The Algorithmic Verification of Reactive Systems

Lecture 8:

Symbolic Verification Algorithms

Background and Principle

- Our goal is to check properties of a transition system obtained from a concurrent program.
- We assume that the properties of interest are expressed in terms of a finite set P of atomic propositions on states.
- Thus, the structures we are dealing with are transitions systems with states labeled by atomic propositions, i.e. Kripke structures

$$K = \langle P, W, R, w^0, L \rangle$$

  where $w^0 \in W$, $R \subseteq W \times W$ and is total, and $L : W \to 2^P$.

- The idea is to represent K *symbolically*, i.e. by formulas, and to compute with these formulas to check properties of K.

- Assume we are only dealing with finite structures. Each state $w \in W$ can thus be encoded by a fixed length binary string (vector)

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

- A set of states can thus be represented by a Boolean function over $b_1, \ldots, b_n$.
- 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 $p(b_1, \ldots, b_n)$ true iff $b_1 b_2 \ldots b_n$ 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 $R(b_1, b_2, \ldots b_n, b'_1, b'_2, \ldots b'_n)$ true iff $b_1 b_2 \ldots b_n$ and $b'_1 b'_2 \ldots b'_n$ 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 $V_{\mathcal{I}} : V \rightarrow W_{\mathcal{I}}$ for free individual variables and an interpretation $PV_{\mathcal{I}} : PV \rightarrow 2^{\bigcup_i W^i_{\mathcal{I}}}$ 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

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

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

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

$$\begin{array}{l}\exists z(I(z)\wedge \\ \quad \mu Z[\lambda(y)(p(y) \vee \exists x(Z(x) \wedge R(y,x)))](z)),\end{array}$$

which amounts to doing a backwards search through the graph.

To check a safety property, one needs to check reachability of an accepting state in the structure obtained from the combination of the program and the automaton for the property. Assuming that the predicate $f(x)$ characterizes accepting states, this can be done by evaluating

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

Checking LTL Properties

To check an LTL property, one adds the automaton for the formulas as an observer to the system. The Kripke structure one obtains is thus a Kripke structure with a set of accepting states for which we have to check repeated reachability.

For this, one first computes the reachability relation R^+.

$$R^+(x,y) \equiv \mu Z^2[\lambda(xy)(R(x,y) \vee \exists z(R(x,z) \wedge Z(z,y)))](x,y)$$

Accepting states nontrivially reachable from themselves are then given by

$$f(x) \wedge R^+(x,x).$$

And, the fact that the Büchi automaton is nonempty is represented by

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

Speeding up the Computation

The following formula speeds up the computation of R^+:

$$R^+(x,y) \equiv \\ \mu Z^2[\lambda(xy)(R(x,y) \vee \exists z(Z(x,z) \wedge Z(z,y)))](x,y)$$

Indeed, at each step it doubles the length of the paths that are considered. However, the formulas obtained might be too large for the computation to succeed.

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)

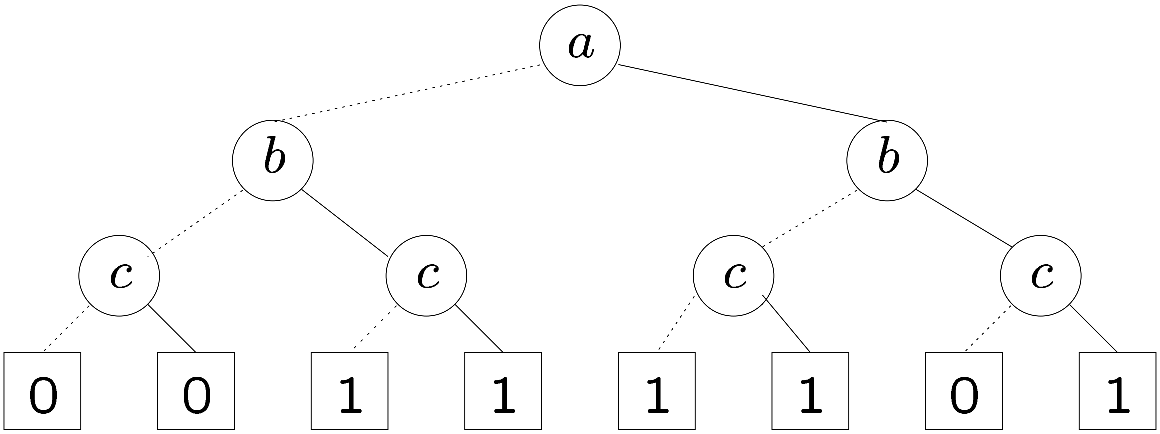
Consider a Boolean function of k variables $b_1, \ldots b_k$ and an ordering of these variables $b_{o(1)}, b_{o(2)}, \ldots, b_{o(k)}$.

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.

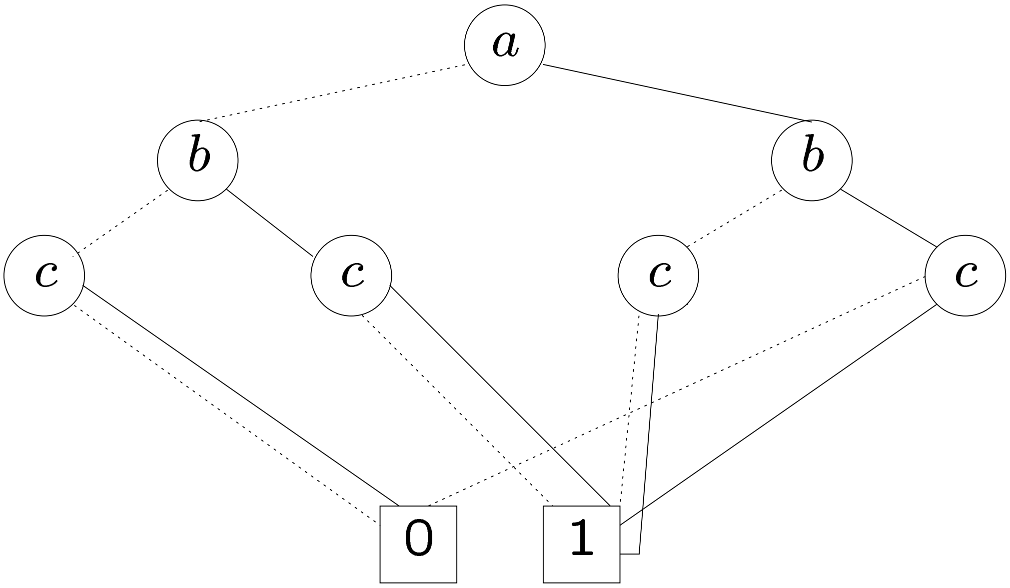
- **Eliminate Redundant Leaf Nodes.** That is, keep at most one leaf node labeled 0 and one leaf node labeled 1.
- **Remove Duplicate Interior Nodes.** If two interior nodes labeled by the same variable have identical successors, eliminate one of these nodes and redirect its incoming edges to the other.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

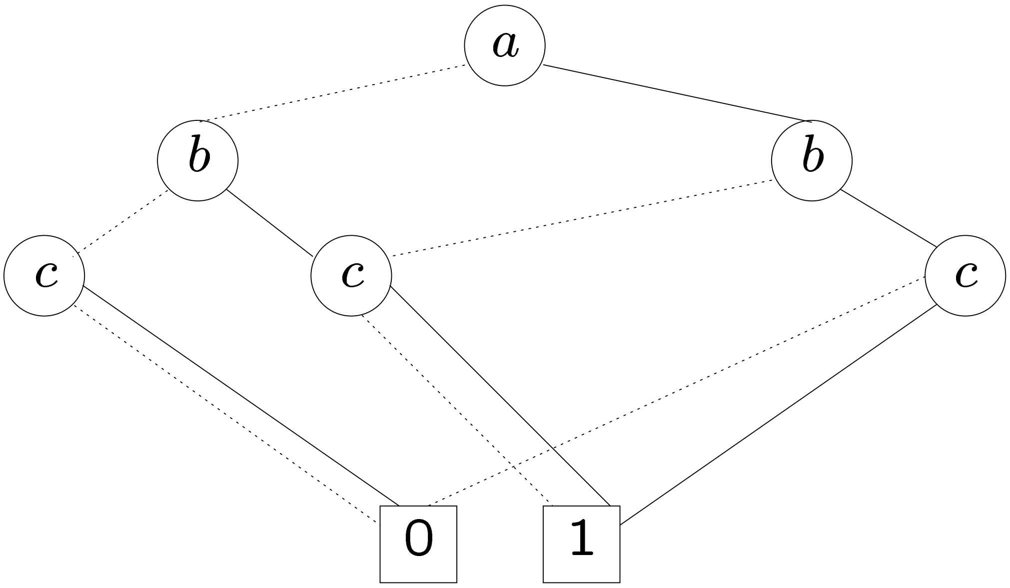
It can be shown that for a given Boolean function and variable order, the maximally reduced OBDD is unique.

Example

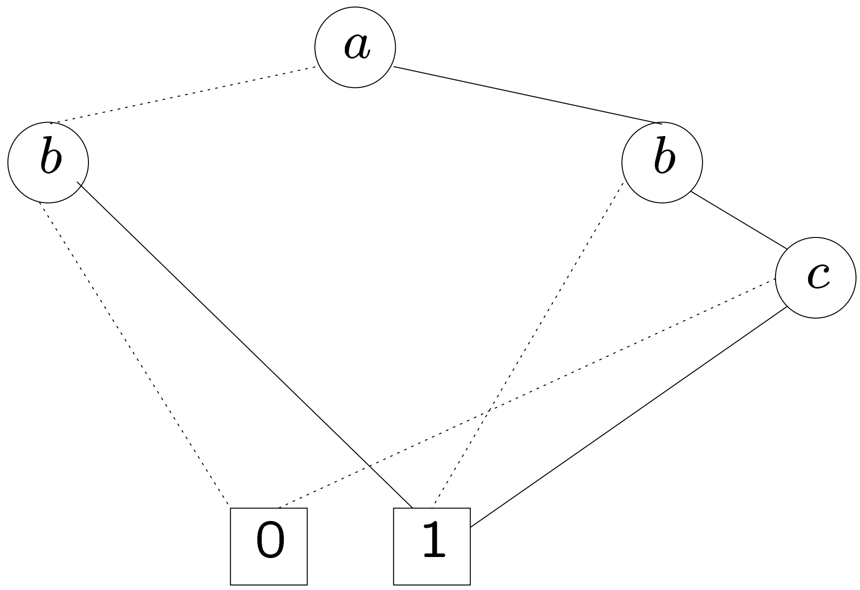
Applying the first transformation to the OBDD for $f(a,b,c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$ yields



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, $var : N \rightarrow V$;
- Low and high successor functions for interior nodes, $lo : N \rightarrow N \cup \{L0, L1\}$ and $hi : N \rightarrow N \cup \{L0, L1\}$ (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).
-
$$lo((n_1, n_2)) = \begin{cases} (lo(n_1), lo(n_2)) \text{ if} \\ \qquad var(n_1) = var(n_2) \\ (lo(n_1), n_2) \text{ if} \\ \qquad var(n_1) < var(n_2) \\ (n_1, lo(n_2)) \text{ if} \\ \qquad var(n_1) > var(n_2) \end{cases}$$

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

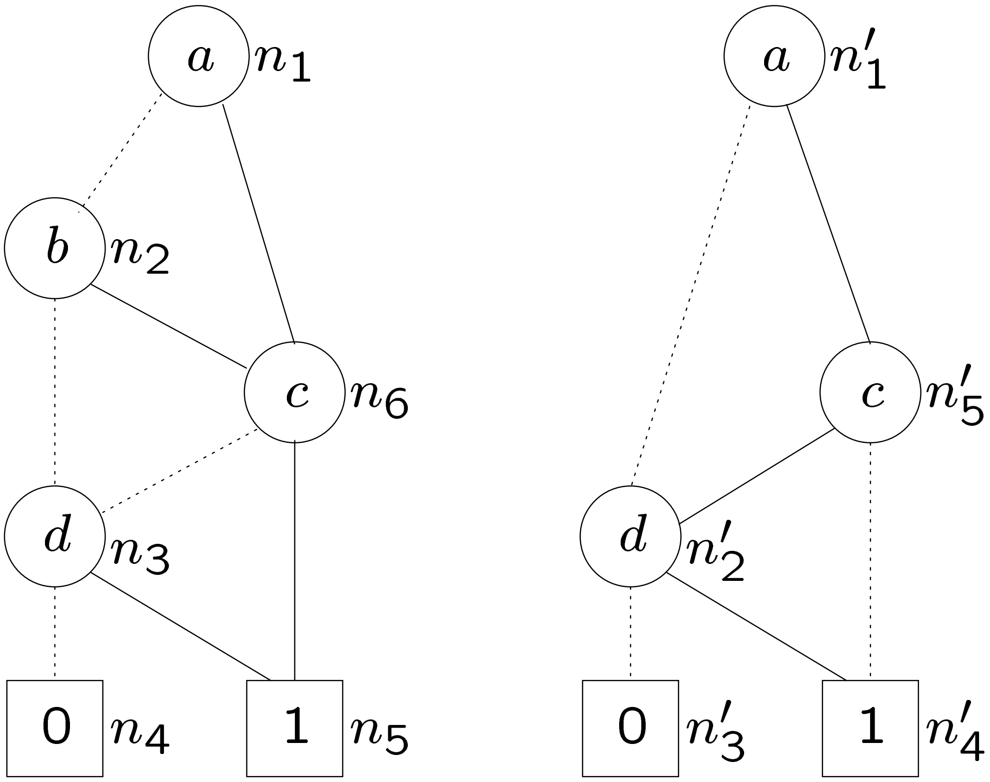
In practice the product is computed incrementally from its root node and only reachable nodes are considered.

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

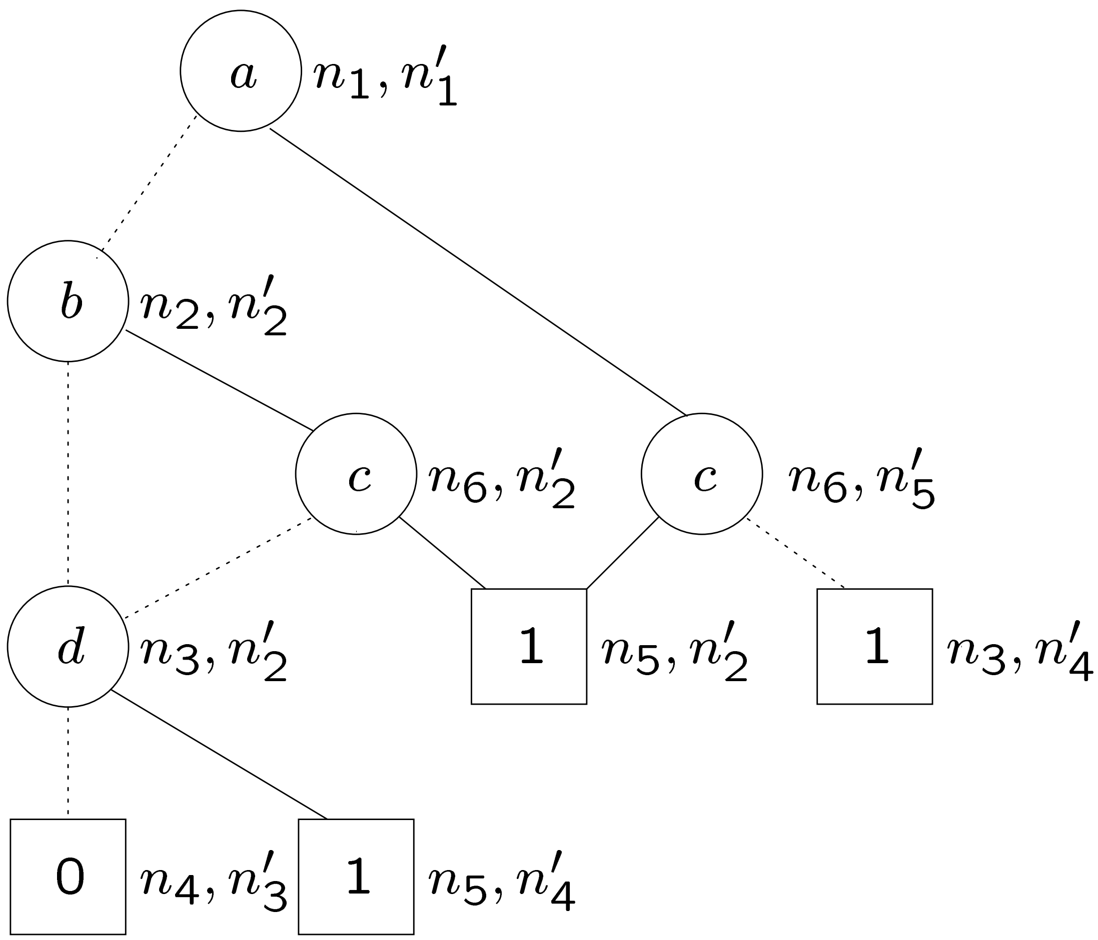
Finally, the OBDD that is obtained is reduced.

Example

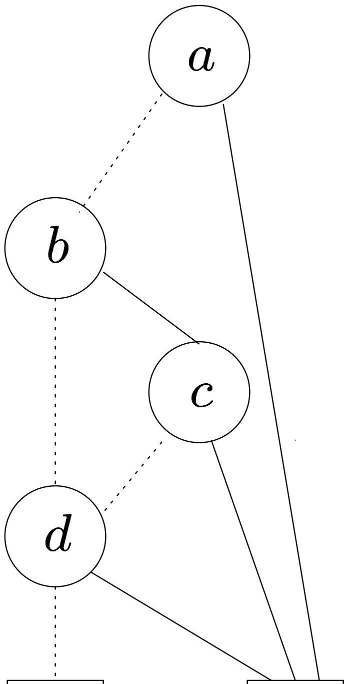
Consider the two following OBDDs



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

Successes and Limits

- OBDD-based symbolic model checking is quite successful for hardware verification
- For software verification, one quickly runs into the problem of very long state vectors (hundreds of bits) which tend to make the OBDDs unmanageable.
- Note that when constructing the OBDDs representing a program, one has to use a state encoding derived from the program text. Because of this, the number of bits used is often much larger than the logarithm of the size of the set of reachable states.