Introduction to Software Verification

Orna Grumberg

Lectures Material
winter 2017-18

Lecture 9

BDD-based Symbolic Model Checking

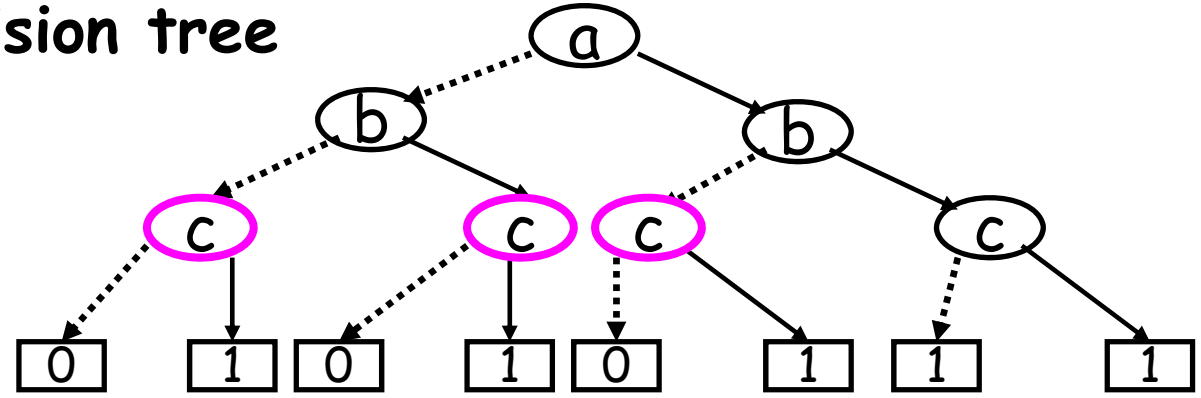
A solution to the state explosion problem:
BDD-based model checking

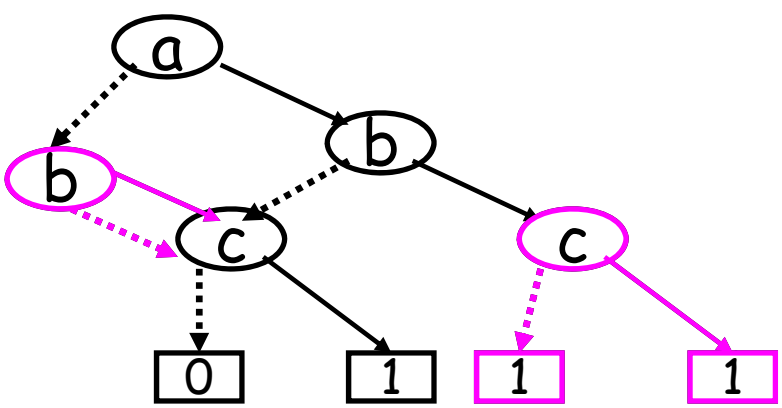
- **Binary Decision Diagrams (BDDs)** are used to represent the **model** and **sets of states**.

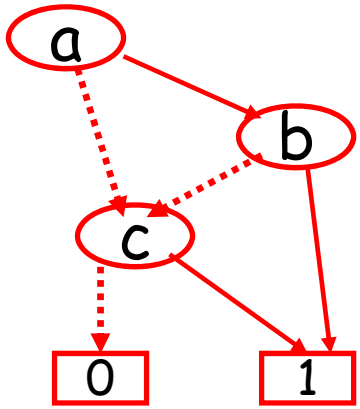
- It can handle systems with **hundreds** of Boolean variables.

BDD for f(a,b,c) = (a ∧ b) ∨ c

Decision tree







BDD

Advantage of BDDs (revisited)

- Often (but not always) **concise** in size
- **Canonical** representation for a given variable ordering
  - Easy to check equivalence between two functions
- A function depends exactly on all variables that appear in its BDD
- Most **Boolean operations** can be performed on BDDs in **polynomial time** in the BDD size

Operations on BDDs

Operations on BDDs - Reduce

Reduce

Given an unreduced BDD:

- Eliminate isomorphic sub-graphs:
  - Eliminate duplicated end nodes
  - Eliminate duplicated internal nodes
- Eliminate redundant nodes

Reduce works bottom-up in linear time in the BDD size

Important remark:

BDD for a complex function is built bottom-up starting from small sub-functions to larger ones

We do not build a full decision tree and then reduce

Operations on BDDs - Restrict

Restrict

Given a BDD for $f(x_1,\ldots x_n)$, build a BDD for

$f|_{xi=b}(x_1,\ldots x_n) = f(x_1,\ldots,x_{i-1},b,x_{i+1},\ldots x_n)$ $\quad b\in\{0,1\}$

Example:

$f(x_1,x_2,x_3,x_4) = (x_1\wedge x_2)\vee(x_3\wedge x_4)$

$f|_{x2=0}(x_1,x_2,x_3,x_4) = (x_1\wedge 0)\vee(x_3\wedge x_4) = (x_3\wedge x_4)$

Operations on BDDs - Apply

- Gets two BDDs, representing functions **f** and **f'** and an operation ***
  - **Over the same variable ordering**
- Returns the BDD representing **f*f'**
- * can be any of 16 binary operations on two Boolean functions

Operations on BDDs - Apply

- Shannon expansion
  for every Boolean function f and a variable x:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1})$$

Notation:

- v,v' are the **roots** of f,f', respectively
- If v,v' are **not end nodes** then var(v)=x, var(v')=x'

Operations on BDDs - Apply

Computing f*f′ :

- Case 1: v and v′ are end nodes

  f*f′ = value(v) * value(v′)

- The BDD for f*f′

  consists of one leaf v″ with
  value(v″) = value(v) * value(v′)

This is the only case where * is taken into account

Operations on BDDs - Apply

Computing f*f′ :

- Case 2: x = x′
- Use Shannon expansion:

$$f*f' = (\neg x \wedge (f|_{x=0} * f'|_{x=0})) \vee (x \wedge (f|_{x=1} * f'|_{x=1}))$$

- Two simpler sub-problems to solve
  - Each depends on one less variable

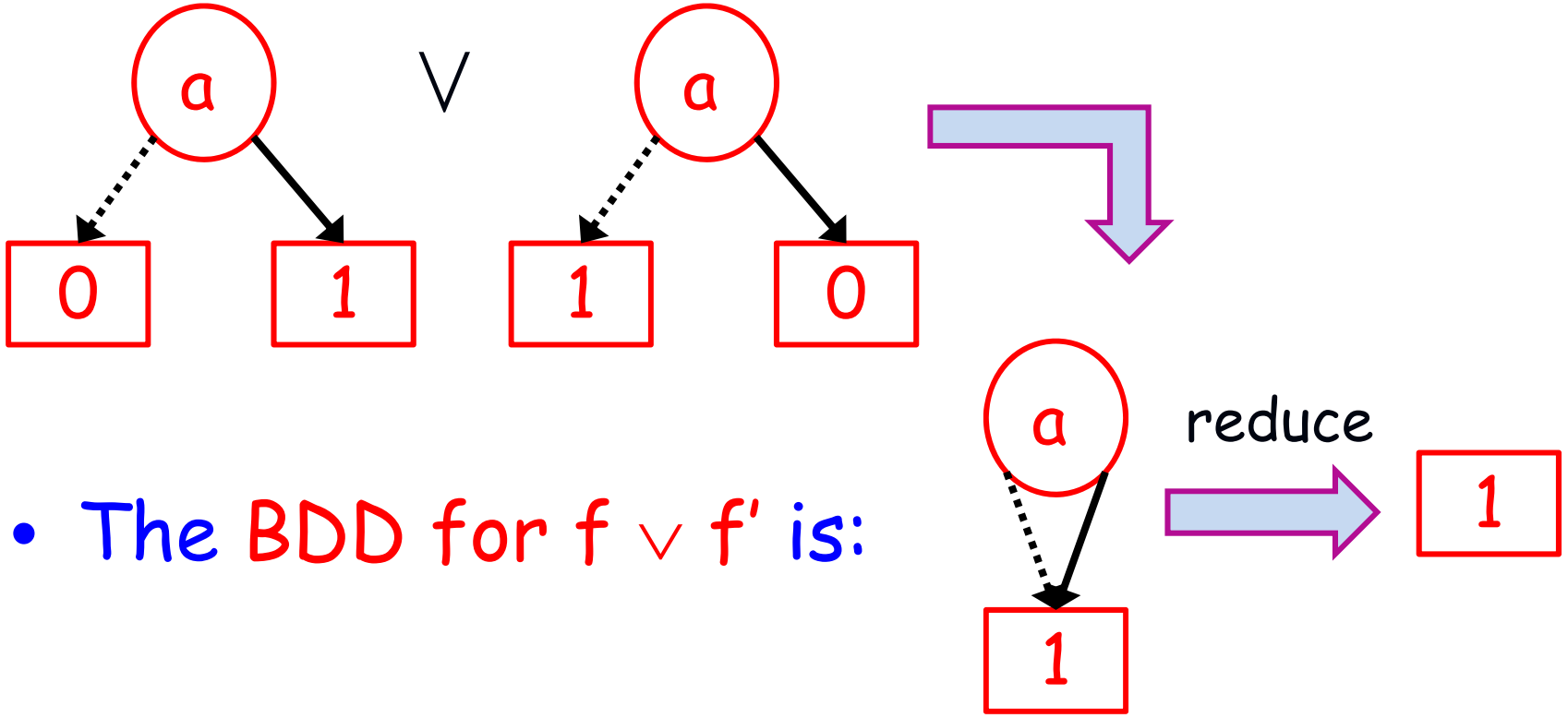
Operations on BDDs - Apply

Computing f*f′ :

- Case 2: x = x′
- The BDD for f*f′
- Root: a new node v″
  - var(v″) = x
  - low(v″) points to the root of the BDD for $(f|_{x=0} * f'|_{x=0})$
  - high(v″) points to the root of the BDD for $(f|_{x=1} * f'|_{x=1})$

Example

- f(a) = a, f'(a) = ¬a, * is ∨



- The BDD for f ∨ f' is:

Operations on BDDs - Apply

Computing f*f′ :

- **Case 3:** $x < x'$
- **x does not appear in f′**

  $f'|_{x=0} = f'|_{x=1} = f'$

- Use Shannon expansion as before:

  $f*f' = (\neg x \wedge (f|_{x=0} * f')) \vee$

  $(x \wedge (f|_{x=1} * f'))$

Operations on BDDs - Apply

Computing f*f' :

- **Case 4**: $x > x'$

  Similar to case 3

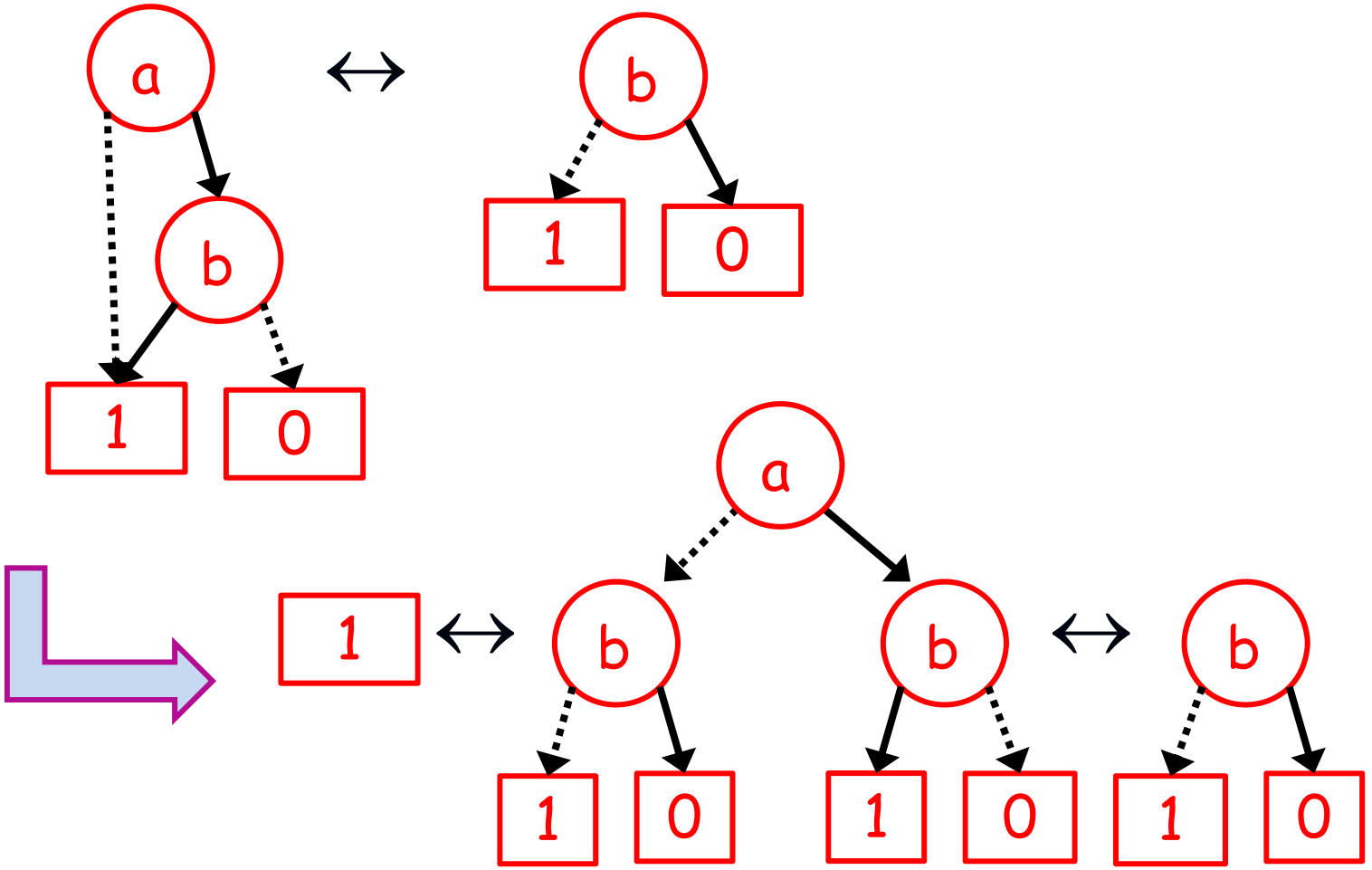
Example

- $f(a,b) = a \rightarrow b$, $f'(a,b) = \neg b$, * is $\leftrightarrow$
- $f \leftrightarrow f' \equiv (a \rightarrow b) \leftrightarrow (\neg b) \equiv (\neg a \vee b) \leftrightarrow (\neg b)$

  $\equiv (\neg a \wedge \neg b)$

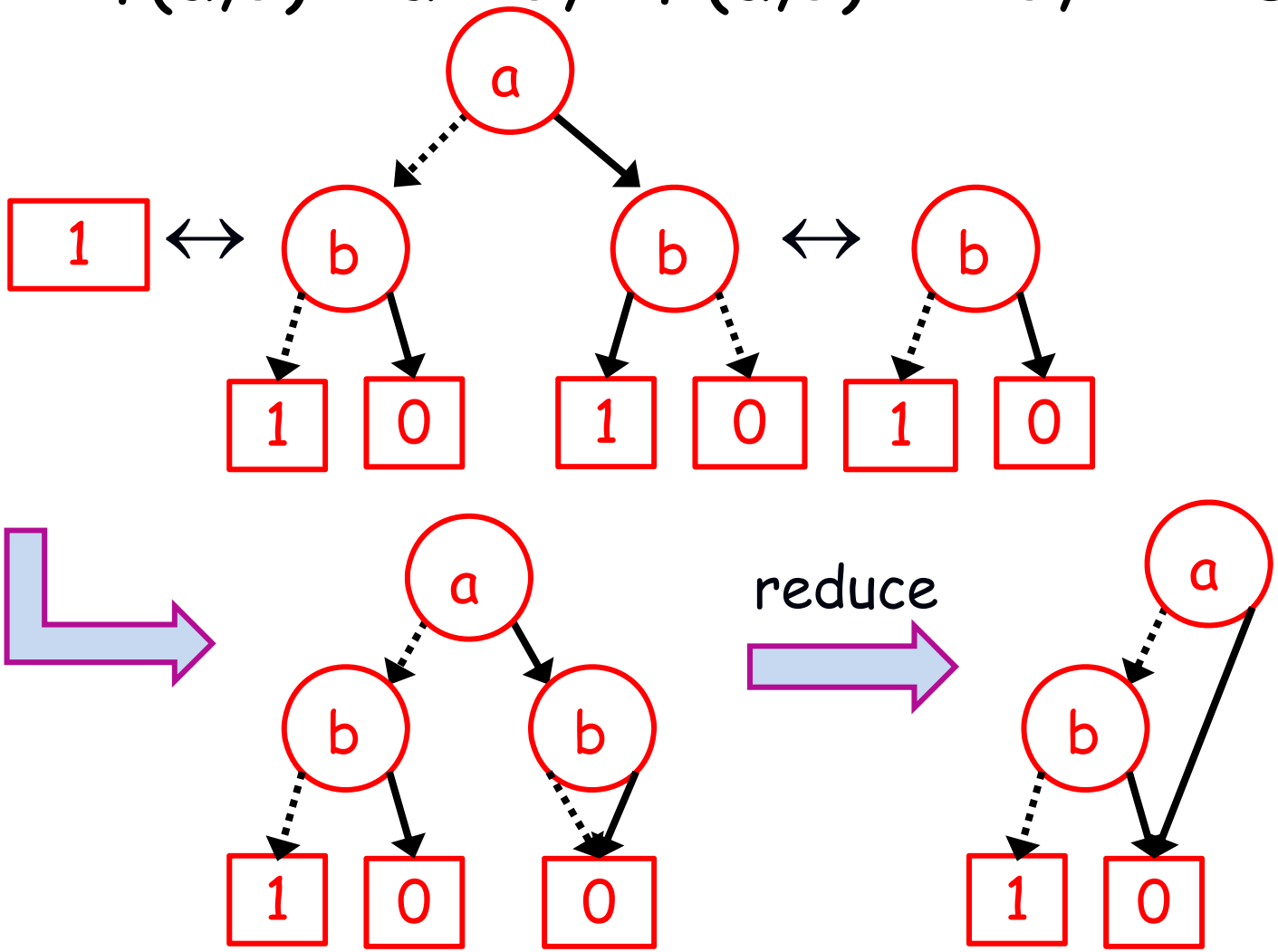
Example

- $f(a,b) = a \rightarrow b$, $f'(a,b) = \neg b$, * is $\leftrightarrow$, $a<b$



Example

- $f(a,b) = a \rightarrow b$, $f'(a,b) = \neg b$, * is $\leftrightarrow$



Complexity of apply
Naive implementation

- two sub-problems for each variable
- exponential in the number of variables

Complexity of apply
Non-naive implementation

Notice:

- Every BDD node u represents a function f_u
- |f|, |f'| denote the number of nodes in the BDD for f, f' respectively

Solution:

- Use hash table with entries:
  - Pointers to (the root node of) the BDDs for g, g', and *
  - Pointer to the resulting BDD for g*g'

Consequences:

- Never redo an operation on the same BDDs
  - **Never solve the same sub-problem twice**
- Never insert into the BDD manager the same BDD twice

Complexity

- The number of different sub-problems is $O(|f|x|f'|)$
  - **Polynomial in the BDD sizes**

Symbolic (BDD-based) Model Checking for CTL

Symbolic (BDD-based) model checking

- Explicit-state model checking applies graph algorithms (for example: BFS, DFS, SCC)
- BDDs are not suitable for that
  - Highly inefficient

- BDD-based model checking manipulates set of states
  - BDD efficiently represents Boolean function which represents a set of states

Operations on sets

- Union of sets $\Rightarrow \vee$ (or) over their BDDs
- Intersection $\Rightarrow \wedge$ (and)
- Complementation $\Rightarrow \neg$ (not)
- Equality of sets $\Rightarrow \leftrightarrow$ (iff)

Two additional operations

- $\exists x_i\ f(x_1,\ldots x_n) = f|_{xi=0} \vee f|_{xi=1}$
- $\forall x_i\ f(x_1,\ldots x_n) = f|_{xi=0} \wedge f|_{xi=1}$
- No additional expressive power
- Can be implemented with apply + restrict
  - Exponential in the number of quantified variables
- Heuristics can be more efficient, but not in the worst case

BDD-based Model Checking

- Accept: Kripke structre M, CTL formula f
- Returns: S_f - the set of states satisfying f

M is given by:

- BDD R(V,V'), representing the transition relation
- BDD p(V), for every $p \in AP$, representing S_p
  - the set of states satisfying p
- $V = (v_1,\ldots v_n)$

BDD-based Model Checking

- The algorithm works from simpler formulas to more complex ones
- When a formula g is handled, the BDD for S_g is built
- A formula is handled only after all its sub-formulas have been handled

BDD-based Model Checking

- For $p \in AP$, return $p(V)$
- For $f = f_1 \wedge f_2$, return $f(V) = f_1(V) \wedge f_2((V)$ (using apply)
- For $f = \neg f_1$, return $f(v) = \neg f_1(V)$

BDD-based Model Checking

- For $f = EX\ f_1$ return

  $f(V) = \exists V' [\ f_1(V') \wedge R(V,V')\ ]$

- This BDD represents all (encoding V of) states that have a successor (with encoding V') in f_1

- Defined as a new BDD operator:
  $EX\ f_1(V) = \exists V' [\ f_1(V') \wedge R(V,V')\ ]$

- This operation is also called pre-image

- Important:
  the formula defines a sequence of BDD operations and therefore is considered as a symbolic algorithm

Model Checking f = EF g

Given: BDDs R(V,V') and g(V):

```
procedure CheckEF (g(V))
    Q(V) := emptyset;  Q'(V) := g(V) ;
    while Q(V) ≠ Q'(V) do
      Q(V) := Q'(V);
      Q'(V) := Q(V) ∨ EX ( Q(V) )
    end while
    f(V) := Q(V) ;   return(f(V))
```

The algorithm applies

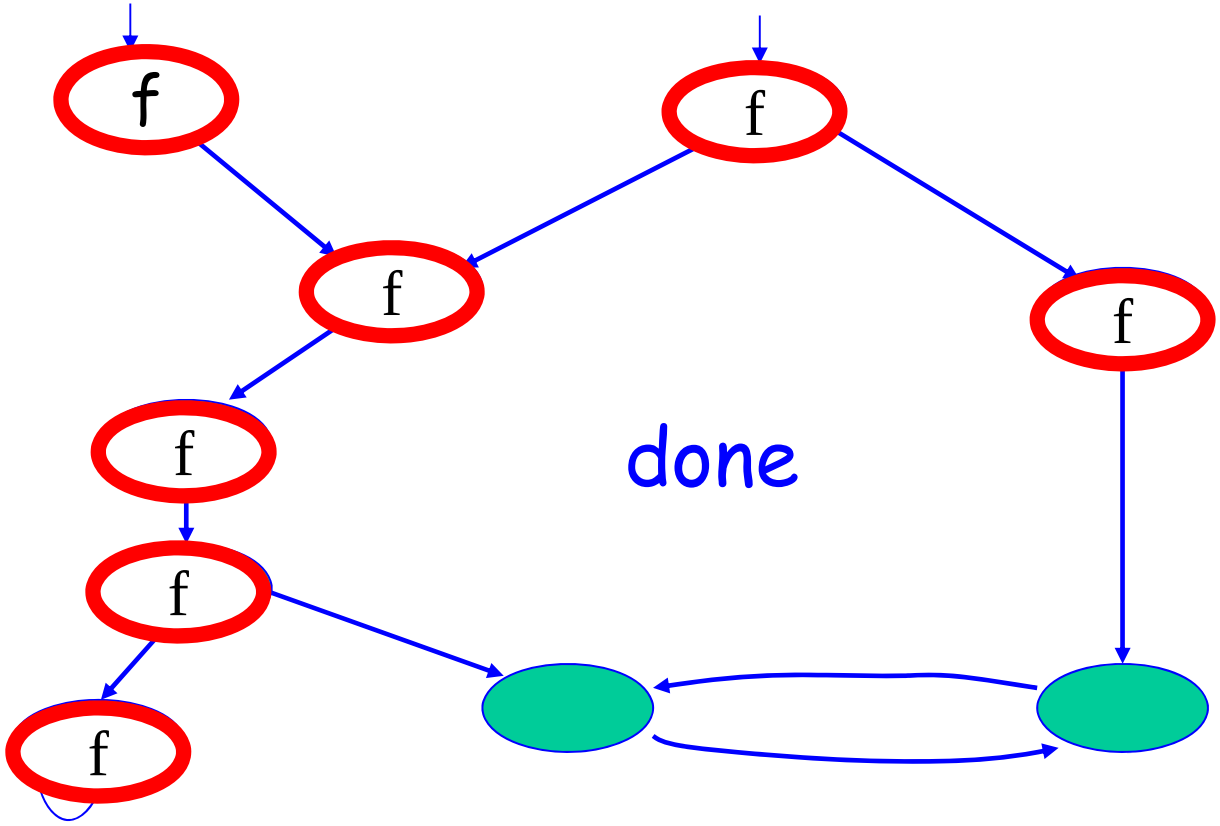
- BDD operations (or $\vee$), and EX
- comparison $Q(V) \neq Q'(V)$ (easy)

Therefore, this is a symbolic algorithm!

The algorithm is based on the equivalence:

$$EF\ g \equiv g \vee EX\ EF\ g$$

Example: f = EF g



Model Checking $f = E[g_1 \; U \; g_2]$

Given: BDDs $R(V,V')$, $g_1(V)$ and $g_2(V)$:

```
procedure CheckEU (g1, g2)
    Q := emptyset;  Q' := g2 ;
    while Q ≠ Q'  do
      Q := Q';
      Q' := Q ∨ (EX(Q) ∧ g1 )
    end while
    f := Q ;   return(f)
```

Model Checking f = EG g

Given: BDDs R(V,V') , g(V)

```
procedure CheckEG (g)
    Q := S ;  Q' := g ;
    while Q ≠ Q' do
      Q := Q';
      Q' := Q ∧ EX (Q)
    end while
    f := Q ;   return( f )
```

Example: f = EG g