Software Verification

Symbolic representations

Ahmed Rezine

IDA, Linköpings Universitet

Spring 2023

Symbolic Model Checking

In these two lectures we discuss:

- symbolic techniques to manipulate sets of states instead of individual states as in explicit model checking
- how such techniques can concisely manipulate (huge, even infinite) state spaces
- today: symbolic (un)bounded model checking with NuSMV

Symbolic Model Checking (cont.)

- the main idea is to use predicates to denote sets of states of the Kripke Structure
- the predicates have to be efficiently represented and manipulated. Today:
  - using Binary Decision Diagrams (BDDs)
  - using SAT sentences

Outline

Binary Decision Diagrams and verification

SAT solving and verification

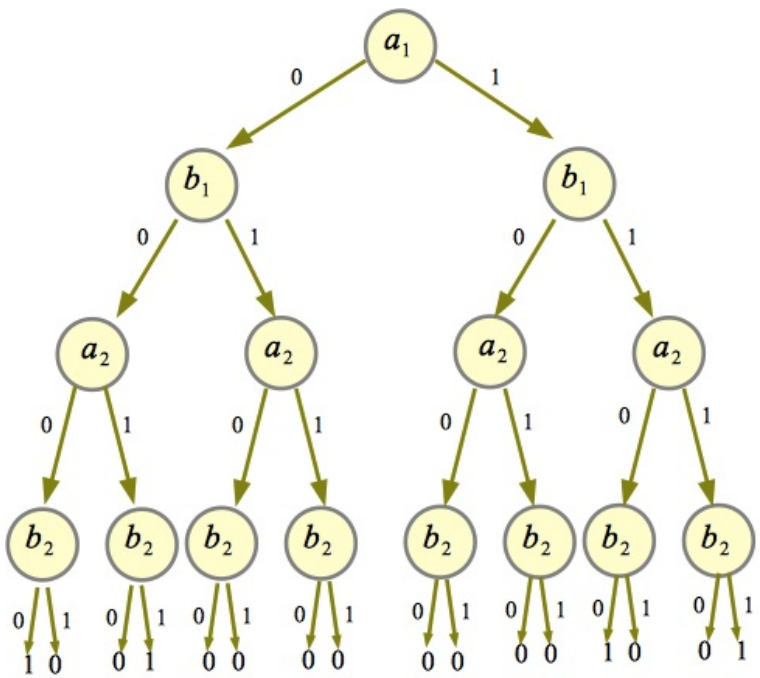
NuSMV

Outline

Binary Decision Trees

A Binary Decision Tree is a rooted directed tree with terminal and non terminal vertices.

- a non terminal vertex v has a variable $var(v)$ and two successors $low(v)$ and $high(v)$.
- a terminal vertex v has a value $value(v)$ in $\{0, 1\}$



$$a_1 \Leftrightarrow b_1 \wedge a_2 \Leftrightarrow b_2$$

Binary Decision Diagrams (BDDs)

A BDD is a rooted, directed acyclic graph with terminal and non-terminal vertices. $value(v)$, $var(v)$, $low(v)$ and $high(v)$ are defined as for binary decision trees. A boolean function $f_v(x_1, \ldots x_n)$ is associated every vertex v.

- if v is a terminal vertex:
  - if $value(v) = 1$ then $f_v(x_1, \ldots x_n) = 1$
  - if $value(v) = 0$ then $f_v(x_1, \ldots x_n) = 0$
- if v is a non terminal vertex with $var(v) = x_i$ then $f_v(x_1, \ldots x_n) = (\neg xi \wedge f_{low(v)}(x_1, \ldots x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots x_n))$

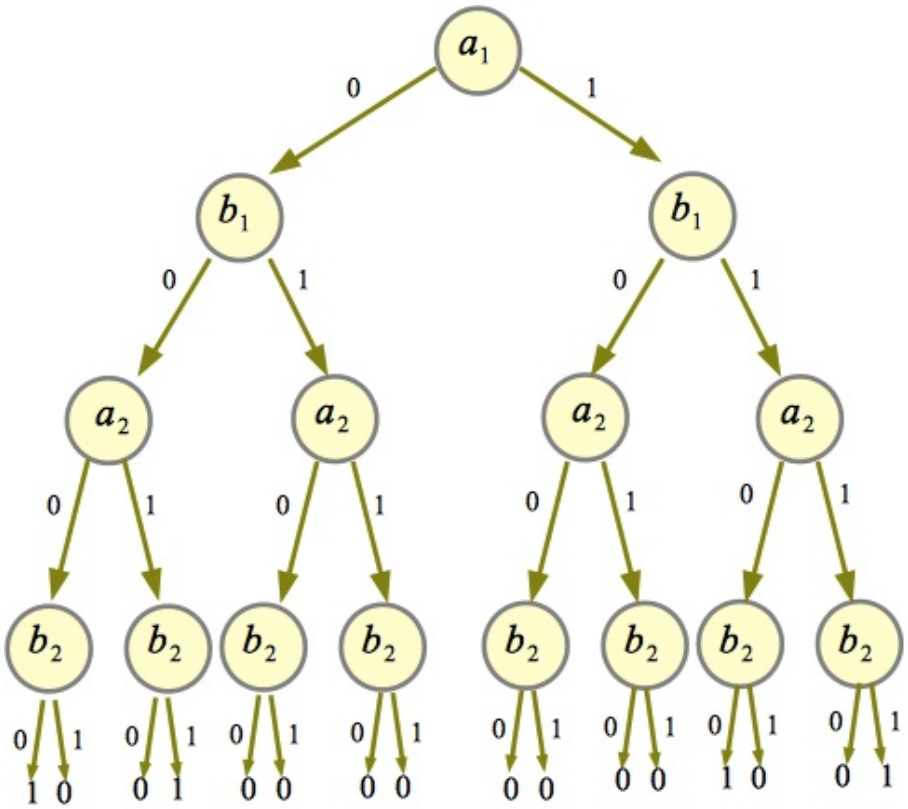
BDDs Canonicity

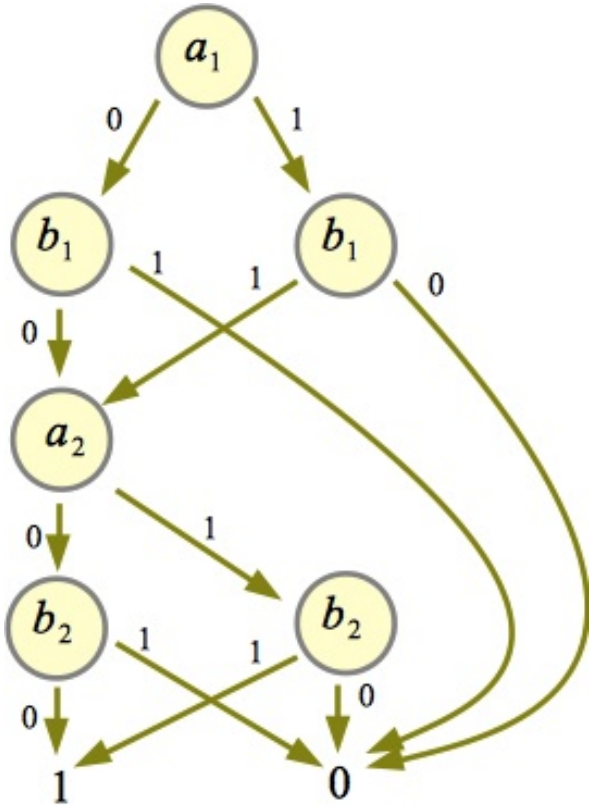
- If it is the case that:
  1. all BDDs have the same order on the variables along each path
  2. each BDD has no redundant vertices or isomorphic subtrees
- then two boolean functions are equivalent iff two corresponding BDDs are isomorphic

BDDs Canonicity: the Reduce Algorithm:

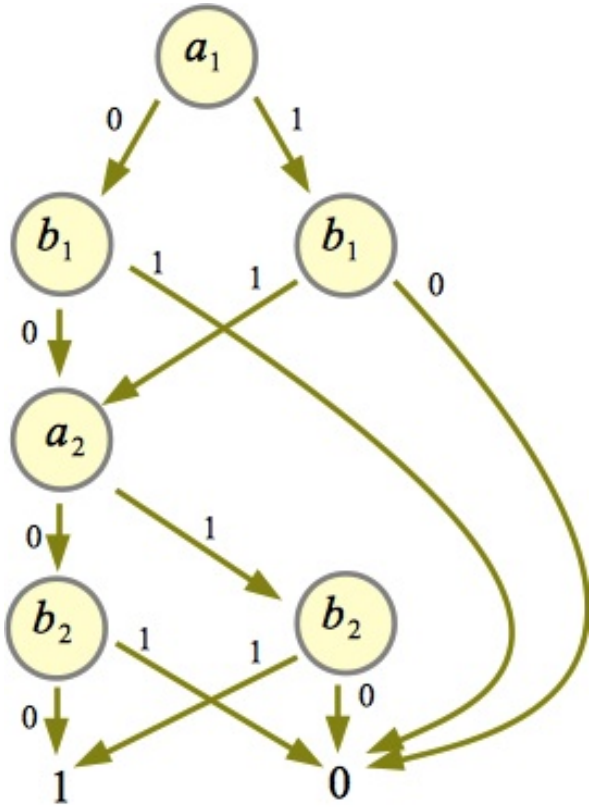
- Starting from a BDD, repeatedly:
  1. Eliminating all terminal vertices but one of each value and redirecting arcs that used to point to deleted vertices to the kept terminal vertex with the same value
  2. Eliminating duplicate non terminals with the same variable, and same low and high arcs (and redirect arcs)
  3. If $low(v) = high(v)$ eliminating v and redirecting arcs to $low(v)$.
- Then we obtain (linear in the size of the original BDD) a canonical representation of boolean formulas

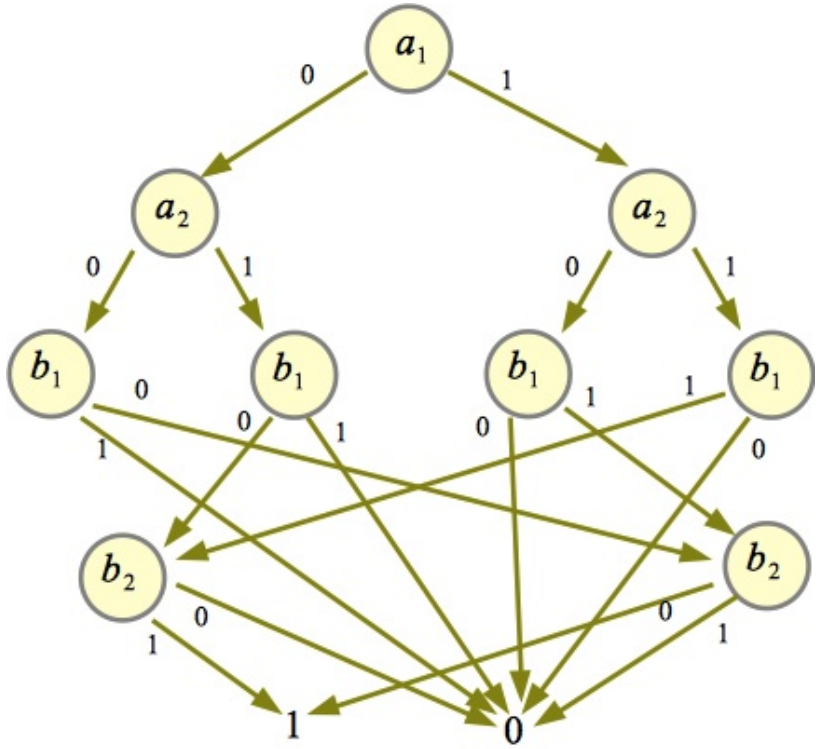
BDDs Canonicity (cont.)





BDDs Order





Operations on BDDs: the Apply Algorithm

Given a binary operation *op* on boolean formulas f, g:

- The *Shanon Expansion* of f with respect to variable x is $f = \neg x.f_{|x \leftarrow 0} + x.f_{|x \leftarrow 1}$
- This also applies to the result of $(f\ op\ g)$, i.e., $(f\ op\ g) = \neg x.(f_{|x \leftarrow 0}\ op\ g_{|x \leftarrow 0}) + x.(f_{|x \leftarrow 1}\ op\ g_{|x \leftarrow 1})$

The Apply Algorithm

Starting from the roots u, v of the BDDs of f, g:

- u, v are terminal vertices: $(f\ op\ g) = value(u)\ op\ value(v)$
- $x = var(u) = var(v)$,
  $(f\ op\ g) = \neg x.(f_{|x \leftarrow 0}\ op\ g_{|x \leftarrow 0}) + x.(f_{|x \leftarrow 1}\ op\ g_{|x \leftarrow 1})$
- $x = var(u) < var(v)$, then g does not depend on x:
  $(f\ op\ g) = \neg x.(f_{|x \leftarrow 0}\ op\ g) + x.(f_{|x \leftarrow 1}\ op\ g)$
- $x = var(v) < var(u)$, then f does not depend on x:
  $(f\ op\ g) = \neg x.(f\ op\ g_{|x \leftarrow 0}) + x.(f\ op\ g_{|x \leftarrow 1})$

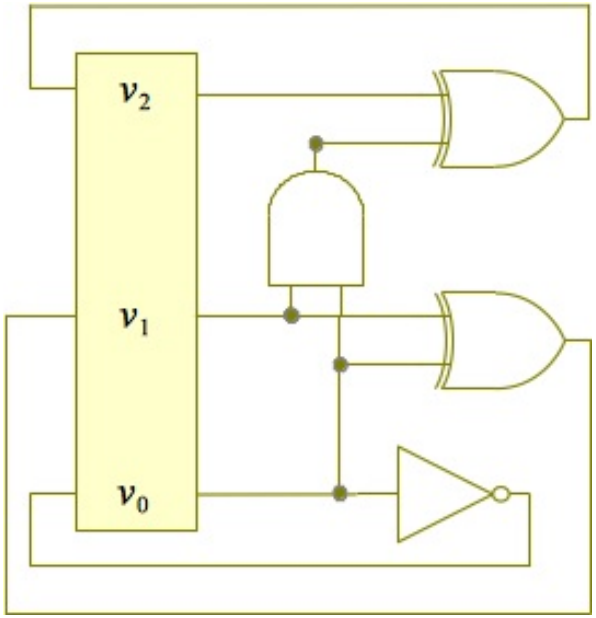
The Apply Algorithm

- Use dynamic programming and memoize for efficiency
- There is a quadratic number of results to store
- The result is not necessarily canonical, may need to use reduce afterwards

Other operations on BDDs

- restrict: compute $f_{|x \leftarrow 0}$. For every node v with $var(v) = x$, redirect each incoming edge to $low(v)$ and delete v.
- exists: compute $\exists x.f$: use $\exists x.f = f_{x \leftarrow 0} + f_{x \leftarrow 1}$, restrict and apply.

Successors and predecessors



$$R(v, v') = \begin{pmatrix} & v'_0 = \neg v_0 \\ \wedge & v'_1 = v_0 \oplus v_1 \\ \wedge & v'_2 = (v_0 \wedge v_1) \oplus v_2 \end{pmatrix}$$

- $\{000, 010, 001, 011\}$ captured by $(\neg v_0)$. $(\exists v. \neg v_0 \wedge R(v, v')) = v'_0$ captures the successors, i.e., $\{100, 110, 101, 111\}$.
- Successors of $f(v)$: $\exists v. f(v) \wedge R(v, v')$
- $\{010, 110, 011, 111\}$ captured by (v'_1). $(\exists v'. v'_1 \wedge R(v, v')) = (v_0 \oplus v_1)$ captures the predecessors, i.e., $\{010, 011, 100, 101\}$.
- Predecessors of $f(v')$: $\exists v'. f(v') \wedge R(v, v')$

$$(\mathbf{EX}\ f) = (\exists v'. f(v') \wedge R(v, v'))$$

Fixpoints and Symbolic Model Checking

$$(\mathbf{EX}\ f) = (\exists v'.f(v') \wedge R(v, v'))$$

Data: monotonic τ
Result: smallest fixpoint of τ
$Q = \tau(false)$;
$Q' = false$;
while $Q \neq Q'$ **do**
    $Q' = Q$;
    $Q = \tau(Q')$;
return Q;

$\mathbf{E}[f_1 \mathbf{U}\ f_2]$ is the least fixpoint of
$\tau(Z) = f_2 \vee (f_1 \wedge (\mathbf{EX}\ Z))$

Data: monotonic τ
Result: greatest fixpoint of τ
$Q = \tau(true)$;
$Q' = true$;
while $Q \neq Q'$ **do**
    $Q' = Q$;
    $Q = \tau(Q')$;
return Q;

$\mathbf{EG}\ f$ is the greatest fixpoint of
$\tau(Z) = f \wedge (\mathbf{EX}\ Z)$

- Other *CTL* formulas can be rewritten using these

Outline

Basics

Sat solvers usually take propositional formulas written in conjunctive normal (cnf):

- variables: propositional variables, e.g., $x, y, z, \ldots$
- literals: variables or their negations, e.g., $x, \overline{x}, y, \ldots$
- clause: disjunction of literals, e.g., $(x \vee \overline{y})$
- cnf formula: conjunction of clauses, e.g., $(x \vee \overline{y}) \wedge (y \vee z) \wedge (\overline{z}) \wedge (\overline{x})$
- a (partial) variable assignment associates (some of the) variables to Boolean values in $\{0, 1\}$

Basics: a well known NP-problem

Given a propositional formula (usually in cnf), does there exist a satisfying assignment (i.e., a variable assignment for which the formula evaluates to true)?

- $x \mapsto 1, y \mapsto 1, z \mapsto 0$ is a satisfying assignment for $(x \vee \overline{y}) \wedge (y \vee z) \wedge (\overline{z})$. The formula is **satisfiable**
- $(x \vee \overline{y}) \wedge (y \vee z) \wedge (\overline{z}) \wedge (\overline{x})$ has no satisfying assignment, it is **unsatisfiable**
- Success story with applications in hardware/software model checking, planning, combinatorial design, test pattern generation, protocol design, bioinformatics, etc.
- From 100 variables and 200 constraints in early 90s to more than a 1,000,000 variables and 5,000,000 constraints

Resolution

$$\frac{(w_1 \vee x) \quad (w_2 \vee \overline{x})}{(w_1 \vee w_2)}$$

- $(w_1 \vee w_2)$ is the resolvant obtained by resolving $(w_1 \vee x) \wedge (w_2 \vee \overline{x})$ on variable x.
- Eliminate one variable at a time. To eliminate a variable x, first add all resolvants that can be obtained by resolving on x.
- Sound (if states unsat then indeed unsat) and complete (if unsat then will state unsat) proof rule for propositional logic.

Resolution

The formula $(x \vee \overline{y}) \wedge (y \vee z) \wedge (\overline{x} \vee z)$

	x	y
$(x \vee \overline{y})$	$(z \vee \overline{y})$	(z)
$(y \vee z)$	$(y \vee z)$	
$(\overline{x} \vee z)$	(z)	
$(x \vee z)$		

is satisfiable (sat for short).

Resolution

The formula $(x \vee \overline{y}) \wedge (y \vee z) \wedge (\overline{x} \vee z) \wedge (\overline{z})$

$$\dfrac{\dfrac{\dfrac{(x \vee \overline{y}) \qquad (y \vee z)}{(x \vee z)} \qquad (\overline{x} \vee z)}{(z)} \qquad (\overline{z})}{()}$$

is unsatisfiable (unsat for short)..

Unit propagation

- A unit is a clause where there is one unassigned literal while all other literals are assigned 0.
- The only chance for the current assignment to satisfy the formula is to assign 1 to unassigned literals in unit clauses
- $(x) \wedge (\overline{x} \vee y) \wedge (\overline{y} \vee z \vee s) \wedge (z \vee \overline{y} \vee \overline{x}) \wedge \ldots$
  - implied assignment: $x = 1$, antecedent (x)
  - implied assignment: $y = 1$, antecedent $(\overline{x} \vee y)$
  - implied assignment: $z = 1$, antecedent $(z \vee \overline{y} \vee \overline{x})$
  - implied assignment: $s = 1$, antecedent $(\overline{y} \vee z \vee s)$

Davis-Putnam-Logemann-Loveland (DPLL)

- decides satisfiability for sat-cnf problems
- introduced in early 60s
- basis of modern sat solvers
- idea: alternate unit propagation, choosing a value for some variable, and recursively checking the result, if does not give satisfying assignment then backtrack (remove) with opposit value

Davis-Putnam-Logemann-Loveland (DPLL)

Algorithm 1: DPLL-recursive(φ, ρ)

Input: φ: cnf formula, ρ: partial assignment

Output: UNSAT or satisfying assignment

$(\varphi, \rho) := \mathit{UnitPropagate}(\varphi, \rho)$;

if φ *contains an empty clause* **then**

  return *UNSAT*;

if φ *has no clauses left* **then**

  Output ρ;

  return SAT;

$l :=$ a literal not assigned by ρ;

if *(DPLL-recursive*$(\varphi[l], \rho \cup \{l\}))$ **then**

  return *SAT*;

return *DPLL-recursive*$(\varphi[\neg l], \rho \cup \{\neg l\})$;

DPLL, Example

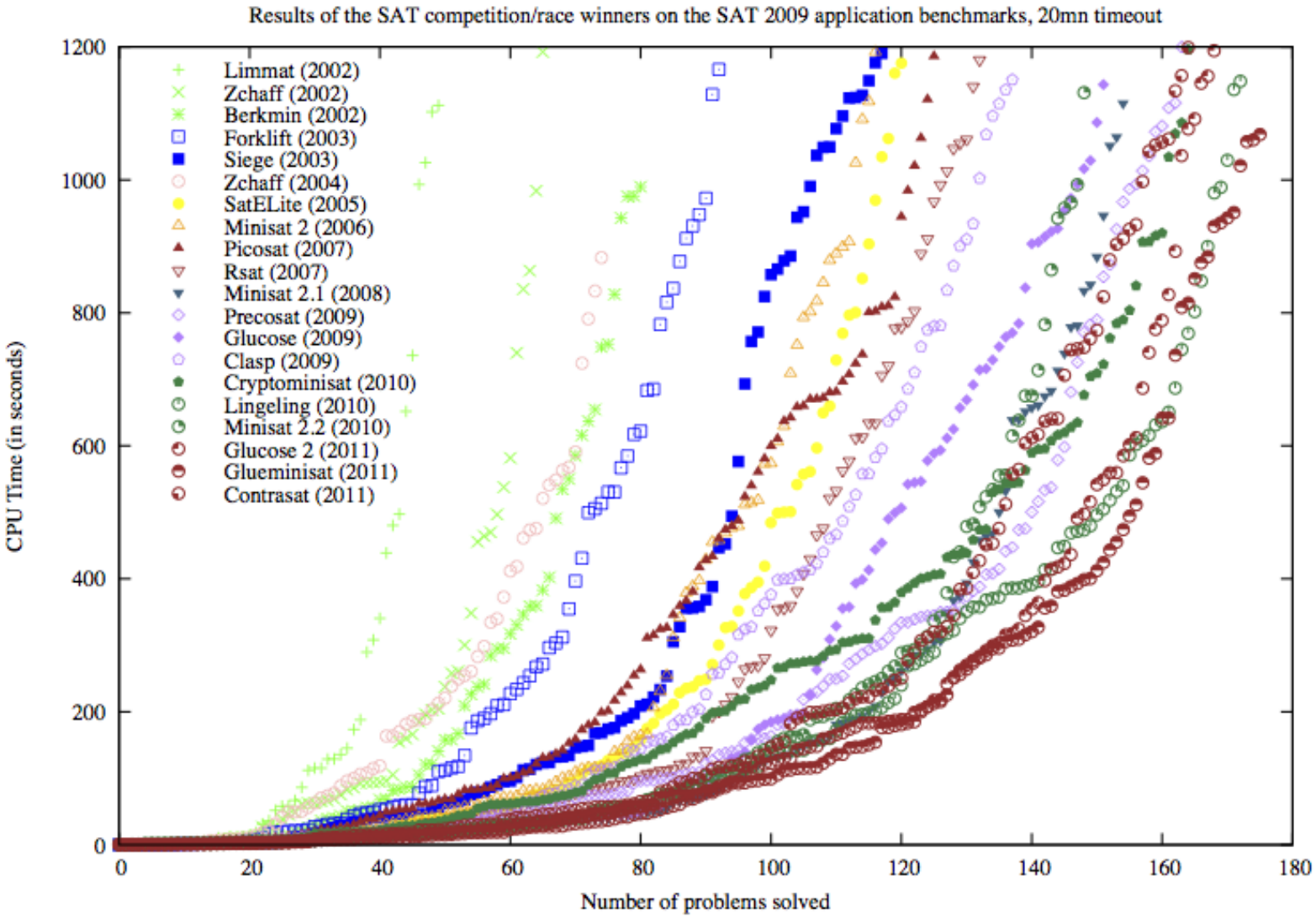
$$\varphi = (x \vee y) \wedge (y \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee z)$$

level	decision	formula
0	$x = 1$	$(1 \vee y) \wedge (y \vee z) \wedge (\overline{1} \vee \overline{y} \vee \overline{z}) \wedge (\overline{1} \vee z)$
1	$y = 1$	$(1 \vee 1) \wedge (1 \vee z) \wedge (\overline{1} \vee \overline{1} \vee \overline{z}) \wedge (\overline{1} \vee z)$
1	$y = 0$	$(1 \vee 0) \wedge (0 \vee z) \wedge (\overline{1} \vee \overline{0} \vee \overline{z}) \wedge (\overline{1} \vee z)$
...		

DPLL, Example 2

$$\varphi = (a \vee \overline{b} \vee d) \wedge (a \vee \overline{b} \vee e) \wedge (\overline{b} \vee \overline{d} \vee \overline{e}) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \overline{d}) \wedge (a \vee b \vee \overline{c} \vee e) \wedge (a \vee b \vee \overline{c} \vee \overline{e})$$

Modern solvers



What made it possible?

- Clause learning with non chronological backtracking
- search restarts
- lazy data-structures
- ...

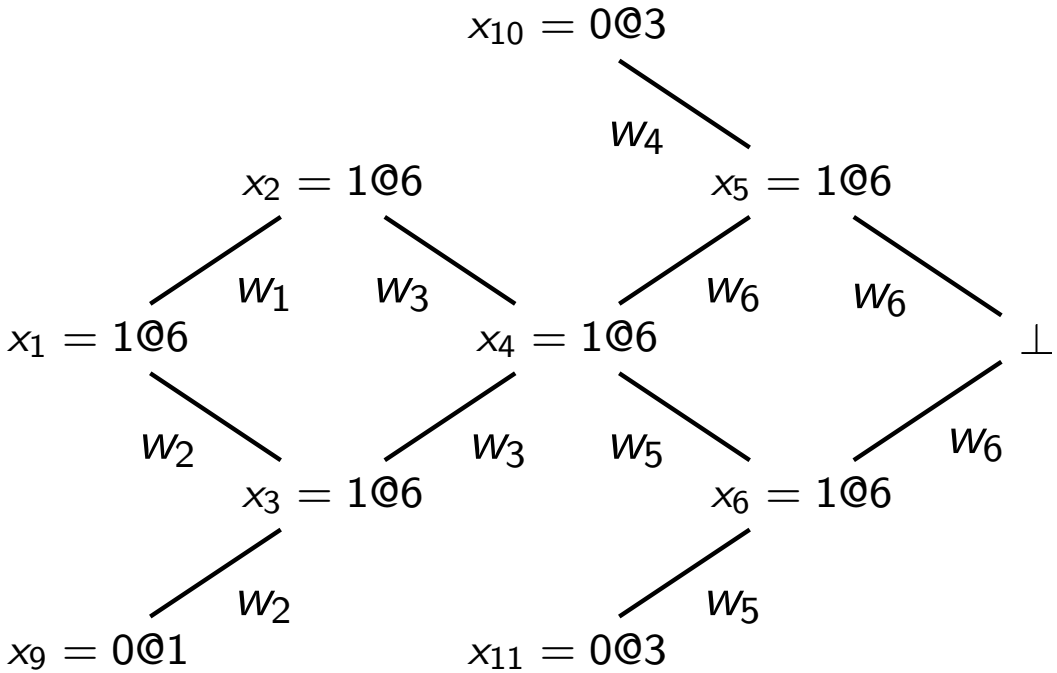
Clause learning

- capture "cause" of encountered conflict as a clause
- the conflict clause is added to the formula
- if deciding $x = 1$ and $y = 0$ led to unsat then remember this by adding $(\overline{x} \vee y)$ to the formula
- learn "reasons" of discovered inconsistencies in order to avoid them in the future

Conflict clauses

- Current assignment: $x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, \ldots$
- Current decision: $x_1 = 1@6$

$$
\begin{aligned}
w_1 &= (\overline{x_1} \vee x_2) \\
w_2 &= (\overline{x_1} \vee x_3 \vee x_9) \\
w_3 &= (\overline{x_2} \vee \overline{x_3} \vee x_4) \\
w_4 &= (\overline{x_4} \vee x_5 \vee x_{10}) \\
w_5 &= (\overline{x_4} \vee x_6 \vee x_{11}) \\
w_6 &= (\overline{x_5} \vee \overline{x_6}) \\
w_7 &= (x_1 \vee x_7 \vee \overline{x_{12}}) \\
w_8 &= (x_1 \vee x_8) \\
w_9 &= (\overline{x_7} \vee \overline{x_8} \vee \overline{x_{13}}) \\
&\ldots.
\end{aligned}
$$



Implication graph

- source nodes of implication graph can be used as a conflict clause
- here, $(\overline{x_1} \vee x_9 \vee x_{10} \vee x_{11})$ can be added as a conflict clause
- better clauses with "unique implication point"
- here, add $(\overline{x_4} \vee x_{10} \vee x_{11})$

Bounded Model Checking

- an alternative approach is to encode erroneous executions as sat formulas, and
- to use sat solvers to establish their satisfiability
- this approach turns out to scale very well, but it can only guarantee correctness up to a given bound
- the approach leverages on the tremendous development in sat solvers' technology

$$BMC(M, p, k) = Init(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

Outline

Introduction

- NuSMV is an open source symbolic model checker
- the latest version is 2.6 and you can get it from `http://nusmv.fbk.eu`
- It uses both BDD and SAT representations
- Performs CTL and LTL model checking
- Can capture (A)Synchronous finite models
- allows for interactive and random simulation

Synchronous: Single Process Example

```
1  MODULE main
2  VAR
3    request : boolean;
4    state : {ready,busy};
5  ASSIGN
6    init(state) := ready;
7    next(state) := case
8                     state = ready & request : busy;
9                     TRUE : {ready,busy};
10                  esac;
11 SPEC
12   AG(request -> AF state = busy)
```

/courses/TDDE34/NuSMV/share/nusmv/examples/smv-dist/short.smv

Synchronous: Binary Counter

```
1  MODULE main
2  VAR
3    bit0 : counter_cell(TRUE);
4    bit1 : counter_cell(bit0.carry_out);
5    bit2 : counter_cell(bit1.carry_out);
6
7  SPEC
8    AG AF bit2.carry_out
9
10
11 MODULE counter_cell(carry_in)
12 VAR
13   value : boolean;
14 ASSIGN
15   init(value) := FALSE;
16   next(value) := value xor carry_in;
17 DEFINE
18   carry_out := value & carry_in;
```

Simulation

```
1  MODULE main
2  VAR
3    request : boolean;
4    state : {ready,busy};
5  ASSIGN
6    init(state) := ready;
7    next(state) := case
8                     state = ready & request : busy;
9                     TRUE : {ready,busy};
10                  esac;
11 SPEC
12   AG(request -> AF state = busy)
```

```
> NuSMV -int short.smv
NuSMV> go
NuSMV> pick_state -r
NuSMV> print_current_state -v
NuSMV> simulate -r -k 3
NuSMV> show_traces -v
NuSMV> simulate -i -k 1
```

Verification

- Use SPEC for CTL specifications
- Use LTLSPEC for LTL specifications
- `> NuSMV semaphore.smv`
- Use fairness to ensure certain sets are visited infinitey often (e.g. that each process is scheduled, using “running”)

Example: counter

```
1  -- A simple counter
2  MODULE main
3  VAR
4    y : 0..15;
5
6  ASSIGN
7    init(y) := 0;
8
9  TRANS
10   case
11     y = 7 :  next(y) = 0;
12     TRUE     :  next(y) = (y + 1) mod 16;
13   esac
14
15 LTLSPEC G (y=4 -> X y=6)
```

```
NuSMV -bmc bmc_tutorial.smv
NuSMV -bmc -bmc_length 4 bmc_tutorial.smv
```

Example: counter

```
1  -- A simple counter
2  MODULE main
3  VAR
4    y : 0..15;
5
6  ASSIGN
7    init(y) := 0;
8
9  TRANS
10   case
11     y = 7 :  next(y) = 0;
12     TRUE     :  next(y) = (y + 1) mod 16;
13   esac
14
15 LTLSPEC ! G F (y = 2)
```

```
NuSMV -bmc bmc_tutorial.smv
NuSMV -int bmc_tutorial.smv
NuSMV> go_bmc
NuSMV> check_ltlspec_bmc_onepb -k -9 -l 0
NuSMV> check_ltlspec_bmc_onepb -k -8 -l 1
NuSMV> check_ltlspec_bmc_onepb -k -9 -l 1
```