The Symbolic Model Checking Algorithm

Gregory Gelfond

Knowledge Representation Lab
Texas Tech University

Overview

- Modeling Domains (Kripke Structures)
- Specifying Properties (Temporal Logic CTL)
  - Syntax
  - Semantics

- CTL Model Checking
  - Ordered Binary Decision Diagrams
  - Quantified Boolean Formulas
  - CTL Model Checking Algorithm

Kripke Structures

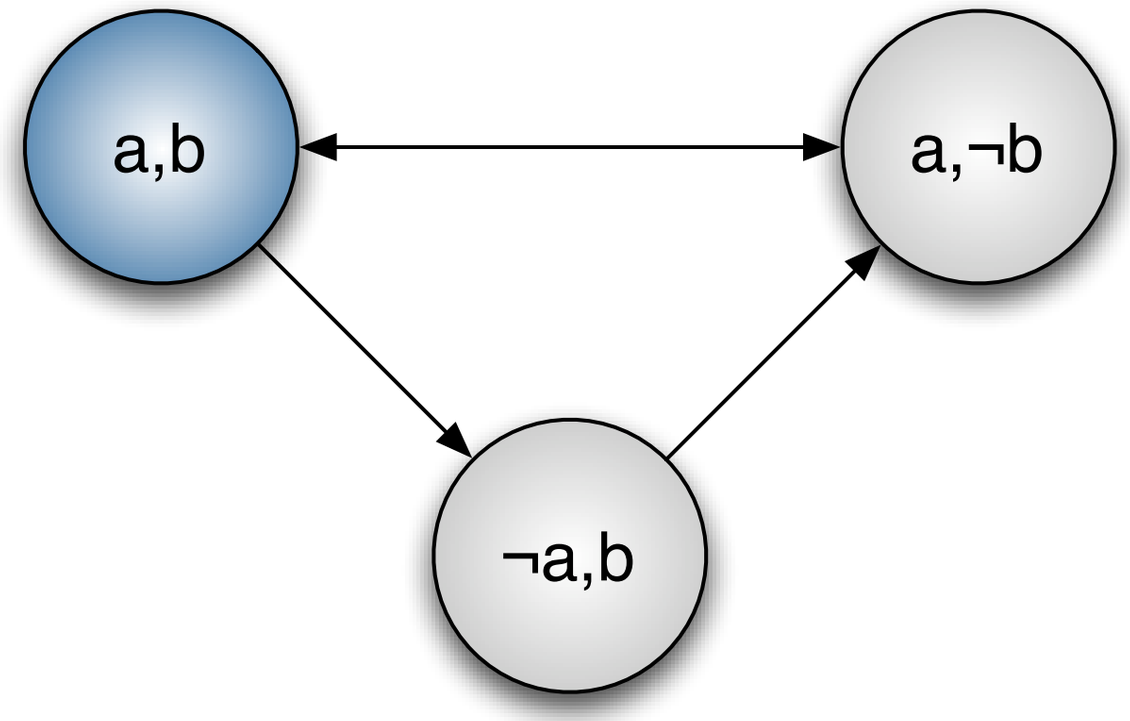
- In model checking, domains are represented by Kripke structures.
- Intuitively, a Kripke structure specifies the states of the world, and transitions from one state to another.

Definition

Let *AP* be a set of atomic propositions. A *Kripke structure*, *M*, over *AP* is a 4-tuple (S, S_0, R, L) where:

- S is a finite set of states
- $S_0 \subseteq S$ is the set of initial states
- $R \subseteq S \times S$ is a *total* transition relation
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state

Example

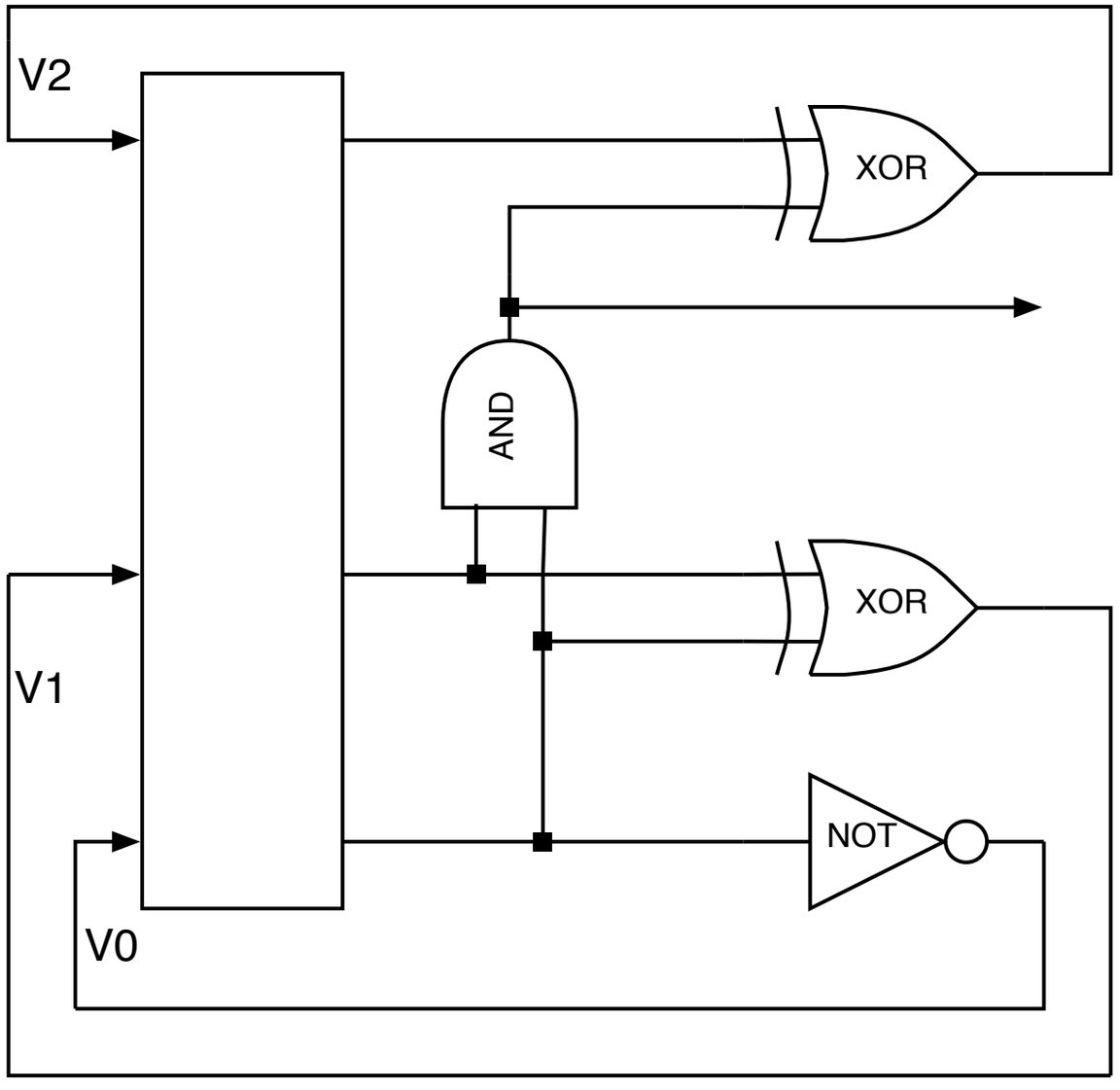


- Each state is labeled with the set of literals true in that state.

- S = {(a,b), (a,¬b), (¬a,b)}

- R = {((a,b),(a,¬b)), ((a,b),(¬a,b)), ((a,¬b),(a,b)), ((¬a,b),(a,¬b))}

Synchronous Modulo-8 Counter



A Kripke structure, *M*, that represents our counter consists of:

- $AP = \{v_i = x : x \in \{0,1\} \text{ and } i \in [0..2]\}$
- $S = \{ (x,y,z) : x, y, z \in \{0,1\} \}$.
- $L(s) = \{v_2 = x, v_1 = y, v_0 = z\}$
- A transition relation, *R*, defined as follows:

First we define the transitions for each state variable:

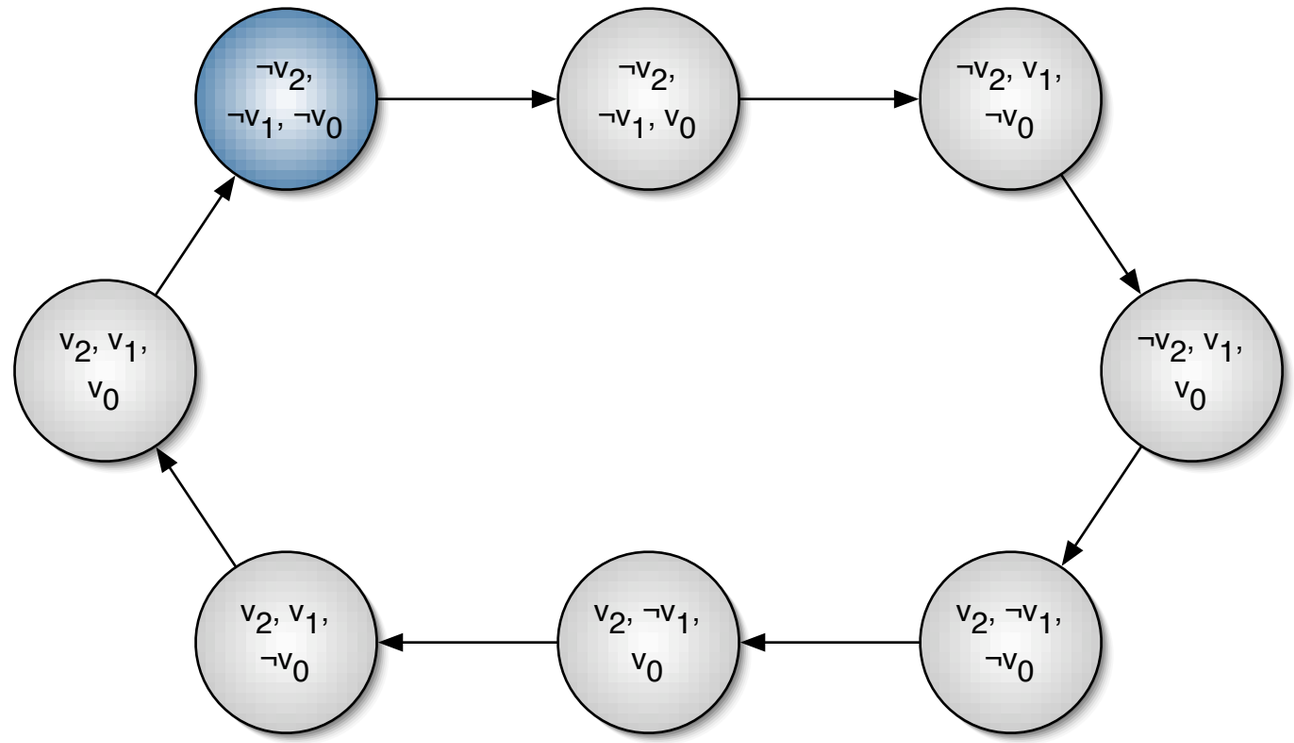
- $R_0(V,V') = (v'_0 \equiv \neg v_0)$
- $R_1(V,V') = (v'_1 \equiv v_0 \oplus v_1)$
- $R_2(V,V') = (v'_2 \equiv (v_0 \wedge v_1) \oplus v_2)$

where v'_i are new variables.

Since our circuit is synchronous, the formula describing the transition relation for the entire circuit is:

$$R_0(V,V') \wedge R_1(V,V') \wedge R_2(V,V')$$

The corresponding Kripke structure is shown on the right:



The Temporal Logic CTL

- In model checking, properties of paths are specified using a *temporal logic*.
- The particular temporal logic we will introduce is called **CTL**.

Syntax

CTL formulas are composed with *path quantifiers*, and *temporal operators*.

Path quantifiers are used to describe the branching structure of trees. There are two path quantifiers:

A - universal path quantifier

E - existential path quantifier

Temporal operators are used to specify properties of a path. There are five temporal operators:

- ***X*** - the "next time" operator (***X****p* specifies that *p* holds in the second state of the path)

- ***F*** - the "future time" operator (***F****p* specifies that *p* holds at some state in the path)

- ***G*** - the "always" operator (***G****p* specifies that *p* holds at every state in the path)

- ***U*** - the "until" operator (*p* ***U*** *g* specifies that *g* holds at some state in the path, and *p* is guaranteed to hold along the path up to the first state in which *g* holds)

- ***R*** - the "release" operator (*p* ***R*** *g* specifies that *g* holds along the path up to and including the first state where *p* holds)

Let *AP* be the set of atomic propositions.

- If $p \in AP$, then p is a formula
- If f and g are formulas, then: $\neg f$, $f \vee g$, and $f \wedge g$, are formulas
- ***EX****f*, ***EF****f*, ***EG****f*, ***E****[f* ***U*** *g]*, ***E****[f* ***R*** *g]* are formulas
- ***AX****f*, ***AF****f*, ***AG****f*, ***A****[f* ***U*** *g]*, ***A****[f* ***R*** *g]* are also formulas

Examples

- The following are examples of valid CTL formulas (p and $q \in AP$):

  p, $p \wedge q$, $\mathbf{AG}p$, $\mathbf{EX}p \vee \mathbf{AG}q$

- The following are not valid formulas:

  $\mathbf{A}p$, $\mathbf{X}q \vee \mathbf{G}r$

Semantics

The semantics of CTL will be given with respect to a Kripke structure *M*. A *path in M*, is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$.

π^i will be used to denote the *suffix* of π starting at s_i.

If *f* is a CTL formula, then $M,s \Rightarrow f$ means that *f* holds in state *s* in the Kripke structure *M*.

The relation $\Rightarrow$ is defined as follows:

- $M,s_0 \Rightarrow p \equiv p \in L(s_0)$
- $M,s_0 \Rightarrow \neg p \equiv \neg(M,s_0 \Rightarrow p)$
- $M,s_0 \Rightarrow p \wedge q \equiv M,s_0 \Rightarrow p$ and $M,s_0 \Rightarrow q$
- $M,s_0 \Rightarrow p \vee q \equiv M,s_0 \Rightarrow p$ or $M,s_0 \Rightarrow q$

- $M,s_0 \Rightarrow \mathbf{EX}p \equiv \exists s_1 : (s_0,s_1) \in$ R, and $M,s_1 \Rightarrow p$

- $M,s_0 \Rightarrow \mathbf{EG}p \equiv \exists$ path π starting at s_0 such that $\forall s_i \in \pi\ M,s_i \Rightarrow p$

- $M,s_0 \Rightarrow \mathbf{E}[p\ \mathbf{U}\ g] \equiv \exists$ path π starting at s_0, $\exists i \geq 0$ such that $M,s_i \Rightarrow g$, and $\forall j : 0 \leq j < i$, $M,s_j \Rightarrow p$

There are seven CTL operators:

- ***AX*** and ***AF***
- ***EF*** and ***AG***
- ***AU***, ***AR*** and ***ER***

Each of these operators can be viewed as shorthand for the following:

- $\mathbf{AX}f \equiv \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f \equiv \mathbf{E}[True\ \mathbf{U}\ f]$
- $\mathbf{AG}f \equiv \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f \equiv \neg\mathbf{EG}(\neg f)$

- $\boldsymbol{A}[f\ \boldsymbol{U}\ g] \equiv \neg\boldsymbol{E}[\neg g\ \boldsymbol{U}\ (\neg f \wedge \neg g)] \wedge \neg\boldsymbol{E}\boldsymbol{G}(\neg g)$

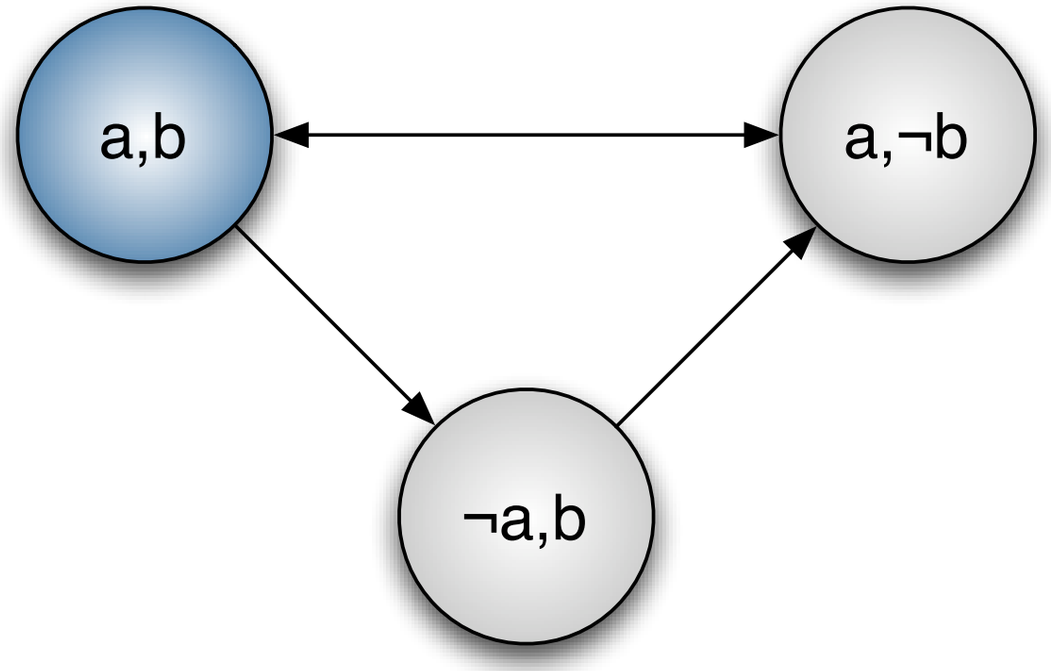
- $\boldsymbol{A}[f\ \boldsymbol{R}\ g] \equiv \neg\boldsymbol{E}[\neg f\ \boldsymbol{U}\ \neg g]$

- $\boldsymbol{E}[f\ \boldsymbol{R}\ g] \equiv \neg\boldsymbol{A}[\neg f\ \boldsymbol{U}\ \neg g]$

Ordered Binary Decision Diagrams

Consider the following Kripke structure *M*:

- Each state is labeled with the set of literals true in that state.
- S = {(a,b), (a,¬b), (¬a,b)}
- R = {((a,b),(a,¬b)),
  ((a,b),(¬a,b)),
  ((a,¬b),(a,b)),
  ((¬a,b),(a,¬b))}



- One method of representing *M* would be to specify the transition relation as a table.

- Unfortunately, for complex structures the table becomes too large.

- Consequently, a more efficient data structure is needed.

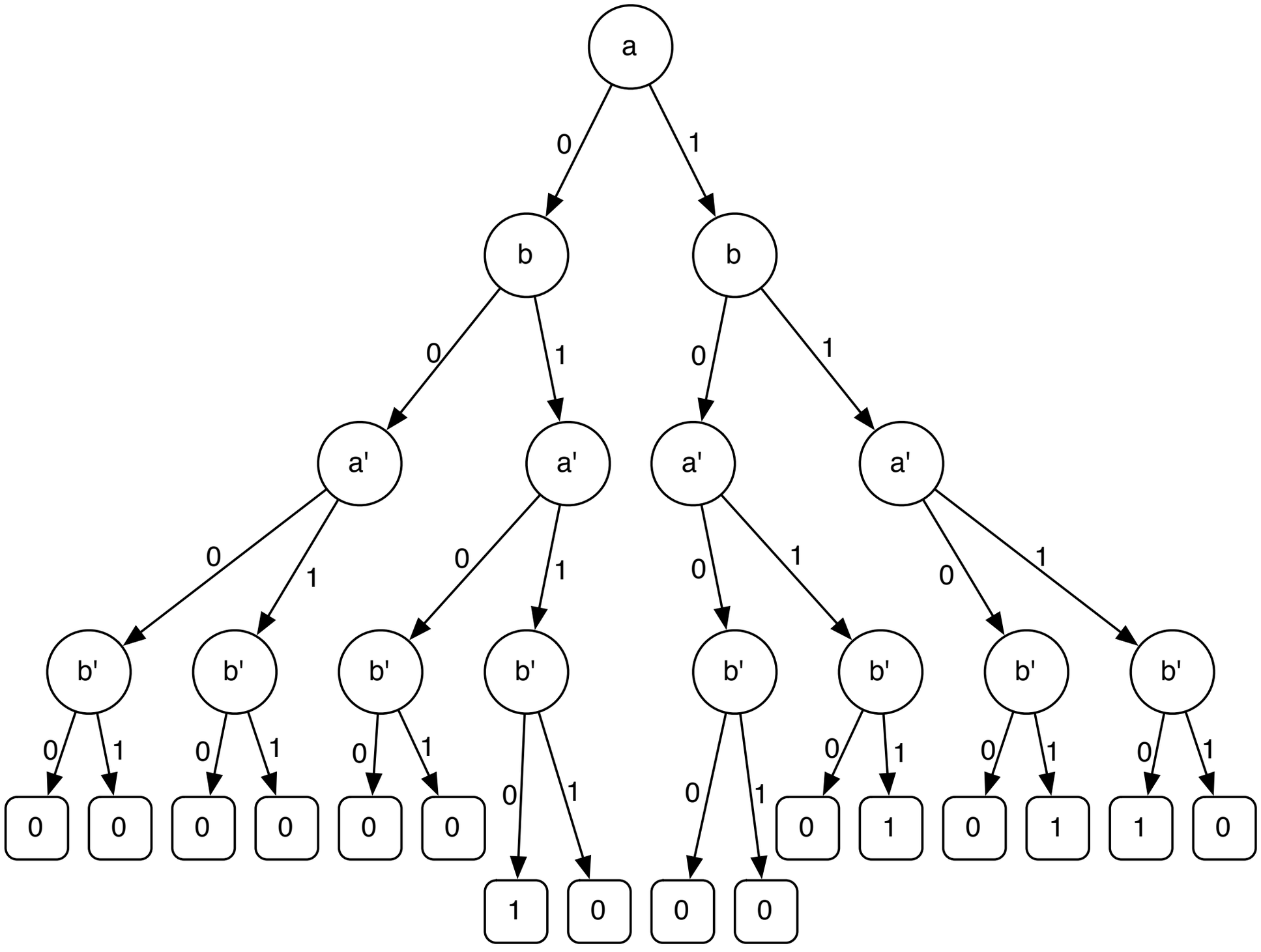
$R = \{((a,b),(a,\neg b)), ((a,b),(\neg a,b)), ((a,\neg b),(a,b)), ((\neg a,b),(a,\neg b))\}$

By introducing a pair of *next-state variables*, a' and b', we can obtain the formula *F(R)*:

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

Valid transitions of *M* are models of *F(R)*.

F(R) can be represented by a binary decision tree:


a
0
1
b
b
0
1
0
1
a'
a'
a'
a'
0
1
0
1
0
1
0
1
b'
b'
b'
b'
b'
b'
b'
b'
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
0
0
0
0
0
1
0
0
0
1
0
1
0
1
1
0

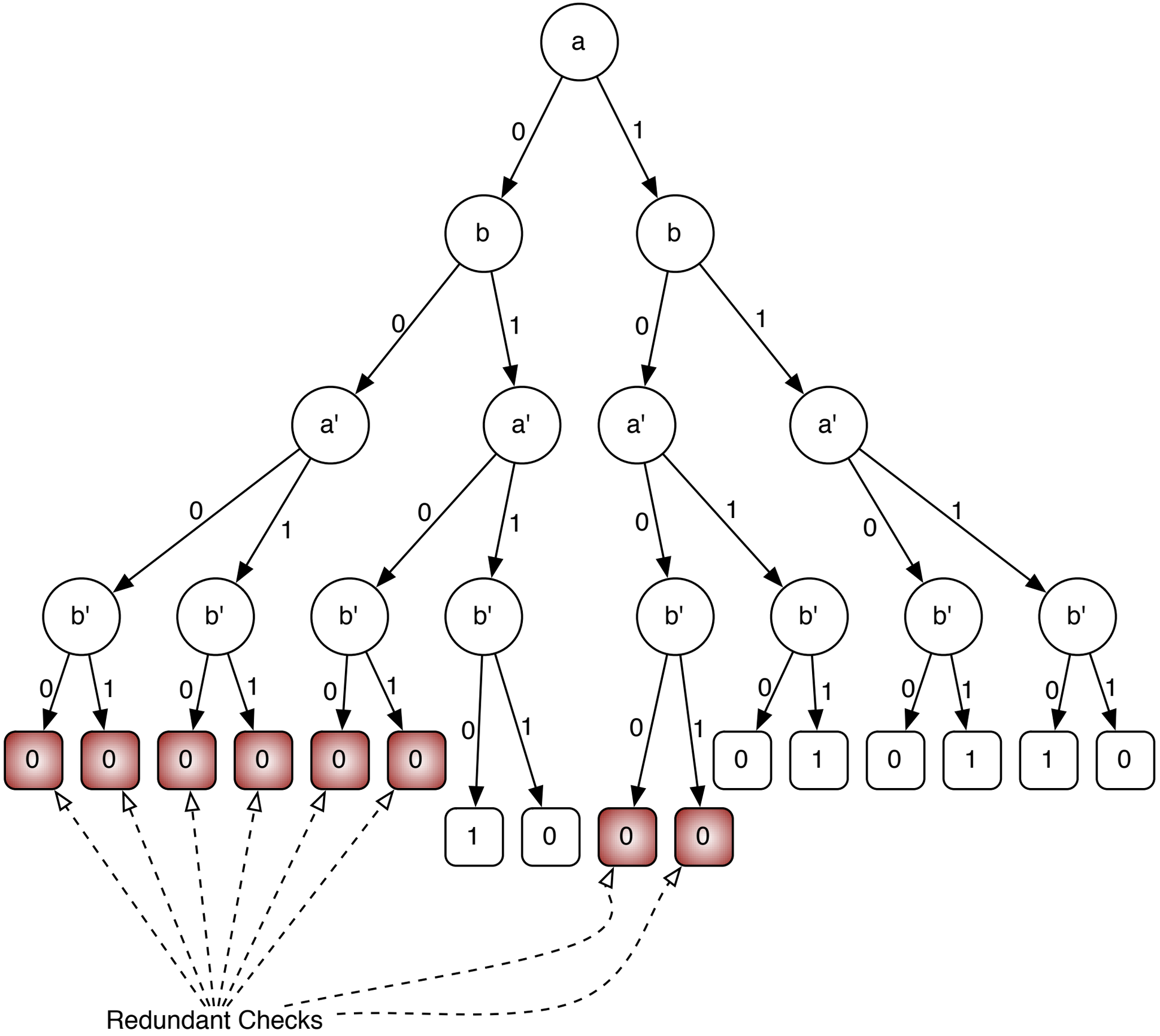
There is a drawback to the binary decision tree in that it stores a great deal of redundant information in the form of equivalent subtrees.

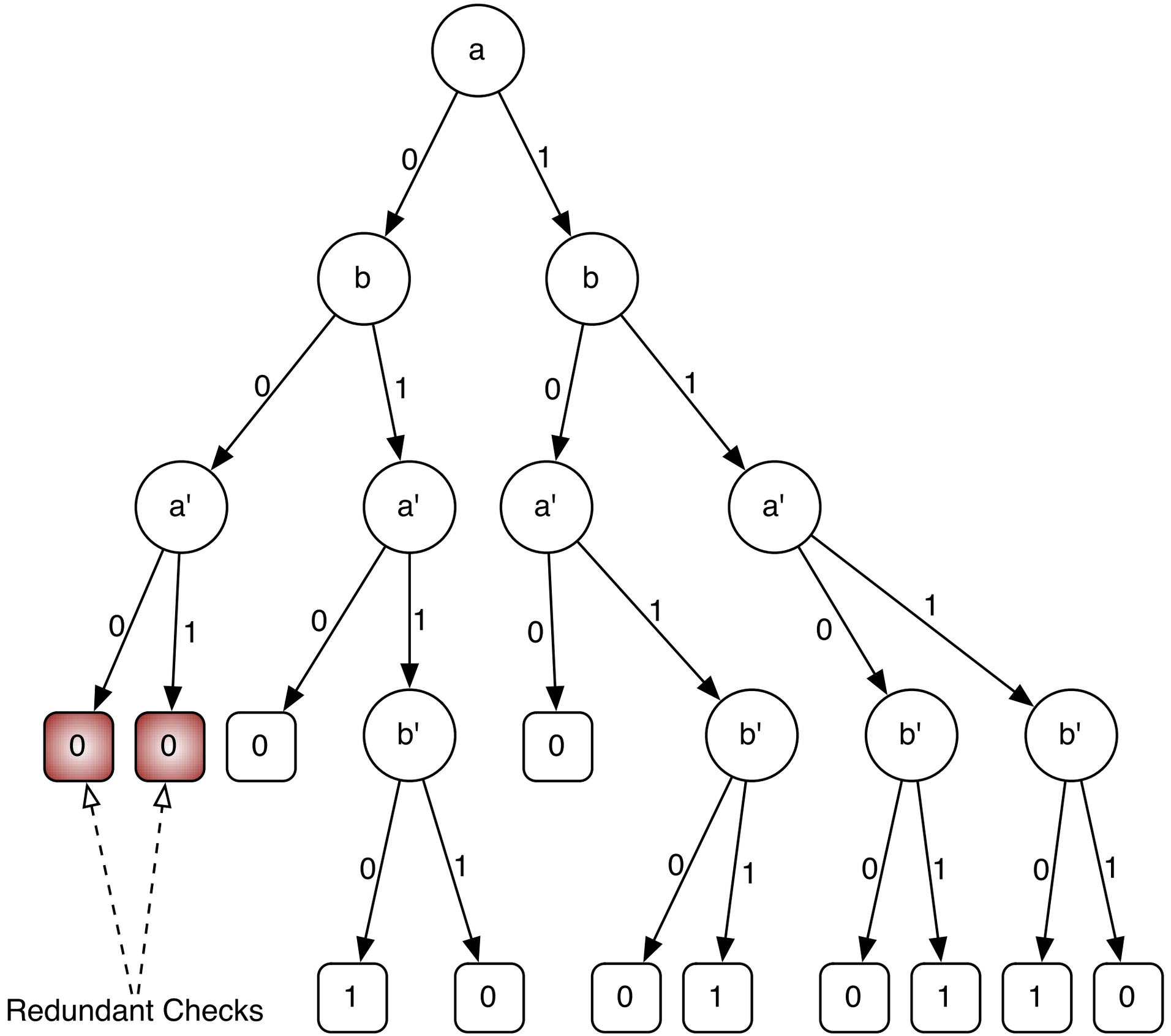
The following algorithm, implemented by a function *Reduce*, takes a binary decision tree as input and removes the redundant subtrees, giving us an *ordered binary decision diagram*:

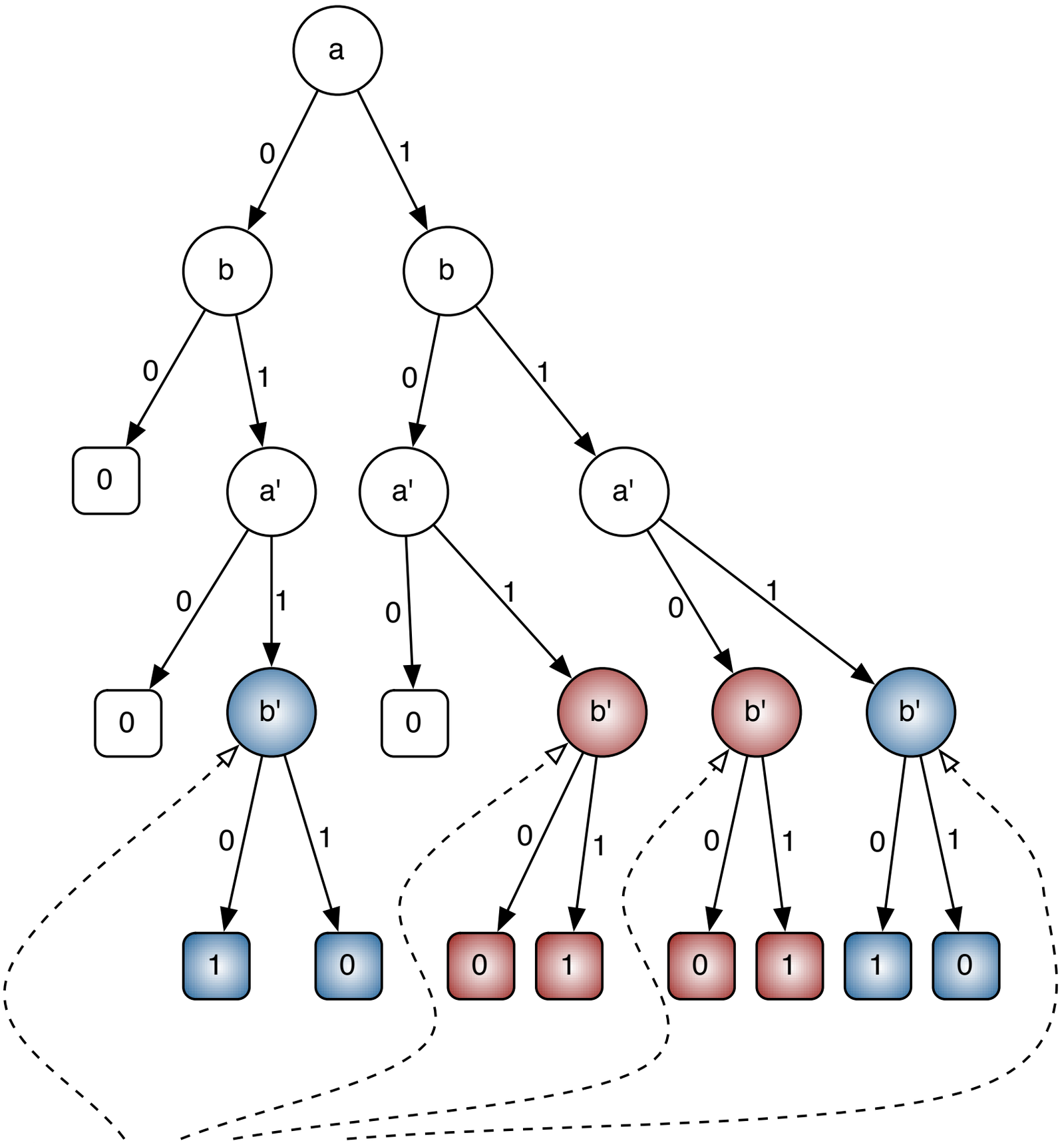
- *Remove duplicate terminals* - we eliminate all but one leaf with a given label, and redirect all arcs to the eliminated vertices to their counterpart.

- *Remove duplicate nonterminals* - if two nonterminals *u* and *v* are roots of identical subtrees, then remove *u* and redirect its incoming arcs to *v*.

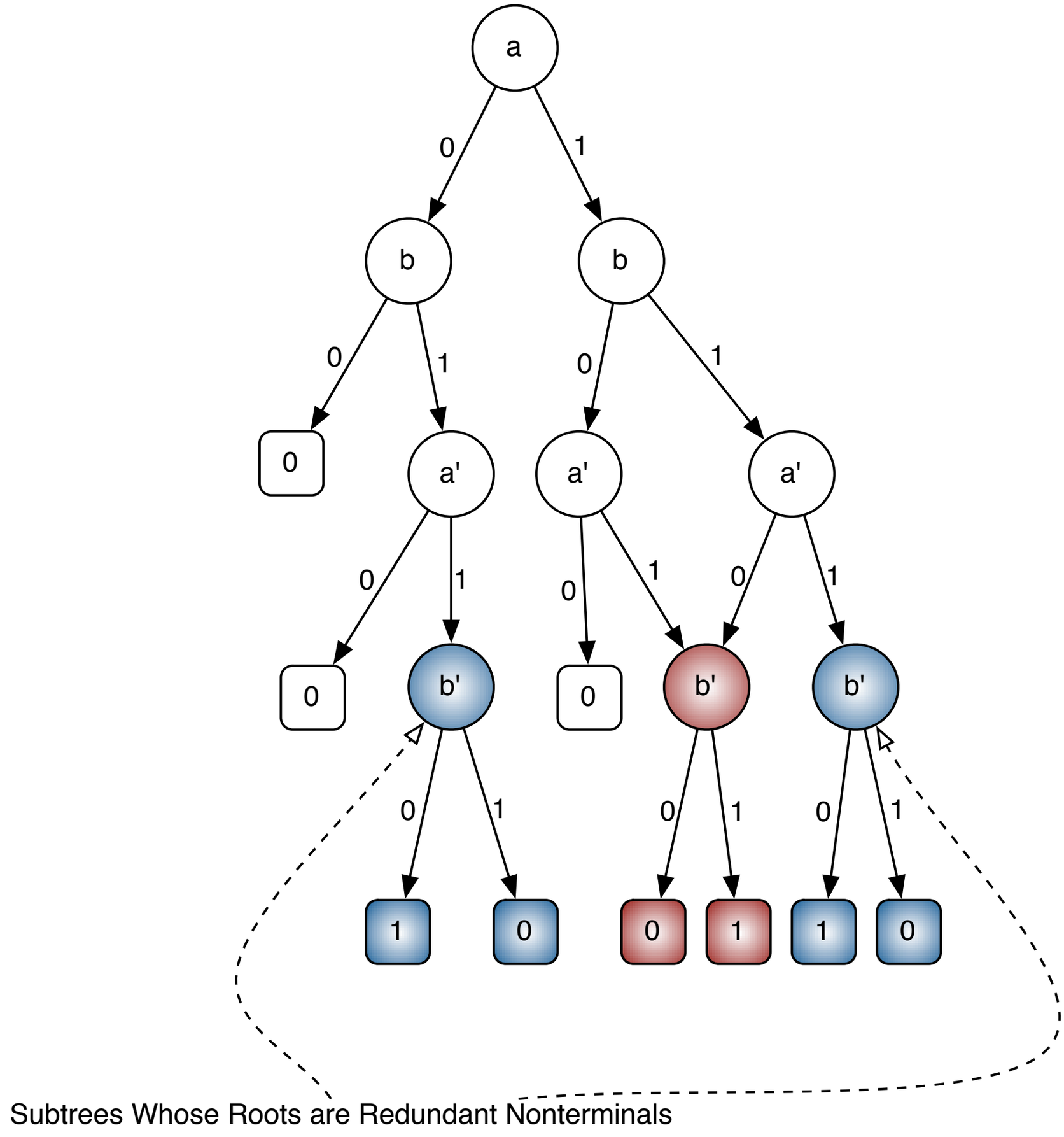
- *Remove redundant checks* - if the children of a nonterminal *v* are roots of identical subtrees, then we remove *v*, and redirect incoming arcs to one of it's children


a
b
b
a'
a'
a'
a'
b'
b'
b'
b'
b'
b'
b'
b'
0
1
Redundant Checks

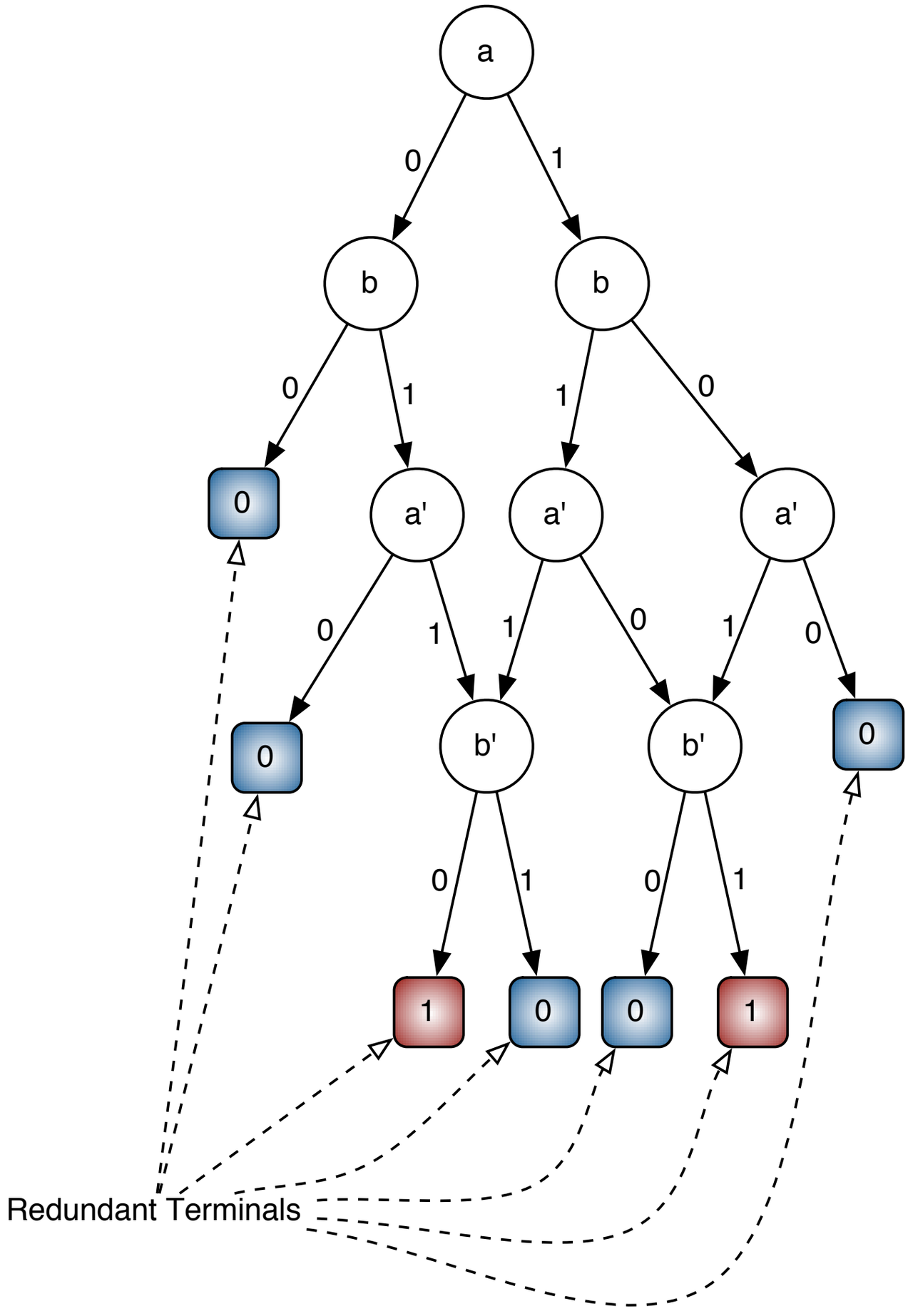

a
0
1
b
b
0
1
0
1
a'
a'
a'
a'
0
1
0
1
0
1
0
1
0
b'
0
b'
b'
b'
0
1
0
1
0
1
0
1
0
1
0
1
0
1
1
0
0
1
0
1
1
0
Redundant Checks

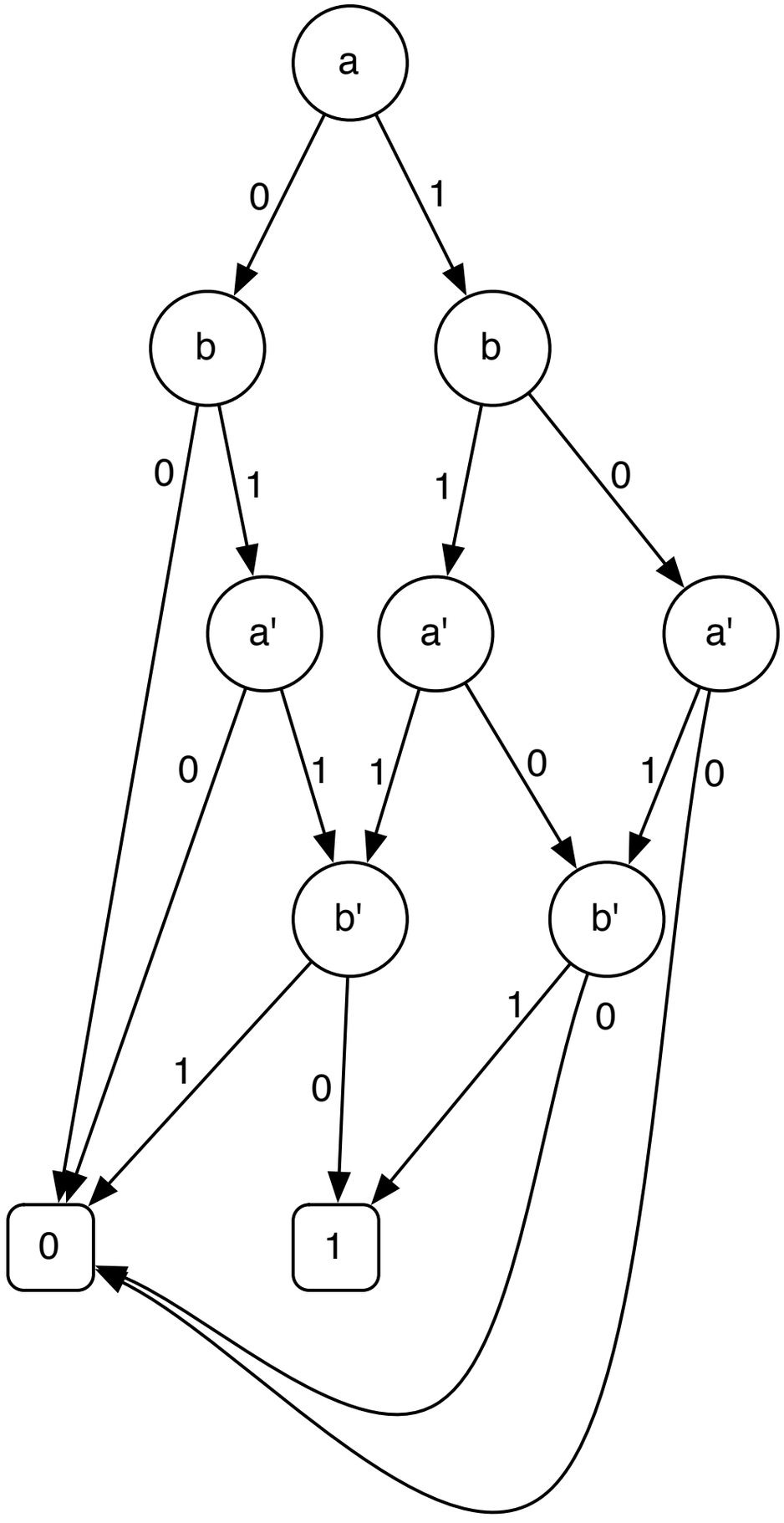


Subtrees Whose Roots are Redundant Nonterminals



Subtrees Whose Roots are Redundant Nonterminals


a
0
1
b
b
0
1
1
0
0
a'
a'
a'
0
1
1
0
1
0
0
b'
b'
0
0
1
0
1
1
0
0
1
Redundant Terminals


a
0
1
b
b
0
1
1
0
a'
a'
a'
0
1
1
0
1
0
b'
b'
1
0
1
0
0
1

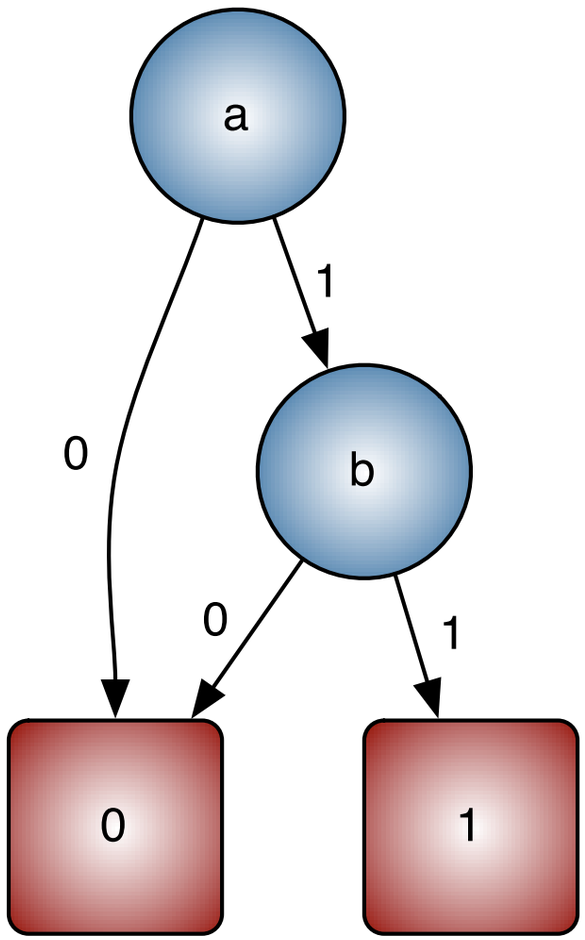
Given an OBDD for a boolean function *F*, we can construct an OBDD for the function that *restricts* the value of an argument *x* of *F* to a boolean value *b* (denoted by $F|_{x=b}$) as follows:

- For any node *v* that has an arc to a node *w* labeled by *x*, redirect the arc to *low(w)* if *b = 0*, and to *high(w)* otherwise.

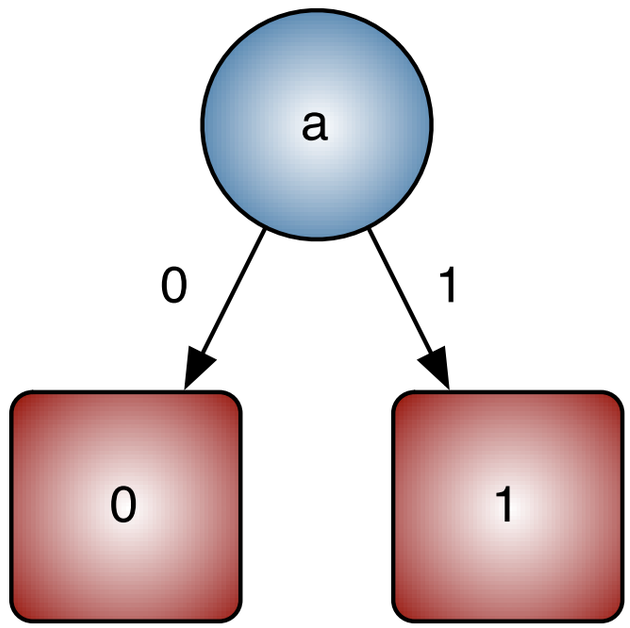
- We then reduce the OBDD as described previously

Example

Let $F = a \wedge b$. The OBDD for F is shown below:



Applying $F|_{b=1}$ yields the following OBDD:



- Let f_1 and f_2 denote boolean functions
- Let v_1 and v_2 denote the roots of the OBDDs representing f_1 and f_2
- Let x_1 and x_2 denote the variables labeling v_1 and v_2
- Let * denote $\wedge$ or $\vee$

Given the OBDD's for f_1 and f_2, we can obtain the OBDD representing $F = f_1 * f_2$ as follows:

- If v_1 and v_2 are terminal nodes then
  $f_1 * f_2 = value(v_1) * value(v_2)$

- If v_1 and v_2 are both labeled by x then we construct a new OBDD whose root is a new node w labeled by x

  - $low(w)$ = the OBDD for $f_1|_{x=0} * f_2|_{x=0}$

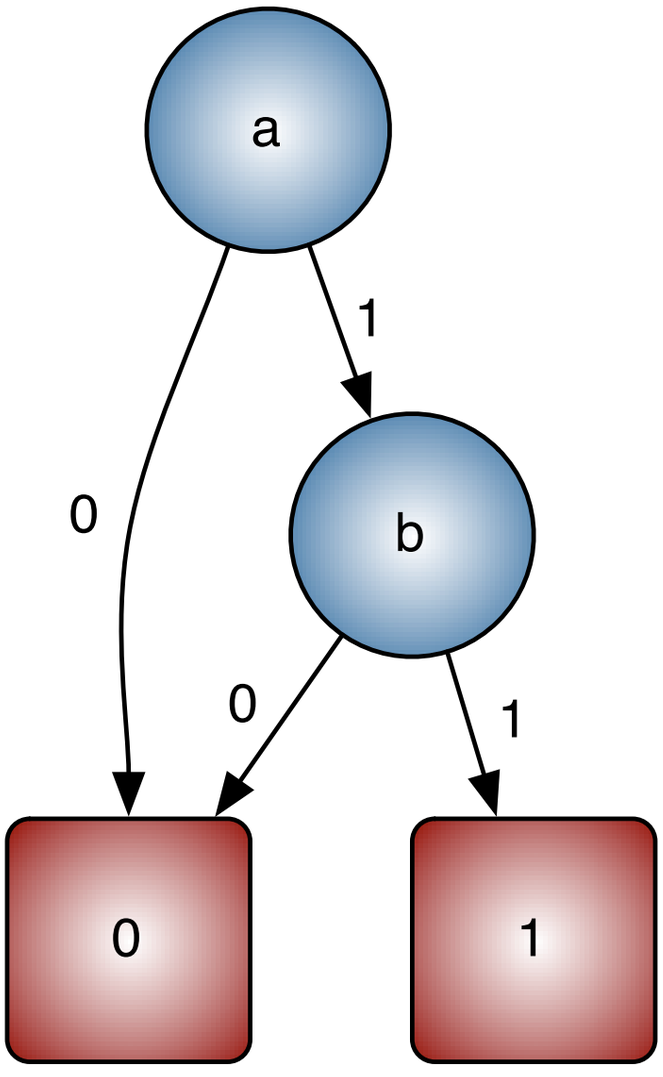
  - $high(w)$ = the OBDD for $f_1|_{x=1} * f_2|_{x=1}$

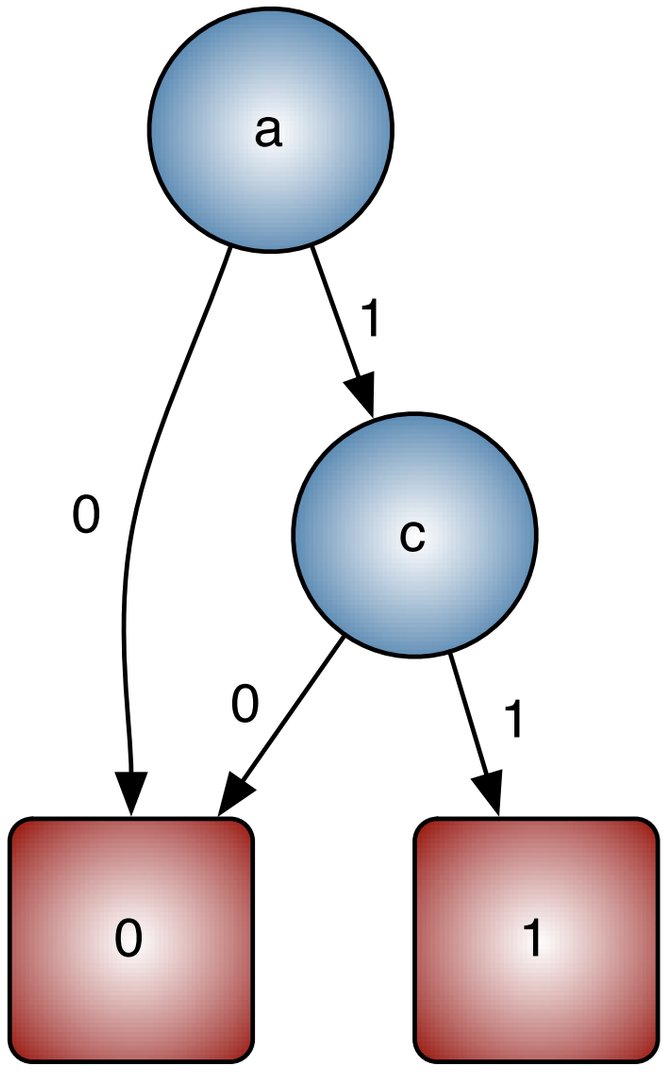
- If $x_1 < x_2$ then we construct a new OBDD whose root is a new node w labeled by x_1
  - *low(w)* = the OBDD for $f_1|_{x_1=0} * f_2$
  - *high(w)* = the OBDD for $f_1|_{x_1=1} * f_2$
- Similarly if $x_1 > x_2$

The preceding algorithm is implemented by the function *Apply*

Example

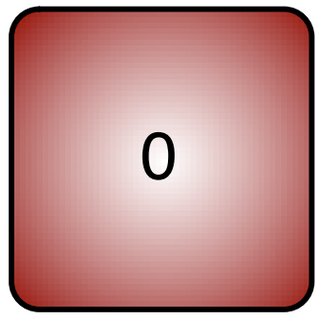
- Consider $f_1 = a \wedge b$ and $f_2 = a \wedge c$
- Let $a < b < c$
- The OBDDs for f_1 and f_2 are as follows:


a
1
0
b
0
1
0
1

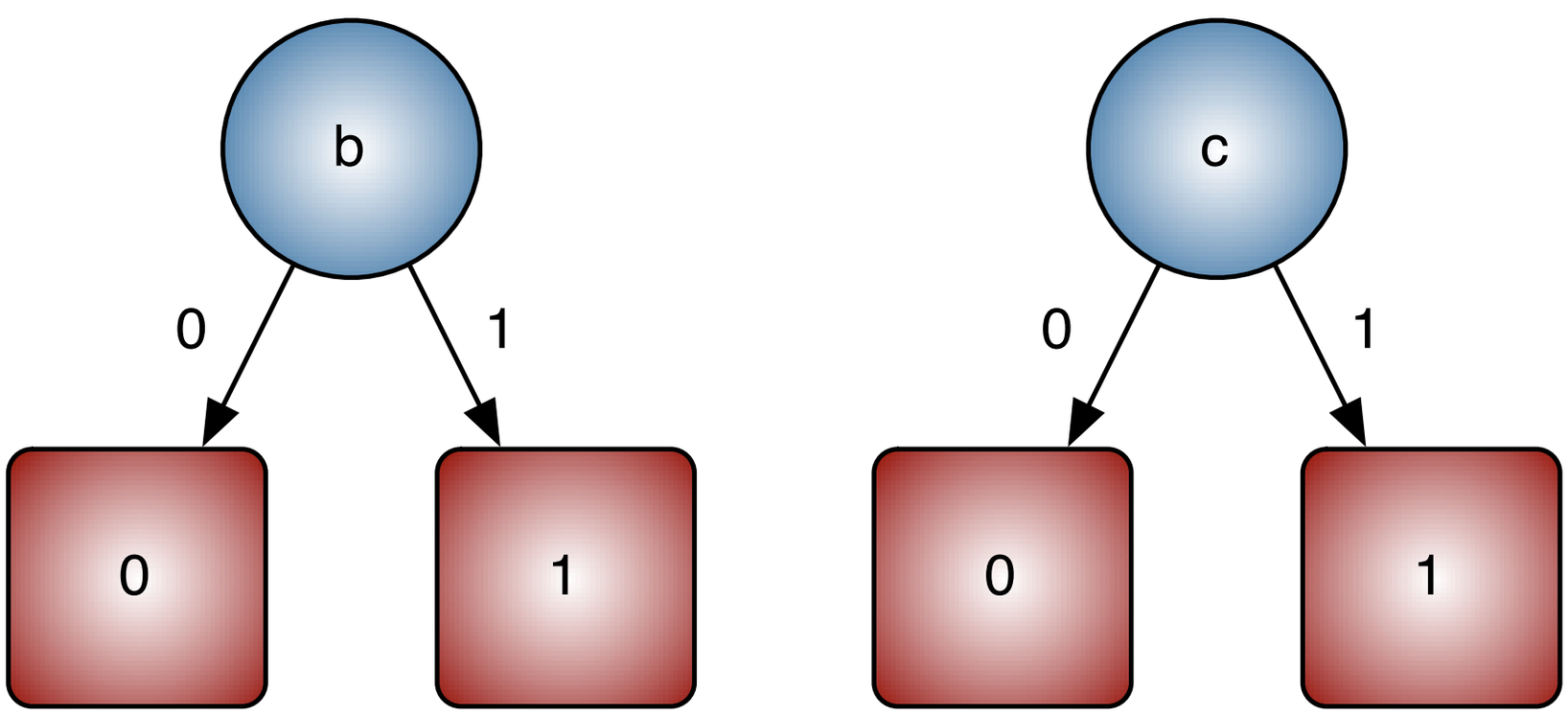

a
1
0
c
0
1
0
1

- The OBDDs have the same root, therefore we apply the second case of the algorithm:

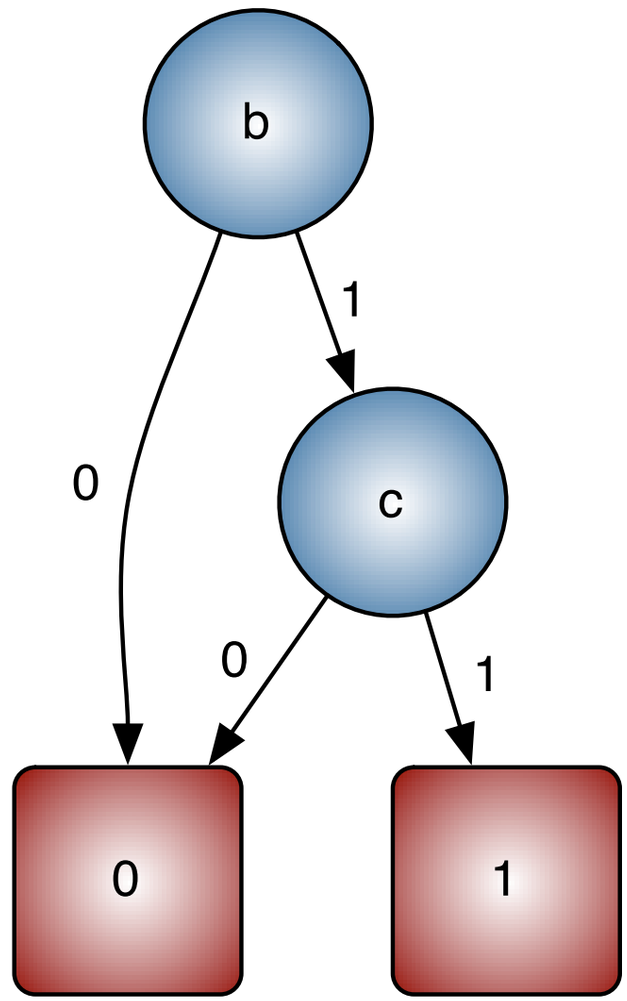
- We obtain the following OBDD for $f_1|_{a=0} \wedge f_2|_{a=0}$:



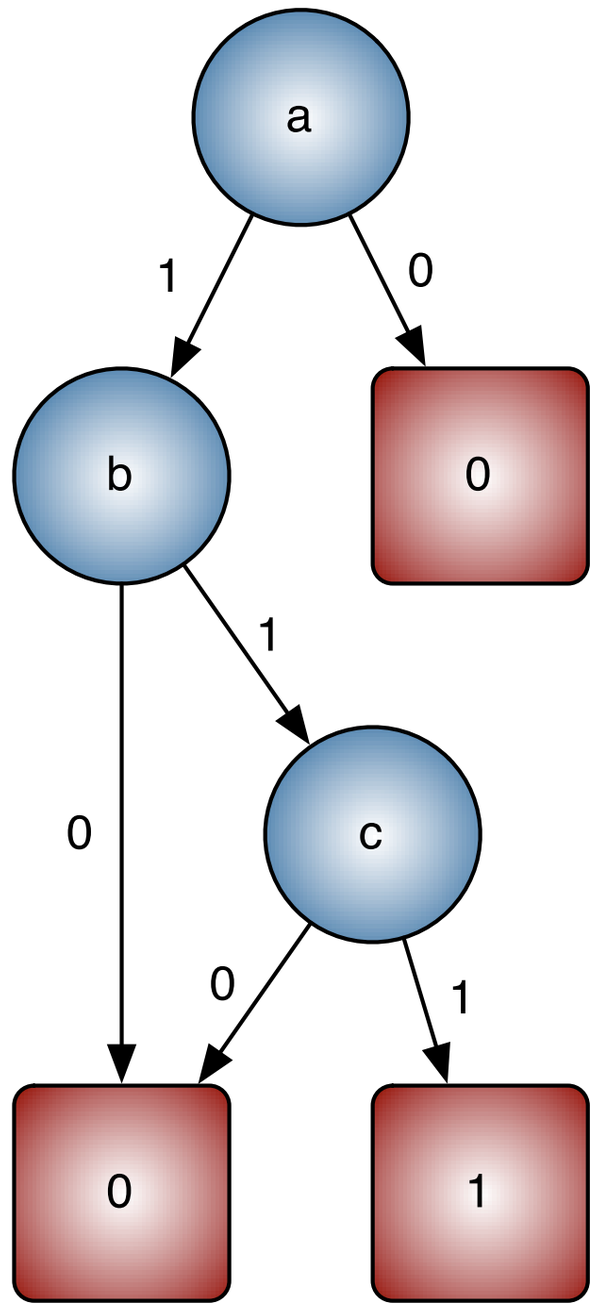
- Similarly we obtain the following OBDDs for $f_1|_{a=1}$ and $f_2|_{a=1}$:



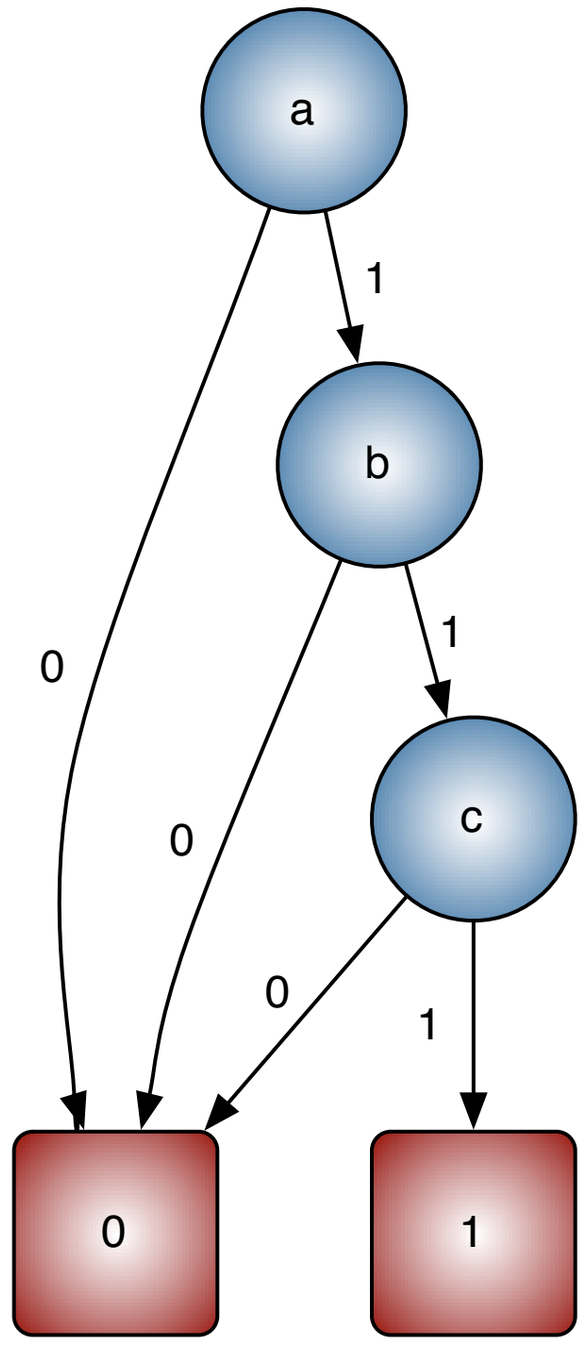
- Combining the OBDDs for $f_1|_{a=1}$ and $f_2|_{a=1}$ yields the following:



- Combing the OBDDs for $f_1|_{a=0} \wedge f_2|_{a=0}$ and $f_1|_{a=1} \wedge f_2|_{a=1}$ gives us the following:



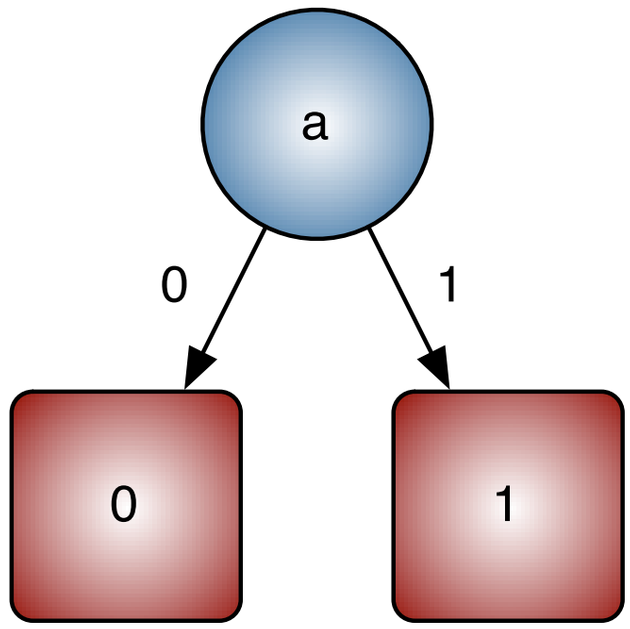
- Reducing the graph gives us the final OBDD:



Given a boolean formula *F*, we can construct the OBDD representing *F* using the following recursive algorithm that is based on the construction of *F*:

- If $F = True$, then the OBDD is a single terminal node labeled by 1.

- If $F = a$, then the OBDD is as follows:



- If $F = \neg g$, we compute the OBDD for g, and invert the terminal nodes.
- Lastly, if $F = f_1 * f_2$ we compute the OBDDs for f_1 and f_2 and then obtain the OBDD for $f_1 * f_2$ as was previously discussed.

Quantified Boolean Formulas

- Given a set $V = \{v_0 \ldots v_{N-1}\}$ of propositional variables, *QBF(V)* is the smallest set of formulas such that:
  - every variable in V is a formula
  - if f and g are formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are formulas
  - if f is a formula, and $v \in V$, then $\exists v f$ and $\forall v f$ are formulas

- A function $\sigma: V \to \{0, 1\}$ is a *truth assignment* for *QBF(V)*.

- If $a \in \{0, 1\}$, then $\sigma[v \leftarrow a](w)$ is defined as follows:

  - a if $v = w$

  - $\sigma(w)$ otherwise

If $f \in QBF(V)$, and σ is a truth assignment, $\sigma \Rightarrow f$ is defined as follows:

- $\sigma \Rightarrow v \equiv \sigma(v) = 1$
- $\sigma \Rightarrow \neg f \equiv \neg(\sigma \Rightarrow f)$
- $\sigma \Rightarrow f \wedge g \equiv \sigma \Rightarrow f$ and $\sigma \Rightarrow g$
- $\sigma \Rightarrow f \vee g \equiv \sigma \Rightarrow f$ or $\sigma \Rightarrow g$
- $\sigma \Rightarrow \exists v f \equiv \sigma[v \leftarrow 0] \Rightarrow f$ or $\sigma[v \leftarrow 1] \Rightarrow f$
- $\sigma \Rightarrow \forall v f \equiv \sigma[v \leftarrow 0] \Rightarrow f$ and $\sigma[v \leftarrow 1] \Rightarrow f$

We have already seen how to represent non-quantified formulas as OBDDs. We can also compute the OBDDs for quantified formulas using the following identities:

- $\exists x f = f|_{x=0} \vee f|_{x=1}$

- $\forall x f = f|_{x=0} \wedge f|_{x=1}$

The CTL Model Checking Algorithm

The CTL model checking algorithm is implemented by a function *Check* that takes a Kripke structure M and a CTL formula f as parameters, and returns an OBDD representing the set $S = \{s : M, s \Rightarrow f\}$.

M satisfies F if the set of initial states belongs to S.

The function *Check* operates as follows:

- If F is an atom, *Check(F)* = the OBDD representing the set of states containing F.
- If $F = \neg f$, $Check(F) = Apply(\neg, Check(f))$
- If $F = f * g$, $Check(F) = Apply(*, Check(f), Check(g))$

- *Check(**EX** f) = CheckEX(Check(f))*

- *Check(**E**[f **U** g]) = CheckEU(Check(f), Check(g))*

- *Check(**EG** f) = CheckEG(Check(f))*

The function *CheckEX* takes as a parameter an OBDD representing the set of states satisfying a formula *f*, and returns the OBDD for the quantified boolean formula $\exists v'[f(v') \wedge R(v,v')]$

The function *CheckEU* takes as parameters the OBDDs representing the sets of states satisfying the formulas *f* and *g*, and returns the OBDD corresponding to

$$\mu Z.\ g \vee (f \wedge \mathbf{EX}\ Z)$$

where $\mu Z.\ g \vee (f \wedge \mathbf{EX}\ Z)$ is the least fixpoint characterization of ***E[f U g]***

The function *CheckEG* takes as parameters the OBDDs representing the set of states satisfying the formula *f*, and returns the OBDD corresponding to

$$\upsilon Z.\, f \wedge \mathbf{EX}\, Z$$

where $\upsilon Z.\, f \wedge \mathbf{EX}\, Z$ is the greatest fixpoint characterization of ***EG***