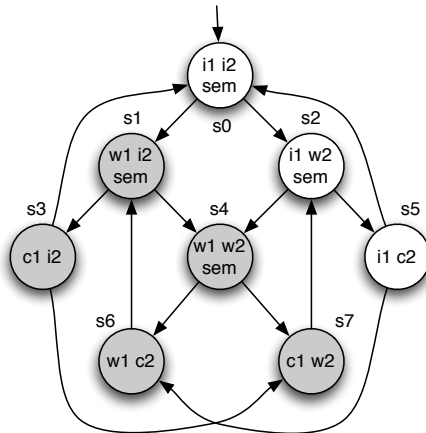
$$\Pi^1(\emptyset) = \llbracket c_1 \rrbracket \cup (\llbracket w_1 \rrbracket \cap R^{-1}(\Pi^0(\emptyset))) = \{s_3, s_7\}$$

# Example: $\llbracket w_1 \text{ EU } c_1 \rrbracket$



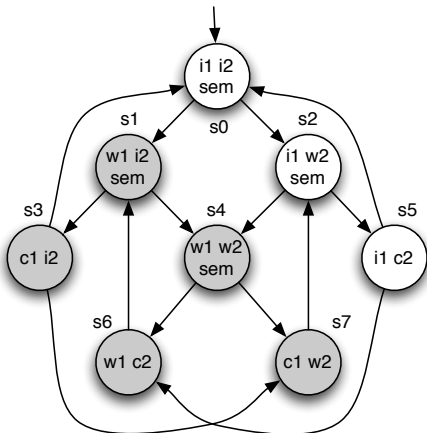
$$\Pi^2(\emptyset) = \llbracket c_1 \rrbracket \cup (\llbracket w_1 \rrbracket \cap R^{-1}(\Pi^1(\emptyset))) = \{s_1, s_3, s_4, s_7\}$$

# Example: $\llbracket w_1 \text{ EU } c_1 \rrbracket$



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

Example:  $\llbracket w_1 \text{ EU } c_1 \rrbracket = \{s_1, s_3, s_4, s_6, s_7\}$

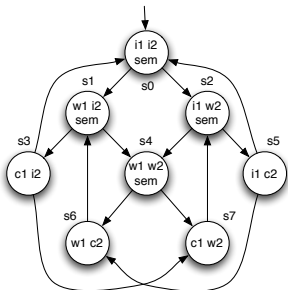


$$\Pi^4(\emptyset) = \llbracket c_1 \rrbracket \cup (\llbracket w_1 \rrbracket \cap R^{-1}(\Pi^3(\emptyset))) = \{s_1, s_3, s_4, s_6, s_7\}$$



# Example: $M \models AG \neg(c_1 \wedge c_2)$

$$AG \neg(c_1 \wedge c_2) \equiv \neg EF \neg\neg(c_1 \wedge c_2) \equiv \neg EF(c_1 \wedge c_2) \equiv \neg(\text{true}EU(c_1 \wedge c_2))$$



$$\llbracket c_1 \rrbracket = \{s_3, s_7\}$$

$$\llbracket c_2 \rrbracket = \{s_5, s_6\}$$

$$\llbracket c_1 \wedge c_2 \rrbracket = \emptyset$$

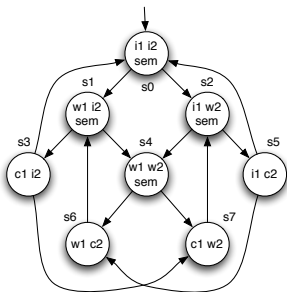
$$\llbracket \text{true}EU(c_1 \wedge c_2) \rrbracket^0 = \emptyset$$

$$\llbracket \text{true}EU(c_1 \wedge c_2) \rrbracket^1 = S \cap (\llbracket c_1 \wedge c_2 \rrbracket \cup R^{-1}(\emptyset))$$

$$\llbracket \neg(\text{true}EU(c_1 \wedge c_2)) \rrbracket = S$$

# Example: $M \not\models \text{AG}(w_1 \supset \text{AF } c_1)$

$$\begin{aligned} \text{AG}(w_1 \supset \text{AF } c_1) &\equiv \text{AG}(\neg w_1 \vee \text{AF } c_1) \equiv \neg \text{EF } \neg(\neg w_1 \vee \text{AF } c_1) \equiv \\ &\neg \text{EF}(w_1 \wedge \neg \text{AF } c_1) \equiv \neg \text{EF}(w_1 \wedge \neg \text{EG } \neg c_1) \equiv \neg \text{EF}(w_1 \wedge \text{EG } \neg c_1) \end{aligned}$$



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

# Complexity Issues

- Given a CTL formula  $f$  and a Kripke structure  $M = (S, I, R, L)$ , the naive model checking algorithm presented above has complexity

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

- With some clever tricks it is possible to lower the complexity to

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

# Model Checking $f \text{ EU } g$

- To compute  $\llbracket f \text{ EU } g \rrbracket$  we start from set  $\llbracket g \rrbracket$  and successively add predecessors that satisfy  $f$ :

```
checkEU ( $\llbracket f \rrbracket$ ,  $\llbracket g \rrbracket$ )  $\equiv$   
   $T \leftarrow \llbracket g \rrbracket$ ;  
   $\llbracket f \text{ EU } g \rrbracket \leftarrow \llbracket g \rrbracket$ ;  
  while  $T \neq \emptyset$   
    choose  $s \in T$ ;  
     $T \leftarrow T - \{s\}$ ;  
    for  $t \in R^{-1}(s)$   
      if  $t \notin \llbracket f \text{ EU } g \rrbracket \wedge t \in \llbracket f \rrbracket$   
         $\llbracket f \text{ EU } g \rrbracket \leftarrow \llbracket f \text{ EU } g \rrbracket \cup \{t\}$ ;  
         $T \leftarrow T \cup \{t\}$ ;  
  return  $\llbracket f \text{ EU } g \rrbracket$ ;
```

## Model Checking EG $f$

- Given a Kripke structure  $M = (S, I, R, L)$ , to model check EG  $f$  it suffices to restrict  $M$  to the states that satisfy  $f$ :

$$M_f = (\llbracket f \rrbracket, I \cap \llbracket f \rrbracket, R \cap (\llbracket f \rrbracket \times \llbracket f \rrbracket), L|_{\llbracket f \rrbracket})$$

### Lemma

$M, s \models \text{EG } f$  iff  $s \in \llbracket f \rrbracket$  and there exists a path in  $M_f$  from  $s$  to some node  $t$  in a *nontrivial strongly connected component* of  $M_f$ .

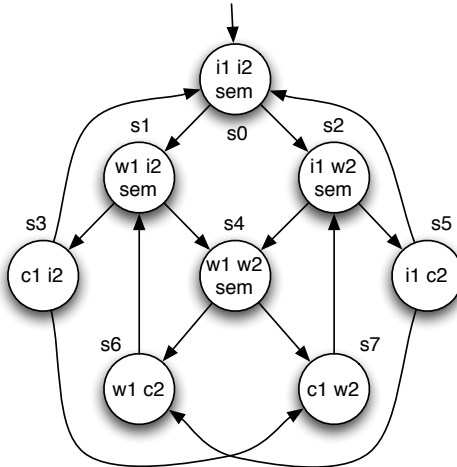
- A SCC (*strongly connected component*)  $C$  is a maximal subgraph where every node is reachable from every other node along a directed path entirely contained in  $C$ .
- $C$  is also *nontrivial* iff it has more than one node or it contains one node with a self-loop.

# Model Checking EG $f$

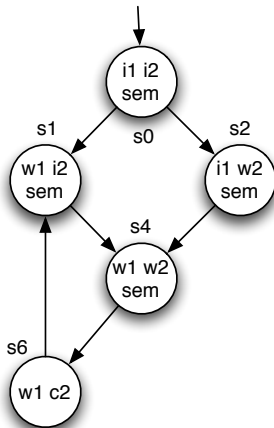
- To compute  $\llbracket \text{EG } f \rrbracket$  we first compute all states belonging to nontrivial SCCs and successively add all predecessors in  $\llbracket f \rrbracket$ .
- **scc** uses Tarjan algorithm with complexity  $O(\llbracket f \rrbracket + |R_f|)$ .

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

# Example: $EG(w_1 \vee sem)$

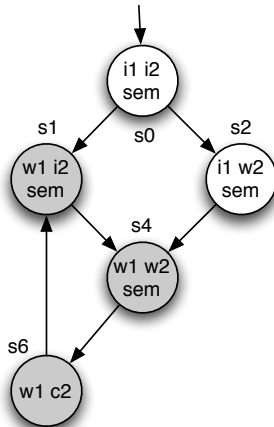


# Example: $EG(w_1 \vee sem)$

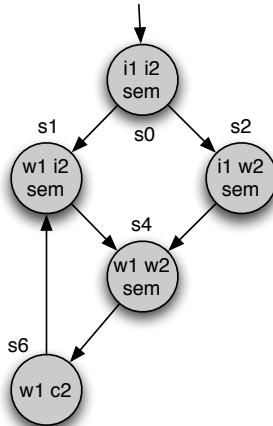




# Example: $EG(w_1 \vee sem)$



# Example: $EG(w_1 \vee sem)$



# Fairness

- Some liveness properties can only be satisfied assuming that some kind of fairness constraints hold in the system.
- For example, evolution in the mutual exclusion with a semaphore only holds if we assume that an agent cannot stay indefinitely in the waiting state.
- In action oriented specification logics, fairness constraints are usually divided in two categories:
  - Weak** A computation is weakly fair to an action iff it cannot be continuously enabled without ever executing.
  - Strong** A computation is strongly fair to an action iff it executes infinitely often whenever it is enabled infinitely often.

# Fairness in CTL

- In CTL we can model fairness constraints by a set of formulas  $f_1, \dots, f_n$  that must hold infinitely often in a path for it to be considered fair.
- A fair Kripke structure is tuple  $M = (S, I, R, L, F)$ , where  $F \subseteq 2^S = \{\llbracket f_1 \rrbracket, \dots, \llbracket f_n \rrbracket\}$ .
- Assuming that  $\text{inf}(\pi)$  extracts all states that occur infinitely often in  $\pi$ , we can define a predicate to test for fairness as follows:

$$\text{fair}(\pi) = \forall P \in F \cdot \text{inf}(\pi) \cap P \neq \emptyset$$

- Semantics of CTL can be easily adapted to capture fairness.

$$M, s \models_F p \quad \Leftrightarrow \quad \exists \pi \in M, \text{fair}(\pi), \pi_0 = s \cdot p \in L(s)$$

$$M, s \models_F A h \quad \Leftrightarrow \quad \forall \pi \in M, \text{fair}(\pi), \pi_0 = s \cdot M, \pi \models h$$

$$M, s \models_F E h \quad \Leftrightarrow \quad \exists \pi \in M, \text{fair}(\pi), \pi_0 = s \cdot M, \pi \models h$$

## Direct Model Checking With Fairness

- To model check the operator EG it suffices to restrict the model to fair SCCs. An SCC is fair iff

$$\forall P \in F \cdot C \cap P \neq \emptyset$$

- Assuming a fair semantics, the formula EG *true* holds in a state *s* iff there is a fair path starting from *s*.
- Given that a path is fair iff any of its suffixes is fair, we can reuse the standard model checking algorithms as follows:

$$\begin{aligned}M, s \models_F p &\equiv M, s \models p \wedge \text{EG } true \\M, s \models_F \text{EX } f &\equiv M, s \models \text{EX}(f \wedge \text{EG } true) \\M, s \models_F f \text{EU } g &\equiv M, s \models f \text{EU } (g \wedge \text{EG } true)\end{aligned}$$

- Complexity of model checking under fairness raises to

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

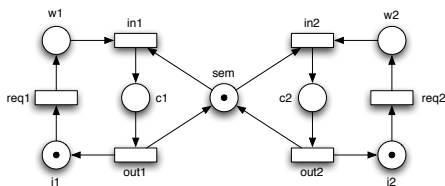
# Symbolic Model Checking

- Although explicit model checking is rather efficient it cannot cope with the state explosion that occurs in many reactive systems.
- Symbolic model checking tackles this problem by avoiding the explicit construction of the state space: the states and the transition relation of a Kripke structure are captured by propositional formulas.
- CTL formulas can also be encoded in propositional logic thanks to the fixpoint definition of temporal operators.
- Model checking of CTL formulas is reduced to checking the validity of propositional formulas.
- This can be done very efficiently by using techniques like *Ordered Binary Decision Diagrams*.

## Encoding states

- When a Kripke structure is derived from an elementary net its states can be seen as models (valuations) of propositional logic, with variables taken from the set of places  $P$ .
- For each state  $s$  it is possible to define a formula that is valid only in the respective model.

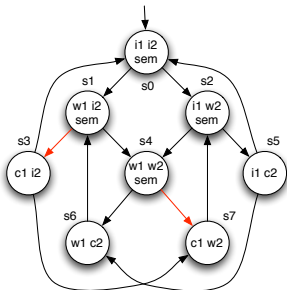
$$\phi_s \equiv (\bigwedge_{X \in S} X) \wedge (\bigwedge_{X \notin S} \neg X)$$



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

# Encoding Transitions

- A similar technique can be used to encode transitions, but we need an additional set of variables  $P'$ : each variable  $x \in P$  has a corresponding next state variable  $x' \in P'$ .
- Models will now be ordered pairs of states  $(s, s')$ : variables in  $P$  should be valued in  $s$ , while variables in  $P'$  should be valued in  $s'$ .



$$\neg i_1 \wedge w_1 \wedge \neg c_1 \wedge \mathbf{sem} \wedge i_2 \wedge \neg w_2 \wedge \neg c_2$$

$$\wedge$$

$$\neg i'_1 \wedge \neg w'_1 \wedge c'_1 \wedge \neg \mathbf{sem}' \wedge i'_2 \wedge \neg w'_2 \wedge \neg c'_2$$

$$\neg i_1 \wedge w_1 \wedge \neg c_1 \wedge \mathbf{sem} \wedge \neg i_2 \wedge w_2 \wedge \neg c_2$$

$$\wedge$$

$$\neg i'_1 \wedge \neg w'_1 \wedge c'_1 \wedge \neg \mathbf{sem}' \wedge \neg i'_2 \wedge w'_2 \wedge \neg c'_2$$



## Encoding Transitions

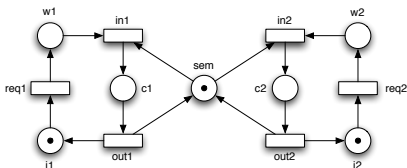
- The formula  $\phi_R$  that encodes the transition relation is the disjunction of all formulas that encode each individual transition.
- With a bit of syntactic sugar we can have a direct encoding for each transition in the net: only variables in the neighborhood of a transition are affected.

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

- The following notation will be used to model the variables whose value is not affected by a transition.

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

# Example



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

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

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

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

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

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

$$\phi_R \equiv req_1 \vee in_1 \vee out_1 \vee req_2 \vee in_2 \vee out_2$$

# Symbolic Model Checking for CTL

- The set of states  $\llbracket f \rrbracket$  where a formula  $f$  is valid is no longer represented extensionally: instead it is represented by a propositional formula that is valid precisely in those states.
- This means that classical connectives are no longer encoded in terms of set operations.
- The validity of temporal operators EG and EU will again be determined by fixpoints:

$$\llbracket \text{EG } f \rrbracket = \nu(\Pi), \text{ where } \Pi(h) = \llbracket f \rrbracket \wedge \llbracket \text{EX } h \rrbracket$$

$$\llbracket f \text{ EU } g \rrbracket = \mu(\Pi), \text{ where } \Pi(h) = \llbracket g \rrbracket \vee (\llbracket f \rrbracket_M \wedge \llbracket \text{EX } h \rrbracket)$$

- Notice that fixpoints are now computed symbolically: for example, to compute a least fixpoint we start with formula *false* and perform disjunctions until two equivalent formulas are computed in successive iterations.

## Model Checking EX $f$

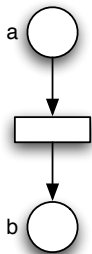
- To model check EX  $f$  an (temporary) existential quantifier is used.

$$\llbracket \text{EX } f \rrbracket = \exists \bar{x}' \cdot \llbracket f \rrbracket' \wedge \phi_R$$

- $\llbracket f \rrbracket'$  is the formula obtained from  $\llbracket f \rrbracket$  by replacing all variables  $x \in P$  by the corresponding  $x' \in P'$ .
- Intuitively,  $\llbracket \text{EX } f \rrbracket$  will be valid in a state  $s$  if there is some valuation to all variables  $x' \in P'$  accessible from  $s$  in which  $f$  is valid.
- The existential quantifier is then eliminated by using the following expansion:

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

# Example



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

$$\begin{aligned} \llbracket EX\ b \rrbracket &\equiv \exists a', b' \cdot \phi_R \wedge b' \\ &\equiv \exists a', b' \cdot \phi_R \\ &\equiv \exists a' \cdot \phi_R|_{b' \leftarrow true} \vee \phi_R|_{b' \leftarrow false} \\ &\equiv \exists a' \cdot (a \wedge \neg b \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a') \\ &\equiv (a \wedge \neg b) \vee (\neg a \wedge b) \end{aligned}$$

$$\begin{aligned} \llbracket EX\ a \rrbracket &\equiv \exists a', b' \cdot \phi_R \wedge a' \\ &\equiv \exists a', b' \cdot (a \wedge \neg b \wedge \neg a' \wedge b' \wedge a') \vee \dots \\ &\equiv false \end{aligned}$$

# Ordered Binary Decision Diagrams

- For symbolic model checking to be effective we need efficient mechanisms to represent, manipulate, and validate propositional formulas.
- *Ordered Binary Decision Diagrams* (OBDDs) are a canonical representation for propositional formulas, where the model checking operations can be implemented efficiently.
- This representation imposes additional constraints on traditional *Binary Decision Diagrams* (BDDs) to achieve canonical forms.

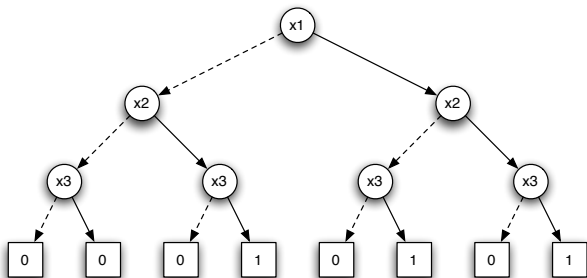
# Binary Decision Diagrams

- A BDD represents a boolean function by a *Directed Acyclic Graph* (DAG) with a single root.
- If the DAG is a tree we have a *Binary Decision Tree*.
- Each terminal node  $v$  is either 0 or 1.
- Each nonterminal node  $v$  is labeled by a variable  $var(v)$  and has two successors:
  - $low(v)$  corresponding to the case where  $v$  is assigned 0.
  - $high(v)$  corresponding to the case where  $v$  is assigned 1.
- Given a valuation for the variables, the value of the formula can be determined by traversing the tree from the root to a terminal node.

# Example

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

$x_1$	$x_2$	$x_3$	$f$
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





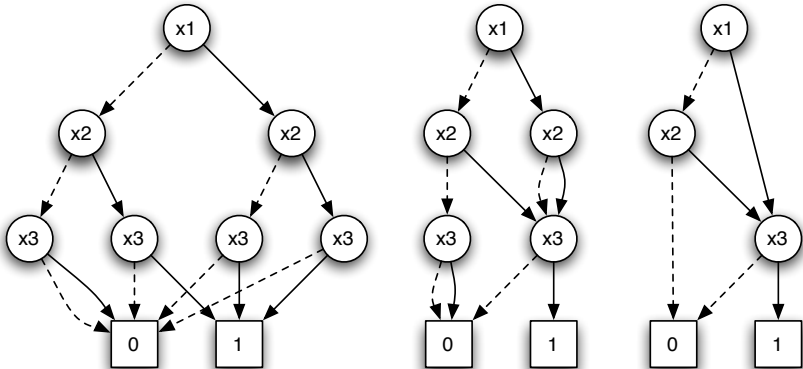
# Ordered Binary Decision Diagrams

- In a ordered BDD a total ordering is imposed on the variables: each path in the graph must respect this order.
- Any ordering is possible, but size of an OBDD can vary significantly with the particular order chosen.
- Besides being ordered, an OBDD must be *reduced*:
  - No nodes with equal successors.
  - No duplicate subtrees.

# Reducing Binary Decision Diagrams

- Given an ordered BDD we can reduce it to an OBDD by successively applying the following transformation rules:
  - *Remove duplicate terminals*: delete all but one terminal with a given value, and redirect all arcs pointing to deleted terminals to the remaining one.
  - *Remove duplicate nonterminals*: if two nodes  $u$  and  $v$  have  $var(u) = var(v)$ ,  $low(u) = low(v)$ , and  $high(u) = high(v)$ , delete one of them and redirect all incoming arcs to the other.
  - *Remove redundant tests*: if nonterminal  $v$  has  $low(v) = high(v)$ , delete  $v$  and redirect all incoming arcs to  $low(v)$ .
- Reducing an ordered BDD can be done in a bottom-up manner by a procedure linear in its size.

# Example



# The Utility of Canonical Representations

- The representation of a boolean function by an OBDD is canonical: given a particular variable ordering two OBDDs that represent the same function are necessarily isomorphic.
- This fact has important consequences for model checking:
  - Checking equivalence is reduced to checking isomorphism.
  - Any tautology is equivalent to the OBDD with a single (terminal) node labeled 1.
  - A formula is satisfiable if its not equivalent to the OBDD with a single (terminal) node labeled 0.
  - If the value of a function does not depend on a particular variable  $x$ , then the OBDD that represents it cannot contain  $x$ .

# Variable Ordering

- The shape and size of an OBDD varies according to the particular ordering imposed on variables.
- This ordering can even change the complexity class of the representation. Consider the following expression.

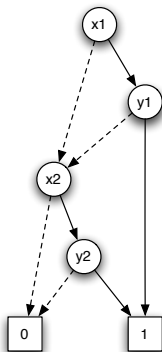
$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \dots \vee (x_n \wedge y_n)$$

- If the ordering is  $x_1 < y_1 < \dots < x_n < y_n$  the number of non-terminals is  $2n$ .
- If the ordering is  $x_1 < \dots < x_n < y_1 < \dots < y_n$  that number raises to  $2(2^n - 1)$ .
- Checking that a particular ordering is optimal is NP-complete.
- Several heuristics have been developed to find good orderings to particular classes of problems.

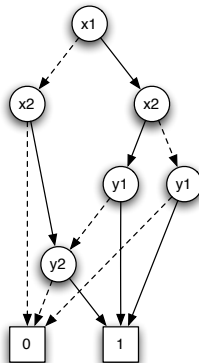
# Example

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

$$x_1 < y_1 < x_2 < y_2$$



$$x_1 < x_2 < y_1 < y_2$$



# Implementing OBDDs

- Its not efficient to compute an ordered BDD from an expression and only afterwards reduce it to obtain an OBDD: the intermediate tree has an exponential size on the number of variables.
- Ideally we want the OBDDs to be reduced incrementally every time an operation is performed.
- When dealing with several expressions we also want to have a single graph with several entry points to increase sharing.
- Equivalence between expressions amounts to test for the same root.

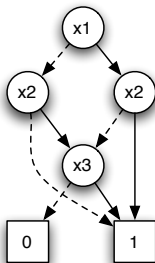
# Implementing OBDDs

- OBDD nodes will be identified by a natural, with 0 and 1 reserved for the terminals.
- Variables  $x_1 < x_2 < \dots < x_n$  have an associated index that determines the ordering.
- A set of OBDDs can be stored in a single table  $T$ , that maps a node  $u$  to a triple of naturals  $(i, l, h)$ , where  $i = \text{var}(u)$ ,  $l = \text{low}(u)$ , and  $h = \text{high}(u)$ .
- Assume that the variable of a terminal node is  $x_{n+1}$ .
- This table has the following methods:
  - $T.\text{init}()$  that initializes  $T$  with nodes 0 and 1.
  - $T.\text{add}(i, l, h)$  that creates a new node and returns its identifier.
  - $T.\text{var}(u)$ ,  $T.\text{low}(u)$ , and  $T.\text{high}(u)$  to search for the attributes of a node.



# Implementing OBDDs

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



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

- To guarantee that no duplicates are created we need to invert  $T$  using a hash table.
- We assume the existence of a function  $mk(i, l, h)$  that creates a node only if it does not exist already.

# Shannon Expansion

- Let  $x \rightarrow y, z$  be an *if-then-else* defined as follows.

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

- It is possible to redefine all boolean expressions using only this operator. Additionally it can be guaranteed that variables appear only on tests and never negated.

$$\neg x \equiv x \rightarrow 0, 1$$

$$x \supset y \equiv x \rightarrow (y \rightarrow 1, 0), 1$$

- This encoding corresponds to a decision tree and can be derived using the *Shannon expansion*.

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

## Implementing Binary Operations

- Given two boolean expressions  $f$  and  $g$ , with root nodes  $u$  and  $v$ , respectively, the OBDD that encodes  $f \star g$  for a given binary operator  $\star$  can be computed by  $apply(\star, u, v)$ .
- Due to Shannon expansion, if both expressions share a variable  $x$  we have

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

- If  $g$  does not depend on  $x$  we have

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

- Since the algorithm is birecursive a memoization table  $G$  is used.
- Negation can be implemented as  $\neg f \equiv f \oplus 1$ .

# Implementing Binary Operations

**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**  $T.var(u) = T.var(v)$

$w \leftarrow mk(T.var(u), aux(T.low(u), T.low(v)),$   
 $aux(T.high(u), T.high(v)))$ ;

**if**  $T.var(u) < T.var(v)$

$w \leftarrow mk(T.var(u), aux(T.low(u), v), aux(T.high(u), v))$ ;

**if**  $T.var(u) > T.var(v)$

$w \leftarrow mk(T.var(v), aux(u, T.low(v)), aux(u, T.high(v)))$ ;

$G.insert(u, v, w)$ ;

**return**  $w$ ;

## Further Reading

- Edmund M. Clarke, Orna Grumberg and Doron A. Peled. *Model Checking*. MIT Press, 2000.
- Edmund M. Clarke, E. Allen Emerson, A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8(2): 244-263. 1986.
- Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* C-35(8):677–691. 1986.
- Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
- Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing* 1: 146–160. 1972.