Translating into SAT

Armin Biere
Johannes Kepler Universität Linz

SAT'16 Industrial Day

Université de Bordeaux
Bordeaux, France

Saturday, 9th July, 2016

optimization of if-then-else chains

original C code

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

$\Downarrow$

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

$\Rightarrow$

optimized C code

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

$\Uparrow$

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

How to check that these two versions are equivalent?

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

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

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

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 original and optimized?

or equivalently:

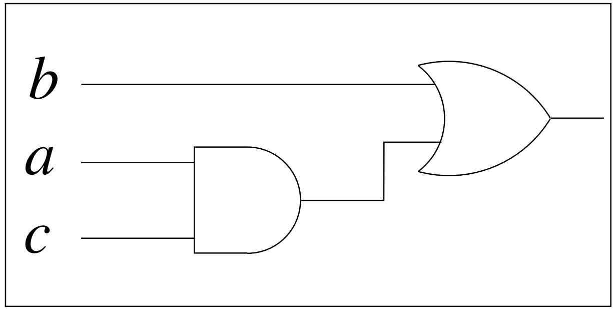
Is the boolean formula $\text{compile}(\mathit{original}) \not\leftrightarrow \text{compile}(\mathit{optimized})$ satisfiable?

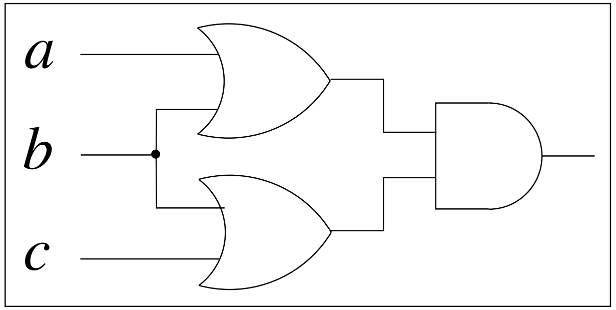
such an assignment would provide an easy to understand counterexample

Note: by concentrating on counterexamples we moved from Co-NP to NP

Note: this is mostly of theoretical interest but in practice there might be big differences if you have many problems and average expected result is only one (SAT or UNSAT)

SAT Example: Circuit Equivalence Checking





$b \vee a \wedge c$

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

equivalent?

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

SAT (Satisfiability) the classical NP complete Problem:

Given a propositional formula f over n propositional variables $V = \{x, y, \ldots\}$.

Is there an assignment $\sigma : V \rightarrow \{0, 1\}$ with $\sigma(f) = 1$?

SAT belongs to NP

There is a non-deterministic Touring-machine deciding SAT in polynomial time:

guess the assignment σ (linear in n), calculate $\sigma(f)$ (linear in $|f|$)

Note: on a real (deterministic) computer this would still require 2^n time

SAT is complete for NP (see complexity / theory class)

Implications for us:
general SAT algorithms are probably exponential in time (unless NP = P)

Definition

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

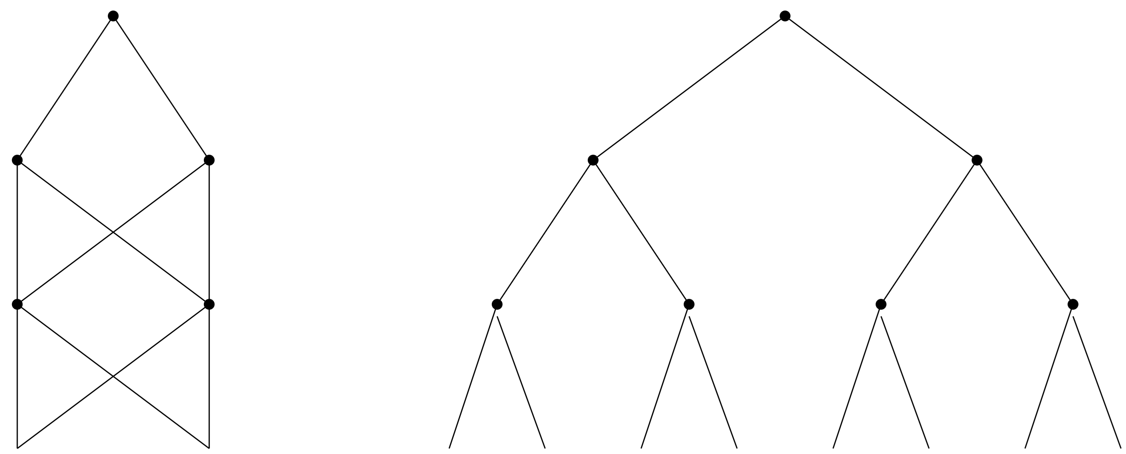
Note 1: two notions for negation: in $\overline{x}$ and $\neg$ as in $\neg x$ for denoting negation.

Note 2: the original SAT problem is actually formulated for CNF

Note 3: SAT solvers mostly also expect CNF as input

- NNF: $\neg$ in front of variables only, arbitrary nested $\wedge$ and $\vee$
- might need to expand non-monotonic operators into $\wedge$ and $\vee$
  - $(a \leftrightarrow b) \equiv (\neg a \wedge \neg b) \vee (a \wedge b)$
  - requires to work with circuit/DAG to avoid exponential explosion
- apply De'Morgan rule to push negations down
  - $\neg(a \wedge b) \equiv \neg a \vee \neg b \qquad \neg(a \vee b) \equiv \neg a \wedge \neg b$
- bottom-up CNF translation
  - $(\bigwedge_i C_i) \wedge (\bigwedge_j D_j)$ is already a CNF
  - $(\bigwedge_i C_i) \vee (\bigwedge_j D_j) \equiv \bigwedge_{i,j}(C_i \vee D_j)$ "clause distribution" (quadratic)
- whole procedure exponential in $\vee/\wedge$ alternation depth
- but might produce compact CNFs for small formulas
  - $(\neg a \wedge \neg b) \vee (a \wedge b) \equiv \cancel{(\neg a \vee a)} \wedge (\neg a \vee b) \wedge (\neg b \vee a) \wedge \cancel{(\neg b \vee b)}$
- NNF to CNF encoding interesting concept but (not really) used in practice

Why are Sharing / Circuits / DAGs important?



DAG may be exponentially more succinct than expanded Tree

Examples: adder circuit, parity, mutual exclusion

Parity Example


```
Boole
parity (Boole a, Boole b, Boole c, Boole d, Boole e,
        Boole f, Boole g, Boole h, Boole i, Boole j)
{
  tmp0 = b ? !a : a;
  tmp1 = c ? !tmp0 : tmp0;
  tmp2 = d ? !tmp1 : tmp1;
  tmp3 = e ? !tmp2 : tmp2;
  tmp4 = f ? !tmp3 : tmp3;
  tmp5 = g ? !tmp4 : tmp4;
  tmp6 = h ? !tmp5 : tmp5;
  tmp7 = i ? !tmp6 : tmp6;
  return j ? !tmp7 : tmp7;
}
```

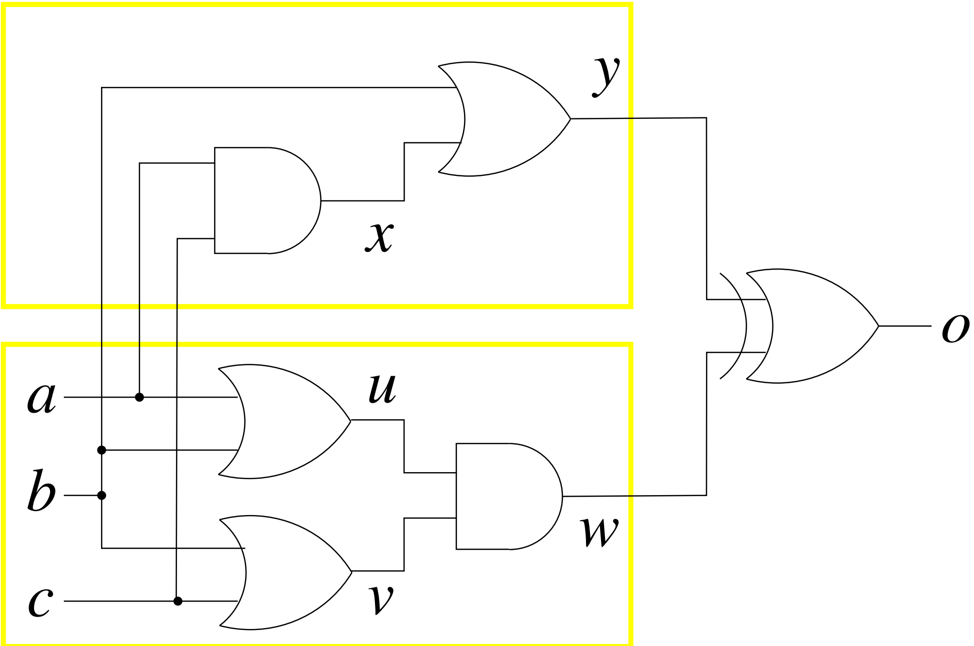
Eliminiate the `tmp...` variables through substitution.

What is the size of the DAG vs the Tree representation?

How to detect Sharing

- through caching of results in algorithms operating on formulas
  (examples: substitution algorithm, generation of NNF for non-monotonic ops)
- when modeling a system: variables are introduced for subformulae
  (then these variables are used multiple times in the toplevel formula)
- structural hashing: detects structural identical subformulae
  (see Signed And Graphs later)
- equivalence extraction: e.g. BDD sweeping, Stålmarcks Method

CNF



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

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

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

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)
- extract multi argument operands (removes variables for intermediate nodes)
- half of AND, OR node constraints can be removed for unnegated nodes
  - a 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 [PlaistedGreenbaum'86] and then [ChambersManoliosVroon'09]
- structural circuit optimizations like in the ABC tool from Berkeley
- however might be simulated on CNF level see [JärvisaloBiereHeule-TACAS'10] and our later discussion on blocked clauses
- compact technology mapping based encoding [EénMishchenkoSörensson'07]

Buggy Program


```
int middle (int x, int y, int z) {
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
      m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

this program is supposed to return the middle (median) of three numbers

Test Suite for Buggy Program


```
middle (1, 2, 3) = 2
middle (1, 3, 2) = 2
middle (2, 1, 3) = 1
middle (2, 3, 1) = 2
middle (3, 1, 2) = 2
middle (3, 2, 1) = 2

middle (1, 1, 1) = 1

middle (1, 1, 2) = 1
middle (1, 2, 1) = 1
middle (2, 1, 1) = 1

middle (1, 2, 2) = 2
middle (2, 1, 2) = 2
middle (2, 2, 1) = 2
```

- This black box test suite has to be generated manually.
- How to ensure that it covers all cases?
- Need to check outcome of each run individually and determine correct result.
- Difficult for large programs.
- Better use specification and check it.

let a be an array of size 3 indexed from 0 to 2

$$
\begin{array}{l}
\quad a[i] = x \wedge a[j] = y \wedge a[k] = z \\
\wedge \\
\quad a[0] \leq a[1] \wedge a[1] \leq a[2] \\
\wedge \\
\quad i \neq j \wedge i \neq k \wedge j \neq k \\
\rightarrow \\
\quad m = a[1]
\end{array}
$$

median obtained by sorting and taking middle element in the order

coming up with this specification is a manual process

Encoding of Middle Program in Logic


```
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
      m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

$$
\begin{array}{ll}
 & (y < z \wedge x < y \rightarrow m = y) \\
\wedge & \\
 & (y < z \wedge x \geq y \wedge x < z \rightarrow m = y) \\
\wedge & \\
 & (y < z \wedge x \geq y \wedge x \geq z \rightarrow m = z) \\
\wedge & \\
 & (y \geq z \wedge x > y \rightarrow m = y) \\
\wedge & \\
 & (y \geq z \wedge x \leq y \wedge x > z \rightarrow m = x) \\
\wedge & \\
 & (y \geq z \wedge x \leq y \wedge x \leq z \rightarrow m = z)
\end{array}
$$

this formula can be generated automatically by a compiler let P be the encoding of the program, and S of the specification

program is correct if "$P \rightarrow S$" is valid

program has a bug if "$P \rightarrow S$" is invalid

program has a bug if negation of "$P \rightarrow S$" is satisfiable (has a model)

program has a bug if "$P \wedge \neg S$" is satisfiable (has a model)

$$
\begin{array}{ll}
(y < z \wedge x < y \rightarrow m = y) & \wedge \\
(y < z \wedge x \geq y \wedge x < z \rightarrow m = y) & \wedge \\
(y < z \wedge x \geq y \wedge x \geq z \rightarrow m = z) & \wedge \\
(y \geq z \wedge x > y \rightarrow m = y) & \wedge \\
(y \geq z \wedge x \leq y \wedge x > z \rightarrow m = x) & \wedge \\
(y \geq z \wedge x \leq y \wedge x \leq z \rightarrow m = z) & \wedge \\
a[i] = x \wedge a[j] = y \wedge a[k] = z & \wedge \\
a[0] \leq a[1] \wedge a[1] \leq a[2] & \wedge \\
i \neq j \wedge i \neq k \wedge j \neq k & \wedge \\
m \neq a[1] &
\end{array}
$$

Encoding with Bit-Vector Logic in SMTLIB2

```
(set-logic QF_AUFBV)
(declare-fun x () (_ BitVec 32)) (declare-fun y () (_ BitVec 32))
(declare-fun z () (_ BitVec 32)) (declare-fun m () (_ BitVec 32))
(assert (=> (and (bvult y z) (bvult x y) ) (= m y)))
(assert (=> (and (bvult y z) (bvuge x y) (bvult x z)) (= m y)))     ; fix last 'y'->'x'
(assert (=> (and (bvult y z) (bvuge x y) (bvuge x z)) (= m z)))
(assert (=> (and (bvuge y z) (bvugt x y) ) (= m y)))
(assert (=> (and (bvuge y z) (bvule x y) (bvugt x z)) (= m x)))
(assert (=> (and (bvuge y z) (bvule x y) (bvule x z)) (= m z)))
(declare-fun i ()(_ BitVec 2)) (declare-fun j ()(_ BitVec 2)) (declare-fun k ()(_ BitVec 2))
(declare-fun a ()(Array (_ BitVec 2) (_ BitVec 32)))
(assert (and (bvule #b00 i) (bvule i #b10) (bvule #b00 j) (bvule j #b10)))
(assert (and (bvule #b00 k) (bvule k #b10)))
(assert (and (= (select a i) x) (= (select a j) y) (= (select a k) z)))
(assert (bvule (select a #b00) (select a #b01)))
(assert (bvule (select a #b01) (select a #b10)))
(assert (distinct i j k))
(assert (distinct m (select a #b01)))
(check-sat)
(get-model)
(exit)
```

```
$ boolector -m middle32-buggy.smt2
sat
(model
  (define-fun x () (_ BitVec 32) #b01100101100011101000011000011001)
  (define-fun y () (_ BitVec 32) #b01100001101010111000011000010101)
  (define-fun z () (_ BitVec 32) #b11101011110110111000110100010110)
  (define-fun m () (_ BitVec 32) #b01100001101010111000011000010101)
  (define-fun i () (_ BitVec 2) #b01)
  (define-fun j () (_ BitVec 2) #b00)
  (define-fun k () (_ BitVec 2) #b10)
  (define-fun a (
   (a_x0 (_ BitVec 2))) (_ BitVec 32)
    (ite (= a_x0 #b00) #b01100001101010111000011000010101
    (ite (= a_x0 #b01) #b01100101100011101000011000011001
    (ite (= a_x0 #b10) #b11101011110110111000110100010110
      #b00000000000000000000000000000000))))
)
2 01100101100011101000011000011001 x
3 01100001101010111000011000010101 y
4 11101011110110111000110100010110 z
5 01100001101010111000011000010101 m
28 01 i
29 00 j
30 10 k
31[00] 01100001101010111000011000010101 a
31[01] 01100101100011101000011000011001 a
31[10] 11101011110110111000110100010110 a
$ boolector middle32-fixed.smt2
unsat
```

- 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 equality check of 4-bit numbers x, y with one bit result e

$$e \leftrightarrow (x = y)$$

$$[e_0]_1 \ \leftrightarrow \ \big([x_3, x_2, x_1, x_0]_4 = [y_3, y_2, y_1, y_0]_4\big)$$

$$e_0 \leftrightarrow \bigwedge_{i=0}^{3} (x_i \leftrightarrow y_i)$$

$$e_0 \leftrightarrow \big((x_3 \leftrightarrow y_3) \wedge (x_2 \leftrightarrow y_2) \wedge (x_1 \leftrightarrow y_1) \wedge (x_0 \leftrightarrow y_0)\big)$$

(strict unsigned) inequality check of 4-bit numbers x, y with one bit result c

$$c \leftrightarrow (x < y)$$

$$[c_0]_1 \;\leftrightarrow\; \big([x_3, x_2, x_1, x_0]_4 < [y_3, y_2, y_1, y_0]_4\big)$$

$$c_0 \leftrightarrow \mathsf{LessThan}(3, x, y)$$

with

$$\begin{array}{lcll} \mathsf{LessThan}(-1, x, y) & = & \bot & \\ \mathsf{LessThan}(\;\; i, x, y) & = & (\neg x_i \wedge y_i) \vee \big((x_i \leftrightarrow y_i) \wedge \mathsf{LessThan}(i-1, x, y)\big) & \text{if } i \leq 0 \end{array}$$

$$c_0 \;\leftrightarrow\; \bar{x}_3 y_3 \vee (x_3 = y_3)(\bar{x}_2 y_2 \vee (x_2 = y_2)(\bar{x}_1 y_1 \vee (x_1 = y_1)\bar{x}_1 y_1))$$

addition of 4-bit numbers x, y with result s also 4-bit

$$s = x + y$$

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

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

where

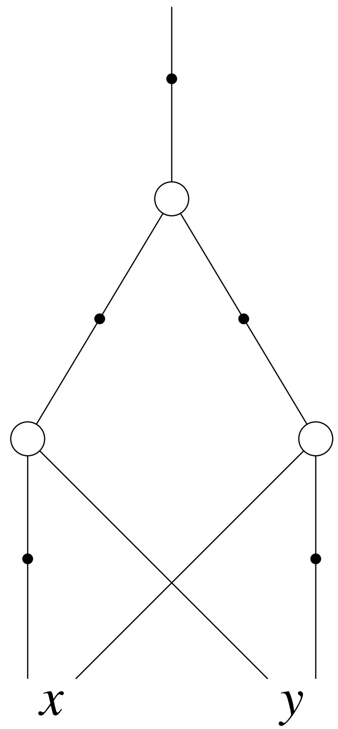
$$
\begin{aligned}
[\, s \,, o \,]_2 &= \text{FullAdder}(x, y, i) \quad \text{with} \\
s &\leftrightarrow x \text{ xor } y \text{ xor } i \\
o &\leftrightarrow (x \wedge y) \vee (x \wedge i) \vee (y \wedge i) \;=\; ((x + y + i) \geq 2)
\end{aligned}
$$

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

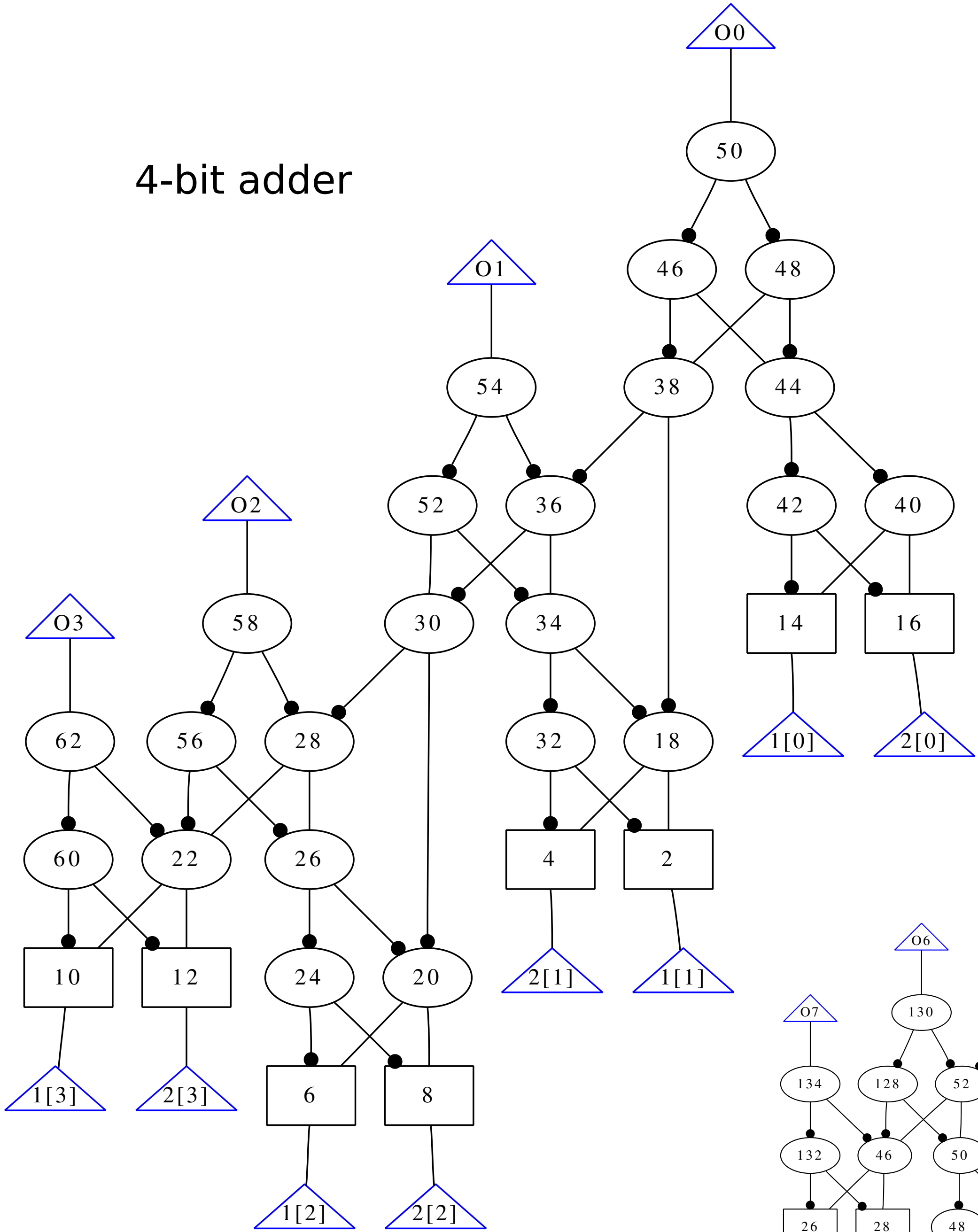
$$x \text{ xor } y \;\equiv\; (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \;\equiv\; \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

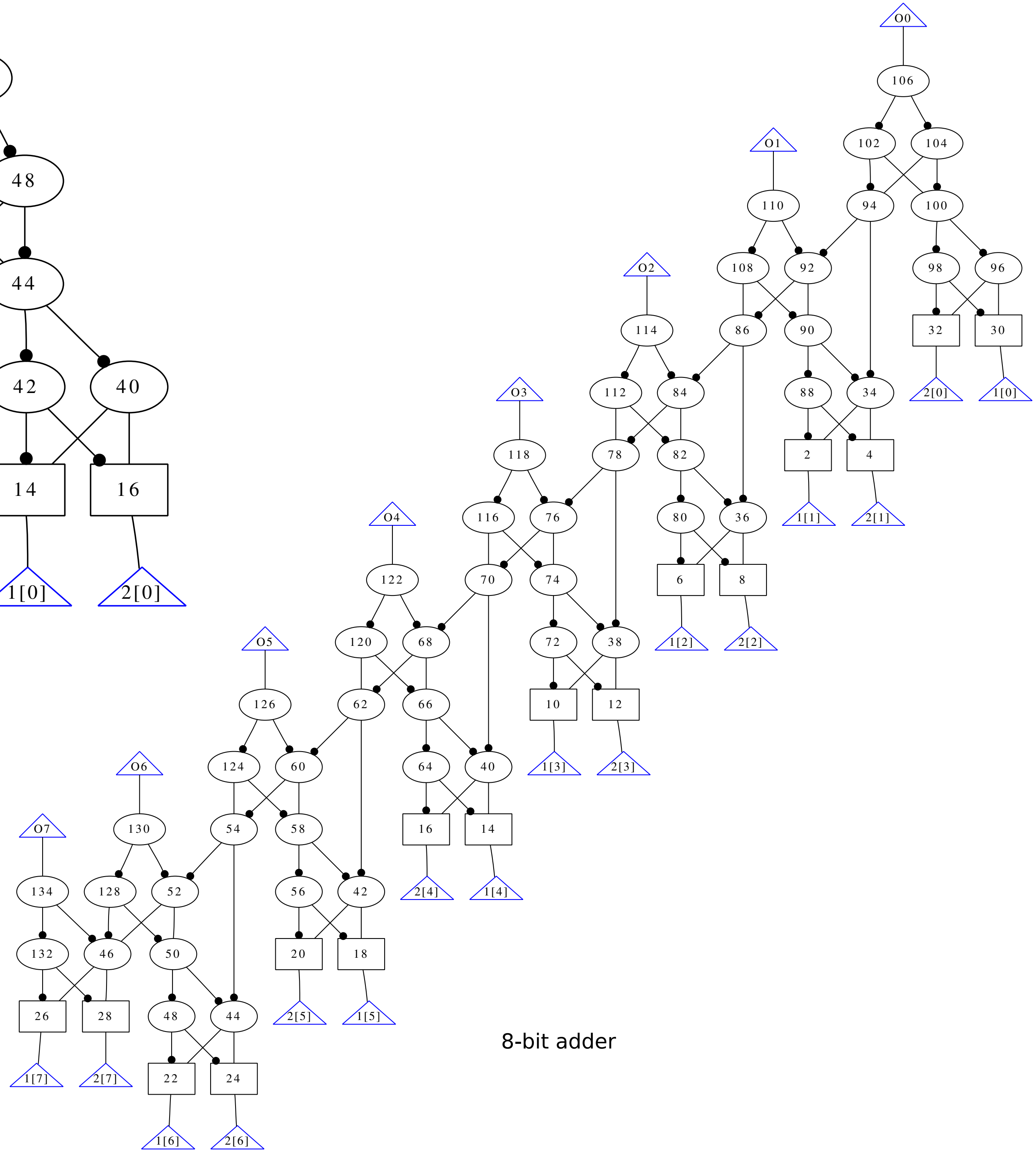
```c
typedef struct AIG AIG;

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:

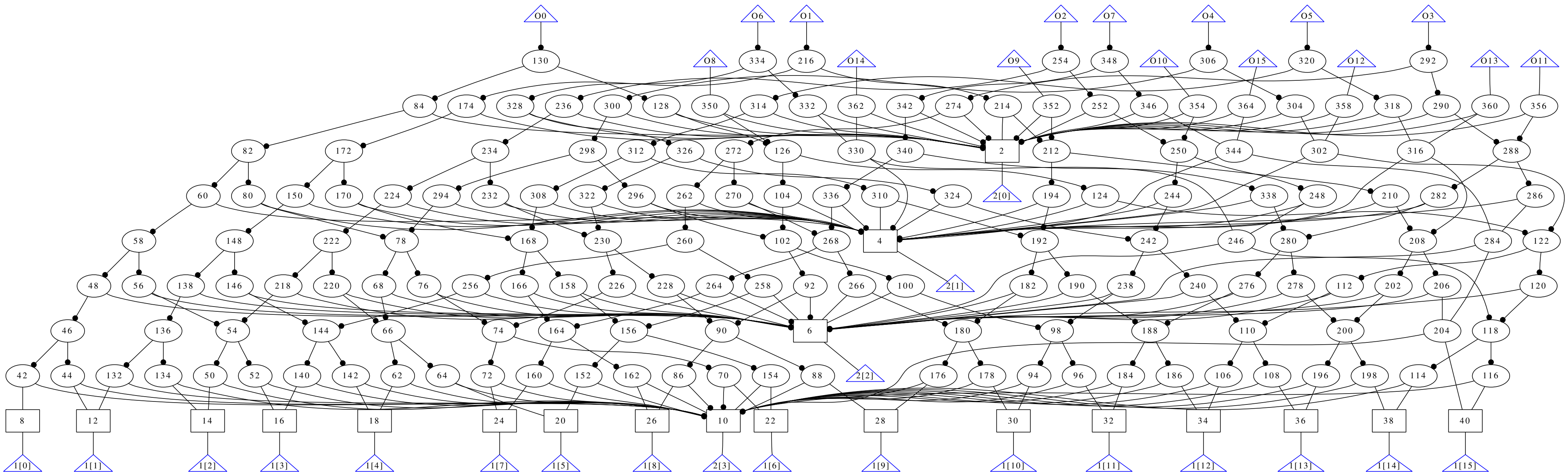
`sizeof(unsigned) == sizeof(void*)`

4-bit adder

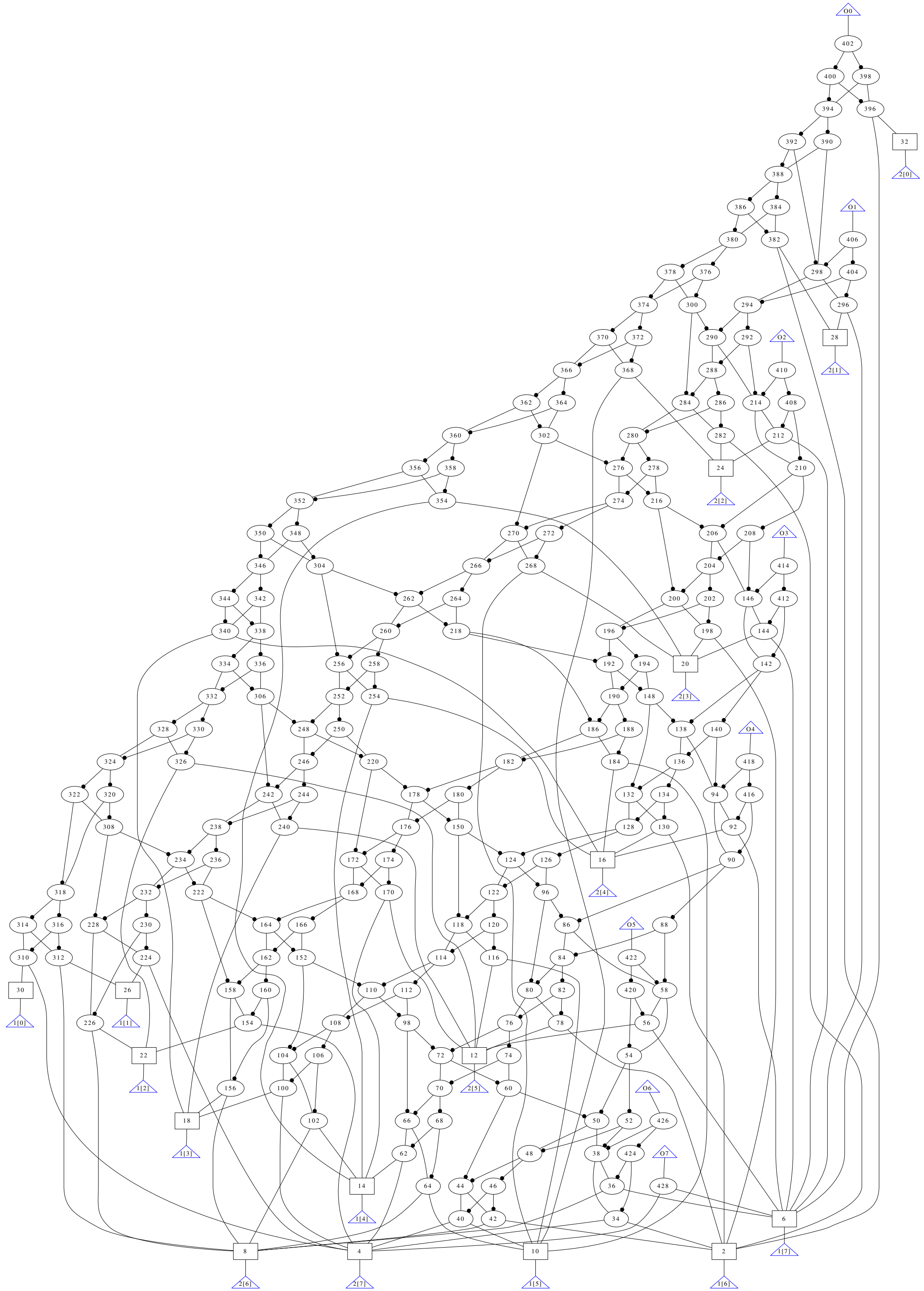




8-bit adder



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]

Structural Hashing and Hyper-Binary Resolution

[HeuleJärvisaloBiere-CPAIOR'13]

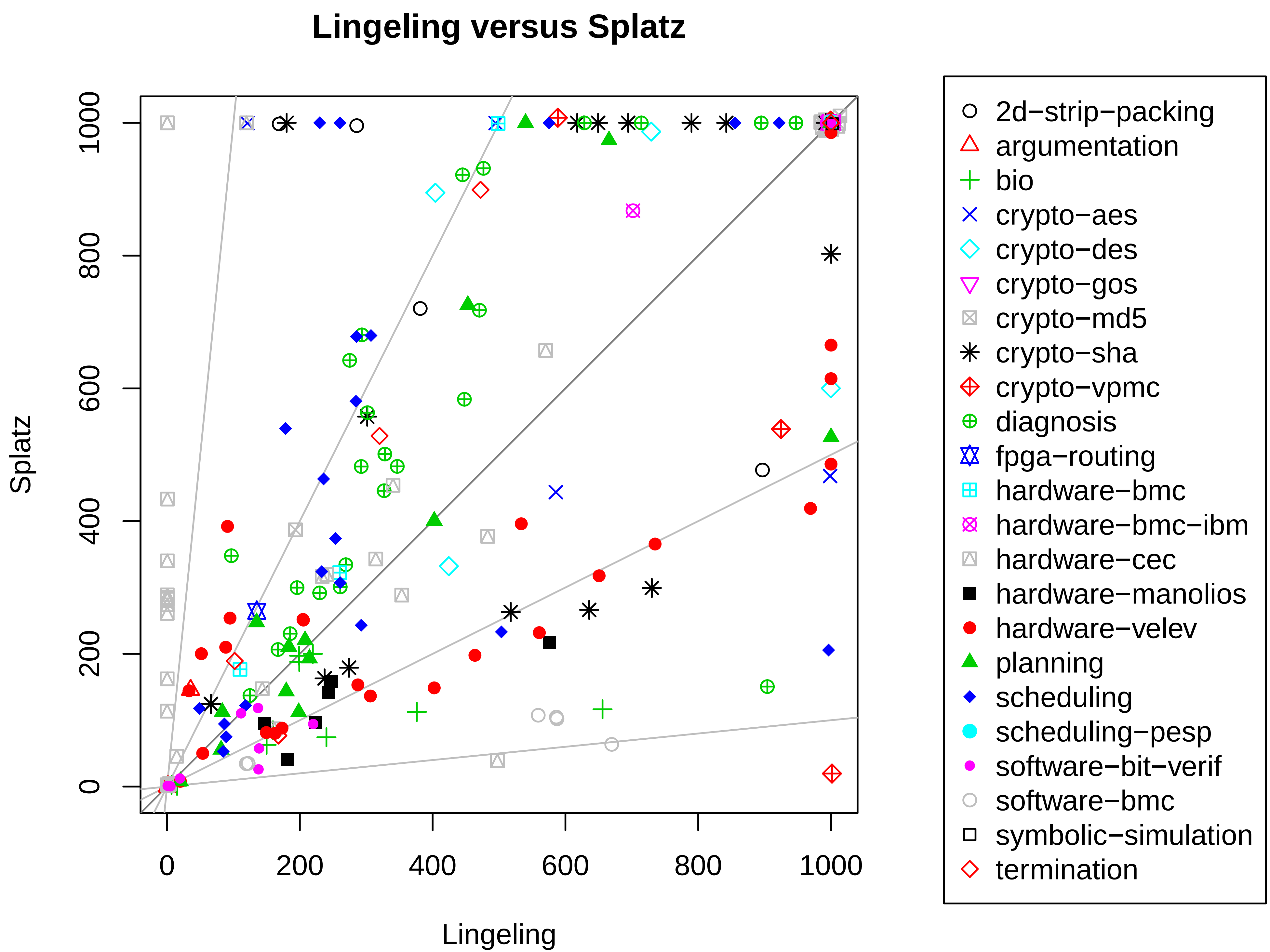
$$\frac{a \leftrightarrow x \wedge y \qquad b \leftrightarrow x \wedge y}{a \leftrightarrow b}$$

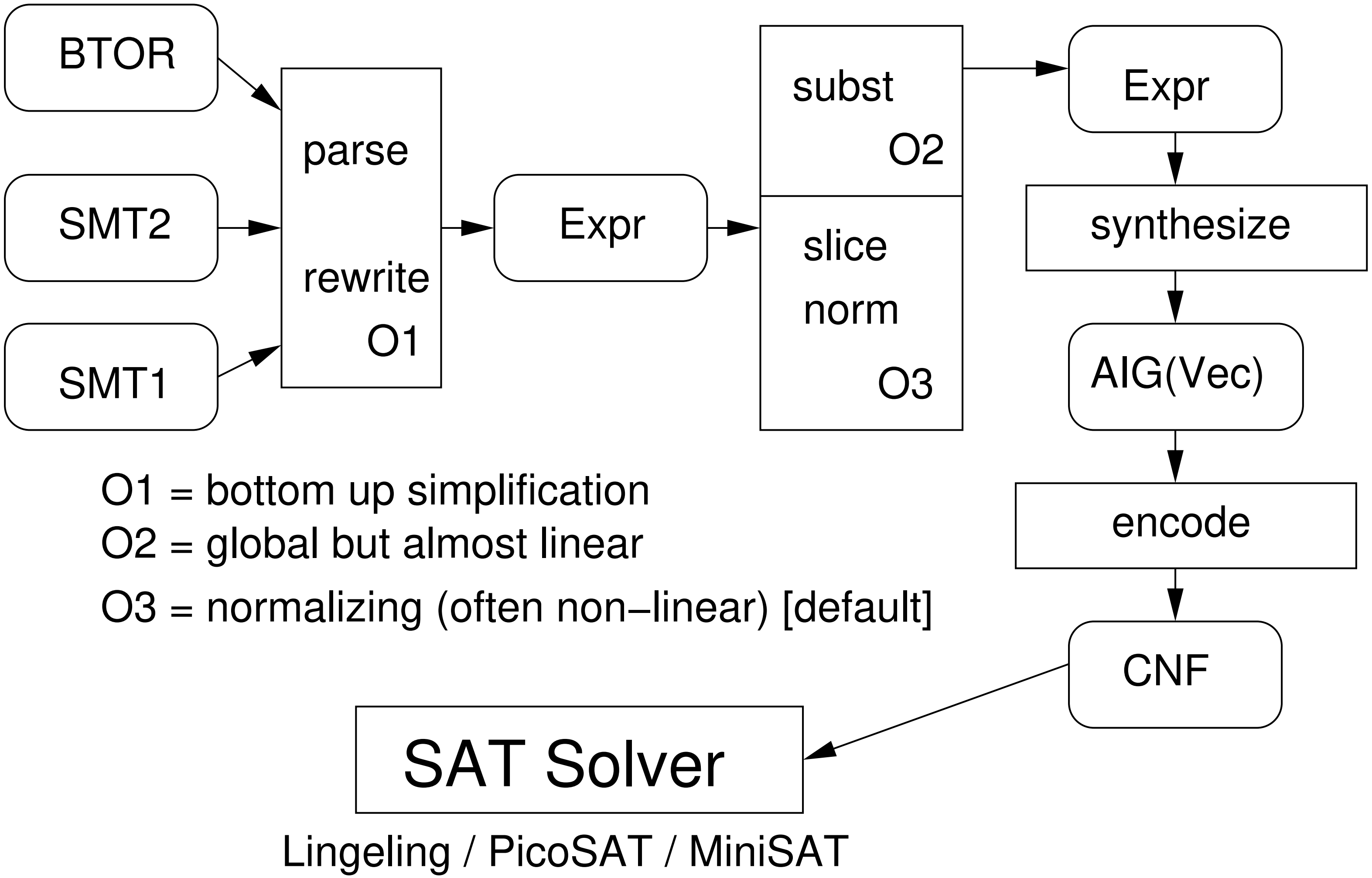
$$(\bar{a} \vee x)(\bar{a} \vee y)(a \vee \bar{x} \vee \bar{y})(\bar{b} \vee x)(\bar{b} \vee y)(b \vee \bar{x} \vee \bar{y})$$

hyper-binary resolve in multiple binary clauses in "parallel":

$$\frac{\bar{a} \vee x \quad \bar{a} \vee y \quad b \vee \bar{x} \vee \bar{y}}{\bar{a} \vee b} \qquad \frac{\bar{b} \vee x \quad \bar{b} \vee y \quad a \vee \bar{x} \vee \bar{y}}{a \vee \bar{b}}$$

thus "in principle" hyper-binary resolution can simulate structural hashing, however ...


Lingeling versus Splatz
Splatz
Lingeling
0
200
400
600
800
1000
2d−strip−packing
argumentation
bio
crypto−aes
crypto−des
crypto−gos
crypto−md5
crypto−sha
crypto−vpmc
diagnosis
fpga−routing
hardware−bmc
hardware−bmc−ibm
hardware−cec
hardware−manolios
hardware−velev
planning
scheduling
scheduling−pesp
software−bit−verif
software−bmc
symbolic−simulation
termination


BTOR
SMT2
SMT1
parse
rewrite
O1
Expr
subst
O2
slice
norm
O3
Expr
synthesize
AIG(Vec)
encode
CNF
O1 = bottom up simplification
O2 = global but almost linear
O3 = normalizing (often non–linear) [default]
SAT Solver
Lingeling / PicoSAT / MiniSAT

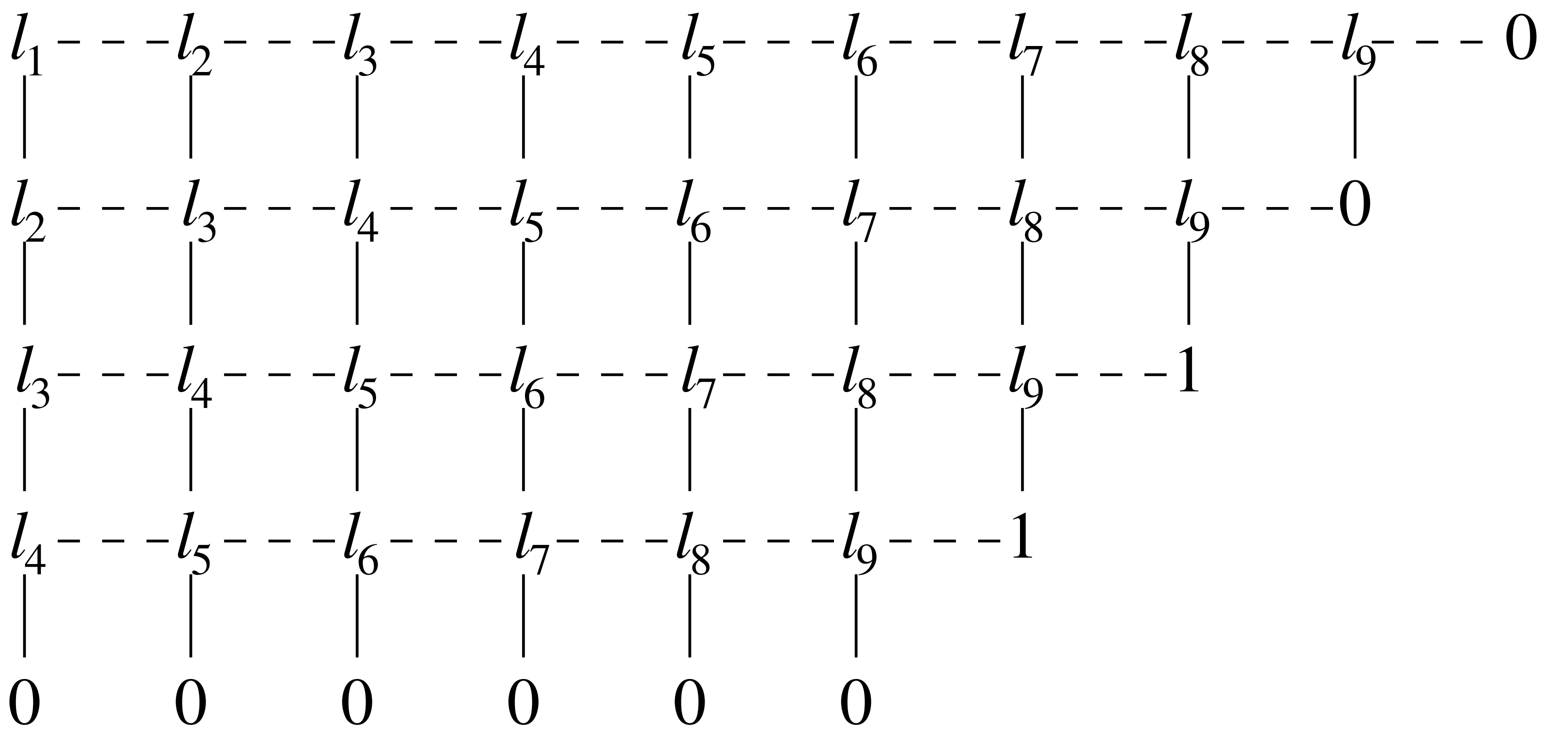
- Tseitin's construction suitable for most kinds of "model constraints"
  - assuming simple operational semantics: encode an interpreter
  - small domains: one-hot encoding large domains: binary encoding
- harder to encode properties or additional constraints
  - temporal logic / fix-points
  - environment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

- 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

BDD based Encoding of Cardinality Constraints

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



“then” edge downward, “else” edge to the right

Bounded Variable Elimination (BVE)

[DavisPutnam60] [EénBiere SAT'05]

- considered to be the most effective preprocessing technique
  - works particularly well on "industrial" formulas
  - usually removes 80% variables and a similar number of clauses
  - bounded: eliminate variable if resulting CNF does not have more clauses
  - replace

    $$\bigwedge_i (x \vee C_i) \wedge \bigwedge_j (\neg x \vee D_j)$$

    by

    $$\bigwedge_{i,j} (C_i \vee D_j)$$

  - ignore tautological $C_i \vee D_j$
  - always for 0, or 1 positive/negative occurrences
  - same for 2 positive and 2 negative occurrences
  - combined with subsumption and strengthening
- simulates NNF compact encodings "at the leafs"

Blocked Clauses

[Kullman'99]

blocked clause $C \in F$

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

fix a CNF F

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

$(a \vee b \vee l)$

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

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

Blocked Clauses and Encoding / Preprocessing Techniques

[JärvisaloBiereHeule-TACAS'10]

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

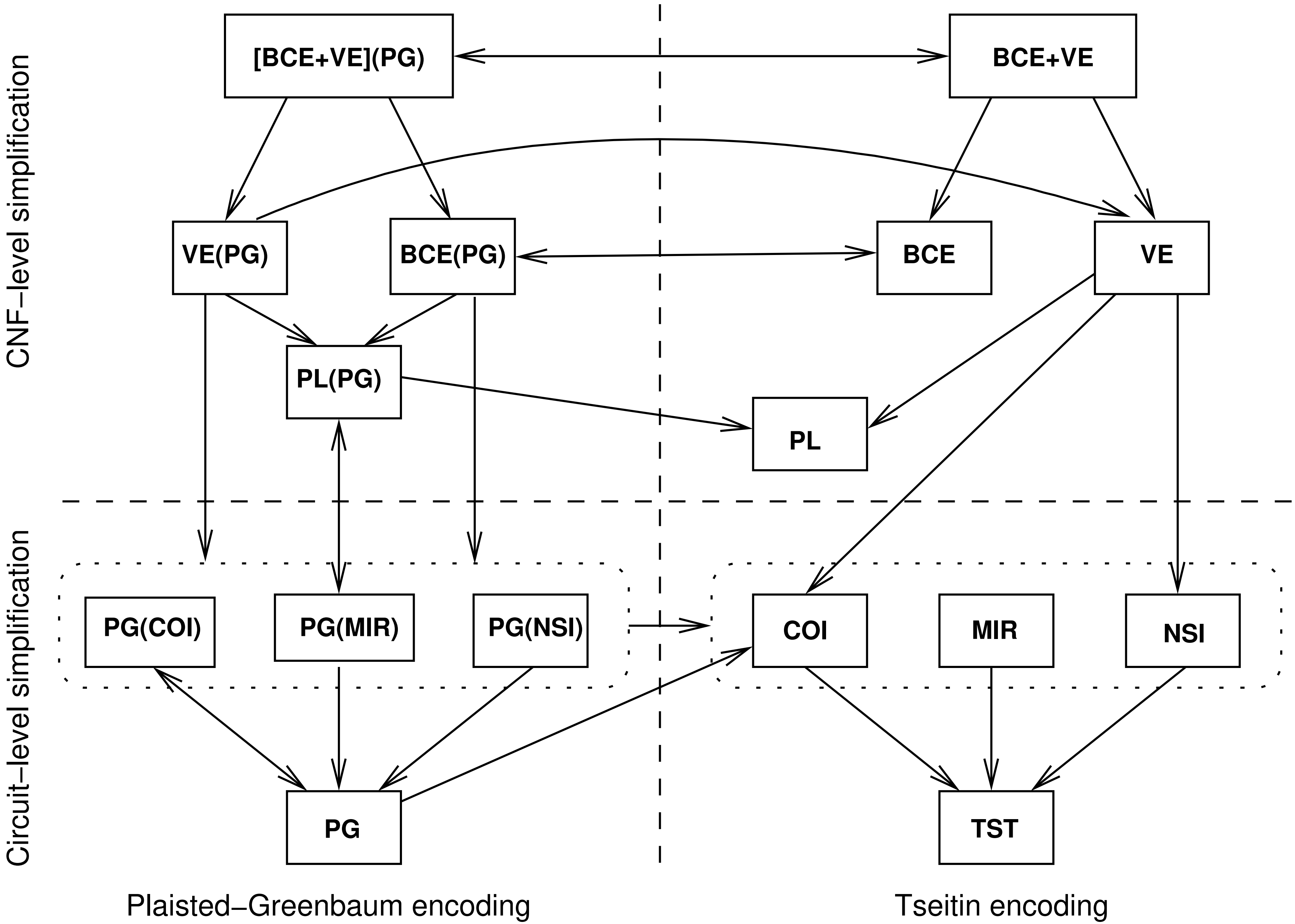
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

(B)VE (Bounded) Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination


[BCE+VE](PG)
BCE+VE
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
CNF−level simplification
Circuit−level simplification
Plaisted−Greenbaum encoding
Tseitin encoding

PrecoSAT [Biere'09], Lingeling [Biere'10], also in CryptoMiniSAT [Soos'09]

- preprocessing can be extremely beneficial
  - most SAT competition solvers use bounded variable elimination (BVE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple idea to benefit from full preprocessing without penalty
  - "preempt" preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time

Benefits of Inprocessing

- special case incremental preprocessing:
  - preprocessing during incremental SAT solving
- allows to use costly preprocessors
  - without increasing run-time "much" in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and distillation even BVE can be costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
  - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding "bad" implementation though ...
- ... and hard(er) to debug and get right [JärvisaloHeuleBiere-IJCAR'12]
- more complex API: `lglfreeze`, `lglmelt` ...