Linear Temporal Logic Semantics. Binary Decision Diagrams. Interpolation

Viktor Kuncak, EPFL

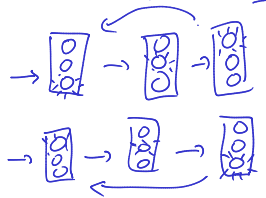
https://lara.epfl.ch/w/fv

History Variables, beyond invariants sets of states in one point of time

How to check properties of past events of a finite-state system?

Example: traffic lights

- $(green\ yellow\ red)*$
- $(red\ yellow\ green)*$



Each of the three single light states is reachable in both cases.

Two traffic lights can are obviously observed as different: different sets of traces.

Ghost state (auxiliary variable): double the state to remember the previous one:

$$green(was : none) \rightarrow yellow(was : green) \rightarrow red(was : yellow) \rightarrow green\ (red)$$

$$red(was : none) \rightarrow yellow(was : red) \rightarrow green(was : yellow) \rightarrow red\ (green)$$

These two systems have different reachable extended states.

We can remember entire past trace (unbounded), or any finite events in the trace.

Such history variables can depend on original state, but do not influence it (we can *monitor* the original system without changing it). We use them to express more properties (just got overflow vs. you have had an overflow).

Fairness and termination properties

If we have only seen a finite trace we cannot necessarily conclude whether the property is true or false.
Examples:

- Execution terminates
- If a customer files a complaint, they will eventually receive an answer.
- Arbiter inside of a hardware bus: every request is eventually served (FIFO vs stack)
- Scheduler: priority (can delay some forever) vs round robin.

Set of infinite traces of a system $M = (S, I, r, A)$
Remember: trace t is an infinite sequence $s_0, a_0, s_1, a_1, s_2, \ldots$ starting from $s_0 \in I$ with steps given by r, $(s_i, a_i, s_{i+1}) \in r$ for all i.

Linear Temporal Logic (LTL)

A sequential circuit as a transition system. B - a boolean formula over state and input vars

$$F ::= B \mid \neg F \mid F_1 \vee F_2 \mid \textbf{next}\, F \mid \textbf{prev}\, F \mid F_1\, \textbf{until}\, F_2 \mid F_1\, \textbf{since}\, F_2$$

Semantics: F holds a position i, written $t, i \models F$:

- $t, i \models B$ iff $[\![B]\!]e = 1$ where e is state at step i of t
- $t, i \models \neg F$ iff not $t, i \models F$
- $t, i \models F_1 \vee F_2$ iff $t, i \models F_1$ or $t, i \models F_2$
- $t, i \models \textbf{next}\, F$ iff $t, i+1 \models F$
- $t, i \models \textbf{prev}\, F$ iff $i > 0$ and $t, i-1 \models F$
- $t, i \models F_1\, \textbf{until}\, F_2$ iff there exists $k \geq i$ such that $t, k \models F_2$ and $t, j \models F_1$ for all j where $i \leq j < k$
- $t, i \models F_1\, \textbf{since}\, F_2$ iff there exists k where $0 \leq k \leq i$ such that $t, k \models F_2$ and $t, j \models F_1$ for all j where $k < j \leq i$

Using LTL: Derived Operators and Example

eventually F: $1\,\mathbf{until}\,F$

globally F: $\neg\,\mathbf{eventually}\,\neg F$

Expand the shorthands above, using the definition of $t, i \models F_1\,\mathbf{until}\,F_2$:

$$\exists k \geq i.\;\; t,k \models F_2 \wedge \forall j.\, i \leq j < k.\; t,j \models F_1$$

Example: $\mathbf{globally}(p \rightarrow \mathbf{eventually}\, q)$

In every trace, for every state of the trace where p holds, either q holds there, or q holds in a some subsequent state of the same trace.

More graphical notation for LTL is also used, e.g.:

next	◯
eventually	◊
globally	□

Traces satisfy $p \rightarrow$ eventually q

but not globally $(p \rightarrow$ eventually $q)$

Linear Temporal Logic Model Checking

There exists an algorithm to check, given sequential circuit and an LTL formula to check whether all traces of the sequential circuit satisfy the formula.

Implemented inside model checkers inside hardware verification tools, as well as standalone model checkers such as nuXmv

`https://nuxmv.fbk.eu/`

Monadic Second-Order Logic of One Successor (S1S)

Conditional Expressions

$(b \rightarrow x) \wedge (\neg b \rightarrow y)$

$b?x:y = (b \wedge x) \vee (\neg b \wedge y)$

We will use them to define BDDs and also to extract interpolants from proofs

Next: Binary Decision Diagrams

In the algorithm to compute reachable states:
states = I
while states' != states do
  states' = states U r[states]

Represent "states" using binary decision diagrams

Computer-Aided Formal Verification (MT 2009)

Binary Decision Diagrams (BDDs)

Daniel Kroening

Oxford University, Computing Laboratory

Version 1.0, 2009

Motivation

UNIVERSITY OF OXFORD

- ✘ The state space of interesting systems is too big, explicit enumeration will fail inevitably.
- ✔ However, the reachable state space isn't random, but follows specific rules
- ✔ We hope to exploit that with data structures that are concise in relevant cases

Binary Decision Diagrams (BDDs)

Randal E. Bryant

- *Binary Decision Diagrams* (BDDs) are a symbolic representation of Boolean functions
- Key idea: specific forms of BDDs are canonical
- [Bryant86] is one of the most-cited papers in computer science

Explicit vs. Symbolic Representations of Functions

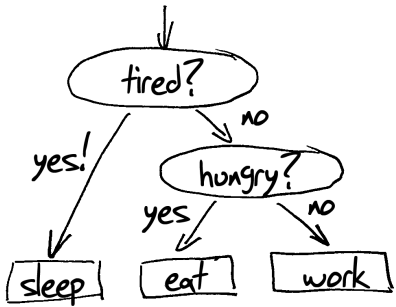
x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Explicit
Obviously gets large

$$f(x,y,z) = x \leftrightarrow (y \wedge z)$$

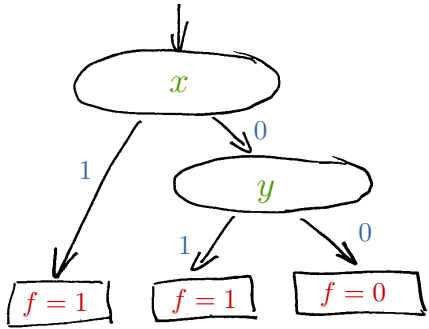
Symbolic
Potentially smaller

Decision Diagrams



- Distinguishes terminal from non-terminal nodes
- The edges are labeled with decisions
- The sink nodes (squares) are labeled with the outcome

Decision Diagrams for Functions

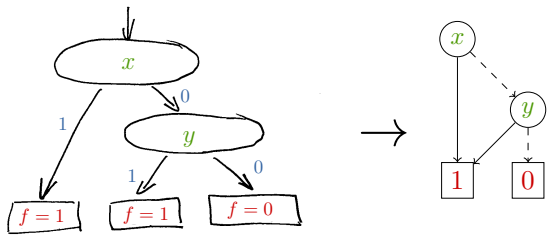


- This encodes $f(x, y) = x \vee y$
- Inner nodes: a variable v
- Out-edges: value for v
- Terminal nodes: function value

BDD

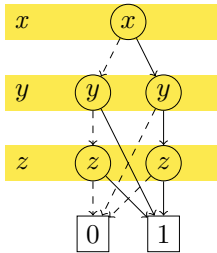
- represents Boolean function $f: \{0,1\}^n \to \{0,1\}$ (n = number of variables)
- as directed acyclic graph (DAG)
  - one root, two leaves (0 and 1), two (colored) edges at inner nodes
- as finite automaton (but no cycles)
  - accepts subset of $\{0,1\}^n$, 1 as accepting state, 0 as dead-end state
- as branching program (but loop-free)
  - `<label>: if <var> goto <label> else goto <label>`
  - `<label>: return [ 0 | 1 ]`

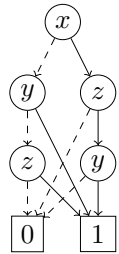
Drawing BDDs



- A solid edge (“high edge”) means the variable is 1
- A dashed edge (“low edge”) means the variable is 0

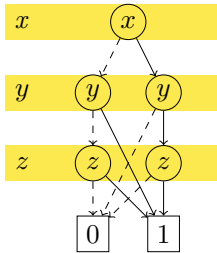
Ordered BDDs

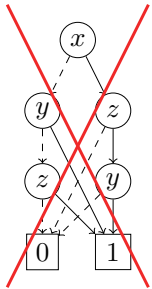




- Assign arbitrary total ordering to variables, e.g., $x < y < z$.
- Variables must appear in that order along all paths.

Ordered BDDs





- Assign arbitrary total ordering to variables, e.g., $x < y < z$.
- Variables must appear in that order along all paths.

Reduced Ordered BDDs (RO-BDDs)

A reduced ordered BDD (RO-BDD) is obtained from an ordered BDD by applying three rules:

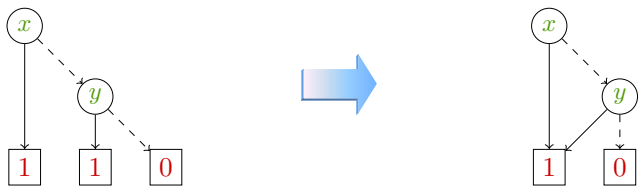
#1: Merge equivalent leaf nodes

#2: Merge isomorphic nodes

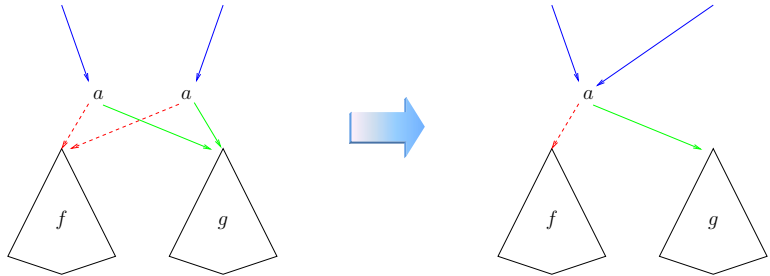
#3: Eliminate redundant tests

Reduction #1: Equivalent Leaf Nodes

This one is trivial:



Reduction #2: Isomorphic Nodes



Merge nodes with the same variable and the same children

Reduction #3: Redundant Tests

Eliminate nodes where both out-edges go into the same node:

Canonicity of BDDs

- Reduced Ordered BDDs (RO-BDDs) [Bryant86]
  - Apply (algebraic) reduction rule until convergence
  - Nodes are ordered with respect to fixed variable order (ordered)

- We restrict our discussion to RO-BDDs (and just call them BDDs)

Theorem Fix variable order, then (RO)BDD for Boolean function f is unique

Proof Obtained from unique minimal automata theorem

Observation: given a BDD it takes constant time to check whether a formula is

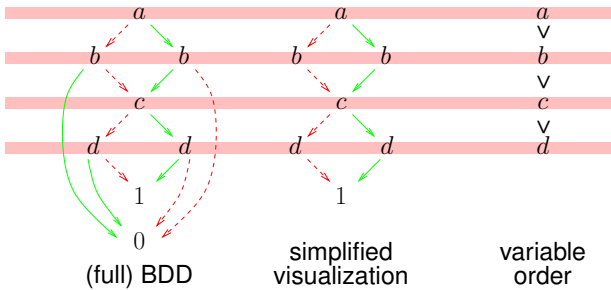
- a tautology,
- inconsistent,
- satisfiable

Aren't these hard?

Variable Ordering and BDD Size

Equality of two 2-bit vectors

$(a, c) = (b, d)$ equivalent to $(a \leftrightarrow b) \wedge (c \leftrightarrow d)$



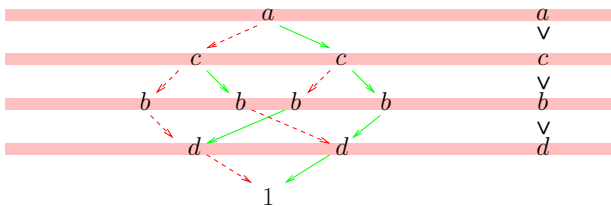
(full) BDD | simplified visualization | variable order

unique BDD for (one) optimal, **interleaved** variable ordering
(linear in the width of the bit vectors)

Variable Ordering and BDD Size

Equality of two 2-bit vectors

$(a, c) = (b, d)$ equivalent to $(a \leftrightarrow b) \wedge (c \leftrightarrow d)$



unique BDD for (one) worst case, **blocked** variable ordering
(exponential in the width of the bit vectors)

Selecting a Good Ordering

✘ Intractable problem

✘ Even when input is an OBDD
(i.e., to find optimum improvement to current ordering)

✔ Application-based heuristics
- Exploit characteristics of application
- E.g., ordering for functions of combinational circuit: traverse circuit graph depth-first from outputs to inputs

Data Structures for BDDs

- Nodes are numbered $0, 1, 2, 3, \ldots$
  (0, 1 are the terminals)

- Variables are numbered $1, 2, 3, \ldots, n$

Data Structures for BDDs

Node table:
$T : u \to (i, l, h)$

Operations:

- init(T)
- u := add(T, i, l, h)
- var(u)
- low(u)
- high(u)

Inverse of node table:
$H : (i, l, h) \to u$

Operations:

- init(H)
- u := lookup(H, i, l, h)
- insert(H, i, l, h, u)

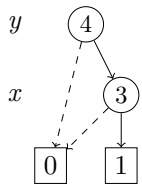
Node Table

Initial State:

$\#$	var	low	$high$
0	$n+1$		
1	$n+1$		

(The $n+1$ variable number for 0/1 will become clear later).

Node Table: Example



#	*var*	*low*	*high*
0	3		
1	3		
2	–	–	–
3	2 *x*	0	1
4	1 *y*	0	3

Inserting a Node into T

MK[T, H](i, l, h)

if $l = h$ **then**

return l;

else if lookup(H, i, l, h)$\neq$ **then**

return lookup(H, i, l, h);

else

u := add(T, i, l, h);
insert(H, i, l, h, u);
return u;

Building a BDD

Goal: build BDD for f.

Use Shannon's expansion as follows

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

to break problem into two subproblems. Solve subproblems recursively.

Building a BDD

BUILD[T,H](f)
    return BUILD2(f, 1);

function BUILD2(f, var) =
    if $var > n$ **then**
        if f is false **then return** 0 **else return** 1;
    else
        f_0 := BUILD2($f[0/x_i]$, $var + 1$);
        f_1 := BUILD2($f[1/x_i]$, $var + 1$);
        return MK[T,H](var, f_0, f_1);
end

Basic Operations on BDDs

- Canonicity implies
  - $f \equiv g$ iff. BDDs for f and g are equal
  - f tautology iff. BDD for f is 1
  - f satisfiable iff. BDD for f is not 0

- Basic operation: RESTRICT
  - $f|_{x=0}$: replace all x nodes by their low-edge sub-tree.
  - $f|_{x=1}$: replace all x nodes by their high-edge sub-tree.

Applying a Function

Conjunction, ...: $f(a, b, c, d) \star g(a, b, c, d)$,
where the symbol $\star$ denotes some binary operator.

Again, use Shannon's expansion as follows

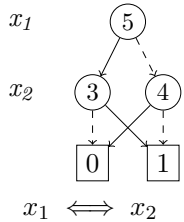
$$f \star g = \neg x \wedge (f|_{x=0} \star g|_{x=0}) \ \vee \ x \wedge (f|_{x=1} \star g|_{x=1})$$

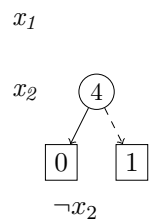
to break problem into two subproblems. Solve subproblems recursively.

Applying a Function

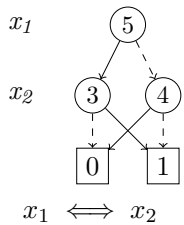
```
function APPLY(u1, u2) =
    if u1, u2 ∈ {0, 1} then
        u := u1 ⋆ u2;
    else if var(u1) = var(u2) then
        u := MK(var(u1), APPLY(low(u1),low(u2)),
                         APPLY(high(u1),high(u2)));
    else if var(u1) < var(u2) then
        u := MK(var(u1), APP(low(u1),u2), APP(high(u1),u2));
    else      (* var(u1) > var(u2) *)
        u := MK(var(u2), APP(u1,low(u2)), APP(u1,high(u2)));
    return u;
```

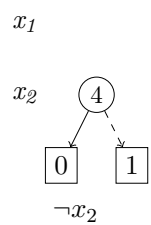
Example





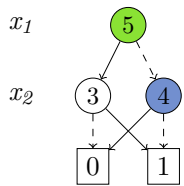
Example





Now compute BDD for $(x_1 \iff x_2) \vee \neg x_2$ using APPLY!

Example (cont.)



APPLY(5, 4):

var(5)=1, var(4)=2

MK(var(5), APP(low(5),4), APP(high(5),4));

MK(1, APP(4,4), APP(3,4)); MK(1, 4, 1);

APP(4, 4):

var(4)=var(4)=2

MK(2, APP(1,1), APP(0,0));

MK(2, 1, 0); =4!

APP(3, 4):

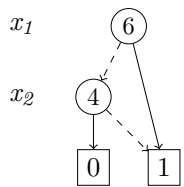
var(3)=var(4)=2

MK(2, APP(0,1), APP(1,0));

MK(2, 1, 1); =1!

Example (cont.)

Final result:



#	*var*	*low*	*high*
0	3		
1	3		
2	–	–	–
3	–	–	–
4	$2\,x_2$	1	0
5	–	–	–
6	$1\,x_1$	4	1

Existential Quantification

For BDD f and a variable x compute BDD for

$$\exists\, x.\ f$$

Existential Quantification

For BDD f and a variable x compute BDD for

$$\exists\, x.\ f$$

This is equivalent to

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

- Two RESTRICT operations
- One call to APPLY

Summary: Binary Decision Diagrams (BDDs)

- Canonical representation of boolean functions
  - Every Boolean function has a unique RO-BDD
  - Try to obtain small graphs by means of sharing

✔ Efficient manipulation algorithms
(e.g., conjunction, quantification, . . .)

✘ Often explode in size