Using High Performance SAT and QBF Solvers

Armin Biere

Institute for Formal Models and Verification

Johannes Kepler University

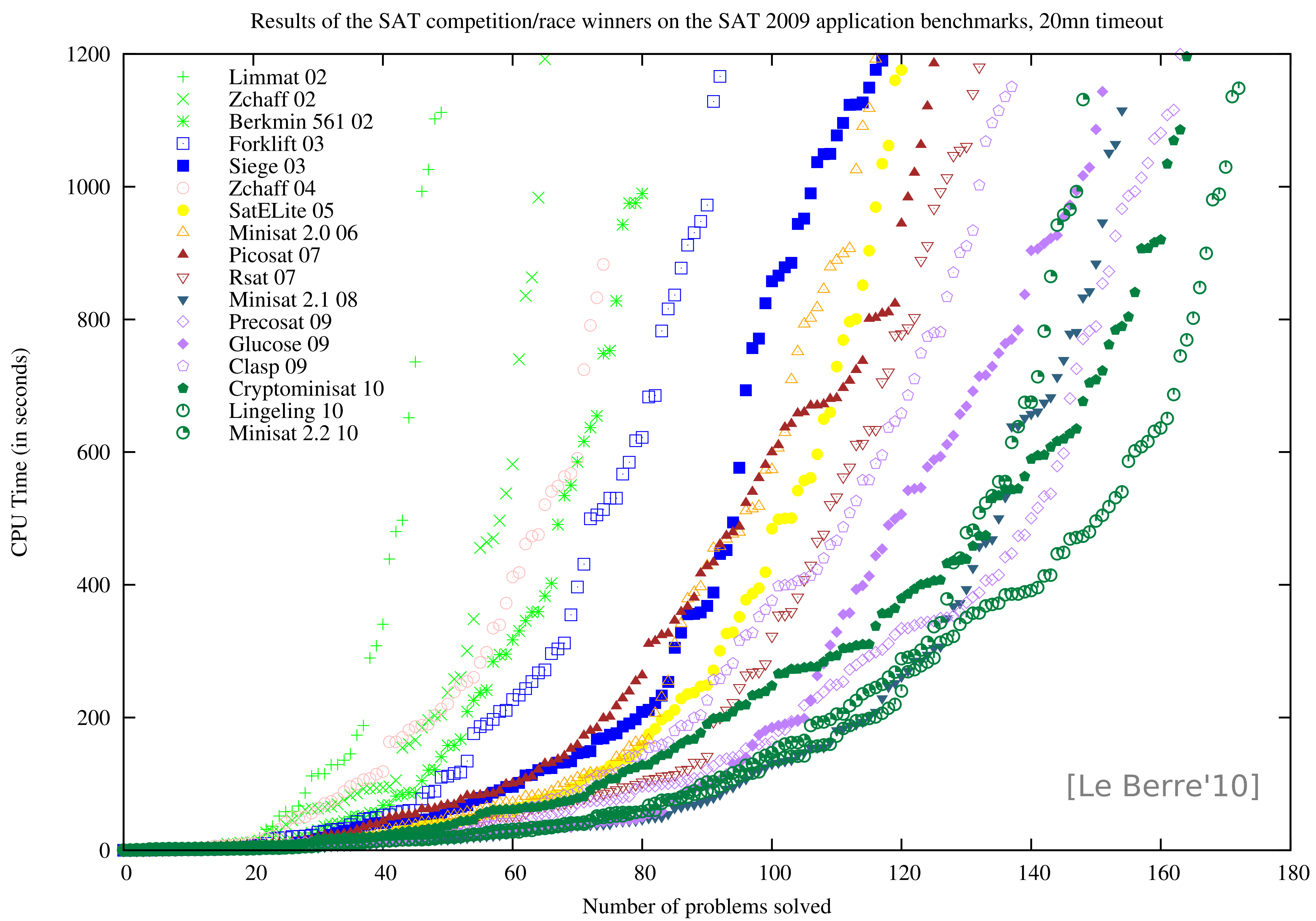
Linz, Austria

Theorem Proving Tools for Program Analysis

Tutorial co-located with POPL 2011

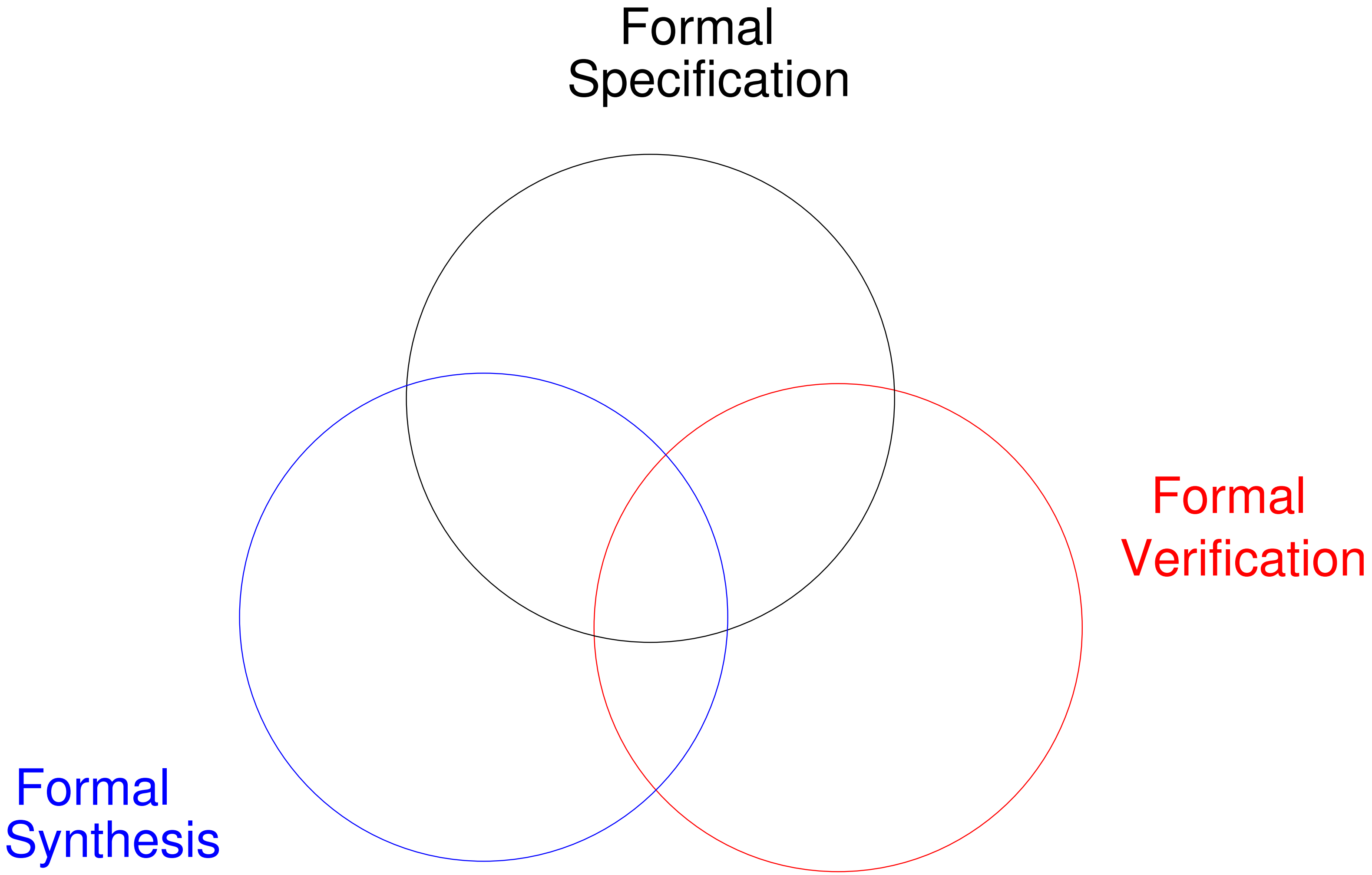
Austin, Texas, USA

Monday, January 24, 2011

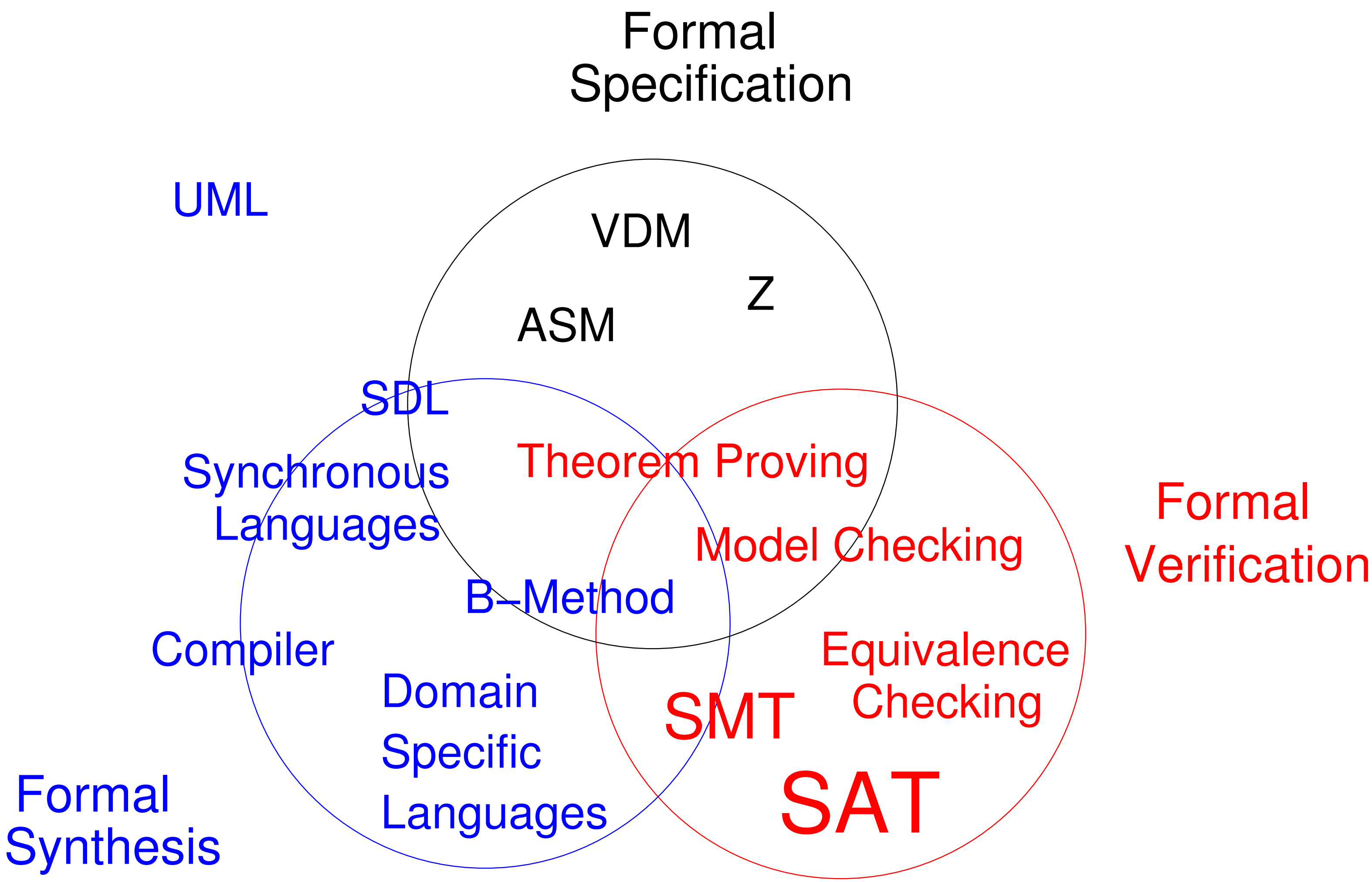


Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout
CPU Time (in seconds)
Number of problems solved
0
200
400
600
800
1000
1200
0
20
40
60
80
100
120
140
160
180
Limmat 02
Zchaff 02
Berkmin 561 02
Forklift 03
Siege 03
Zchaff 04
SatELite 05
Minisat 2.0 06
Picosat 07
Rsat 07
Minisat 2.1 08
Precosat 09
Glucose 09
Clasp 09
Cryptominisat 10
Lingeling 10
Minisat 2.2 10
[Le Berre'10]

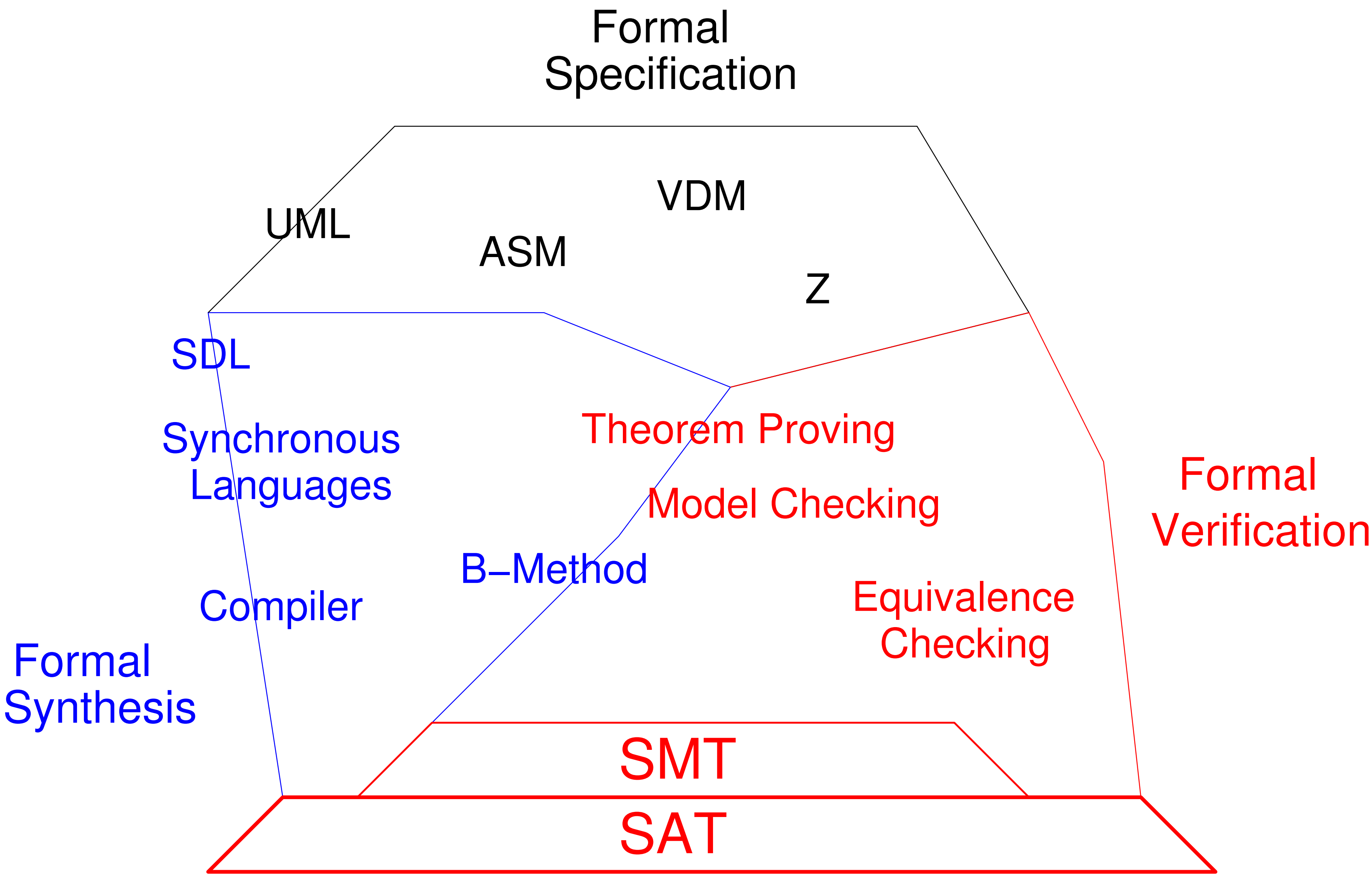
Formal Methods in Computer Science



Formal Methods and Satisfiability (SAT)



SAT as Core Technology



Dress Code of a Tutorial Speaker as Satisfiability Problem

- propositional logic:
  - variables **tie** **shirt**
  - negation $\neg$ (not)
  - disjunction $\vee$ disjunction (or)
  - conjunction $\wedge$ conjunction (and)
- three conditions / clauses:
  - clearly one should not wear a **tie** without a **shirt** $\neg\textbf{tie} \vee \textbf{shirt}$
  - not wearing a **tie** nor a **shirt** is impolite $\textbf{tie} \vee \textbf{shirt}$
  - wearing a **tie** and a **shirt** is overkill $\neg(\textbf{tie} \wedge \textbf{shirt}) \quad \equiv \quad \neg\textbf{tie} \vee \neg\textbf{shirt}$
- is the formula $(\neg\textbf{tie} \vee \textbf{shirt}) \wedge (\textbf{tie} \vee \textbf{shirt}) \wedge (\neg\textbf{tie} \vee \neg\textbf{shirt})$ satisfiable?

What is SAT?

- a class of rather low-level kind of problems:
  - propositional variables only, e.g. either hold (true) or not (false)
  - logic operators $\neg$, $\vee$, $\wedge$, actually restricted to conjunctive normal form (CNF)
  - but no quantifiers such as “for all such things”, or “there is one such thing”
  - can we find an assignment of the variables to true or false, such that a set of clauses is satisfied simultaneously
- theory: it is **the** standard NP complete problem [Cook’70]
- encoding: how to get your problem into CNF
- simplifying: how can the problem or the CNF be simplified (preprocessing)
- solving: how to implement fast solvers

Short SAT Solver History

- Davis and Putnam procedure
  - DP: elimination procedure [DavisPutnam'60]
  - DPLL: splitting [DavisLogemannLoveland'62]
- modern SAT solvers are mostly based on DPLL, actually $\boxed{\text{CDCL}}$

  CDCL = Conflict Driven Clause Learning
  - learning: GRASP [MarquesSilvaSakallah'96], RelSAT [BayardoSchrag'97]
  - watched literals, VSIDS: [mz]Chaff [MoskewiczMadiganZhaoZhangMalik-DAC'01]
  - improved heuristics: MiniSAT [EénSörensson-SAT'03] actually version from 2005
- preprocessing is still a hot topic:
  - most practical solvers use SatELite style preprocessing [EénBiere'05] DP
  - *inprocessing* in fastest available solvers PrecoSAT, Lingeling, CryptoMiniSAT, ...

What is Satisfiability Modulo Theory (SMT)?

- satisfiability solving for first order formulae
  - extension of SAT but interpreted over fixed theories
  - originally without quantifiers *but* quantifiers are important
  - fully automatic decision procedures which also can provide models
- theories of interest
  - equality, uninterpreted functions
  - real / integer arithmetic
  - bit-vectors, arrays
- particularly important are bit-vectors and arrays for HW/SW verification
  - our SMT solver Boolector ranked #1 in this category (SMT 2008 + 2009)

Applications of SAT and SMT

- $\boxed{\textit{bounded model checking}}$ in electronic design automation (EDA)
  - routinely used for falsification in all major design houses
  - unbounded extensions also use SAT, e.g. *sequential* equivalence checking
- SAT as working horse in *static software verification*
- static device driver verification at Microsoft (SLAM, SDV)
  - predicate abstraction with SMT solvers
  - spurious counter example checking
- *software configuration*, e.g. Eclipse IDE ships with SAT4J MaxSAT
- cryptanalysis and other *combinatorial problems* (bio-informatics)

What are Quantified Boolean Formulas (QBF)?

- QBF can be seen as extension to SAT:
  - existentially quantified variables as in SAT
  - but some variables can be universally quantified
- QBF is the *the classical* PSPACE complete problem
  - as SAT is *the* NP-complete problem
  - two other important PSPACE complete problems:
    - ∗ (Propositional) Linear Temporal Logic (LTL) satisfiability
    - ∗ symbolic model checking / symbolic reachability

$$\underbrace{\forall \mathbf{tie}[\exists \mathbf{shirt}[\overbrace{(\mathbf{tie} \vee \mathbf{shirt}) \wedge (\neg \mathbf{tie} \vee \neg \mathbf{shirt})}^{\mathbf{tie} \neq \mathbf{shirt}}]]}_{\text{satisfiable}} \quad \not\equiv \quad \underbrace{\exists \mathbf{tie}[\forall \mathbf{shirt}[\overbrace{(\mathbf{tie} \vee \mathbf{shirt}) \wedge (\neg \mathbf{tie} \vee \neg \mathbf{shirt})}^{\mathbf{tie} \neq \mathbf{shirt}}]]}_{\text{unsatisfiable}}$$

QBF Semantics and State-of-the-Art

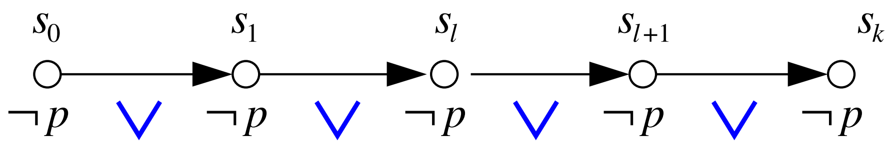
- semantics given as expansion of quantifiers

$$\exists x[f] \;\equiv\; f[0/x] \vee f[1/x] \qquad\qquad \forall x[f] \;\equiv\; f[0/x] \wedge f[1/x]$$

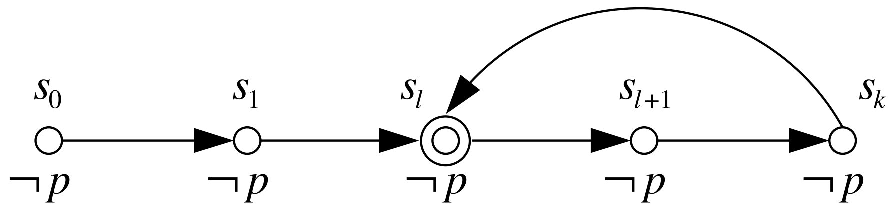
- expansion as translation from SAT to QBF is exponential
  - SAT problems have only existential quantifiers
  - expansion of universal quantifier can double formula size
- large number of different approaches to solve QBF   versus “mono-culture” in SAT
  - scalability for practically interesting problem still an issue
  - nevertheless first real applications appear, e.g. black-box equivalence checking
  - steady progress: currently fastest solvers DepQBF and Qube

[BiereCimattiClarkeZhu-TACAS'99]

- look only for counter example made of k states   "k" = the bound



or



- simple for *safety properties* p is invariantly true   (e.g. $p = \neg B$)

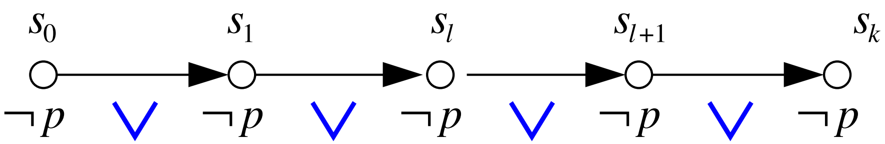
$$I(s_0) \wedge T(s_0, s_1)) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

- harder for *liveness properties* p is eventually true

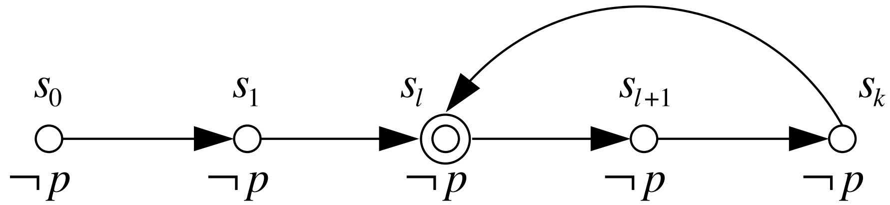
$$I(s_0) \wedge T(s_0, s_1)) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigwedge_{i=0}^{k} \neg p(s_i) \wedge \exists l \; T(s_k, s_l)$$

[BiereCimattiClarkeZhu-TACAS'99]

- look only for counter example made of k states   "k" = the bound



or

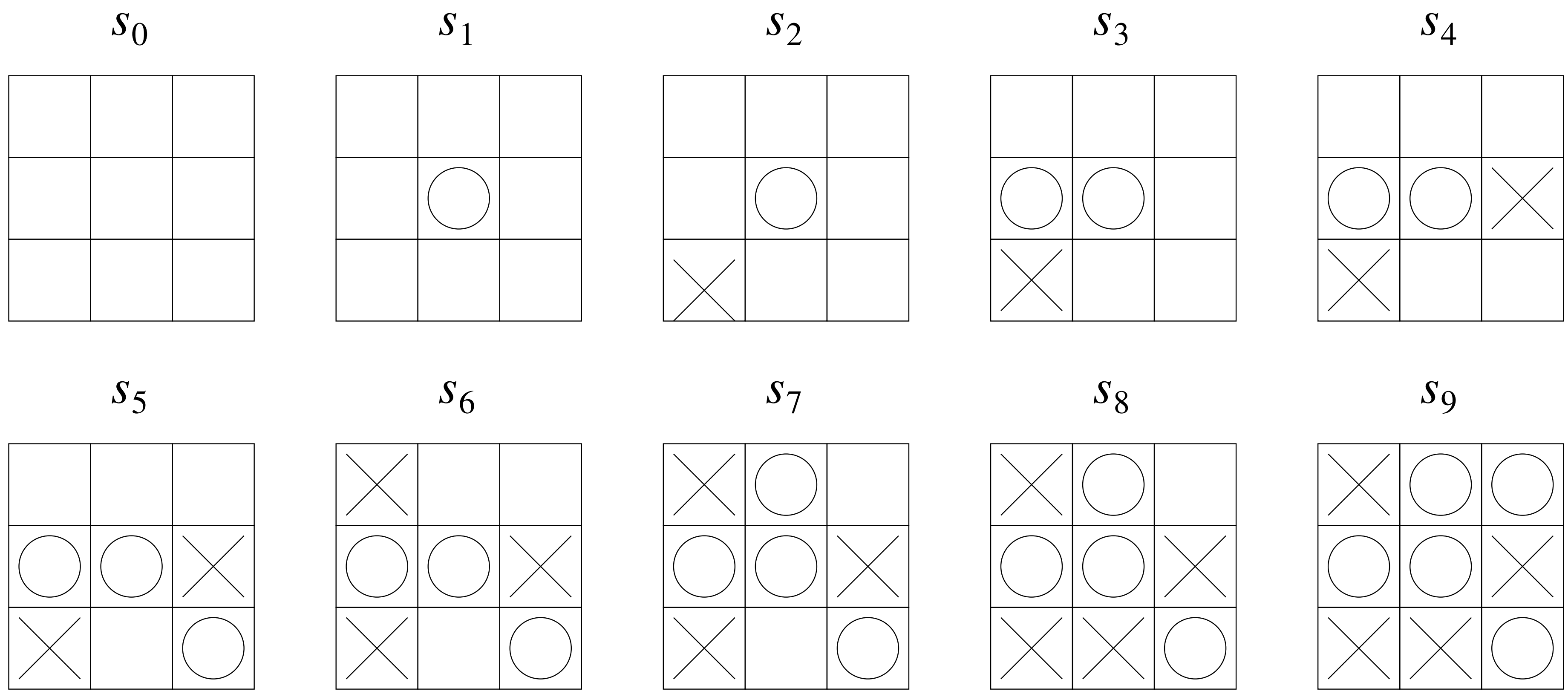


- simple for *safety properties* p is invariantly true   (e.g. $p = \neg B$)

$$I(s_0) \wedge T(s_0, s_1)) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

- harder for *liveness properties* p is eventually true

$$I(s_0) \wedge T(s_0, s_1)) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigwedge_{i=0}^{k} \neg p(s_i) \wedge \bigvee_{l=0}^{k} T(s_k, s_l)$$


s_0
s_1
s_2
s_3
s_4
s_5
s_6
s_7
s_8
s_9

No Winning Strategy for Tic-Tac-Toe

$$
\begin{array}{l}
\not\models \quad \forall s_0[empty(s_0) \rightarrow \\
\quad \exists x_1[circle(s_0,x_1,s_1) \wedge \\
\quad\quad \forall y_2[cross(s_1,y_2,s_2) \rightarrow \\
\quad\quad\quad \exists x_3[circle(s_2,x_3,s_3) \wedge \\
\quad\quad\quad\quad \forall y_4[cross(s_3,y_4,s_4) \rightarrow \\
\quad\quad\quad\quad\quad \exists x_5[circle(s_4,x_5,s_5) \wedge \\
\quad\quad\quad\quad\quad\quad \forall y_6[cross(s_5,y_6,s_6) \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad \exists x_7[circle(s_6,x_7,s_7) \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad \forall y_8[cross(s_7,y_8,s_8) \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists x_9[circle(s_8,x_9,s_9) \wedge win_{circle}(s_9)]]]]]]]]]
\end{array}
$$

x_i, y_i plays (4 bits each)

s_i configurations

(9×3 bits each)

Example: Equivalence Checking If-then-else Chains

original code | optimized code

```
if(!a && !b) h();
else if(!a) g();
else f();
```

$\Downarrow$

```
if(!a) {
  if(!b) h();
  else g();
} else f();
```

$\Rightarrow$

```
if(a) f();
else {
  if(!b) h();
  else g(); }
```

$\Uparrow$

```
if(a) f();
else if(b) g();
else h();
```

How to check that these two versions are equivalent?

1. represent procedures as *independent* boolean variables

$$
\begin{array}{ll}
\mathit{original} := & \mathit{optimized} := \\
\quad \textbf{if } \neg a \wedge \neg b \textbf{ then } h & \quad \textbf{if } a \textbf{ then } f \\
\quad \textbf{else if } \neg a \textbf{ then } g & \quad \textbf{else if } b \textbf{ then } g \\
\quad \textbf{else } f & \quad \textbf{else } h
\end{array}
$$

2. compile if-then-else chains into boolean formulae

$$\text{compile}(\textbf{if } x \textbf{ then } y \textbf{ else } z) \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z)$$

3. check equivalence of boolean formulae

$$\boxed{\text{compile}(\mathit{original}) \quad \Leftrightarrow \quad \text{compile}(\mathit{optimized})}$$

$$\begin{aligned} \textit{original} \;&\equiv\; \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{if } \neg a \textbf{ then } g \textbf{ else } f \\ &\equiv\; (\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge \textbf{if } \neg a \textbf{ then } g \textbf{ else } f \\ &\equiv\; (\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \;\vee\; a \wedge f) \end{aligned}$$

$$\begin{aligned} \textit{optimized} \;&\equiv\; \textbf{if } a \textbf{ then } f \textbf{ else } \textbf{if } b \textbf{ then } g \textbf{ else } h \\ &\equiv\; a \wedge f \;\vee\; \neg a \wedge \textbf{if } b \textbf{ then } g \textbf{ else } h \\ &\equiv\; a \wedge f \;\vee\; \neg a \wedge (b \wedge g \;\vee\; \neg b \wedge h) \end{aligned}$$

$$(\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \;\vee\; a \wedge f) \quad\Leftrightarrow\quad a \wedge f \;\vee\; \neg a \wedge (b \wedge g \;\vee\; \neg b \wedge h)$$

How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

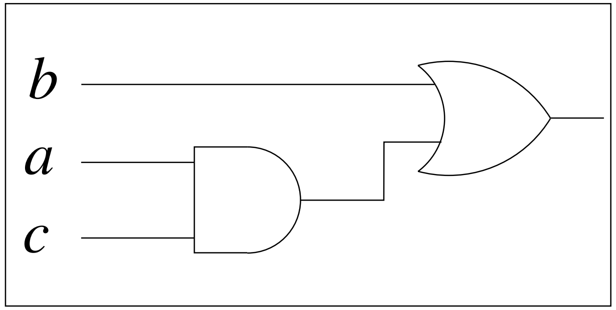
Is there an assignment to a, b, f, g, h,
which results in different evaluations of $\mathit{original}$ and $\mathit{optimized}$?

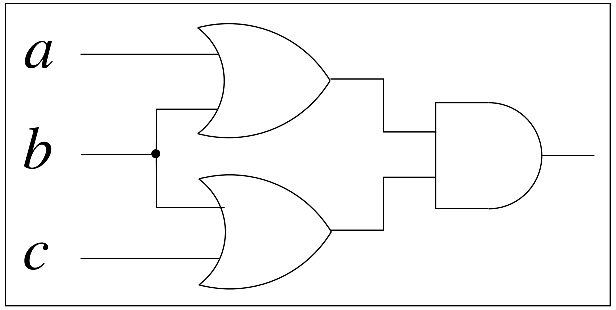
or equivalently:

Is the boolean formula $\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})$ satisfiable?

such an assignment would provide an easy to understand counterexample

SAT Example: Circuit Equivalence Checking





$b \vee a \wedge c$ $\quad\quad$ $(a \vee b) \wedge (b \vee c)$

equivalent?

$$b \vee a \wedge c \quad \Leftrightarrow \quad (a \vee b) \wedge (b \vee c)$$

Conjunctive Normal Form

Definition formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

each *clause* C is a disjunction of literals

$$C = L_1 \vee \ldots \vee L_m$$

and each *literal* is either a plain variable x or a negated variable $\overline{x}$.

Example $(a \vee b \vee c) \wedge (\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c})$

Note 1: two notions for negation: in $\overline{x}$ and $\neg$ as in $\neg x$ for denoting negation.

Note 2: original SAT problem is actually formulated for CNF

Note 3: solvers (mostly) expect CNF as input

DIMACS Format Example 1

- common ASCII file format of SAT solvers, used by SAT competitions
- variables are represented as natural numbers, literals as integers
- header "`p cnf <vars> <clauses>`", comment lines start with "`c`"

In order to show the *validity* of $\quad b \vee a \wedge c \quad \Leftarrow \quad (a \vee b) \wedge (b \vee c)$

negate, $\quad \overline{(b \vee a \wedge c)} \quad \wedge \quad (a \vee b) \wedge (b \vee c)$

simplify and show *unsatisfiability* of $\quad \neg b \wedge (\neg a \vee \neg c) \quad \wedge \quad (a \vee b) \wedge (b \vee c)$

```
c the first two lines are comments
c ex1.cnf: a=1, b=2, c=3
p cnf 3 4
-2 0
-1 -3 0
1 2 0
2 3 0
```

PicoSAT API for Constructing CNFs Example 1

```
// compile with: gcc -o ex1 ex1.c picosat.o
#include "picosat.h"
#include <stdio.h>
int main () {
  int res;
  picosat_init ();

  picosat_add (-2); picosat_add (0);
  picosat_add (-1); picosat_add (-3); picosat_add (0);
  picosat_add (1); picosat_add (2); picosat_add (0);
  picosat_add (2); picosat_add (3); picosat_add (0);

  res = picosat_sat (-1);

  if (res == 10) printf ("s SATISFIABLE\n");
  else if (res == 20) printf ("s UNSATISFIABLE\n");
  else printf ("s UNKNOWN\n");

  picosat_reset ();
  return res;
}
```

Satisfying Assignments Example 2

assume invalid equivalence resp. implication: $(a \vee b) \quad \Rightarrow \quad (a \text{ xor } b)$

its negation $(a \vee b) \quad \wedge \quad (a = b)$

as CNF $(a \vee b) \quad \wedge \quad (\neg a \vee b) \wedge (\neg b \vee a)$

```
c ex2.cnf: a=1,b=2
p cnf 2 3
1 2 0
-1 2 0
-2 1 0
```

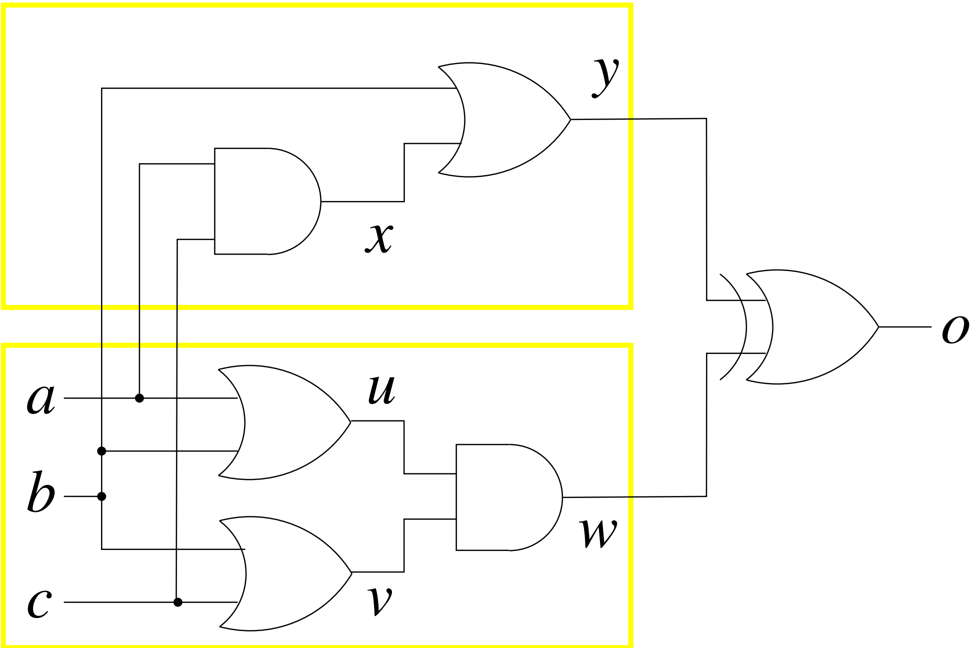
SAT solver then allows to extract one satisfying assignment:

```
$ picosat ex2.cnf
s SATISFIABLE
v 1 2 0
```

this is the *only one* since “assuming” the opposite values individually is UNSAT

```
$ picosat ex2.cnf -a -1; picosat ex2.cnf -a -2
s UNSATISFIABLE
s UNSATISFIABLE
```

CNF



$$
\begin{array}{r}
o \;\wedge \\
(x \leftrightarrow a \wedge c) \;\wedge \\
(y \leftrightarrow b \vee x) \;\wedge \\
(u \leftrightarrow a \vee b) \;\wedge \\
(v \leftrightarrow b \vee c) \;\wedge \\
(w \leftrightarrow u \wedge v) \;\wedge \\
(o \leftrightarrow y \oplus w)
\end{array}
$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Algorithmic Description of Tseitin Transformation

[Tseitin'68]

1. generate a new variable x_s for each non input circuit signal s

2. for each gate produce complete input / output constraints as clauses

3. collect all constraints in a big conjunction

the transformation is *satisfiability equivalent*:

the result is satisfiable iff and only the original formula is satisfiable

not equivalent to the original formula: it has new variables

just *project* satisfying assignment onto the original variables

Negation:
$$
\begin{aligned}
x \leftrightarrow \overline{y} \quad &\Leftrightarrow \quad (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee \overline{y}) \wedge (y \vee x)
\end{aligned}
$$

Disjunction:
$$
\begin{aligned}
x \leftrightarrow (y \vee z) \quad &\Leftrightarrow \quad (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\
&\Leftrightarrow \quad (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z)
\end{aligned}
$$

Conjunction:
$$
\begin{aligned}
x \leftrightarrow (y \wedge z) \quad &\Leftrightarrow \quad (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\
&\Leftrightarrow \quad (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x)
\end{aligned}
$$

Equivalence:
$$
\begin{aligned}
x \leftrightarrow (y \leftrightarrow z) \quad &\Leftrightarrow \quad (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow \quad (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow \quad (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\
&\Leftrightarrow \quad (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x)
\end{aligned}
$$

Optimizations for Tseitin Transformation

- goal is smaller CNF less variables, less clauses, so easier to solve (?!)

- extract multi argument operands to remove variables for intermediate nodes

- half of AND, OR node constraints/clauses can be removed for *unnegated* nodes [PlaistedGreenbaum'86]

  - node occurs negated if it has an ancestor which is a negation

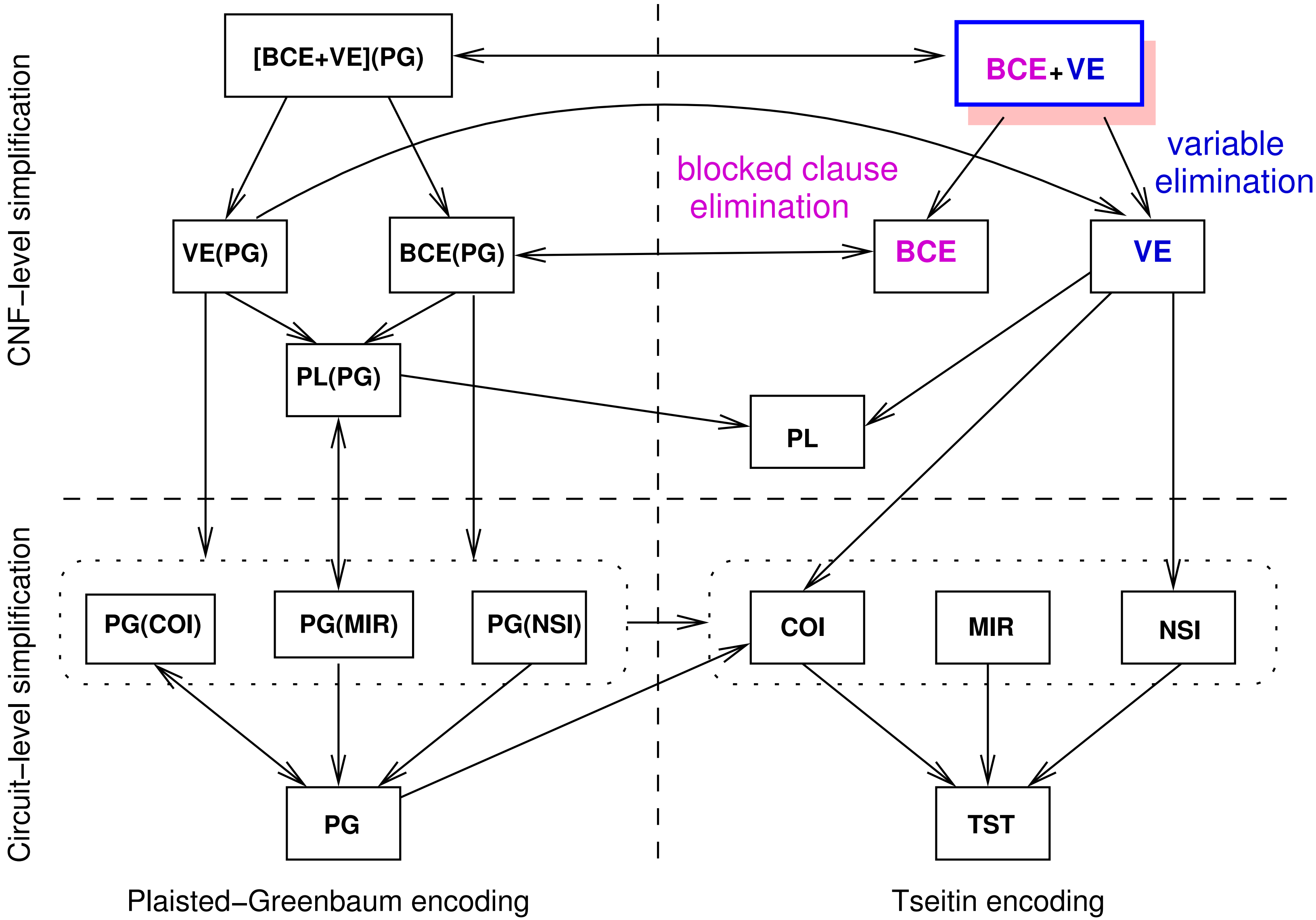
  - half of the constraints determine parent assignment from child assignment

  - those are unnecessary if node is not used negated

  - those have to be carefully applied to DAG structure

- further structural circuit optimizations ...

CNF Blocked Clause Elimination simulates many encoding / circuit optimizations



[JärvisaloBiereHeule-TACAS'10]

Intermediate Representations

- encoding directly into CNF is hard, so we use intermediate levels:

  1. application level
  2. bit-precise semantics world-level operations: bit-vector theory
  3. bit-level representations such as AIGs or vectors of AIGs
  4. CNF

- encoding application level formulas into word-level: as generating machine code

- word-level to bit-level: bit-blasting similar to hardware synthesis

- encoding “logical” constraints is another story

Bit-Blasting of 4-Bit Addition

addition of 4-bit numbers x, y with result s also 4-bit: $s = x + y$

$$[s_3, s_2, s_1, s_0]_4 \;=\; [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

$$\begin{aligned}
[s_3, \cdot\,]_2 &= \mathsf{FullAdder}(x_3, y_3, c_2) \\
[s_2, c_2]_2 &= \mathsf{FullAdder}(x_2, y_2, c_1) \\
[s_1, c_1]_2 &= \mathsf{FullAdder}(x_1, y_1, c_0) \\
[s_0, c_0]_2 &= \mathsf{FullAdder}(x_0, y_0, \mathit{false})
\end{aligned}$$

where

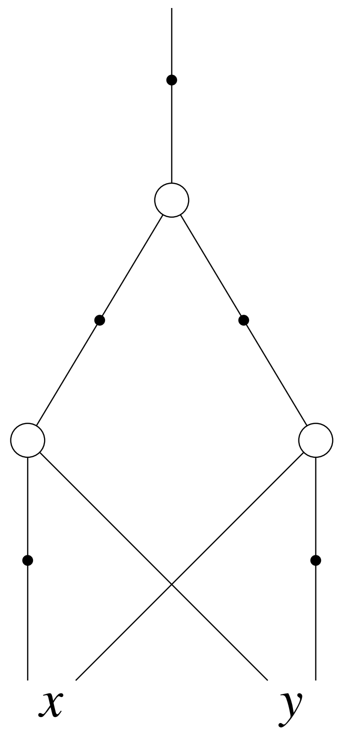
$$\begin{aligned}
[\, s \,, o \,]_2 &= \mathsf{FullAdder}(x, y, i) \quad \text{with} \\
s &= x \text{ xor } y \text{ xor } i \\
o &= (x \wedge y) \vee (x \wedge i) \vee (y \wedge i) \;=\; ((x + y + i) \geq 2)
\end{aligned}$$

And-Inverter-Graphs (AIG)

- widely adopted bit-level intermediate representation
  - see for instance our AIGER format   http://fmv.jku.at/aiger
  - used in Hardware Model Checking Competition (HWMCC)
  - also used in the *structural track* in SAT competitions
  - many companies use similar techniques
- basic logical operators: *conjunction* and *negation*
- DAGs: nodes are conjunctions,   negation/sign as *edge attribute*

  bit stuffing: signs are compactly stored as LSB in pointer
- automatic sharing of isomorphic graphs, constant time (peep hole) simplifications
- *or even*   SAT sweeping, full reduction, etc . . .   see ABC system from Berkeley

XOR as AIG



negation/sign are edge attributes

not part of node

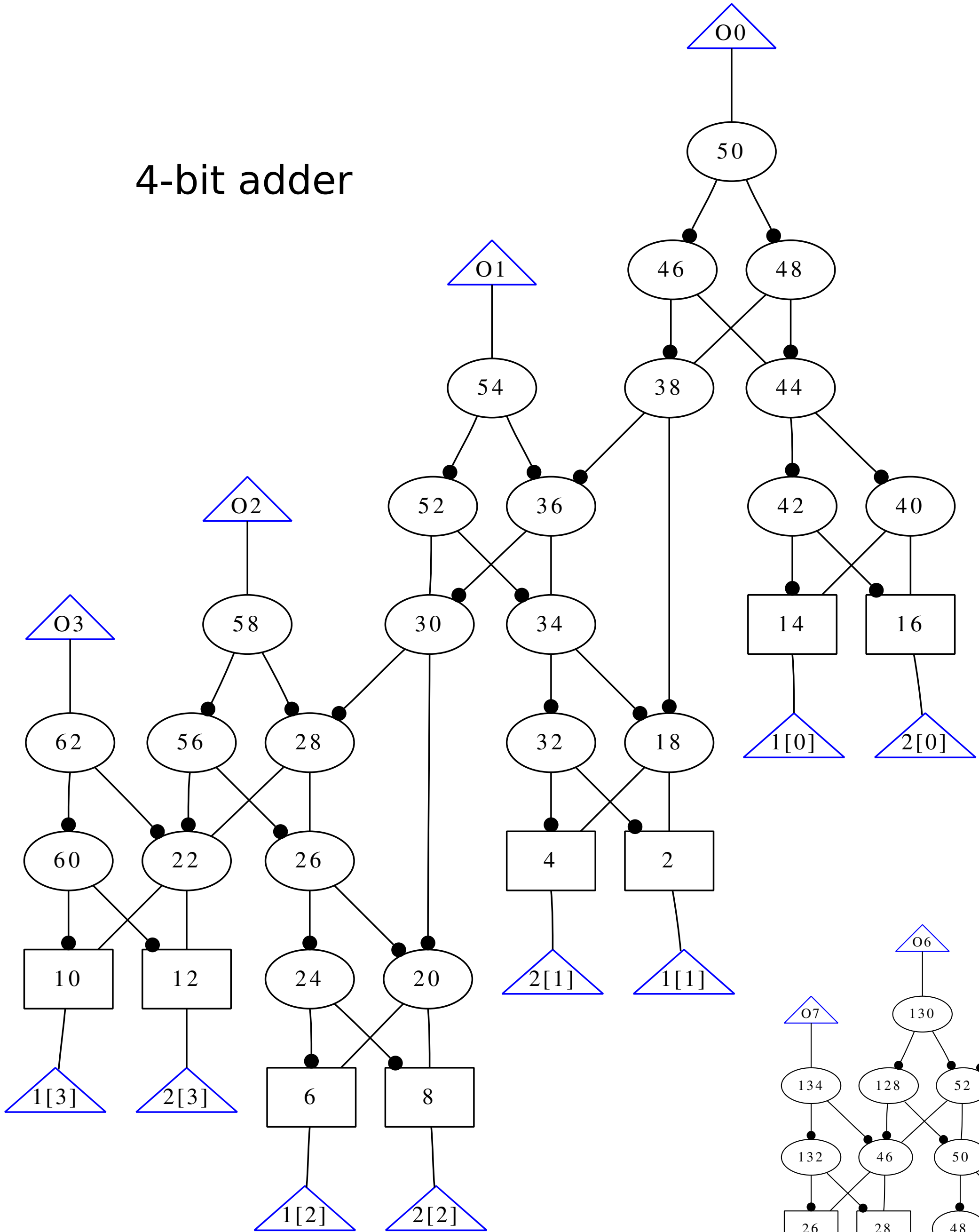
$$x \text{ xor } y \;\equiv\; (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \;\equiv\; \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

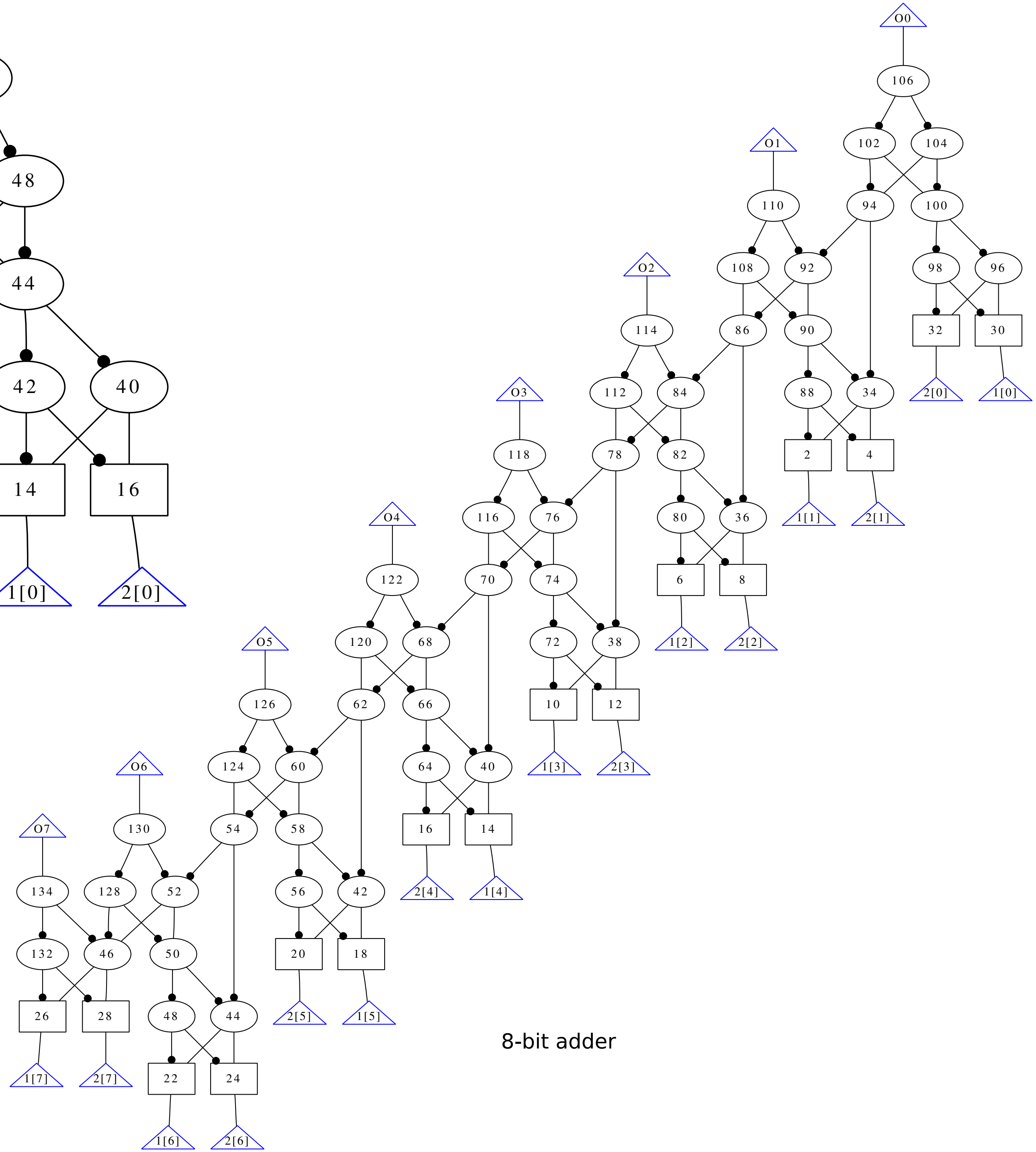
```c
typedef struct AIG AIG;

struct AIG
{
  enum Tag tag;                    /* AND, VAR */
  void *data[2];
  int mark, level;                 /* traversal */
  AIG *next;                       /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)
```

assumption for correctness:

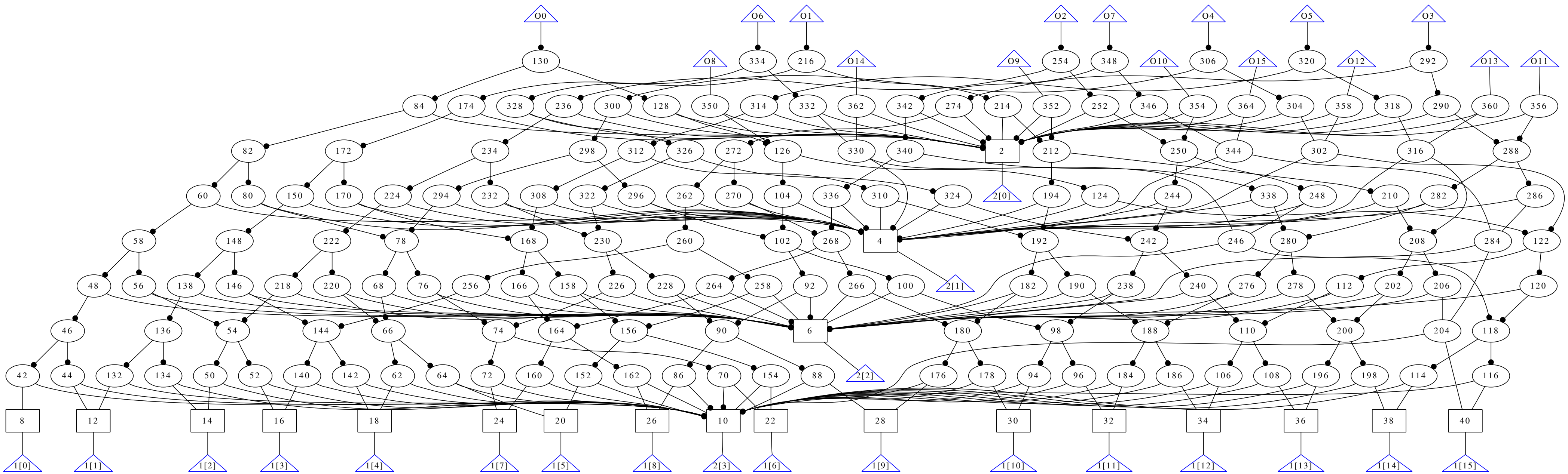
`sizeof(unsigned) == sizeof(void*)`

4-bit adder

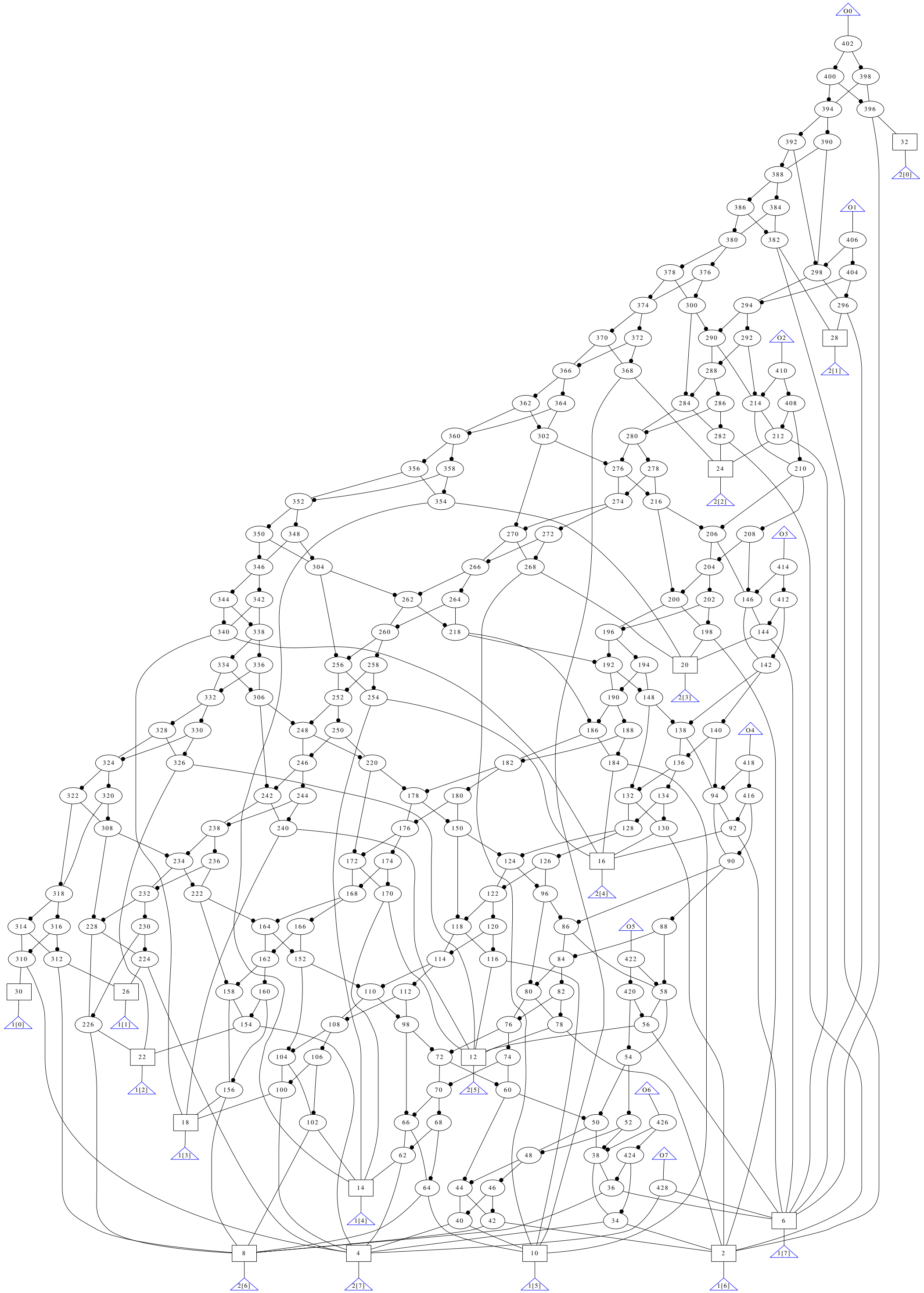




8-bit adder



bit-vector of length 16 shifted by bit-vector of length 4



O0
O1
O2
O3
O4
O5
O6
O7
2[0]
2[1]
2[2]
2[3]
2[4]
2[5]
2[6]
2[7]
1[0]
1[1]
1[2]
1[3]
1[4]
1[5]
1[6]
1[7]

Encoding Logical Constraints

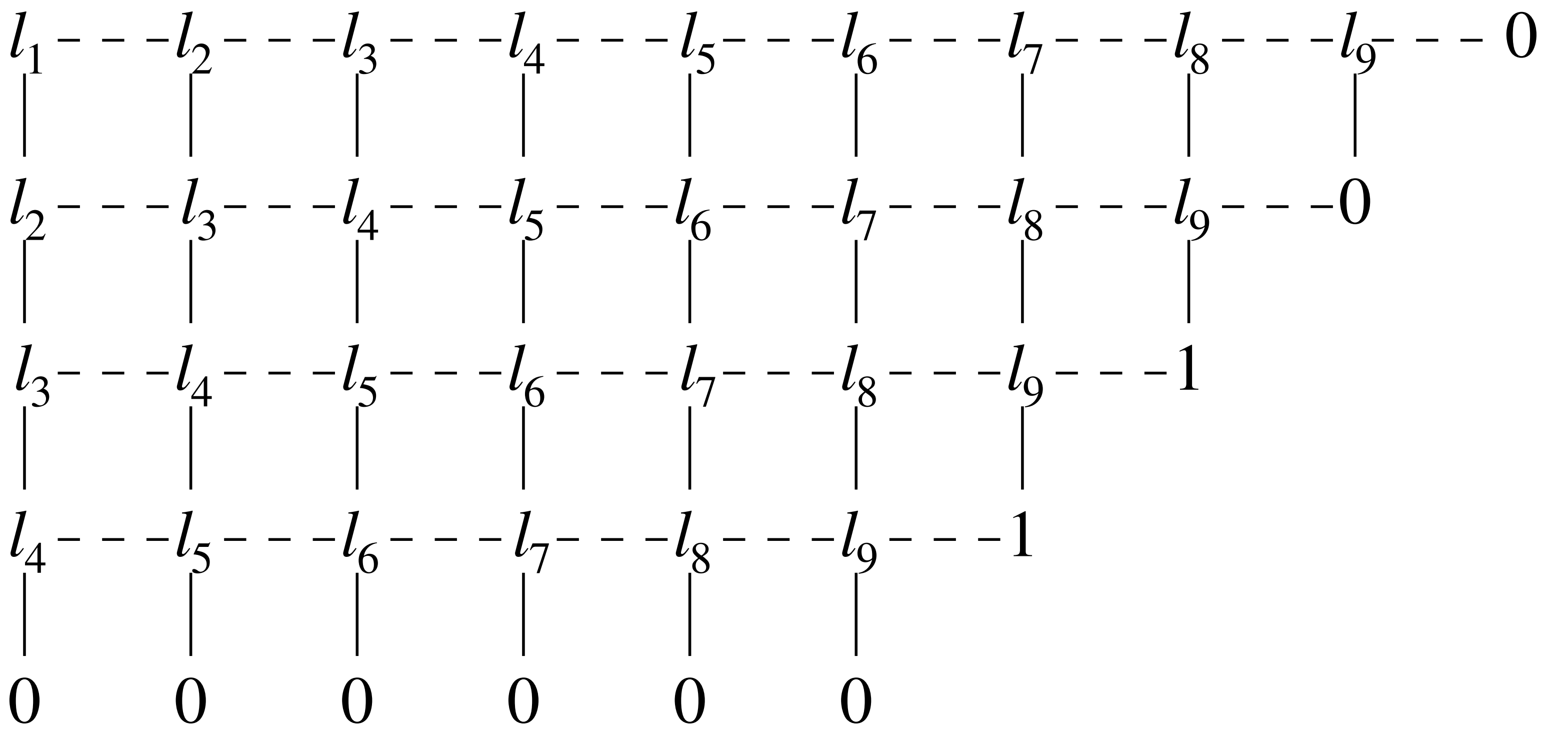
- Tseitin's construction suitable for most kinds of "model constraints"
  - assuming simple operational semantics: encode an interpreter
  - small domains: *one-hot encoding* large domains: *binary encoding*
- harder to encode *properties* or additional *constraints*
  - temporal logic / fix-points
  - environment constraints
- example for fix-points / recursive equations: $x = (a \vee y)$, $y = (b \vee x)$
  - has unique *least* fix-point $x = y = (a \vee b)$
  - and unique *largest* fix-point $x = y = \mathit{true}$ but unfortunately
  - only largest fix-point can be (directly) encoded in SAT otherwise need ASP

Example of Logical Constraints: Cardinality Constraints

- given a set of literals $\{l_1, \ldots l_n\}$
  - constraint the *number* of literals assigned to *true*
  - $|\{l_1, \ldots, l_n\}| \geq k$ or $|\{l_1, \ldots, l_n\}| \leq k$ or $|\{l_1, \ldots, l_n\}| = k$
- multiple encodings of cardinality constraints
  - naïve encoding exponential: *at-most-two* quadratic, *at-most-three* cubic, etc.
  - quadratic $O(k \cdot n)$ encoding goes back to Shannon
  - linear $O(n)$ parallel counter encoding [Sinz'05]
  - for an $O(n \cdot \log n)$ encoding see Prestwich's chapter in our Handbook of SAT
- generalization *Pseudo-Boolean* constraints (PB), e.g. $2 \cdot \overline{a} + \overline{b} + c + \overline{d} + 2 \cdot e \geq 3$

  actually used to handle MaxSAT in SAT4J for configuration in Eclipse

$$2 \leq |\{l_1, \ldots, l_9\}| \leq 3$$



"then" edge downward, "else" edge to the right

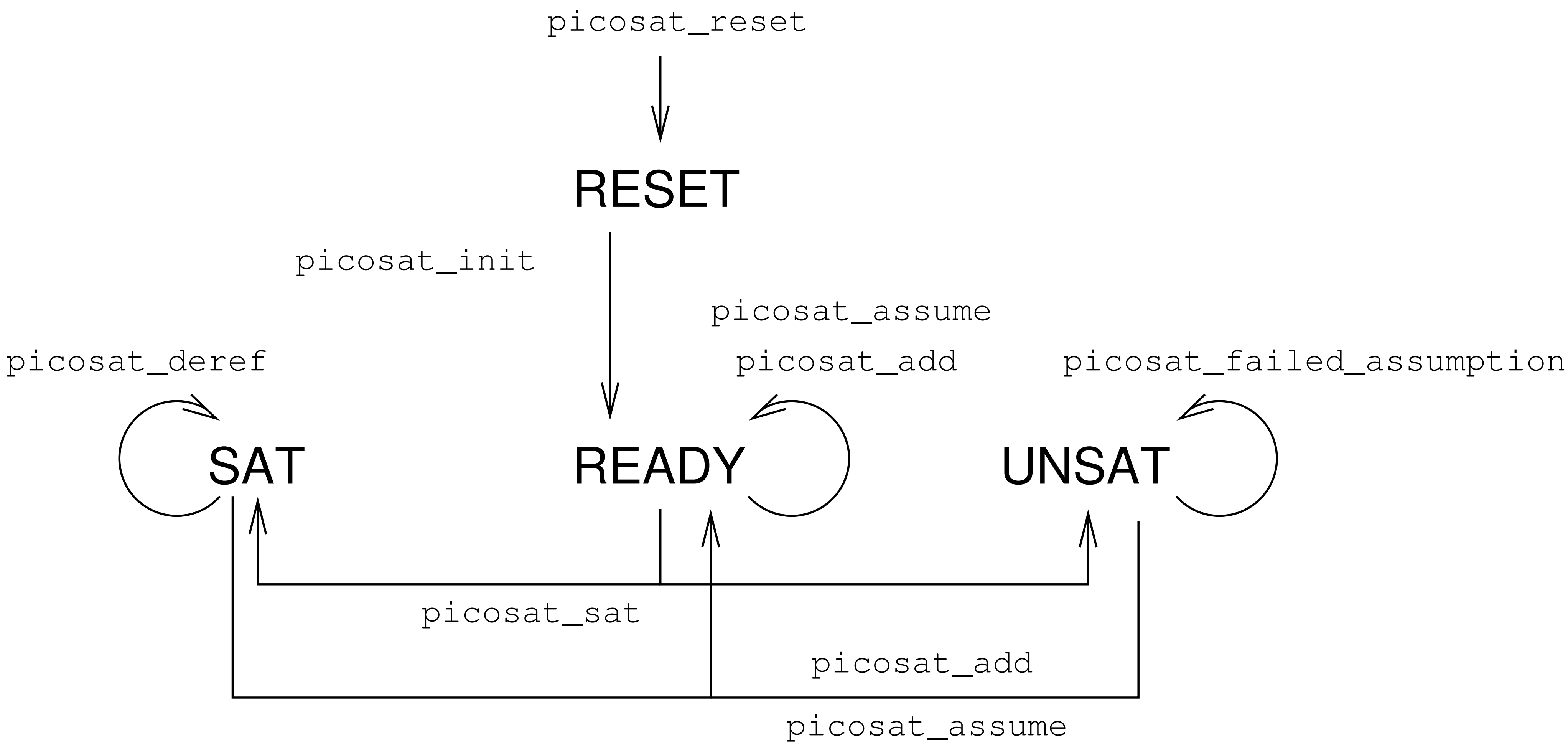
Example 2 with PicoSAT API

```
// compile with: gcc -o ex2 ex2.c picosat.o
#include "picosat.h"
#include <stdio.h>
#include <assert.h>
int main () {
  int res, a, b;
  picosat_init ();
  picosat_add (1); picosat_add (2); picosat_add (0);
  picosat_add (-1); picosat_add (2); picosat_add (0);
  picosat_add (-2); picosat_add (1); picosat_add (0);
  assert (picosat_sat (-1) == 10); // SATISFIABLE
  a = picosat_deref (1); b = picosat_deref (2);
  printf ("v %d %d\n", a*1, b*2);
  picosat_assume (-a*1); assert (picosat_sat (-1) == 20);//UNSAT
  picosat_assume (-b*2); assert (picosat_sat (-1) == 20);//UNSAT
  return res;
}
```

Adding a Blocking Clause to Block Current Solution

```
static void block_current_solution (void) {
  int max_idx = picosat_variables (), i;

  // since 'picosat_add' resets solutions
  // need to store it first:
  signed char * sol = malloc (max_idx + 1);
  memset (sol, 0, max_idx + 1);

  for (i = 1; i <= max_idx; i++)
    sol[i] = (picosat_deref (i) > 0) ? 1 : -1;

  for (i = 1; i <= max_idx; i++)
    picosat_add ((sol[i] < 0) ? i : -i);
  picosat_add (0);

  free (sol);
}
```

Simplified PicoSAT API



picosat_set...   picosat_inconsistent   picosat_deref_toplevel

Failed Assumptions

- two ways to implement incremental SAT solvers
  - push / pop as in SMT solvers   partial support in SATIRE, zChaff, PicoSAT
    - ∗ clauses associated with context and pushed / popped in a stack like manner
    - ∗ pop discards clauses of current context
  - most common: assumptions   [ClaessenSörensson'03]   [EénSörensson'03]
    - ∗ allows to use set of literals as assumptions
    - ∗ force SAT solver to first pick assumption as decisions
    - ∗ more flexible, since assumptions can be reused
    - ∗ assumptions are only valid for the next SAT call
- failed assumptions: sub set of assumptions inconsistent with CNF

Example: Bit-Vector Under-Approximation

- goal: reduce size of bit-vector constants in satisfying assignments

- refinement approach: for each bit-vector variable only use an "effective width"

  - example: 4-bit vector $[x_3, x_2, x_1, x_0]$ and effective width 2 use $[x_1, x_1, x_1, x_0]$

  - either encode from scratch with x_3 and x_2 replaced by x_1 (1)

  - or add $x_3 = x_1$ and $x_2 = x_1$ after push (2)

  - or add $a_x^2 \rightarrow x_3 = x_1$ and $a_x^2 \rightarrow x_2 = x_1$ and assume fresh literal a_x^2 (3)

- if satisfiable then a solution with small constants has been found

  otherwise increase eff. width of bit-vectors where it was used to derive UNSAT

  under-approximations not used then formula UNSAT "used" = "failed assumption"

- in (3) constraints are removed by forcing assumptions to the opposite value
  by adding a unit clause, e.g. $\neg a_x^2$ in next iteration

Clausal Cores

- clausal core (or unsatisfiable sub set) of an unsatisfiable formula
  - clauses used to derive the empty clause
  - may include not only original but also learned clauses
  - similar application as in previous under-approximation example
  - but also useful for diagnosis of inconsistencies
- variable core
  - sub set of variables occurring in clauses of a clausal core
- these cores are not unique and not necessary minimimal
- minimimal unsatisfiable sub set (MUS) = clausal core where no clause can be removed

PicoMUS

- PicoMUS is a MUS extractor based on PicoSAT
  - uses several rounds of clausal core extraction for preprocessing
  - then switches to assumption based core minimization using

    `picosat_failed_assumptions`
  - source code serves as a good example on how to use cores / assumptions

- new MUS track in this year's SAT 2011 competition
  - with high- and low-level MUS sub tracks

Examples for Core and MUS Extraction


```
c ex3.cnf
p cnf 6 10
1 2 3 0
1 2 -3 0
1 -2 3 0
1 -2 -3 0
4 5 6 0
4 5 -6 0
4 -5 6 0
4 -5 -6 0
-1 -4 0
1 4 0
```

```
$ picosat ex3.cnf -c core
s UNSATISFIABLE
$ cat core
p cnf 6 9
2 3 1 0
2 -3 1 0
-2 3 1 0
-2 -3 1 0
6 5 4 0
5 -6 4 0
6 4 -5 0
4 -6 -5 0
-1 -4 0
```

```
c ex4.cnf
p cnf 6 11
1 2 3 0
1 2 -3 0
1 -2 3 0
1 -2 -3 0
4 5 6 0
4 5 -6 0
4 -5 6 0
4 -5 -6 0
-1 -4 0
-1 4 0
-1 -4 0
```

```
$ picomus ex4.cnf mus
s UNSATISFIABLE
$ cat mus
p cnf 6 6
1 2 3 0
1 2 -3 0
1 -2 3 0
1 -2 -3 0
-1 4 0
-1 -4 0
```

Proof-Traces and TraceCheck

- core extraction in PicoSAT is based on tracing proofs
  - enabled by `picosat_enable_trace_generation`
  - maintains “dependency graph” of learned clauses
  - kept in memory, so fast core generation
- traces can also written to disk in various formats
  - RUP format by Allen Van Gelder (SAT competition)
  - or format of TraceCheck tool
- TraceCheck can check traces for correctness
  - orders clauses and antecedents to generate and check resolution proof
  - (binary) resolution proofs can be dumped

QDIMACS Example

same as DIMACS except that we have additional quantifiers:

```
c SAT
p cnf 3 4
a 1 0
e 2 3 0
-1 -2 3 0
-1 2 -3 0
1 2 3 0
1 -2 -3 0
```

```
c UNSAT
p cnf 4 8
a 1 2 0
e 3 4 0
-1 -3 4 0
-1 3 -4 0
1 3 4 0
1 -3 -4 0
-2 -3 4 0
-2 3 -4 0
2 3 4 0
2 -3 -4 0
```

DepQBF API

```
/* Create and initialize solver instance. */
QDPLL *qdpll_create (void);

/* Delete and release all memory of solver instance. */
void qdpll_delete (QDPLL * qdpll);

/* Ensure var table size to be at least 'num'. */
void qdpll_adjust_vars (QDPLL * qdpll, VarID num);

/* Open a new scope, where variables can be added by 'qdpll_add'.
   Returns nesting of new scope.
   Opened scope can be closed by adding '0' via 'qdpll_add'.
   NOTE: will fail if there is an opened scope already.
*/
unsigned int qdpll_new_scope (QDPLL * qdpll, QDPLLQuantifierType qtype);

/* Add variables or literals to clause or opened scope.
   If scope is opened, then 'id' is interpreted as a variable ID,
   otherwise 'id' is interpreted as a literal.
   NOTE: will fail if a scope is opened and 'id' is negative.
*/
void qdpll_add (QDPLL * qdpll, LitID id);

/* Decide formula. */
QDPLLResult qdpll_sat (QDPLL * qdpll);
```

Wish List for QBF Solvers

- extraction of "certificates"
  - satisfying assignment to outer-most existential variables
  - resp. in general skolem-functions for satisfiable instances
  - falsifying assignment to outer-most universal variables
- incremental QBF solving
- API for preprocessing / inprocessing
- beside steady progress more scalability

Summary

- SAT and QBF
- conjunctive normal form (CNF) and (Q)DIMACS format
- encoding of models and logical constraints
- PicoSAT, PicoMUS, TraceCheck, DepQBF
- examples and use cases
- APIs

What is missing?

- how SAT can handle millions of variables routinely
- whether there will still be progress in SAT
- AIGER format, proof (trace) formats
- the full API of PicoSAT (see `picosat.h` for more details)
- skipped API of other Solvers, in particular (Crypto)MiniSAT, SAT4J
- tricky issues with incremental pre/in-processing such as “freezing variables”
- compact QBF encodings [JussilaBiere’06]

ed topics have attracted researchers from various disciplines. Logic, nning, scheduling, operations research and combinatorial optimization, s on the theme of complexity, and much more, they all are connected

AT stems from actual solving: The increase in power of modern SAT years has been phenomenal. It has become the key enabling technology of both computer hardware and software. Bounded Model Checking ware is now probably the most widely used model checking technique. t it finds are just satisfying instances of a Boolean formula obtained by depth a sequential circuit and its specification in linear temporal logic. to software verification is a much more difficult problem on the frontier promising approach for languages like C with finite word-length integers s in BMC but with a decision procedure for the theory of bit-vectors n procedures for bit-vectors that I am familiar with ultimately make use ndle complex formulas.

more complicated theories, like linear real and integer arithmetic, are ification. Most of them use powerful SAT solvers in an essential way.

ving is a key technology for 21st century computer science. I expect on all theoretical and practical aspects of SAT solving will be extremely nd researchers and will lead to many further advances in the field.

Edmund Clarke

ystems University Professor of Computer Science and Professor of Electrical at Carnegie Mellon University, is one of the initiators and main contributors ing, for which he also received the 2007 ACM Turing Award.

larke was one of the first researchers to realize that SAT solving has the the most important technologies in model checking.

ISBN 978-1-58603-929-5

185

HANDBOOK

of satisfiability

Editors:
Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

Frontiers in Artificial Intelligence a

HANDBOO

f satisfia

Editors:
Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

Blocked Clauses

[Kullman'99]

blocked clause $C \in F$

all clauses in F with $\bar{l}$

fix a CNF F

$(a \vee b \vee l)$

$(\bar{l} \vee \bar{a} \vee c)$

$(\bar{l} \vee \bar{b} \vee d)$

since all resolvents of C on l are tautological C can be removed

Proof

assignment σ satisfying $F \backslash C$ but not C

can be extended to a satisfying assignment of F by flipping value of l

[JärvisaloBiereHeule-TACAS'10]

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

NSI Non-Shared Inputs reduction

PG polarity based encoding [PlaistedGreenbaum'86]

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination

Boolector: Lemmas-on-Demand + Underapproximation

[BrummayerBiere'09]