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 $\pi = s_0 s_1 s_2 \ldots$, such that $s_0 = s$ and $\forall i \geq 0 \cdot s_{i+1} \in R(s_i)$.
- Given a path π its i-th state will be denoted by π_i.
- The suffix of π starting at its i-th state will be denoted by π^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):

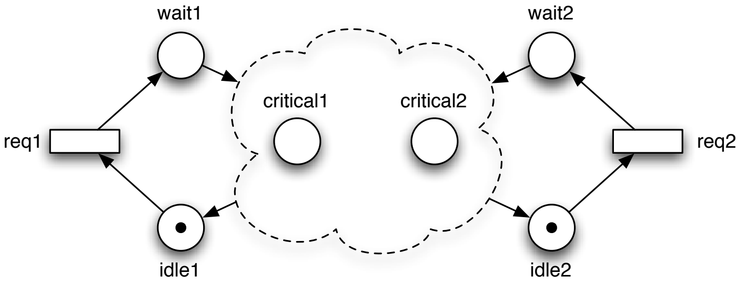
- Any subset $M \subseteq P$ is a global state of Σ.
- A transition t is enabled in a given state M iff ${}^\bullet t \subseteq M$ and $t^\bullet \cap (M - {}^\bullet t) = \emptyset$. This fact will be denoted by $M \xrightarrow{t}$.
- When a transition fires all tokens from the pre-conditions are consumed 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:

$$\begin{aligned} S &= 2^P \\ I &= \{M_0\} \\ R &= \{(M, M') \mid \exists t \in T \cdot M \xrightarrow{t} M'\} \cup \{(M, M) \mid \nexists t \in T \cdot M \xrightarrow{t}\} \\ L &= \mathrm{id} \end{aligned}$$

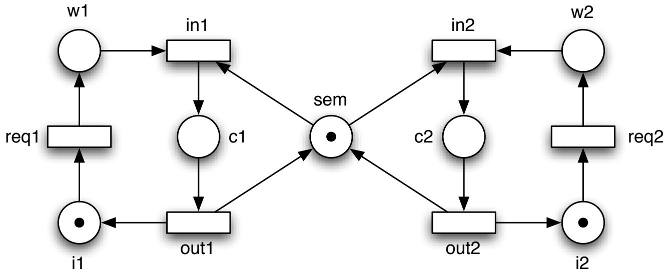
Mutual Exclusion



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.

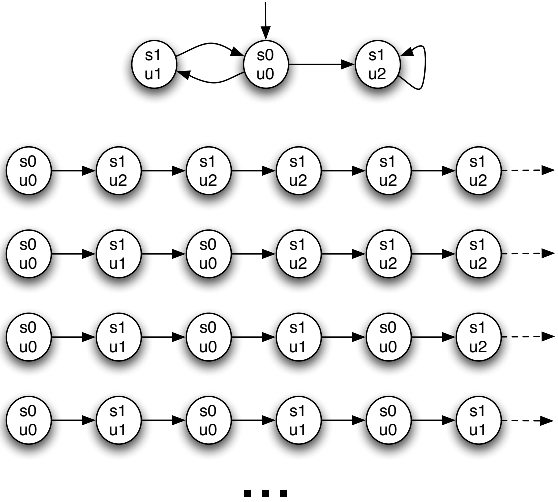
Models of Time

- There are two basic models of time in temporal logic:

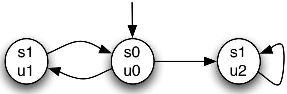
Linear Time	The behavior of the system is the set of all infinite paths starting in initial states.
Branching Time	The behavior of the system is the set of all infinite computation trees unrolled from initial states.

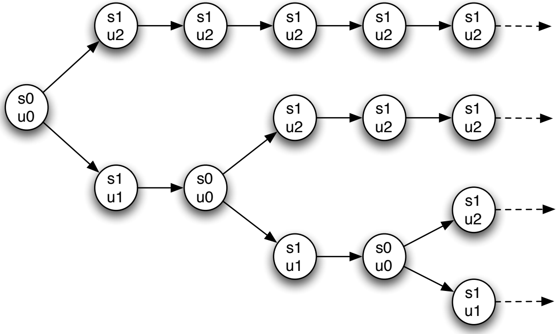
- Both can be determined from a Kripke structure.

Linear Time


s1
u1
s0
u0
s1
u2
s0 u0 → s1 u2 → s1 u2 → s1 u2 → s1 u2 → s1 u2
s0 u0 → s1 u1 → s0 u0 → s1 u2 → s1 u2 → s1 u2
s0 u0 → s1 u1 → s0 u0 → s1 u1 → s0 u0 → s1 u2
s0 u0 → s1 u1 → s0 u0 → s1 u1 → s0 u0 → s1 u1
...

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:

$\mathsf{A}\, f$	f holds for all computation paths.
$\mathsf{E}\, f$	f holds for some computation path.

- Temporal operators:

$\mathsf{X}\, f$	f holds in the next state.
$\mathsf{F}\, f$	Eventually (or in the future) f holds.
$\mathsf{G}\, f$	f always (or globally) holds.
$f\ \mathsf{U}\ g$	g eventually holds and until then f always holds.
$g\ \mathsf{R}\ f$	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 $\Box$, 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 $\mathsf{E}\, f$, and $\mathsf{A}\, f$ 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 $\neg f$, $f \vee g$, $f \wedge g$, $f \supset g$, $\mathsf{X}\, f$, $\mathsf{F}\, f$, $\mathsf{G}\, f$, $f\ \mathsf{U}\ g$, and $g\ \mathsf{R}\ f$ 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):

$$\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s) \\
M, s \models \neg f & \Leftrightarrow & M, s \not\models f \\
M, s \models f \vee g & \Leftrightarrow & M, s \models f \text{ or } M, s \models g \\
M, s \models f \wedge g & \Leftrightarrow & M, s \models f \text{ and } M, s \models g \\
M, s \models f \supset g & \Leftrightarrow & M, s \not\models f \text{ or } M, s \models g \\
M, s \models \mathsf{A}\, h & \Leftrightarrow & \forall\, \pi \in M, \pi_0 = s \cdot M, \pi \models h \\
M, s \models \mathsf{E}\, h & \Leftrightarrow & \exists\, \pi \in M, \pi_0 = s \cdot M, \pi \models h
\end{array}$$

Semantics of Path Formulas

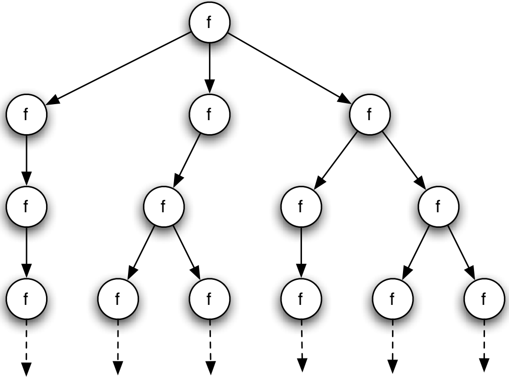
- If f is a path formula, $M, \pi \models f$ means that f holds along path π in M. The relation $\models$ is defined inductively as follows (f and g are path formulas, and h is a state formula):

$$
\begin{array}{lcl}
M, \pi \models h & \Leftrightarrow & M, \pi_0 \models h \\
M, \pi \models \neg f & \Leftrightarrow & M, \pi \not\models f \\
M, \pi \models f \vee g & \Leftrightarrow & M, \pi \models f \text{ or } M, \pi \models g \\
M, \pi \models f \wedge g & \Leftrightarrow & M, \pi \models f \text{ and } M, \pi \models g \\
M, \pi \models f \supset g & \Leftrightarrow & M, \pi \not\models f \text{ or } M, \pi \models g \\
M, \pi \models \mathrm{X}\, f & \Leftrightarrow & M, \pi^1 \models f \\
M, \pi \models \mathrm{F}\, f & \Leftrightarrow & \exists i \geq 0 \cdot M, \pi^i \models f \\
M, \pi \models \mathrm{G}\, f & \Leftrightarrow & \forall i \geq 0 \cdot M, \pi^i \models f \\
M, \pi \models f \,\mathrm{U}\, g & \Leftrightarrow & \exists i \geq 0 \cdot M, \pi^i \models g \text{ and } \forall 0 \leq j < i \cdot M, \pi^j \models f \\
M, \pi \models g \,\mathrm{R}\, f & \Leftrightarrow & \forall i \geq 0 \cdot (\exists 0 \leq j < i \cdot M, \pi^j \models g) \text{ or } M, \pi^i \models f
\end{array}
$$

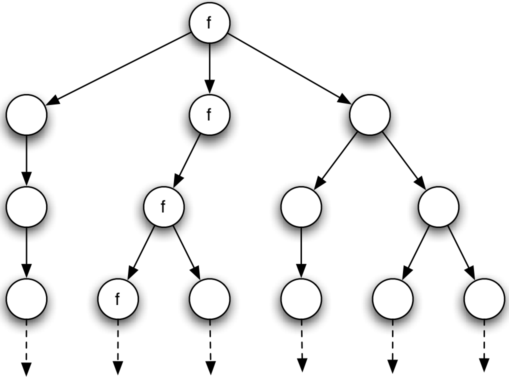
LTL and CTL

- LTL is a linear time sublogic of CTL* in which all formulas are of the form $\mathsf{A}\,f$, 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 $\neg f$, $f \vee g$, $f \wedge g$, $f \supset g$, $\mathsf{X}\,f$, $\mathsf{F}\,f$, $\mathsf{G}\,f$, $f\,\mathsf{U}\,g$, and $g\,\mathsf{R}\,f$ 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 $\mathsf{X}\,f$, $\mathsf{F}\,f$, $\mathsf{G}\,f$, $f\,\mathsf{U}\,g$, and $g\,\mathsf{R}\,f$ are path formulas.

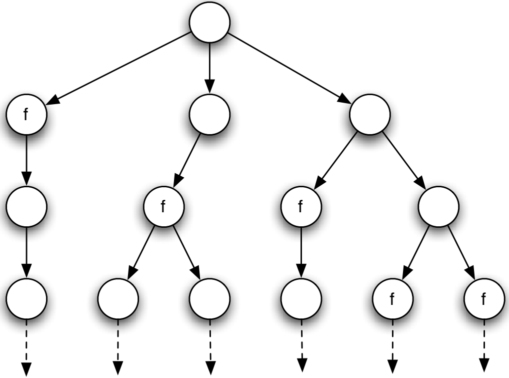
Basic CTL operators: AG *f*



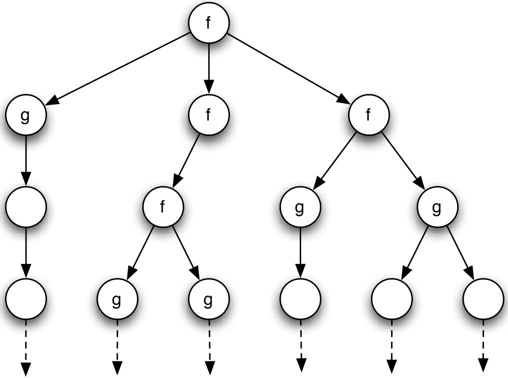
Basic CTL operators: EG *f*



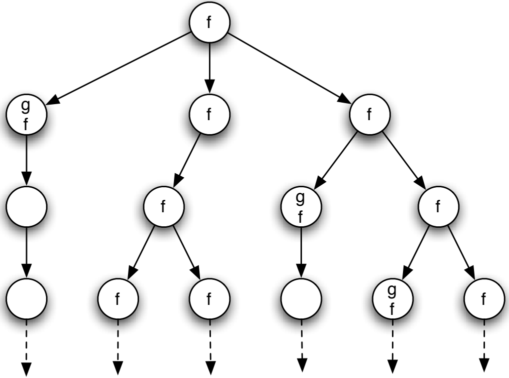
Basic CTL operators: AF *f*



Basic CTL operators: *f* AU *g*



Basic CTL operators: *g* AR *f*

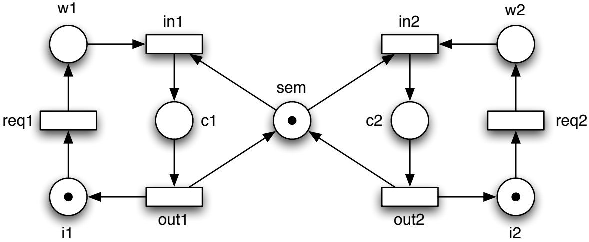


Minimal Set of CTL Operators

- All CTL formulas can be expressed using five operators: ¬, ∨, EX, EU e EG.

$$
\begin{aligned}
f \wedge g &\equiv \neg(\neg f \vee \neg g)\\
f \supset g &\equiv \neg f \vee g\\
\mathrm{AX}\, f &\equiv \neg\, \mathrm{EX}\, \neg f\\
\mathrm{EF}\, f &\equiv \mathit{true}\ \mathrm{EU}\ f\\
\mathrm{AG}\, f &\equiv \neg\, \mathrm{EF}\, \neg f\\
\mathrm{AF}\, f &\equiv \neg\, \mathrm{EG}\, \neg f\\
f\ \mathrm{AR}\ g &\equiv \neg(\neg f\ \mathrm{EU}\ \neg g)\\
f\ \mathrm{ER}\ g &\equiv \mathrm{EG}\, g \vee g\ \mathrm{EU}\ (f \wedge g)\\
f\ \mathrm{AU}\ g &\equiv \neg(\neg f\ \mathrm{ER}\ \neg g)
\end{aligned}
$$

Examples of CTL formulas



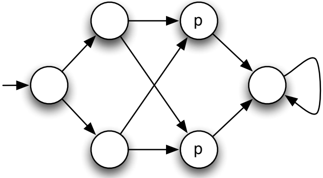
- Mutual exclusion: $\mathrm{AG}\,\neg(c_1 \wedge c_2)$
- Evolution: $\mathrm{AG}(w_1 \supset \mathrm{AF}\,c_1) \wedge \mathrm{AG}(w_2 \supset \mathrm{AF}\,c_2)$
- Reversibility: $\mathrm{AG}\,\mathrm{EF}(i_1 \wedge i_2 \wedge sem \wedge \ldots)$
- No takeover: $\mathrm{AG}((w_1 \wedge i_2) \supset (c_1\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

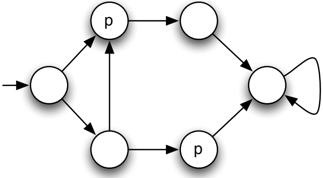
LTL vs CTL

- 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:

$$\mathsf{AG}\,\mathsf{EF}\,\mathit{init}$$

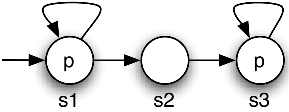
- LTL formulas are also not equivalent to the CTL formulas obtained by preceding each temporal operator by A. For example, $\mathsf{AF}\,\mathsf{AX}\,p$ and $\mathsf{F}\,\mathsf{X}\,p$ have different semantics.

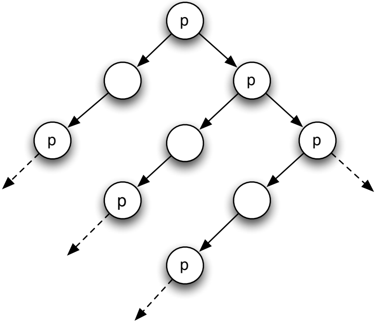




LTL vs CTL

- Although a computation tree is more expressive than a set of computations, there are properties that can only be expressed in LTL.
- For example, F G 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:

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

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

$$M \models f \Leftrightarrow I \subseteq [\![f]\!]_M$$

- 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 ¬, ∨, 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:

$$\begin{aligned}
[\![p]\!]_M &= L^{-1}(p) = \{s \in S \mid p \in L(s)\} \\
[\![\neg f]\!]_M &= S - [\![f]\!]_M \\
[\![f \vee g]\!]_M &= [\![f]\!]_M \cup [\![g]\!]_M
\end{aligned}$$

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

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

Model Checking EG: A Naive Approach

- A recursive algorithm to model check $\mathrm{EG}\, f$ can be derived from its expansion law:

$$\mathrm{EG}\, f \equiv f \wedge \mathrm{EX}\, \mathrm{EG}\, f$$

- $[\![\mathrm{EG}\, f]\!]_M$ is the largest solution to the following recursive equation in the domain $(2^S, \subseteq)$.

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

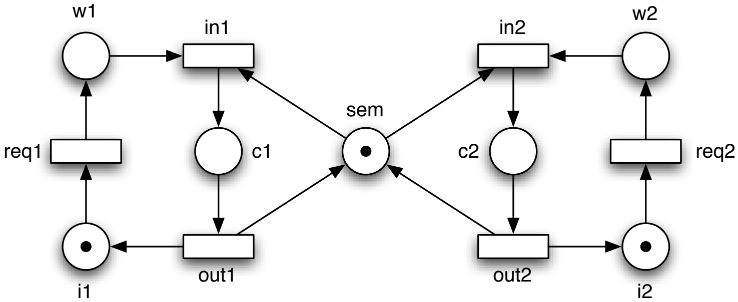
- Alternatively, $[\![\mathrm{EG}\, f]\!]_M$ is the largest fixpoint of $\Pi : 2^S \to 2^S$:

$$[\![\mathrm{EG}\, f]\!]_M = \nu(\Pi), \text{ where } \Pi(X) = [\![f]\!]_M \cap R^{-1}(X)$$

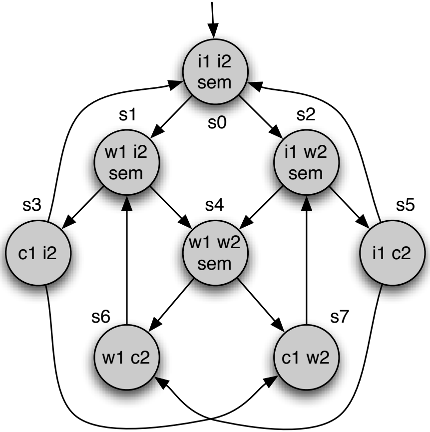
- This set can be computed as the limit of the following series:

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

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

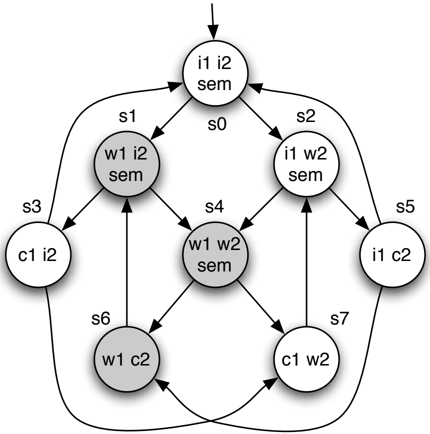


Example: $[\![EG\ w_1]\!]$



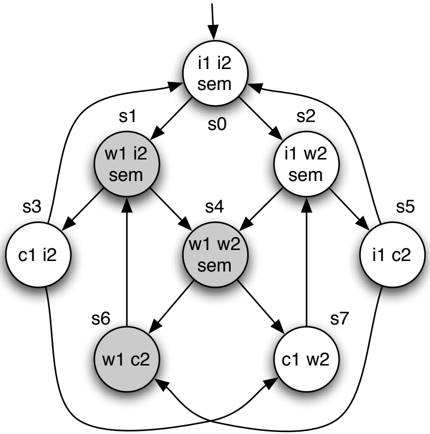
$$\Pi^0(S) = S$$

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



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

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



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

Model Checking EU: A Naive Approach

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

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

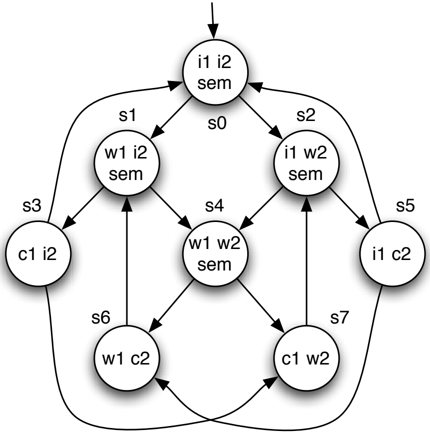
- $[\![f \text{ EU } g]\!]_m$ is the least fixpoint of $\Pi : 2^S \rightarrow 2^S$.

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

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

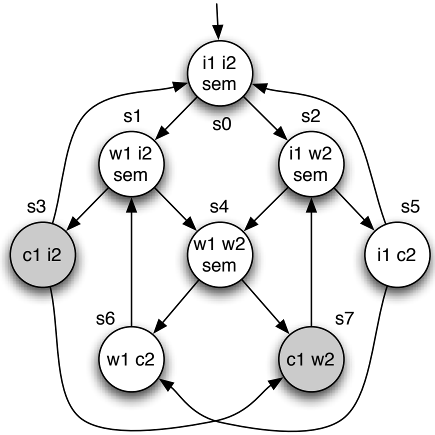
$$\emptyset, \Pi(\emptyset), \Pi(\Pi(\emptyset)), \Pi(\Pi(\Pi(\emptyset))), \ldots$$

Example: $[\![w_1 \text{ EU } c_1]\!]$



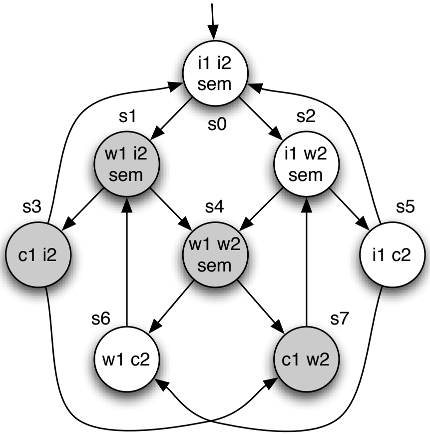
$$\Pi^0(\emptyset) = \emptyset$$

Example: $[\![w_1 \text{ EU } c_1]\!]$



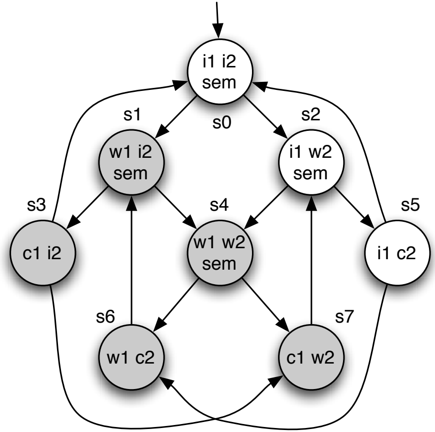
$$\Pi^1(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^0(\emptyset))) = \{s_3, s_7\}$$

Example: $[\![w_1 \text{ EU } c_1]\!]$



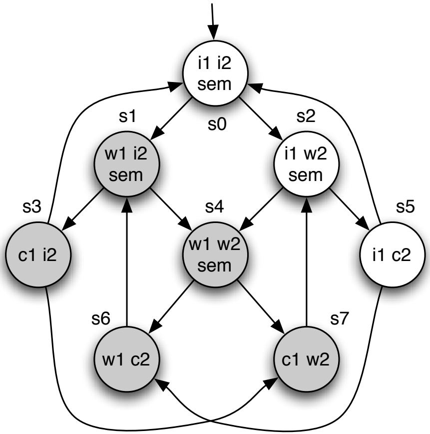
$$\Pi^2(\emptyset) = [\![c_1]\!] \cup ([\![w_1]\!] \cap R^{-1}(\Pi^1(\emptyset))) = \{s_1, s_3, s_4, s_7\}$$

Example: $[\![w_1 \text{ EU } c_1]\!]$



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

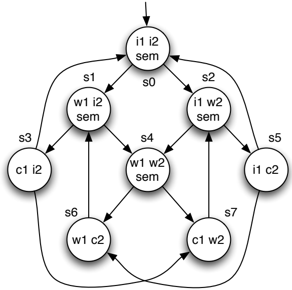
Example: $[\![w_1 \text{ EU } c_1]\!] = \{s_1, s_3, s_4, s_6, s_7\}$



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

Example: $M \models \mathrm{AG}\,\neg(c_1 \wedge c_2)$

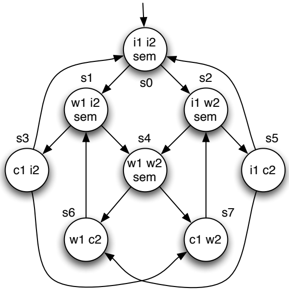
$$\mathrm{AG}\,\neg(c_1 \wedge c_2) \equiv \neg\,\mathrm{EF}\,\neg\neg(c_1 \wedge c_2) \equiv \neg\,\mathrm{EF}(c_1 \wedge c_2) \equiv \neg(\mathit{true}\,\mathrm{EU}(c_1 \wedge c_2))$$



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

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

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



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

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
$$\mathcal{O}(|f| \cdot |S| \cdot (|S| + |R|))$$
- With some clever tricks it is possible to lower the complexity to
$$\mathcal{O}(|f| \cdot (|S| + |R|))$$

Model Checking f EU g

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

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

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:

$$M_f = ([\![f]\!], I \cap [\![f]\!], R \cap ([\![f]\!] \times [\![f]\!]), L|_{[\![f]\!]})$$

Lemma

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

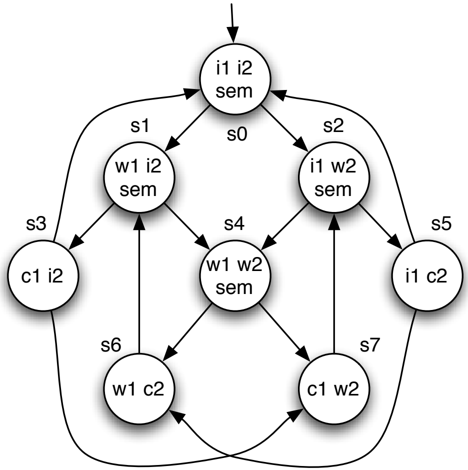
- A SCC (*strongly connected component*) C is a maximal subgraph where every node is reachable from every other node along a directed path entirely contained in C.
- C is also *nontrivial* iff it has more than one node or it contains one node with a self-loop.

Model Checking EG *f*

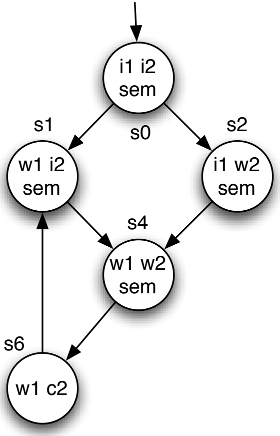
- To compute $[\![\text{EG}\, f]\!]$ we first compute all states belonging to nontrivial SCCs and successively add all predecessors in $[\![f]\!]$.
- **scc** uses Tarjan algorithm with complexity $O(|[\![f]\!]| + |R_f|)$.

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

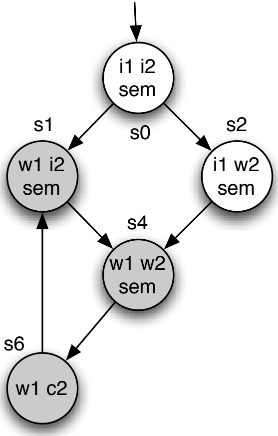
Example: EG($w_1 \vee sem$)



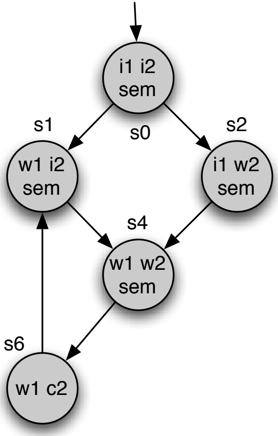
Example: EG($w_1 \lor sem$)



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 $f_1, \ldots, f_n$ 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 $F \subseteq 2^S = \{[\![f_1]\!], \ldots, [\![f_n]\!]\}$.
- Assuming that $\mathrm{inf}(\pi)$ extracts all states that occur infinitely often in π, we can define a predicate to test for fairness as follows:

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

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

$$\begin{array}{lcl} M, s \models_F p & \Leftrightarrow & \exists\, \pi \in M, \mathrm{fair}(\pi), \pi_0 = s \cdot p \in L(s) \\ M, s \models_F \mathsf{A}\, h & \Leftrightarrow & \forall\, \pi \in M, \mathrm{fair}(\pi), \pi_0 = s \cdot M, \pi \models h \\ M, s \models_F \mathsf{E}\, h & \Leftrightarrow & \exists\, \pi \in M, \mathrm{fair}(\pi), \pi_0 = s \cdot M, \pi \models h \end{array}$$

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:

$$\begin{array}{rcl} M, s \models_F p & \equiv & M, s \models p \wedge \mathrm{EG}\, \mathit{true} \\ M, s \models_F \mathrm{EX}\, f & \equiv & M, s \models \mathrm{EX}(f \wedge \mathrm{EG}\, \mathit{true}) \\ M, s \models_F f\ \mathrm{EU}\ g & \equiv & M, s \models f\ \mathrm{EU}\ (g \wedge \mathrm{EG}\, \mathit{true}) \end{array}$$

- Complexity of model checking under fairness raises to

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

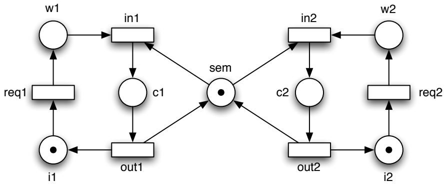
Symbolic Model Checking

- 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 a 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.

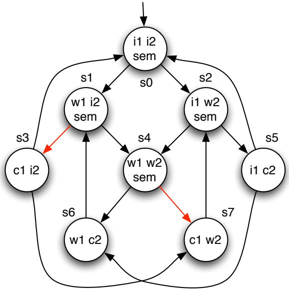
$$\phi_s \equiv (\wedge_{x \in s} x) \wedge (\wedge_{x \notin s} \neg x)$$



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

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 valuated in s, while variables in P' should be valuated in s'.



$$\neg i_1 \land w_1 \land \neg c_1 \land sem \land i_2 \land \neg w_2 \land \neg c_2$$

$$\land$$

$$\neg i'_1 \land \neg w'_1 \land c'_1 \land \neg sem' \land i'_2 \land \neg w'_2 \land \neg c'_2$$

$$\neg i_1 \land w_1 \land \neg c_1 \land sem \land \neg i_2 \land w_2 \land \neg c_2$$

$$\land$$

$$\neg i'_1 \land \neg w'_1 \land c'_1 \land \neg sem' \land \neg i'_2 \land w'_2 \land \neg c'_2$$

Encoding Transitions

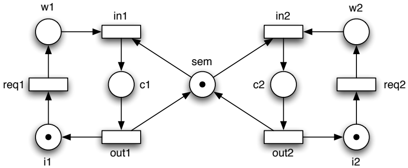
- The formula ϕ_R that encodes the transition relation is the disjunction of all formulas that encode each individual transition.
- With a bit of syntactic sugar we can have a direct encoding for each transition in the net: only variables in the neighborhood of a transition are affected.

$$\begin{gathered} w_1 \wedge \neg c_1 \wedge sem \wedge \neg w_1' \wedge c_1' \wedge \neg sem' \\ \wedge \\ i_1' = i_1 \wedge i_2' = i_2 \wedge w_2' = w_2 \wedge c_2' = c_2 \end{gathered}$$

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

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

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 $[\![f]\!]$ 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:

$$[\![\mathrm{EG}\, f]\!] = \nu(\Pi), \text{ where } \Pi(h) = [\![f]\!] \wedge [\![\mathrm{EX}\, h]\!]$$

$$[\![f \,\mathrm{EU}\, g]\!] = \mu(\Pi), \text{ where } \Pi(h) = [\![g]\!] \vee ([\![f]\!]_M \wedge [\![\mathrm{EX}\, h]\!])$$

- 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

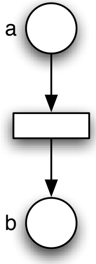
- To model check EX f an (temporary) existential quantifier is used.

$$[\![\mathrm{EX}\, f]\!] = \exists \overline{x}' \cdot [\![f]\!]' \wedge \phi_R$$

- $[\![f]\!]'$ is the formula obtained from $[\![f]\!]$ by replacing all variables $x \in P$ by the corresponding $x' \in P'$.
- Intuitively, $[\![\mathrm{EX}\, f]\!]$ 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:

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

Example



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

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

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

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 constrains on traditional *Binary Decision Diagrams* (BDDs) to achieve canonical forms.

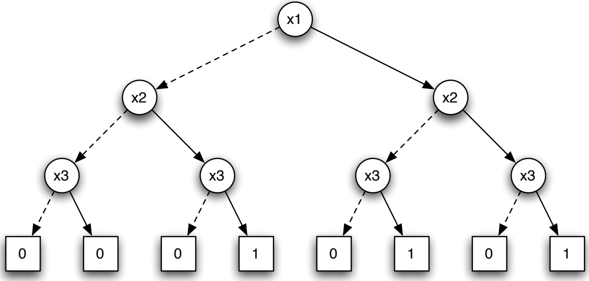
Binary Decision Diagrams

- 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



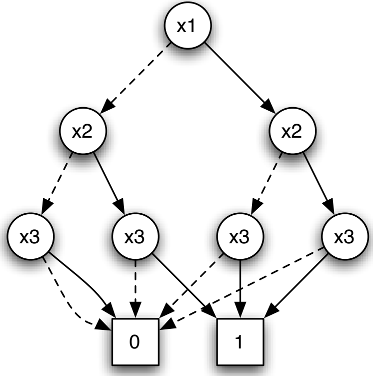
Ordered Binary Decision Diagrams

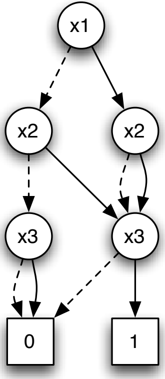
- In a ordered BDD a total ordering is imposed on the variables: each path in the graph must respect this order.
- Any ordering is possible, but size of an OBDD can vary significantly with the particular order chosen.
- Besides being ordered, an OBDD must be *reduced*:
  - No nodes with equal successors.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

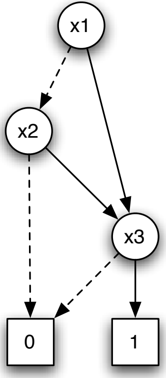
Reducing Binary Decision Diagrams

- 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.

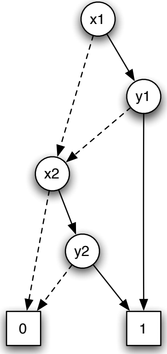
$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$$

    - If the ordering is $x_1 < y_1 < \ldots < x_n < y_n$ the number of non-terminals is $2n$.
    - If the ordering is $x_1 < \ldots < x_n < y_1 < \ldots < y_n$ 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.

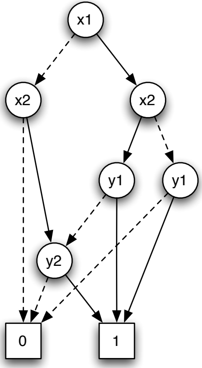
Example

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

$x_1 < y_1 < x_2 < y_2$



$x_1 < x_2 < y_1 < y_2$



Implementing OBDDs

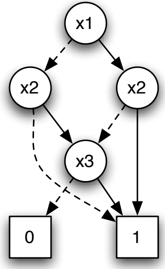
- Its not efficient to compute an ordered BDD from an expression and only afterwards reduce it to obtain an OBDD: the intermediate tree has an exponential size on the number of variables.
- Ideally we want the OBDDs to be reduced incrementally every time an operation is performed.
- When dealing with 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 $x_1 < x_2 < \ldots < x_n$ 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 $i = var(u)$, $l = low(u)$, and $h = high(u)$.
- Assume that the variable of a terminal node is x_{n+1}.
- This table has the following methods:
  - $T.init()$ that initializes T with nodes 0 and 1.
  - $T.add(i, l, h)$ that creates a new node and returns its identifier.
  - $T.var(u)$, $T.low(u)$, and $T.high(u)$ 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 to 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.