An Introduction to Model Checking

Pierre Wolper

Université de Liège

Overview

- Model checking: the problem

- Temporal logics

- Algorithms for model checking

  - Linear-time and the automata-theoretic approach

  - Branching-time and the symbolic approach

- Summary and conclusions

Model checking: the problem

- One is given a program which can be represented in terms of a finite transition system $K = \langle P, W, R, w^0, L \rangle$, where
  - P is a set of atomic propositions,
  - W is a finite set of states,
  - $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $\langle w, w' \rangle \in R$),
  - w^0 is an initial state, and
  - $L : W \to 2^P$ maps each state to the set of atomic propositions true in that state.

- The problem is to check whether K satisfies some properties expressed in a temporal logic.

Model checking: the challenge

- K can be very large in terms of the size of the program text (state-space explosion problem), which is the main practical limit to the use of model checking.

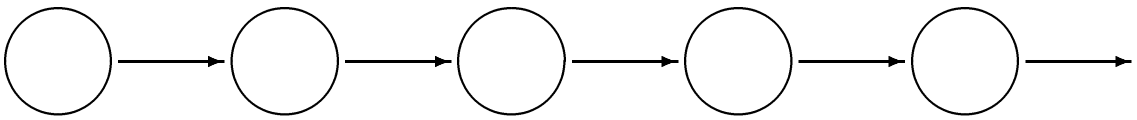
- Much of the work on model checking has been concerned with computing and representing K.

A First Temporal Logic: Linear Time

- Language for describing properties of infinite sequences,
- Extension of propositional logic (or first-order logic),
- Uses *Temporal Operators* to describe temporal (sequencing) properties.

Linear Temporal Logic: Interpretations

- Temporal logic is interpreted on sequences of states:



- each state in the sequence gives a truth value to atomic propositions,
- a formula is given a meaning in a state (world, interpretation) of a sequence,
- *temporal operators* indicates in which states the formula (or parts thereof) should be interpreted.

Temporal Operators

1. $\bigcirc$ which is read "at the next time" (in the next state of the sequence)

2. $\Box$ which is read "always in the future" (in all future states of the sequence)

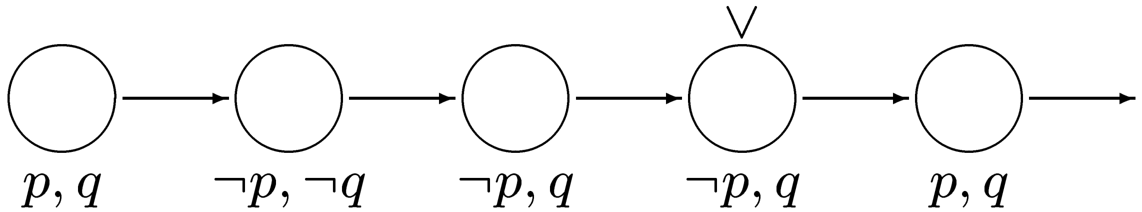
3. $\Diamond$ which is read "eventually" (in some future state of the sequence)

4. U which is read "until" (binary operator)

5. $\tilde{U}$ which is read "releases" (binary operator)

Examples

In the marked state of the sequence



the following formulas are *true*:

$$\neg p \wedge q$$

$$\bigcirc(p \wedge q)$$

the following formula is *false*:

$$\square\, p$$

More Examples

- $\Box(p \supset \bigcirc q)$ is satisfied by all sequences in which each state where p is true is immediately followed by a state in which q is true.

- $\Box(p \supset \bigcirc(\neg q\, U\, r))$ is satisfied by all sequences such that if p is true in a state, then q remains false from the next state on and until the first state where r is true, which must occur.

- The formula $\Box \Diamond p$ is satisfied by all sequences in which p is true infinitely often in the future

- The formula $p\, \tilde{U}\, q$ is satisfied by all sequences in which q is always true unless this obligation is *released* by p being true in a previous state.

Formal Syntax of Temporal Logic

The formulas of linear-time temporal logic (LTL) built from a set of atomic propositions P are the following.

- **true**, **false**, p, or $\neg p$, for all $p \in P$;
- $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are LTL formulas;
- $\bigcirc \varphi_1$, $\varphi_1 \, U \, \varphi_2$, or $\varphi_1 \, \tilde{U} \, \varphi_2$, where φ_1 and φ_2 are LTL formulas.

Two operators are then defined as abbreviations:

- $\Diamond \varphi = \mathbf{true} \, U \, \varphi$ ("eventually").
- $\Box \varphi = \mathbf{false} \, \tilde{U} \, \varphi$ ("always").

Semantics of Temporal Logic

The semantics of LTL is defined with respect to paths $\pi : \mathbf{N} \rightarrow 2^P$. For a path π, π^i represents the suffix of π obtained by removing its i first states, i.e. $\pi^i(j) = \pi(i+j)$. The rules given the truth of a formula in the first state of a path π are the following.

- For all π, we have $\pi \models$ **true** and $\pi \not\models$ **false**.
- $\pi \models p$ for $p \in P$ iff $p \in \pi(0)$.
- $\pi \models \neg p$ for $p \in P$ iff $p \notin \pi(0)$.
- $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$.
- $\pi \models \varphi_1 \vee \varphi_2$ iff $\pi \models \varphi_1$ or $\pi \models \varphi_2$.
- $\pi \models \bigcirc \varphi_1$ iff $\pi^1 \models \varphi_1$.
- $\pi \models \varphi_1 \, U \, \varphi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \varphi_2$ and for all $0 \leq j < i$, we have $\pi^j \models \varphi_1$.
- $\pi \models \varphi_1 \, \tilde{U} \, \varphi_2$ iff for all $i \geq 0$ such that $\pi^i \not\models \varphi_2$, there exists $0 \leq j < i$ such that $\pi^j \models \varphi_1$.

In the logic we have defined, negation is only applied to atomic propositions. This restriction can be lifted with the help of the following relations

$$\pi \not\models \varphi_1 \, U \, \varphi_2 \text{ iff } \pi \models (\neg\varphi_1) \, \tilde{U} (\neg\varphi_2)$$

$$\pi \not\models \varphi_1 \, \tilde{U} \, \varphi_2 \text{ iff } \pi \models (\neg\varphi_1) \, U (\neg\varphi_2)$$

$$\pi \not\models \bigcirc \varphi_1 \text{ iff } \pi \models \bigcirc \neg\varphi_1$$

A linear time temporal logic formula is a description of a set of infinite sequences (those that satisfy it)

Describing Infinite Sequences with Automata

- A finite word of length n over an alphabet Σ is a mapping $w : \{0, \ldots n-1\} \to \Sigma$

- An automaton on finite words is a description of a set of finite words (those that it accepts).

- An infinite word (ω-word) over an alphabet Σ is a mapping $w : \mathbf{N} \to \Sigma$

- An automaton on infinite words is a description of a set of infinite words (those that it accepts).

Note: Automata are often introduced as a model of computation. Here, we will just view them as a description of a set of words.

Automata on Infinite Words

Same structure as automata on finite words:

$A = \{\Sigma, S, \delta, S_0, F\}$ where

- Σ is an alphabet,
- S is a set of states,
- $\delta : S \times \Sigma \to S$ (deterministic) or
  $\delta : S \times \Sigma \to 2^S$ (nondeterministic) is a transition function,
- $S_0 \subseteq S$ is a set of initial states, and
- $F \subseteq S$ is a set of accepting states.

What changes is the semantics.

When does an Automaton accept a Word?

From now on, we consider the more general case of nondeterministic automata.

Finite words $w : \{0, \dots n-1\} \to \Sigma$

The word w is accepted by the automaton if there is a labeling of the word by states

$$\rho : \{0, \dots n\} \to S$$

such that

- $\rho(0) \in S_0$ (the initial label is an initial state),
- $\forall 0 \leq i \leq n-1,\ \rho(i+1) \in \delta(w(i), \rho(i))$ (the labeling is compatible with the transition relation),
- $\rho(n) \in F$ (the last label is an accepting state).

Don't think of automata as programs, but as a formalism for defining sets of words that has a very simple semantics.

Infinite words $w : \mathbf{N} \rightarrow \Sigma$

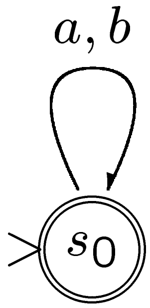
The word w is accepted by the automaton if there is a labeling of the word by states

$$\rho : \mathbf{N} \rightarrow S$$

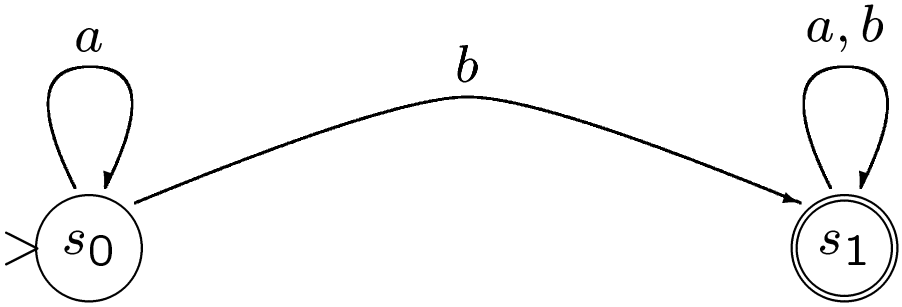
such that

- $\rho(0) \in S_0$ (the initial label is an initial state),
- $\forall 0 \leq i,\ \rho(i+1) \in \delta(w(i), \rho(i))$ (the labeling is compatible with the transition relation),
- $inf(\rho) \cap F \neq \emptyset$ where $inf(\rho)$ is the set of states that appear infinitely often in ρ (the set of repeating states intersects F).

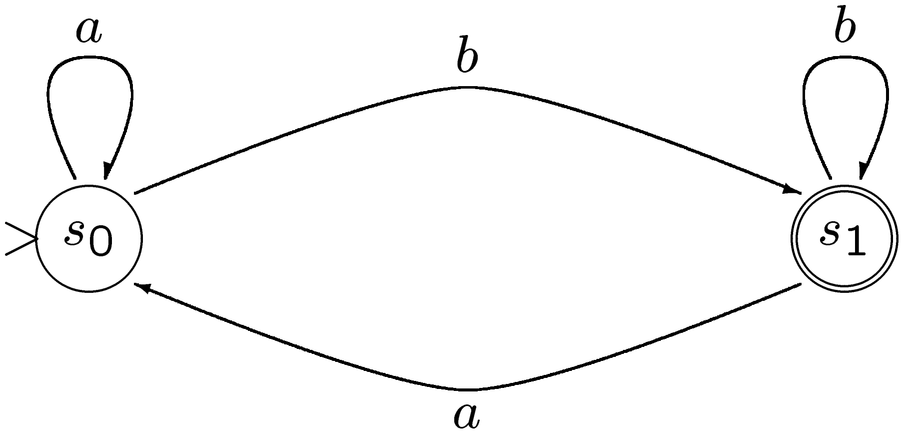
Examples



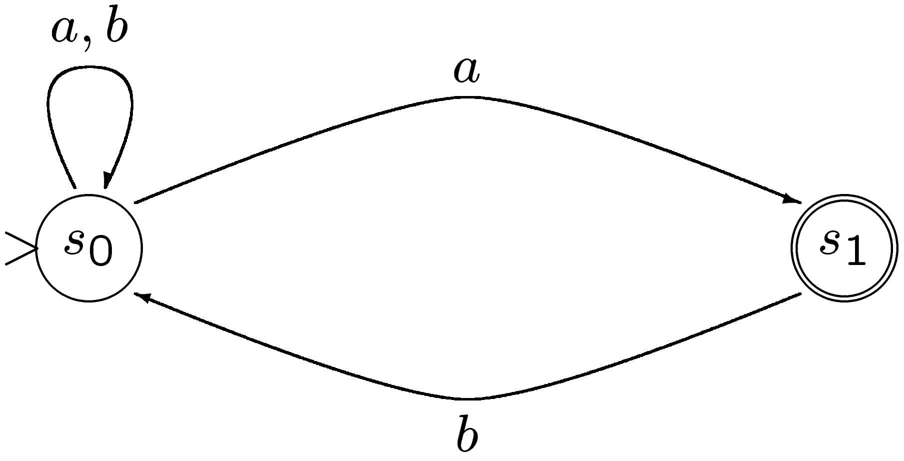
All words over $\Sigma = \{a, b\}$



An automaton accepting $a^{\star}b(a \cup b)^{\omega}$



All words containing b infinitely often



Infinitely often ab

Other types of Acceptance conditions

Büchi: $F \subseteq S$,
$\inf(\rho) \cap F \neq \emptyset$.

Generalized Büchi: $\mathcal{F} \subseteq 2^S$,
i.e. $F = \{F_1, \ldots, F_m\}$
For each F_i, $\inf(\rho) \cap F_i \neq \emptyset$.

Rabin: $F \subseteq 2^S \times 2^S$,
i.e. $F = \{(G_1, B_1), (G_2, B_2), \ldots, (G_m, B_m)\}$
For some pair $(G_i, B_i) \in F$, $\inf(\rho) \cap G_i \neq \emptyset$ and $\inf(\rho) \cap B_i = \emptyset$

Streett: $F \subseteq 2^S \times 2^S$,
i.e. $F = \{(G_1, B_1), (G_2, B_2), \ldots, (G_m, B_m)\}$
For all pairs $(G_i, B_i) \in F$, $\inf(\rho) \cap G_i = \emptyset$ or $\inf(\rho) \cap B_i \neq \emptyset$

For nondeterministic automata, all define the ω-regular languages :

$$\bigcup_i \alpha_i \beta_i^\omega$$

where α_i and β_i are finite-word regular languages and ω denotes infinite repetition.

For deterministic automata, Büchi and Generalized Büchi conditions are weaker.

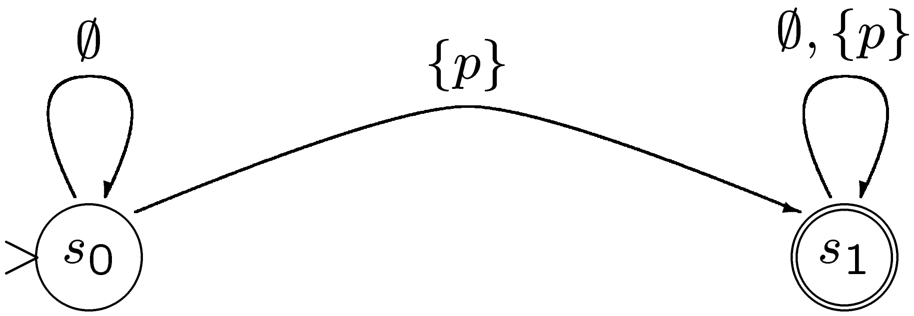
Properties of Büchi Automata

- Closed under union, intersection, projection, and complementation.
- Nonemptiness easy to decide:
  - Check if some accepting state is accessible from an initial state and (nontrivially) from itself.
  - Linear time (compute the strongly connected components).
  - NLOGSPACE.
- For Rabin conditions, nonemptiness can also be decided in polynomial time. The simplest algorithms involve a translation to nondeterministic Büchi automata.
- For Street automata, the translation to nondeterministic Büchi automata is exponential.

From LTL to Automata

Given a LTL formula φ built from a set of atomic propositions P, construct an automaton on infinite words over the alphabet 2^P that accepts exactly the infinite sequences satisfying φ.

Example:



An automaton for $\Diamond\, p$

Idea of the Construction

To determine if a sequence $\pi : \mathbf{N} \rightarrow 2^P$ satisfies a formula φ, one can proceed by labeling it with subformulas of φ in a way that is compatible with LTL semantics. The subformulas of φ are the *closure* of φ ($cl(\varphi)$).

- $\varphi \in cl(\varphi)$,
- $\varphi_1 \wedge \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$,
- $\varphi_1 \vee \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$,
- $\bigcirc \varphi_1 \in cl(\varphi) \Rightarrow \varphi_1 \in cl(\varphi)$,
- $\varphi_1 \, U \, \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$.
- $\varphi_1 \, \tilde{U} \, \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$.

$$cl(\Diamond \neg p) = \{\Diamond \neg p, \neg p\}$$

What are the labeling rules?

For every position $i \geq 0$ the labeling $\tau : \mathbf{N} \rightarrow 2^{cl(\varphi)}$ has to satisfy the following:

- For $p \in P$, if $p \in \tau(i)$ then $p \in \pi(i)$, and if $\neg p \in \tau(i)$ then $p \notin \pi(i)$

- if $\varphi_1 \wedge \varphi_2 \in \tau(i)$ then $\varphi_1 \in \tau(i)$ and $\varphi_2 \in \tau(i)$.

- if $\varphi_1 \vee \varphi_2 \in \tau(i)$ then $\varphi_1 \in \tau(i)$ or $\varphi_2 \in \tau(i)$.

This takes care of propositions and boolean connectives. What about the temporal operators?

Labeling Rules for Temporal Operators

Even though the temporal operators, have a semantics that looks at the whole sequence, it can (mostly) be reduced to local conditions.

$$
\begin{array}{rcl}
p\,U\,q & \equiv & (q \vee (p \wedge \bigcirc(p\,U\,q))) \\
p\,\tilde{U}\,q & \equiv & (q \wedge (p \vee \bigcirc(p\,\tilde{U}\,q)))
\end{array}
$$

We thus add the following labeling rules for all positions $i \geq 0$:

- if $\bigcirc\,\varphi_1 \in \tau(i)$ then $\varphi_1 \in \tau(i+1)$.
- if $\varphi_1\,U\,\varphi_2 \in \tau(i)$ then either $\varphi_2 \in \tau(i)$, or $\varphi_1 \in \tau(i)$ and $\varphi_1\,U\,\varphi_2 \in \tau(i+1)$.
- if $\varphi_1\,\tilde{U}\,\varphi_2 \in \tau(i)$ then $\varphi_2 \in \tau(i)$, and either $\varphi_1 \in \tau(i)$ or $\varphi_1\,\tilde{U}\,\varphi_2 \in \tau(i+1)$

There is only one missing condition that concerns formulas called *eventualities*

$$(\varphi_1 \, U \, \varphi_2) \;\; \equiv \;\; \varphi_1 \wedge (\varphi_2 \vee \bigcirc(\varphi_1 \, U \, \varphi_2))$$

φ_2, has to be true at some point!

Interpreting Labelings

A sequence π satisfies a formula φ if one can find a labeling τ satisfying

- The conditions above,
- $\varphi \in \tau(0)$, and
- If an eventuality formula $\varphi_1 \, U \, \varphi_2$ is in $\tau(i)$, then for some $j \geq i$, $\varphi_2 \in \tau(j)$.

An automaton on infinite words is just a labeling rule. Actually, we have defined an automaton on infinite words that accepts the models of a temporal logic formula!

The Automaton

It is a generalized Büchi automaton
$A_\varphi = (\Sigma, S, \delta, S_0, \mathcal{F})$ where

- $\Sigma = 2^P$,
- The set of states S is the set of possible labels, i.e. the subsets s of $2^{cl(\varphi)}$ that satisfy
  - if $\varphi_1 \wedge \varphi_2 \in \mathrm{s}$ then $\varphi_1 \in \mathrm{s}$ and $\varphi_2 \in \mathrm{s}$.
  - if $\varphi_1 \vee \varphi_2 \in \mathrm{s}$ then $\varphi_1 \in \mathrm{s}$ or $\varphi_2 \in \mathrm{s}$.

- The transition function δ checks that the propositional labeling matches the one in the sequence being considered and that the rules for the temporal operators are satisfied. Thus, $\mathbf{t} \in \delta(\mathbf{s}, \mathbf{a})$ iff

  - For all $p \in P$, if $p \in \mathbf{s}$ then $p \in \mathbf{a}$.
  - For all $p \in P$, if $\neg p \in \mathbf{s}$ then $p \notin \mathbf{a}$.
  - if $\bigcirc \varphi_1 \in \mathbf{s}$ then $\varphi_1 \in \mathbf{t}$.
  - if $\varphi_1 \, U \, \varphi_2 \in \mathbf{s}$ then either $\varphi_2 \in \mathbf{s}$, or $\varphi_1 \in \mathbf{s}$ and $\varphi_1 \, U \, \varphi_2 \in \mathbf{t}$.
  - if $\varphi_1 \, \tilde{U} \, \varphi_2 \in \mathbf{s}$ then $\varphi_2 \in \mathbf{s}$ and either $\varphi_1 \in \mathbf{s}$, or $\varphi_1 \, U \, \varphi_2 \in \mathbf{t}$.

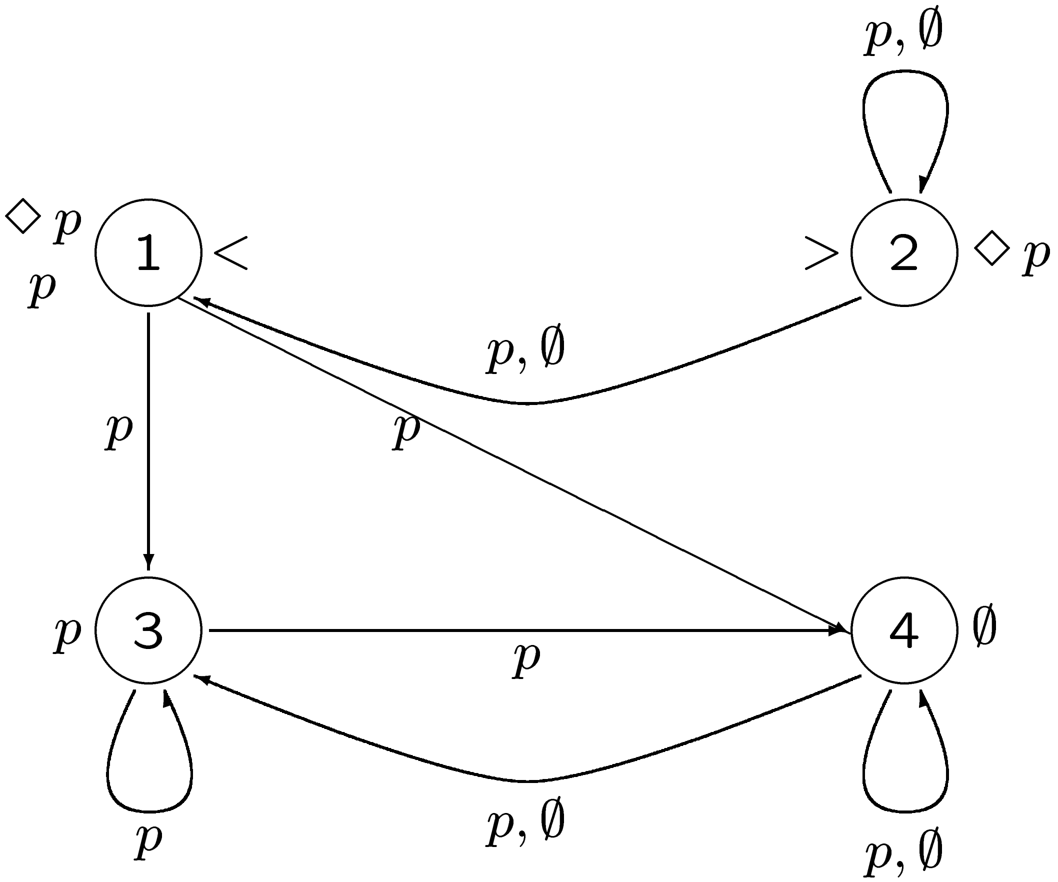
- The initial states are those that contain φ, $S_0 = \{\mathbf{s} \in S \mid \varphi \in \mathbf{s}\}$.

- For the accepting condition, we need to impose that, for every eventuality formula $\varphi_1 \, U \, \varphi' \equiv e(\varphi') \in cl(\varphi)$, any state that contains that formula is followed by a state that contains φ'.

  - We have the property that, from every state in which $e(\varphi')$ appears, the transition relation ensures that this formula keeps appearing until the first state in which φ' appears.
  - It is thus sufficient to require that, for every eventuality $e(\varphi')$, one goes infinitely often either through a state in which it does not appear, or through a state in which the eventuality and φ' both appear.
  - The acceptance condition is thus a generalized Büchi condition. If the eventualities are $e_1(\varphi_1), \ldots e_m(\varphi_m)$, it is $\mathcal{F} = \{\Phi_1, \ldots, \Phi_m\}$ where $\Phi_i = \{\mathbf{s} \in S \mid e_i, \varphi_i \in \mathbf{s} \vee e_i \not\in \mathbf{s}\}$.

Example

The automaton for $\Diamond p$



$$\mathcal{F} = \{\{1, 3, 4\}\}$$

Note: Transitions leading to states with more (nonatomic) elements of the closure than necessary were not included.

Improvements to the Construction

- Build the automaton incrementally from the formula φ by decomposing and expanding the temporal operators.
- Merge states which only differ by atomic propositions since these are determined by the labels of the transitions.
- When possible, detect inconsistent nodes without fully expanding to the propositional level.

Checking Linear-Time Temporal Logic Properties

- To check that a transition system satisfies a linear-time temporal logic property, one needs to check that *all* its infinite behaviors satisfy this property.
- One checks that all these behaviors are *models* of the property. This procedure is thus called *model-checking*.
- To obtain a model checking procedure, the idea is to use the automaton on infinite words for the property as an observer for the system.
- However, this automaton is nondeterministic and it accepts a behavior is *there exists* some computation for which it accepts.
- Checking that *for all* behaviors of the transition system, *there exists* some accepting computation of the observer is problematic.
- The trick is to first complement the property to be checked.

A Model-Checking Algorithm for LTL

- One is given a transition system K and a LTL property f.
- The first step is to build an automaton on infinite words $A_{\neg f}$ that accepts exactly all infinite sequences that *do not* satisfy f.
- $A_{\neg f}$ is then composed with K and one checks whether this transition system is empty of not (accepts at least one infinite word). Any accepted infinite word is a behavior of the system that does not satisfy the property to be checked.

Complexity of LTL Model Checking:
Linear in the size of the transition system (NLOGSPACE),
Exponential in the size of the LTL property (PSPACE).

Branching Time Temporal Logic

Branching time temporal logic are interpreted over *infinite trees* (varying degree) in which each node is a state assigning truth values to the atomic propositions.

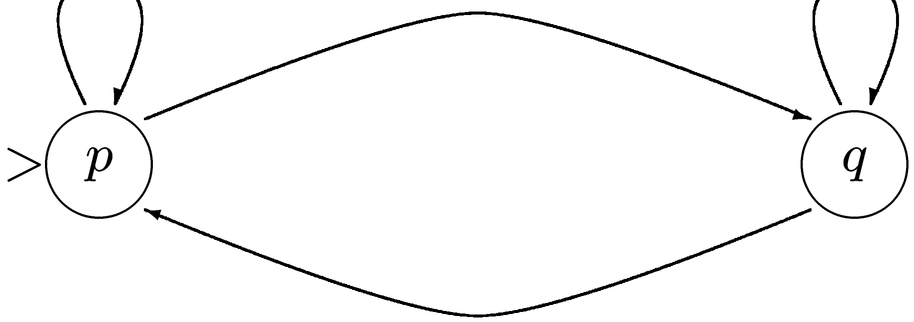
More generally, branching time temporal logic is interpreted over Kripke structures $K = \langle P, W, R, w^0, L \rangle$, where

- P is a set of atomic propositions,
- W is a set of states,
- $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $\langle w, w' \rangle \in R$),
- w^0 is an initial state, and
- $L : W \to 2^P$ maps each state to the set of atomic propositions true in that state.

From a state of a Kripke structure leave a number of *paths*, i.e. infinite sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, $\langle w_i, w_{i+1} \rangle \in R$.

Note that by identifying $\pi(i)$ with $L(w_i)$, a path in a Kripke structure defines an LTL interpretation.

Example



Path Quantifiers

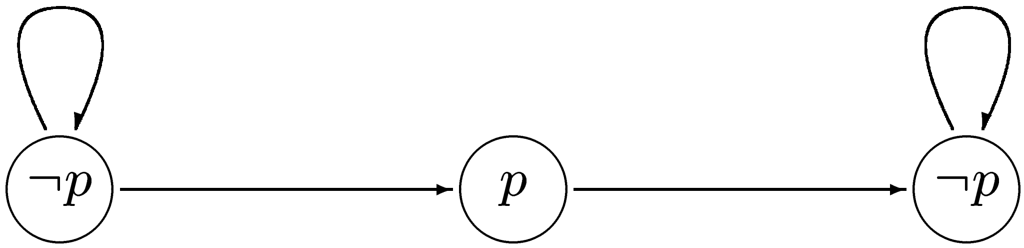
Branching-time temporal logic is linear-time temporal logic extended with *path quantifiers*:

- A for all paths,
- E for some paths.

Path quantifiers are applied to linear-time formulas which then become *state formulas* and can be used as atoms in a new path formula.

Examples

$A \Box p$, $E \Box p$, $A(p \supset \bigcirc E \Box p)$, $A \Box E \bigcirc p$,
$A(\Box \Diamond p)$, $E(\Box \Diamond p)$, $E \Box E \Diamond p$.



Formal Syntax of Branching-time Temporal Logic

Syntax

We first define the more general logic CTL*. A CTL* state formula is either:

- **true**, **false**, p, or $\neg p$, for all $p \in P$;
- $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are CTL* state formulas;
- $A\psi$ or $E\psi$, where ψ is a CTL* path formula.

A CTL* path formula is either:

- A CTL* state formula;
- $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\bigcirc\, \psi_1$, $\psi_1 U \psi_2$, or $\psi_1 \tilde{U} \psi_2$, where ψ_1 and ψ_2 are CTL* path formulas.

CTL* is the set of state formulas generated by the above rules.

A restricted form of Branching-time Temporal Logic

The logic *CTL* is a restricted subset of CTL* in which the temporal operators must be immediately preceded by a path quantifier.

Formally, it is the subset of CTL* obtained by defining the path formulas as follows.

A CTL path formula is of the form

- $\bigcirc \varphi_1$, $\varphi_1 \, U \, \varphi_2$, or $\varphi_1 \, \tilde{U} \, \varphi_2$, where φ_1 and φ_2 are CTL state formulas.

Examples

- CTL formulas: $A \Box p$, $E \Box p$, $A \Box E \bigcirc p$, $E \Box E \Diamond p$.
- CTL* formulas: $A(p \supset \bigcirc E \Box p)$, $A(\Box \Diamond p)$, $E(\Box \Diamond p)$.

Semantics

Given a Kripke structure $K = \langle P, W, R, w^0, L \rangle$, the truth of a CTL* formula in a state w is defined as follows.

- For all w, we have $w \models$ **true** and $w \not\models$ **false**.
- $w \models p$ for $p \in P$ iff $p \in L(w)$.
- $w \models \neg p$ for $p \in P$ iff $p \notin L(w)$.
- $w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$.
- $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$.
- $w \models A\psi$ iff for every path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, we have $\pi \models \psi$.
- $w \models E\psi$ iff there exists a path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, such that $\pi \models \psi$.

- $\pi \models \varphi$ for a state formula φ, iff $w_0 \models \varphi$ where $\pi = w_0, w_1, \ldots$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$.
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$.
- $\pi \models \bigcirc \psi$ iff $\pi^1 \models \psi$.
- $\pi \models \psi_1 \, U \, \psi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \psi_2$ and for all $0 \leq j < i$, we have $\pi^j \models \psi_1$.
- $\pi \models \psi_1 \, \tilde{U} \, \psi_2$ iff for all $i \geq 0$ such that $\pi^i \not\models \psi_2$, there exists $0 \leq j < i$ such that $\pi^j \models \psi_1$.

Symbolic Model Checking : The Idea

- Since we are only dealing with finite structures, each state $w \in W$ can be encoded by a fixed length binary string (vector)

$$b_1 b_2 \dots b_n \text{ with } b_i \in \{0, 1\}$$

- A set of states can thus be represented by a Boolean function over $b_1, \dots, b_n$.
- For instance, for each $p \in P$, the set of states w such that $p \in L(w)$ can be represented by a Boolean function $p(b_1, \dots, b_n)$ true iff $b_1 b_2 \dots b_n$ is the encoding of a state w such that $p \in L(w)$.
- Similarly, the transition relation R can be represented by a Boolean function over twice as many Boolean variables $R(b_1, b_2, \dots b_n, b'_1, b'_2, \dots b'_n)$ true iff $b_1 b_2 \dots b_n$ and $b'_1 b'_2 \dots b'_n$ are encodings of states w and w' such that $R(w, w')$.
- Properties of K are then checked by computing with these Boolean functions.

Problems to Address

To make the idea work we need to address several questions.

- How does one check properties (e.g. temporal logic properties) of a Kripke structure represented by Boolean functions? What computations are required?
- What operations do these computations have to perform on Boolean functions?
- What is a good representation of Boolean functions that allows the required operations to be performed efficiently?

We will now address these three questions.

Computing with Symbolically Represented Structures : A Description language

In order to describe the computations that need to be performed, we will first introduce a simple logical language to describe Kripke structures.

Given a set P of atomic propositions, our language is built from the following elements:

- A set $V = \{x, y, z, \ldots\}$ of individual variables which will represent states,
- An interpreted predicate $I(x)$ that will be used to distinguish initial states,
- An interpreted unary predicate $p(x)$ for each atomic proposition $p \in P$,
- An interpreted binary predicate $R(x, y)$ representing the transition relation of the structure,

- A set $PV = \{X, Y, Z, \ldots\}$ of predicate variables with arbitrary numbers of arguments. If needed for clarity, the fact that a predicate variable X has n arguments will be made explicit by writing X^n.

We then define a first set of *simple* formulas.

- An interpreted predicate or a predicate variable applied to the required number of individual variables is a formula;
- If f and g are formulas, then so are $f \vee g$, $f \wedge g$, $\neg f$;
- If x is an individual variable, then $\exists x f$ and $\forall x f$ are formulas.

Examples

$$\forall x x'(p(x) \wedge R(x, x') \supset q(x'))$$

$$\exists x'(p(x') \wedge R(x, x'))$$

We now wish to extend the definition of the logic to allow the recursive definition of predicates. The idea is that these definitions are recursive equation on predicate variables which represent their least or greatest fixpoint solutions.

We proceed as follows.

Consider a n-ary predicate variable X^n. Let f be a formula in which X^n appears monotonically (under the scope of an even number of negations).

To use f as a definition of X^n (intuitively X^n is a solution of $X^n \equiv f(X^n)$), we need to specify which individual variables occurring in f should be used as arguments in the definition of X^n.

This is done with the following constructs.

- If f is a formula and $x_1, \ldots x_n$ are individual variables, then $\lambda x_1, \ldots x_n f$ is an n-ary predicate definition.

- If pd^n is an n-ary predicate definition monotone in X^n, then $\mu X^n[pd^n]$ is an n-ary (least-fixpoint) predicate definition and $\nu X^n[pd^n]$ is an n-ary (greatest-fixpoint) predicate definition.

- If pd^n is an n-ary (fixpoint) predicate definition and $x_1, \ldots, x_n$ are individual variables not occurring free in pd^n, then $pd^n(x_1, \ldots x_n)$ is a formula.

Example

$$\exists y(p(y) \wedge$$
$$\mu X^1[\lambda x(I(x) \vee \exists x'(X^1(x') \wedge R(x', x)))](y)$$

Semantics of the Description Language

Let f be a formula of our language built from a set of atomic propositions P. An interpretation $\mathcal{I}$ for f is a Kripke structure $K_{\mathcal{I}} = (P, W_{\mathcal{I}}, w^0_{\mathcal{I}}, R_{\mathcal{I}}, L_{\mathcal{I}})$ augmented with in interpretation $V_{\mathcal{I}} : V \to W_{\mathcal{I}}$ for free individual variables and an interpretation $PV_{\mathcal{I}} : PV \to 2^{\bigcup_i W^i_{\mathcal{I}}}$ for free predicate variables that respects arities.

Truth in an interpretation $\mathcal{I}$ is then given by the following rules.

- $I(x)$ is true iff $V_{\mathcal{I}}(x) = w^0$,
- $p(x)$ is true iff $p \in L_{\mathcal{I}}(V_{\mathcal{I}}(x))$,
- $R(x, y)$ is true iff $(V_{\mathcal{I}}(x), V_{\mathcal{I}}(y)) \in R_{\mathcal{I}}$,
- $X^n(x_1, \ldots, x_n)$ is true iff $(V_{\mathcal{I}}(x_1), \ldots, V_{\mathcal{I}}(x_n)) \in PV_{\mathcal{I}}(X^n)$
- The usual rules for Boolean operators.
- $\exists x f$ is true iff there is some interpretation $\mathcal{I}'$ that differs from $\mathcal{I}$ only by the state assigned to x such that f is true in $\mathcal{I}'$.

For n-ary predicate definitions, the semantics is a subset of W^n. Specifically, given an interpretation $\mathcal{I}$,

- For $\lambda x_1, \ldots x_n f$, $[\![\lambda x_1, \ldots x_n f]\!]_{\mathcal{I}}$ is the set of $(w_1, \ldots, w_n)$ such that f is true in $\mathcal{I}$ extended with $V_{\mathcal{I}}(x_1) = w_1$, ..., $V_{\mathcal{I}}(x_n) = w_n$,
- $[\![\mu X^n[pd^n]]\!]_{\mathcal{I}}$ is the smallest subset LFP of W^n such that when $\mathcal{I}$ is extended with $PV_{\mathcal{I}}(X^n) = LFP$, the semantics of pd^n is also LFP.
- $[\![\nu X^n[pd^n]]\!]_{\mathcal{I}}$ is the greatest subset GFP of W^n such that when $\mathcal{I}$ is extended with $PV_{\mathcal{I}}(X^n) = GFP$, the semantics of pd^n is also GFP.

And finally, we have

- $pd^n(x_1, \ldots x_n)$ is true in an interpretation $\mathcal{I}$ iff $(V_{\mathcal{I}}(x_1), \ldots V_{\mathcal{I}}(x_n)) \in [\![pd^n]\!]_{\mathcal{I}}$.

Evaluating Formulas of the Description Language

Imagine now that an interpretation is given by Boolean formulas. This means that, for some encoding of the states of the Kripke structure by fixed length strings of bits, the interpreted predicates, the value of individual variables, and the value of predicate variables are given by Boolean formulas.

A simple formula (no fixpoints) can then be directly transformed into a quantified Boolean formula representing its meaning.

Since quantification on Boolean variables can be eliminated

$$\exists b f(b) \equiv f(\mathbf{true}) \lor f(\mathbf{false})$$

this means that a simple formula can be transformed into a Boolean formula representing its meaning.

For general formulas, we also have to deal with predicate definitions.

- For a predicate definition $\lambda x_1, \ldots x_n f$ the formula is simply the formula for f in which the individual variables $x_1, \ldots x_n$ are not interpreted.
- For a least fixpoint predicate definition $\mu X[pd]$, one computes the sequence of formulas

  $$\emptyset \subseteq pd(\emptyset) \subseteq pd^2(\emptyset) \subseteq pd^3(\emptyset) \subseteq \ldots$$

  which converges (up to Boolean equivalence) after a finite number of steps since we are working on a finite domain.
- For a greatest fixpoint predicate definition $\nu X^n[pd^n]$, one computes the sequence of formulas

  $$W_{\mathcal{I}} \supseteq pd(W_{\mathcal{I}}) \supseteq pd^2(W_{\mathcal{I}}) \supseteq pd^3(W_{\mathcal{I}}) \supseteq \ldots$$

Example: Computing Reachable States Symbolically

The set of reachable states of a symbolically defined Kripke structure is given by

$$\mu Z[\lambda(y)(I(y) \vee \exists x(Z(x) \wedge R(x, y)))](z).$$

Note that evaluating this formula only requires a number of steps equal to the diameter of the Kripke structure. However, the Boolean formulas appearing in the computation *can become huge*.

To check if a state satisfying p is reachable, one could compute the value of

$$\exists z(I(z) \wedge \\ \mu Z[\lambda(y)(p(y) \vee \exists x(Z(x) \wedge R(y, x)))](z)),$$

which amounts to doing a backwards search through the graph.

Checking Branching-Time (CTL) Formulas

For checking a CTL formula φ on a Kripke structure, we proceed by computing predicate definitions corresponding to the formulas of the closure of φ, starting with the innermost subformulas of φ.

Assuming the computation has already been done for the immediate subformulas φ_1 and φ_2 of the formula being considered, we use the following definitions.

- For $E \bigcirc \varphi_1$, we have

$$\lambda x(\exists y(R(x,y) \wedge \varphi_1(y)))$$

- For $A \bigcirc \varphi_1$, we have

$$\lambda x(\forall y(R(x,y) \supset \varphi_1(y)))$$

- For $E(\varphi_1 \, U \, \varphi_2)$, we have

  $\mu Z[\lambda x(\varphi_2(x) \vee (\varphi_1(x) \wedge \exists y(R(x,y) \wedge Z(y))))$

- For $E(\varphi_1 \, \tilde{U} \, \varphi_2)$, we have

  $\nu Z[\lambda x(\varphi_2(x) \wedge (\varphi_1(x) \vee \exists y(R(x,y) \wedge Z(y))))$

- For $A(\varphi_1 \, U \, \varphi_2)$, we have

  $\mu Z[\lambda x(\varphi_2(x) \vee (\varphi_1(x) \wedge \forall y(R(x,y) \supset Z(y))))$

- For $A(\varphi_1 \, \tilde{U} \, \varphi_2)$, we have

  $\nu Z[\lambda x(\varphi_2(x) \wedge (\varphi_1(x) \vee \forall y(R(x,y) \supset Z(y))))$

Note that the computation is done backwards even for AU and $E\tilde{U}$ formulas. This is possible efficiently since we are handling a set of states at each step.

Required Boolean Operations

To use the verification method we have outlined above, we need to be able to perform the following operations on Boolean formulas.

1. Taking Boolean combinations of formulas.
2. Existential and Universal quantification.
3. Checking implication (to know when to terminate the computation of a fixpoint).
4. Computing the truth value of closed formulas (to get the result of the verification).

Items (1) and (2) are trivial if the required operations are in the language. However items (3) and (4) are then hard to compute.

It is more effective to restrict formulas to a normal from for which (3) and (4) are easy and to perform the computation required to reduce formulas to normal form each time (1) and (2) are needed.

A representation of Boolean Functions: Ordered Binary Decision Diagrams (OBDD)

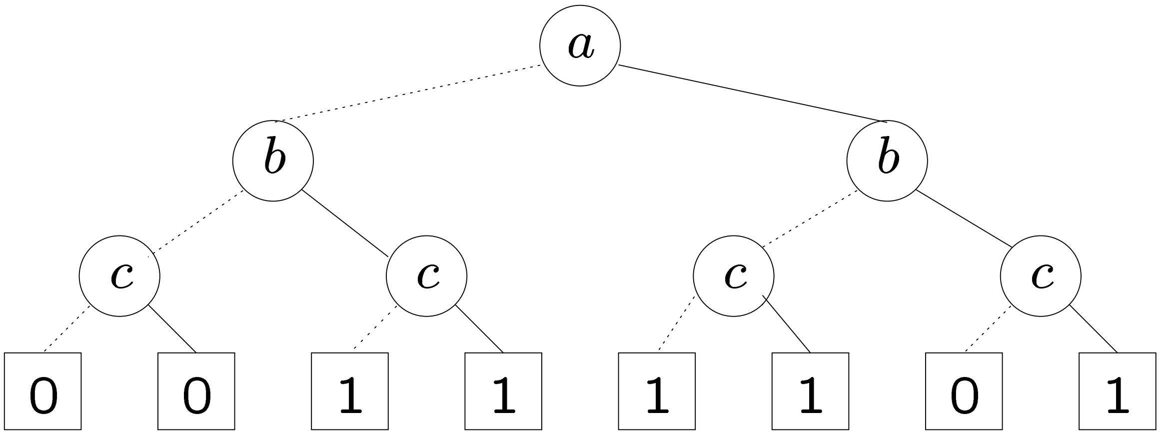
Consider a Boolean function of k variables $b_1, \ldots b_k$ and an ordering of these variables $b_{o(1)}, b_{o(2)}, \ldots, b_{o(k)}$.

Any Boolean function $f(b_1, \ldots b_k)$ can be represented by a full binary tree of depth k in which

- The leaves are labeled by 0 (false) or 1 (true).
- The interior nodes at depth $i-1$ are labeled by $b_{o(i)}$ (the root has depth 0).
- The outgoing edges of each interior node are respectively labeled by 0 and 1.
- The label of a leaf is the value of the function f for input values corresponding to the labels on the path leading to that leaf.

Example

For $f(a, b, c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$, the tree is



Dotted lines represent a value 0 for the variable labeling the node from which they originate; solid lines represent a value 1.

Reducing OBDDs

The following rules can be used to reduce OBDDs.

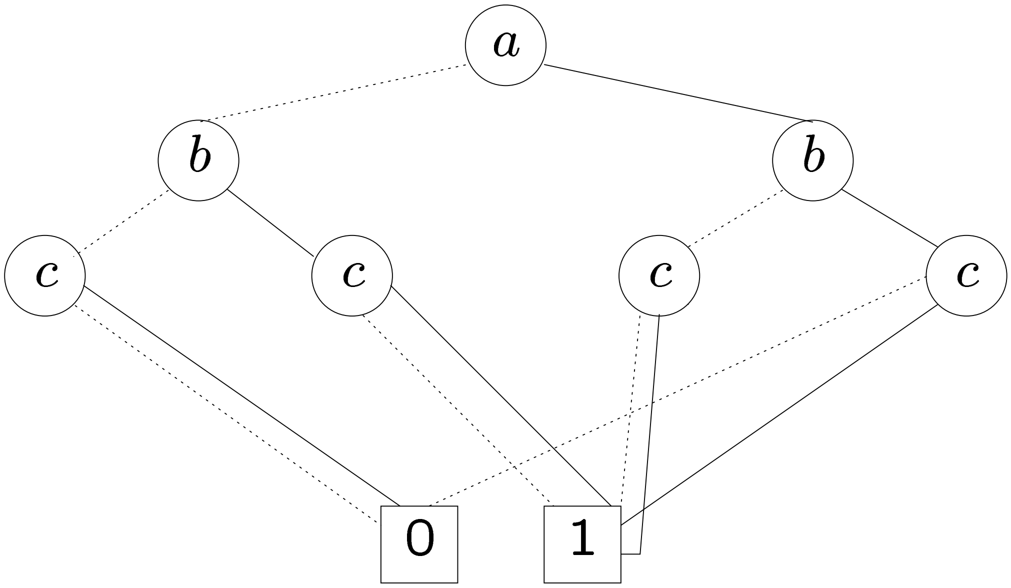
- **Eliminate Redundant Leaf Nodes.** That is, keep at most one leaf node labeled 0 and one leaf node labeled 1.
- **Remove Duplicate Interior Nodes.** If two interior nodes labeled by the same variable have identical successors, eliminate one of these nodes and redirect its incoming edges to the other.
- **Remove Redundant Tests.** If both successors of a node are the same node, eliminate that node and redirect its incoming edges to the unique successor.

These rules are applied until no further transformations are possible.

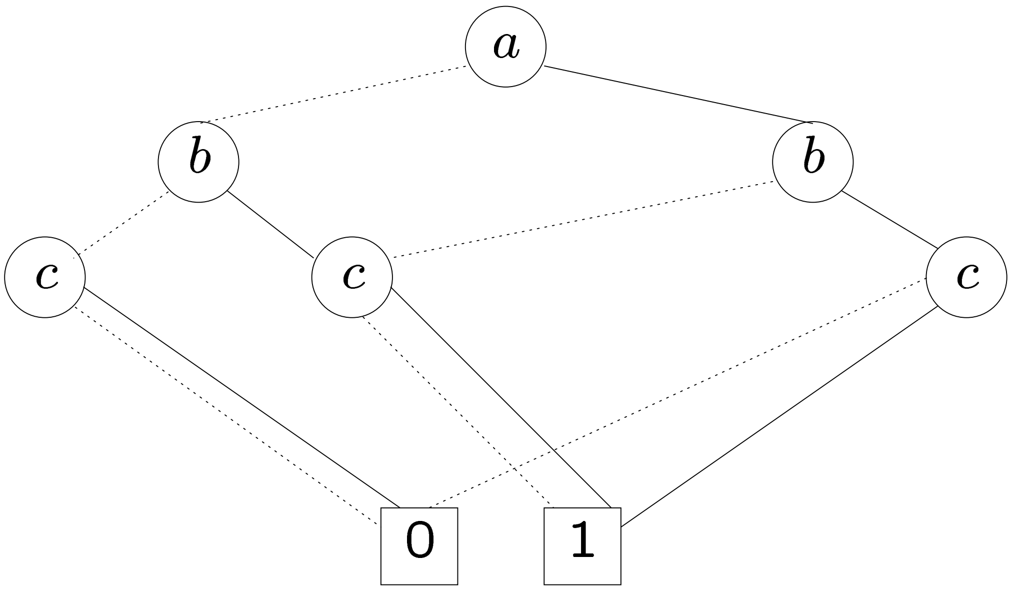
It can be shown that for a given Boolean function and variable order, the maximally reduced OBDD is unique.

Example

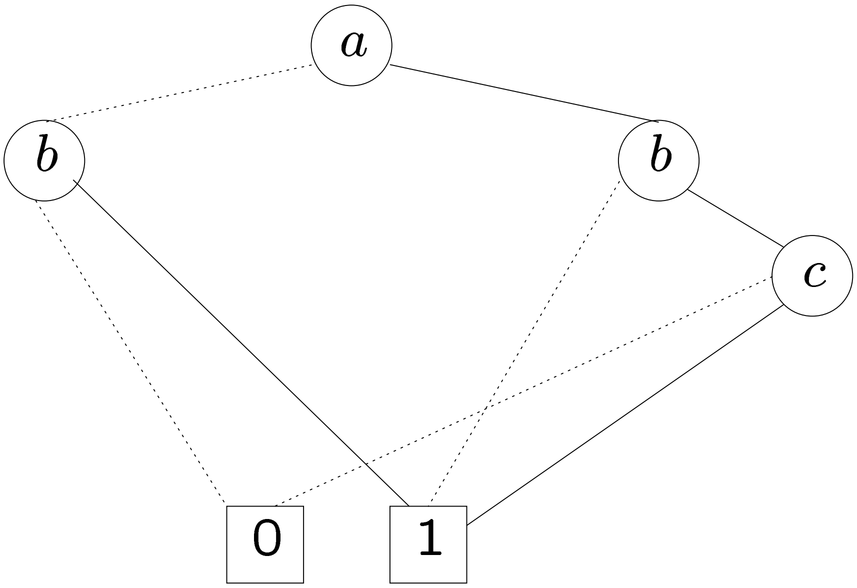
Applying the first transformation to the OBDD for $f(a,b,c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$ yields



Applying the second leads to



And applying the third yields



Operations on OBDDs

Complementation
Complement the labels of leaves.

Binary Boolean Operations ($\wedge, \vee$)
These operations are performed by computing a product of the OBDDs representing their arguments. Precisely, assume the OBDDs for the arguments are defined by

- An ordered set V of variables;
- A set of *interior nodes* N;
- The 0 and 1 leaves: $L0$ and $L1$;
- A labeling of interior nodes by variables, $var : N \to V$;
- Low and high successor functions for interior nodes, $lo : N \to N \cup \{L0, L1\}$ and $hi : N \to N \cup \{L0, L1\}$ (leaf nodes have implicitly themselves as low and high successors).

Given OBDDs $B_1 = (V, N_1, var_1, lo_1, hi_1)$ and $B_1 = (V, N_1, var_1, lo_1, hi_1)$, the product OBDD $B_1 \times B_2 = (V, N, var, lo, hi)$ is defined by

- $N = N_1 \times N_2$,
- $var((n_1, n_2)) = min\{var(n_1), var(n_2)\}$ (treat $L0$ and $L1$ as labeled by a dummy maximal variable).
- $$lo((n_1, n_2)) = \begin{cases} (lo(n_1), lo(n_2)) \text{ if} \\ \qquad var(n_1) = var(n_2) \\ (lo(n_1), n_2) \text{ if} \\ \qquad var(n_1) < var(n_2) \\ (n_1, lo(n_2)) \text{ if} \\ \qquad var(n_1) > var(n_2) \end{cases}$$

- $$hi((n_1,n_2)) = \begin{cases} (hi(n_1), hi(n_2)) \text{ if} \\ \qquad var(n_1) = var(n_2) \\ (hi(n_1), n_2) \text{ if} \\ \qquad var(n_1) < var(n_2) \\ (n_1, hi(n_2)) \text{ if} \\ \qquad var(n_1) > var(n_2) \end{cases}$$

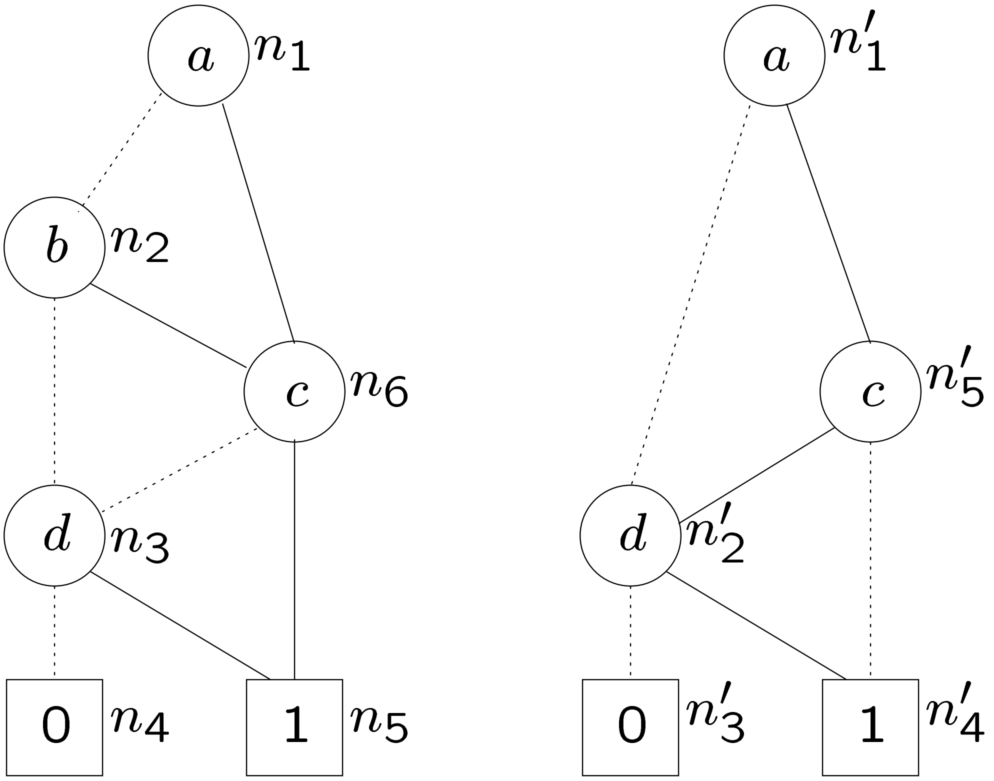
In practice the product is computed incrementally from its root node and only reachable nodes are considered.

Nodes with at least one component that is a leaf are then simplified according to the Boolean function being computed.

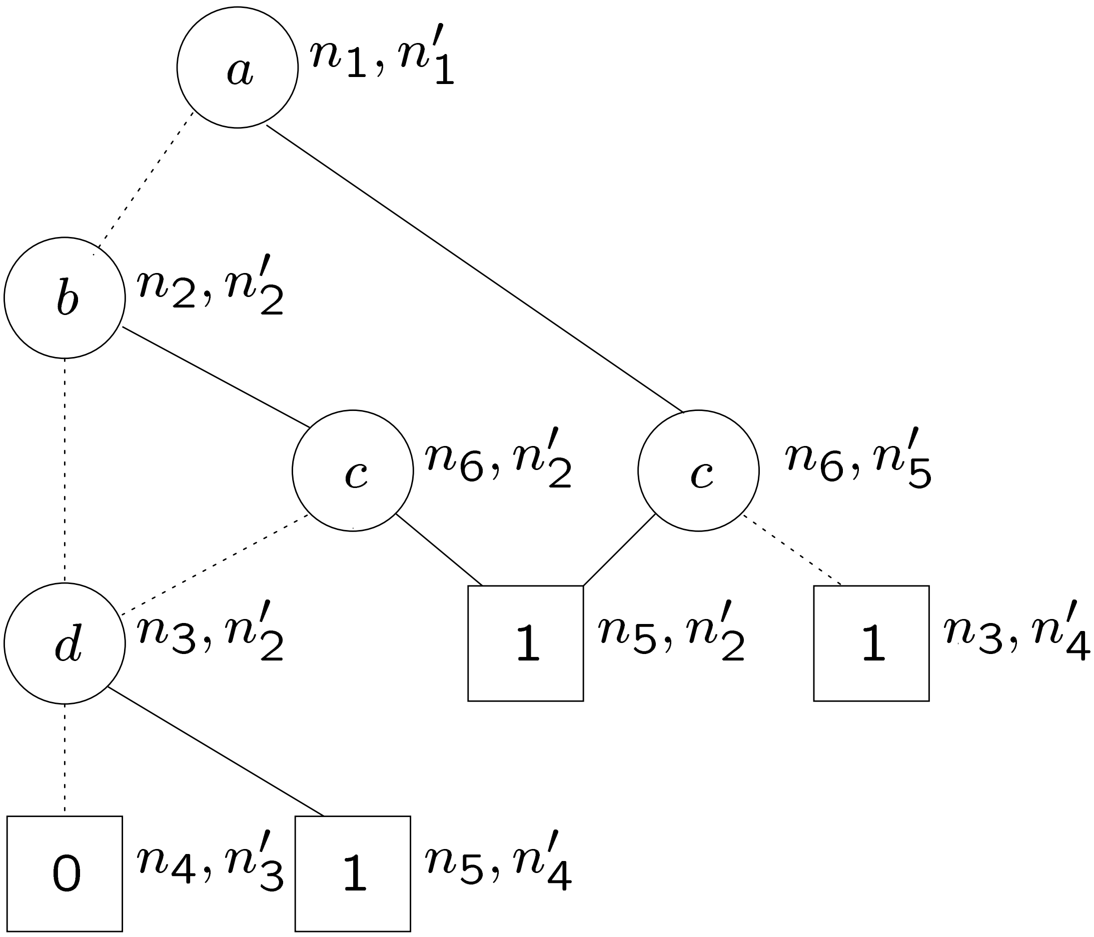
Finally, the OBDD that is obtained is reduced.

Example

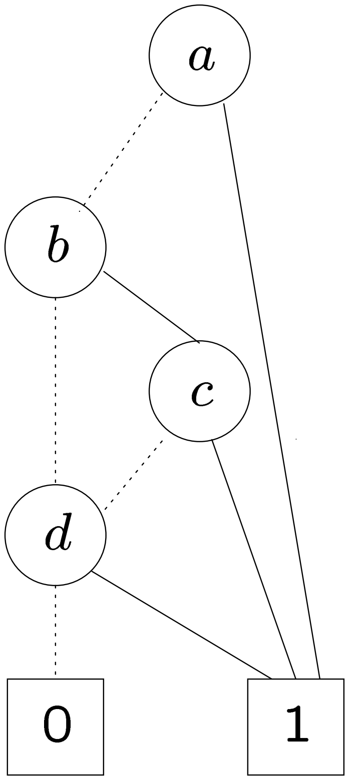
Consider the two following OBDDs



Their product simplified for $\vee$ is



And reduced it becomes



Quantification

To implement quantification, one uses the *restrict* operator that forces the value of a given variable to be 0 or 1. The notation for the restriction of a variable x in a function f is

$$f|_{x \leftarrow 0} \text{ or } f|_{x \leftarrow 1}$$

To compute the effect of restricting a variable x in a function represented by a OBDD, one eliminates all nodes labeled by x and redirects their incoming edges to the appropriate successor.

For quantification one then uses the following identities

$$\begin{aligned} \forall x f &\equiv f|_{x \leftarrow 0} \wedge f|_{x \leftarrow 1} \\ \exists x f &\equiv f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1} \end{aligned}$$

Implication

To check if $f_1 \supset f_2$, one computes the OBDD for $\neg f_1 \vee f_2$ and checks that the result is valid (is the Boolean function whose value is always 1).

Checking for Validity, Satisfiability, Unsatisfiability

- Validity: all accessible leaves are labeled 1.
- Satisfiability: at least one accessible leaf is labeled 1.
- Unsatisfiability: all accessible leaves are labeled 0.

Complexity

- Given two reduced OBDDs f_1 and f_2, all Boolean operations (including the final reduction steps) can be compute in time $O(|f_1| \cdot |f_2|)$ ($O(|f_1|^2)$ for quantification).
- This does not prevent the OBDDs from becoming very large after a number of operations.
- Some functions do not have compact OBDD representations.
- Variable order can hae a very substantial order on the size of OBDDs.

A whole technology for efficiently manipulating OBDDs has been developed

Model Checking: Conclusions

- Successes: hardware, protocols, reactive systems, . . .
- Limit: all model checking techniques, enumerative or symbolic, fail when the state descriptors of the system are larger than a few K *bits*.
- Conclusions: model checking can only work on models, not on programs.