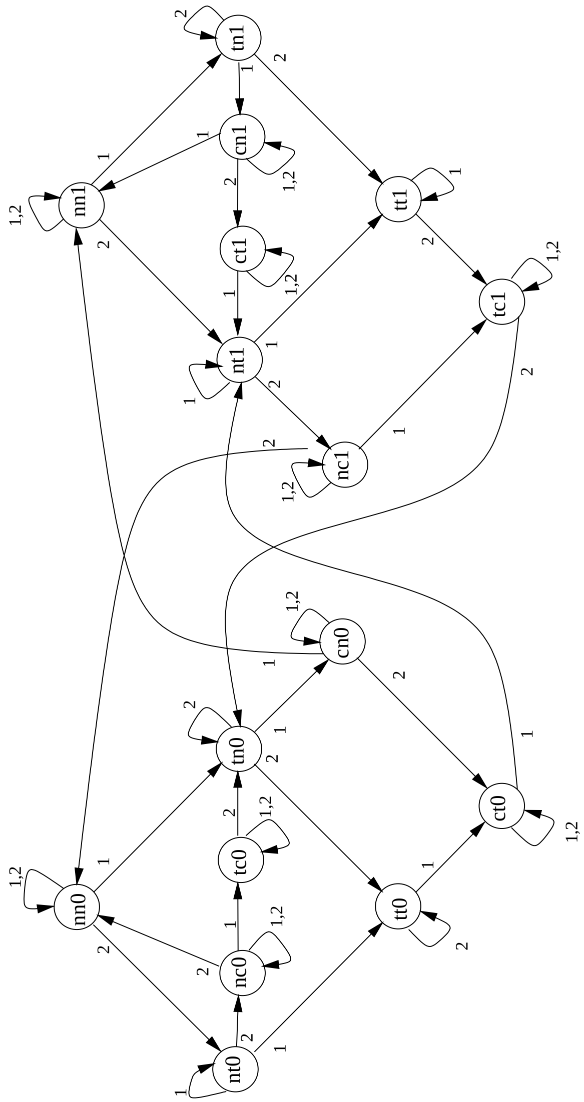
Mutual Exclusion Again

`st` – status of the process (critical section, or not, or trying)
`other-st` – status of the other process
`turn` – ensures that they take turns

```
MODULE main
  VAR
    pr1 : process prc(pr2.st, turn, 0);
    pr2 : process prc(pr1.st, turn, 1);
    turn : boolean;
  ASSIGN
    init(turn) := 0;
  --safety
  SPEC AG!((pr1.st = c) & (pr2.st = c))
  --liveness
  SPEC AG((pr1.st = t) -> AF (pr1.st = c))
  SPEC AG((pr2.st = t) -> AF (pr2.st = c))
  --no strict sequencing
  SPEC EF(pr1.st = c & E[pr1.st = c U
            (!pr1.st = c &
              E[! pr2.st = c U pr1.st = c ])])
```


Model (Cont'd)

```
MODULE prc(other-st, turn, myturn)
  VAR
    st : {n, t, c};
  ASSIGN
    init(st) := n;
    next(st) :=
      case
        (st = n) : {t, n};
        (st = t) & (other-st = n) : c;
        (st = t) & (other-st = t)
                 & (turn = myturn) : c;
        (st = c) : {c, n};
        1 : st;
      esac;
    next(turn) :=
      case
        turn = myturn & st = c  : !turn;
        1        : turn;
      esac;
  FAIRNESS running
  FAIRNESS !(st = c)
```



Comments:

• The labels in the slide above denote the process which can make the move.

• Variable `turn` was used to differentiate between states s_3 and s_9, so we now distinguish between `ct0` and `ct1`. But transitions out of them are the same.

• Removed the assumption that the system moves on each tick of the clock. So, the process can get stuck, and thus the fairness constraint.

• In general, what is the difference between the single fairness constraint $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$ and n fairness constraints ψ_1, ψ_2, etc., written on separate lines under `FAIRNESS`?

Notion of Fairness

Fairness: a path p is *fair* w.r.t. property ψ if ψ is true on p infinitely often.

We may want to evaluate A and E constraints only over those paths.

Example: each process will run infinitely often; a process can stay in a critical section arbitrarily long, as long as it eventually leaves.

Two types of fairness: simple
 Property ϕ is true infinitely often.
and compound
 If ϕ is true infinitely often, then ψ is also true infinitely often.

SMV can deal only with simple fairness.

Formal Definition of Fairness

Let $C = \{\psi_1, \psi_2, ..., \psi_n\}$ be a set of n fairness constraints. A computation path $s_0, s_1, \cdots$ is *fair* w.r.t. C if for each i there are infinitely many j s.t. $s_j \models \psi_i$, that is, each ψ_i is true infinitely often along the path.

We use A_C and E_C for the operators A and E restricted to fair paths.

$\mathsf{E}_C\mathsf{U}$, $\mathsf{E}_C\mathsf{G}$ and $\mathsf{E}_C\mathsf{X}$ form an adequate set.

Also, a path is fair iff any suffix of it is fair. Finally,

$$\mathsf{E}_C[\phi \mathsf{U} \psi] = \mathsf{E}[\phi \mathsf{U}(\psi \wedge \mathsf{E}_C\mathsf{G}\top)]$$

$$\mathsf{E}_C\mathsf{X}\phi = \mathsf{EX}(\phi \wedge \mathsf{E}_C\mathsf{G}\top)$$

We only need an algorithm for $\mathsf{E}_C\mathsf{G}\phi$.

Algorithm for $\mathsf{E}_C\mathsf{G}\phi$

• Restrict the graph to states satisfying ψ; of the resulting graph, we want to know from which states there is a fair path.

• Find the maximal *strongly connected components* (SCCs) of the restricted graph;

• Remove an SCC if, for some ψ_i, it does not contain a state satisfying ψ_i. The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach one has a fair path from it.

• Use backwards breadth-first searching to find the states on the restricted graph that can reach a fair SCC.

Complexity: $O(n \times |f| \times (S + |R|))$ (still linear in the size of the model and formula).

Guidelines for Modeling with SMV

• Identify inputs from the environment.

• Make sure that the environment is non-deterministic. All constraints on the environment should be carefully justified.

• Determine the states of the system and its outputs. Model them in terms of the environmental inputs.

• Specify fairness criteria, if any. Justify each criterium. Remember that you can over-specify the system. Fairness may not be implementable, and in fact may result in no behaviors.

• Specify correctness in CTL. Comment each CTL property in English.

• Ensure that CTL properties are not satisfied vacuously. That is, each universally-quantified property should be paired up with an existentially-specified property. Also check that LHSs of implications are not always false.

Examples:
AG (a) —
AG $(a \rightarrow b)$ —

Binary Decision Diagrams

- Representation of Boolean Functions
- BDDs, OBDDs, ROBDDs
- Operations
- Model-Checking over BDDs

Readings: 6.1-6.3 of Huth, Ryan

Boolean Functions

Boolean functions: $\mathcal{B} = \{0,1\}$,
$f : \mathcal{B} \times \cdots \times \mathcal{B} \to \mathcal{B}$

Boolean expressions:
$t ::= x \mid 0 \mid 1 \mid \neg t \mid t \wedge t \mid t \vee t \mid t \to t | t \leftarrow t$

Truth assignments: ρ,
$[v_1 x_1, v_2/x_2, \cdots, v_n/x_n]$

Satisfiable: Exists ρ s.t. $t[\rho] = 1$

Tautology: Forall ρ, $t[\rho] = 1$

Truth Tables

xyz	$x \to y, z$
000	0
001	1
010	0
011	1
100	0
101	0
110	1
111	1

$x_1 \cdots x_n$	$f(x_1, \cdots x_n)$
$0 \cdots 0$	1
$0 \cdots 1$	0
$\vdots$	$\vdots$
$1 \cdots 1$	0

2^n entries

What is a good representation of boolean functions?

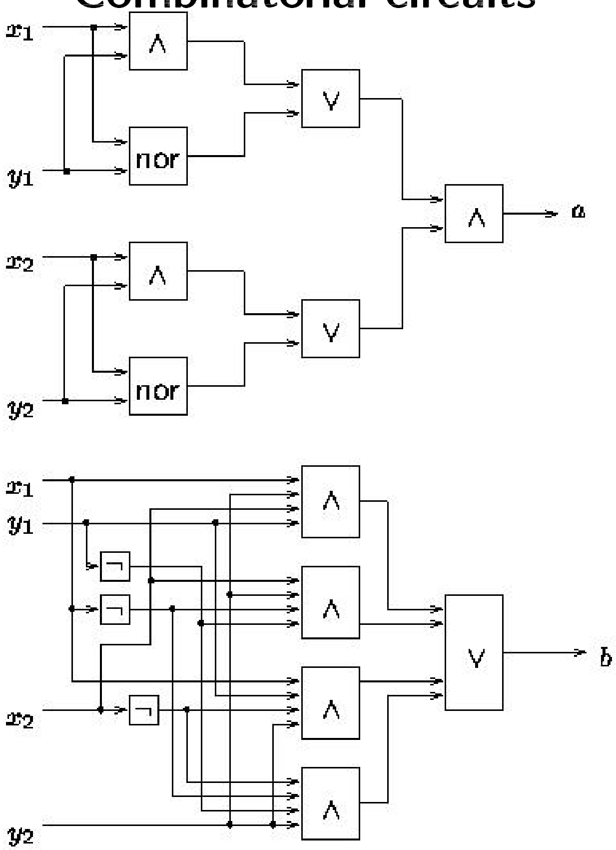
Perfect representation is hopeless:

Theorem 1 (Cook's Theorem)
Satisfiability of Boolean expressions is NP-complete.

(Tautology-checking is co-NP-complete)

Good representations are
compact and
efficient
on real-life examples

Combinatorial circuits



Are these equivalent? Do they represent a tautology? Are they satisfiable?

Shannon Expansion

Def: $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$

x is the test expression and thus this is an if-then-else.

We can represent all operators using if-then-else on unnegated variables and constants 0(false) and 1(true). This is called INF.

Shannon expansion w.r.t. x:

$t = x \rightarrow t[1/x], t[0/x]$

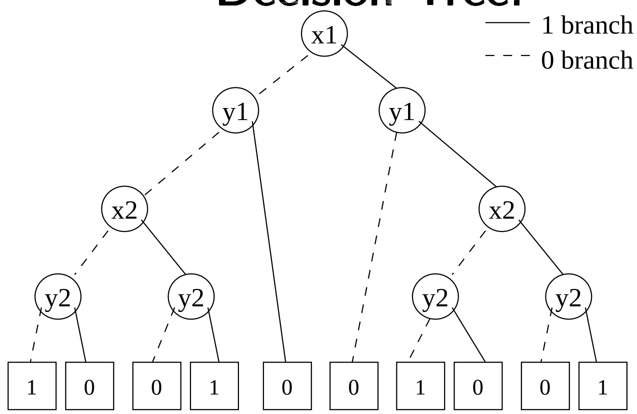
Any boolean expression is equivalent to an expression in INF.

Example

$t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. Represent this in INF form with order x_1, y_1, x_2, y_2.

$t = x_1 \to t_1, t_0$
$t_0 = y_1 \to 0, t_{00}$
  (since $x_1 = 1, y_1 = 0 \to t = 0$)
$t_1 = y_1 \to t_{11}, 0$
  (since $x_1 = 0, y_1 = 1 \to t = 0$)
$t_{00} = x_2 \to t_{001}, t_{000}$
$t_{11} = x_2 \to t_{111}, t_{000}$
$t_{000} = y_2 \to 0, 1 \qquad (x_1 = 0, y_1 = 0, x_2 = 0)$
$t_{001} = y_2 \to 1, 0 \qquad (x_1 = 0, y_1 = 0, x_2 = 1)$
$t_{110} = y_2 \to 0, 1 \qquad (x_1 = 1, y_1 = 1, x_2 = 0)$
$t_{111} = y_2 \to 1, 0 \qquad (x_1 = 1, y_1 = 1, x_2 = 1)$

Decision Tree:

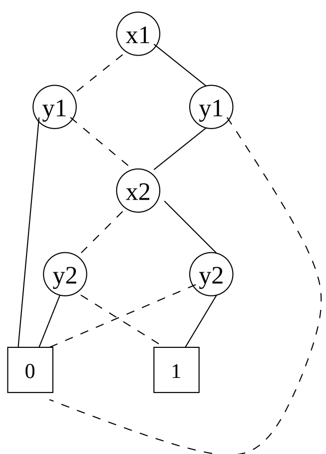


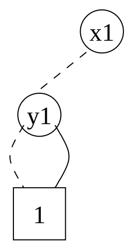
Lots of common subexpressions:
 - identify them!

BDDs – directed acyclic graph of Boolean expressions. If the variables occur in the same ordering on all paths from root to leaves, we call this OBDD.

Example OBDD

OBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with ordering $x_1 < y_1 < x_2 < y_2$





If an OBDD does not contain any redundant tests, it is called ROBDD.

ROBDDs

A *Binary Decision Diagram* is a rooted, directed, acyclic graph (V, E). V contains (up to) two terminal vertices, $0, 1 \in V$. $v \in V \backslash \{0, 1\}$ are non-terminal and have attributes *var*(v), and *low*(v), *high*$(v) \in V$.

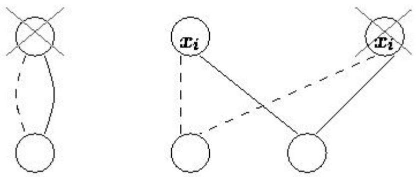
A BDD is *ordered* if on all paths from the root the variables respect a given total order.

A BDD is *reduced* if for all non-terminal vertices u, v,

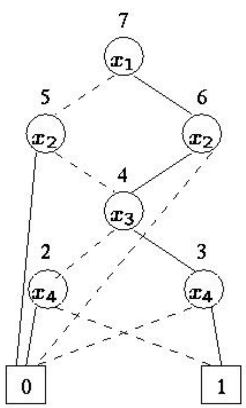
1) *low*$(u) \neq$ *high*(u)

2) *low*$(u) =$ *low*(v), *high*$(u) =$ *high*(v), *var*(u) $=$ *var*(v) implies $u = v$.

ROBDD Examples



reducedness



$(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$

Canonicity of ROBDDs

Lemma 1 (Canonicity lemma) For any function $f : \mathcal{B}^n \to \mathcal{B}$ there is exactly one ROBDD b with variables $x_1 < x_2 < \cdots < x_n$ such that

$$t_b[v_1/x_1, \cdots, v_n/x_n] = f(v_1, \cdots, v_n)$$

for all $(v_1, ..., v_n) \in \mathcal{B}^n$.

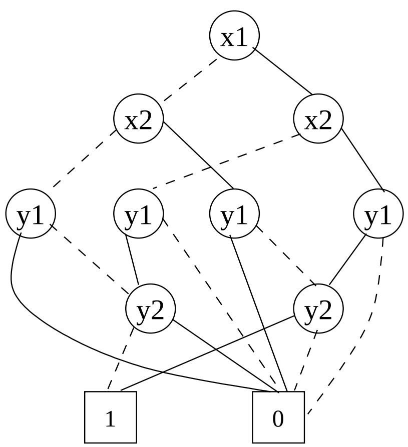
Consequences:

- b is a tautology if and only if $b = \boxed{1}$
- b is satisfiable if and only if $b \neq \boxed{0}$

But..

The size of ROBDD depends *significantly* on the chosen variable ordering!

Example: ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with ordering $x_1 < x_2 < y_1 < y_2$



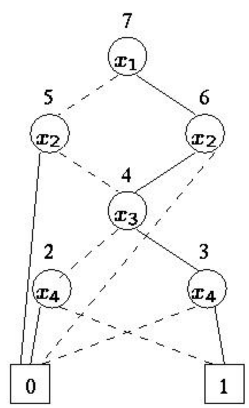
Under ordering $x_1 < y_1 < x_2 < y_2$ had 6 nodes.

Furthermore...

- The size according to one ordering may be exponentially smaller than another ordering.
- Figuring out the optimal ordering of variables is co-NP-complete.
- Some functions have small size independent of ordering, e.g. parity.
- Some functions have large size independent of ordering, e.g., multiplication

Implementing BDDs

{root: integer; var, low, high: array of integer;}



	var	*low*	*high*
0	?	?	?
1	?	?	?
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

Helper Functions: Makenode and Hashing

Makenode ensures reducedness using a hash table

$H : (i, l, h) \to u$

Makenode(H, *max*, b, i, l, h)

if $l = h$ **then return** l
else if *member*(H, i, l, h)
  then return *lookup*(H, i, l, h)
else *max* $\leftarrow$ *max* $+ 1$
  b.*var*(*max*) $\leftarrow i$
  b.*low*(*max*) $\leftarrow l$
  b.*high*(*max*) $\leftarrow h$
  insert$(H, i, l, h,$ *max*$)$
  return *max*

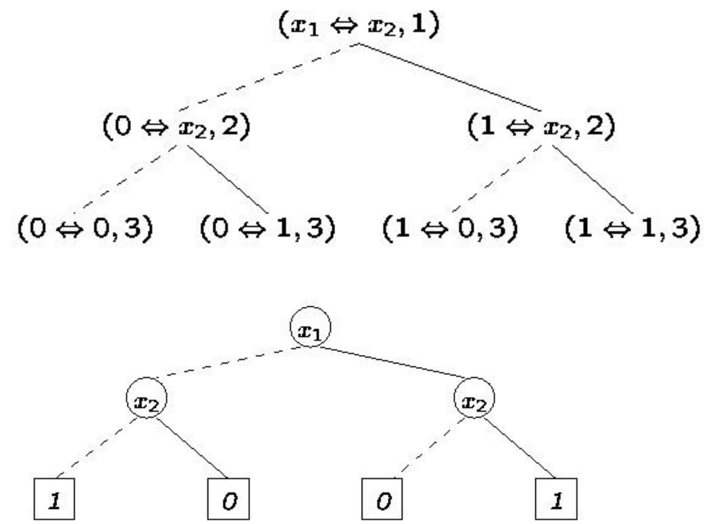
Build

Build: Maps a Boolean expression t into an ROBDD.

function *Build*(t)
$H \leftarrow$ *emptytable*; *max* $\leftarrow 1$
b.*root* $\leftarrow$ *build'*$(t,\ 1)$
return b

function *build'*$(t,\ i)$
if $i > n$ **then**
  if t is false **then return** 0
    else return 1
  else $l \leftarrow$ *build'*$(t[0/x_i], i+1)$
    $h \leftarrow$ *build'*$(t[1/x_i], i+1)$
    return *makenode*$(H,$ *max*$,\ b, i, l, h)$

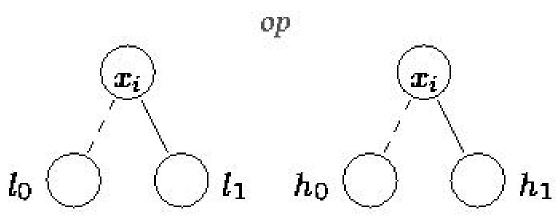
Build Example

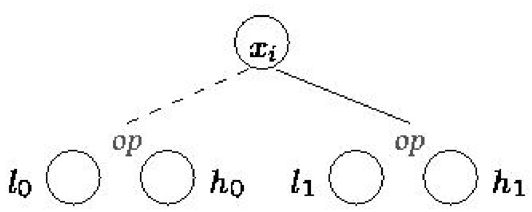


Boolean Operations on ROBDDs

Ordering: $x_1 < \cdots < x_n$

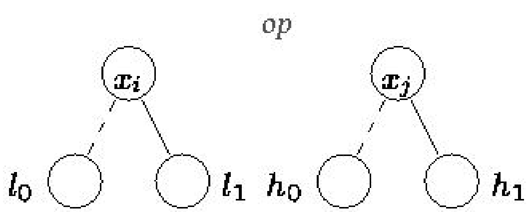
$(x_i \to l_1, l_0)$ *op* $(x_i \to h_1, h_0) =$
$x_i \to (l_1 \text{ op } h_1), (l_0 \text{ op } h_0)$

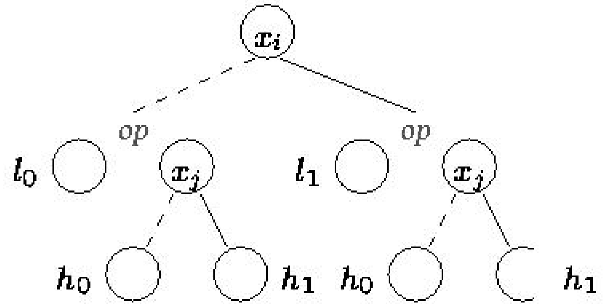




Boolean Operations on ROBDDs(Cont'd)

If $x_i < x_j$:
$(x_i \to l_1, l_0)$ *op* $(x_j \to h_1, h_0) =$
$x_i \to (l_1 \text{ op } (x_j \to h_1, h_0)),$
$(l_0 \text{ op } (x_j \to h_1, h_0))$

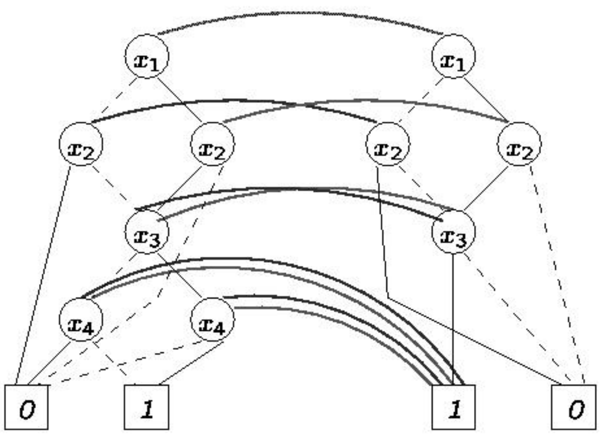




Function *Apply*

Used to perform operations on two ROBDDs.

Example:



Can be either recursive or using dynamic programming.

Other Operations on ROBDDs

Restrict – $b[v/x]$"
given a truth assignment for x, compute ROBDD for b

Size – *size*$(b) = |\{\rho \mid b[\rho] = 1\}|$
"number of valid truth assignments"

Anysat – *anysat*$(b) = \rho$, for some ρ with $b[\rho] = 1$
"give a satisfying assignment"

Compose – *compose*$(b, x, b') = b[x/b']$
"substitute b' for all free occurrences of x"

Existential quantification – $\exists x.b = b[x/0] \vee b[x/1]$

Using dynamic hash-table implementation, can get amortized cost for operations to be $O(1)$.

Representing Boolean Functions

Representation of boolean functions	compact?	satisf'ty	validity
Prop. formulas	often	hard	hard
Formulas in DNF	sometimes	easy	hard
Formulas in CNF	sometimes	hard	easy
Ordered truth tables	never	hard	hard
Reduced OBDDs	often	easy	easy

Representation of boolean functions	$\wedge$	$\vee$	$\neg$
Prop. formulas	easy	easy	easy
Formulas in DNF	hard	easy	hard
Formulas in CNF	easy	hard	hard
Ordered truth tables	hard	hard	hard
Reduced OBDDs	medium	medium	easy

Uses of ROBDDs

Symbolic reasoning about:

- Combinatorial circuits
- Sequential circuits
- Automata
- Program analysis (theorem-proving)

and

- Temporal logic model checking