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**

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

**Optimized C Code**

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

<table>
<thead>
<tr>
<th>Original C Code</th>
<th>Optimized C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(!a &amp;&amp; !b) h();</td>
<td>if(a) f();</td>
</tr>
<tr>
<td>else if(!a) g();</td>
<td>else if(b) g();</td>
</tr>
<tr>
<td>else f();</td>
<td>else h();</td>
</tr>
</tbody>
</table>

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

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

How to check that these two versions are equivalent?
Compilation

original \equiv \text{if } \neg a \land \neg b \text{ then } h \text{ else if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land \text{if } \neg a \text{ then } g \text{ else } f
\equiv (\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f)

optimized \equiv \text{if } a \text{ then } f \text{ else if } b \text{ then } g \text{ else } h
\equiv a \land f \lor \neg a \land \text{if } b \text{ then } g \text{ else } h
\equiv a \land f \lor \neg a \land (b \land g \lor \neg b \land h)

(\neg a \land \neg b) \land h \lor \neg (\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \iff a \land f \lor \neg a \land (b \land g \lor \neg b \land 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 \(\text{compile(} \text{original}\text{)} \not\equiv \text{compile(} \text{optimized}\text{)}\) 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

Translating into SAT @ Industrial Day SAT’16
SAT (Satisfiability) the classical NP complete Problem:

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

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

**SAT belongs to NP**

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

- guess the assignment $\sigma$ (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)
Conjunctive Normal Form

**Definition**

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

\[ C_1 \land C_2 \land \ldots \land C_n \]

Each clause \( C \) is a disjunction of literals

\[ C = L_1 \lor \ldots \lor L_m \]

And each literal is either a plain variable \( x \) or a negated variable \( \bar{x} \).

**Example**

\[(a \lor b \lor c) \land (\bar{a} \lor \bar{b}) \land (\bar{a} \lor \bar{c})\]

**Note 1:** two notions for negation: in \( \bar{x} \) 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
CNF Encoding through Negation Normalform

- **NNF:** \( \neg \) in front of variables only, arbitrary nested \( \land \) and \( \lor \)

- might need to expand non-monotonic operators into \( \land \) and \( \lor \)
  - \((a \leftrightarrow b) \equiv (\neg a \land \neg b) \lor (a \land b)\)
  - requires to work with circuit/DAG to avoid exponential explosion

- apply De’Morgan rule to push negations down
  - \(\neg (a \land b) \equiv \neg a \lor \neg b\)
  - \(\neg (a \lor b) \equiv \neg a \land \neg b\)

- bottom-up CNF translation
  - \((\land_i C_i) \land (\land_j D_j) \) is already a CNF
  - \((\land_i C_i) \lor (\land_j D_j) \equiv \land_{i,j} (C_i \lor D_j)\) “clause distribution” (quadratic)

- whole procedure exponential in \( \lor/\land \) alternation depth

- but might produce compact CNFs for small formulas
  - \((\neg a \land \neg b) \lor (a \land b) \equiv (\neg a \lor a) \land (\neg a \lor b) \land (\neg b \lor a) \land (\neg b \lor b)\)

- 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
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;
  \textbf{return} j ? !tmp7 : tmp7;
}

Eliminate the \texttt{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
Example of Tseitin Transformation: Circuit to CNF

\[
\begin{align*}
& \bigwedge o \\
& \quad \bigwedge (x \leftrightarrow a \land c) \\
& \quad \bigwedge (y \leftrightarrow b \lor x) \\
& \quad \bigwedge (u \leftrightarrow a \lor b) \\
& \quad \bigwedge (v \leftrightarrow b \lor c) \\
& \quad \bigwedge (w \leftrightarrow u \land v) \\
& \quad \bigwedge (o \leftrightarrow y \oplus w)
\end{align*}
\]

\[
\begin{align*}
& o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \\
& o \land (\overline{x} \lor a) \land (\overline{x} \lor c) \land (x \lor \overline{a} \lor \overline{c}) \land \ldots
\end{align*}
\]
Tseitin Transformation: Input / Output Constraints

Negation: 
\[ x \leftrightarrow \bar{y} \iff (x \rightarrow \bar{y}) \land (\bar{y} \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y}) \land (y \lor x) \]

Disjunction: 
\[ x \leftrightarrow (y \lor z) \iff (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \]
\[ \iff (\bar{y} \lor x) \land (\bar{z} \lor x) \land (\bar{x} \lor y \lor z) \]

Conjunction: 
\[ x \leftrightarrow (y \land z) \iff (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \]
\[ \iff (\bar{x} \lor y) \land (\bar{x} \lor z) \land ((y \lor z) \lor x) \]
\[ \iff (\bar{x} \lor y) \land (\bar{x} \lor z) \land (\bar{y} \lor \bar{z} \lor x) \]

Equivalence: 
\[ x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land ((y \leftrightarrow z) \rightarrow x) \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land ((y \land z) \lor (\bar{y} \land \bar{z})) \rightarrow x \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land ((y \land z) \lor (\bar{y} \land \bar{z})) \lor x \]
\[ \iff (\bar{x} \lor \bar{y} \lor z) \land (\bar{x} \lor \bar{z} \lor y) \land (\bar{y} \lor \bar{z} \lor x) \land (y \lor z \lor x) \]
- 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]
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
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.
Specification for Middle

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

$$a[i] = x \land a[j] = y \land a[k] = z \land a[0] \leq a[1] \land a[1] \leq a[2] \land i \neq j \land i \neq k \land j \neq k \rightarrow m = a[1]$$

Median obtained by sorting and taking middle element in the order coming up with this specification is a manual process.
```cpp
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;
```

\[(y < z \land x < y \rightarrow m = y)\] \land \[(y < z \land x \geq y \land x < z \rightarrow m = y)\] \land \[(y < z \land x \geq y \land x \geq z \rightarrow m = z)\] \land \[(y \geq z \land x > y \rightarrow m = y)\] \land \[(y \geq z \land x \leq y \land x > z \rightarrow m = x)\] \land \[(y \geq z \land x \leq y \land x \leq z \rightarrow m = z)\]

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 “\( P \land \neg S \)” is satisfiable (has a model)

\[
(y < z \land x < y \rightarrow m = y) \land \\
(y < z \land x \geq y \land x < z \rightarrow m = y) \land \\
(y < z \land x \geq y \land x \geq z \rightarrow m = z) \land \\
(y \geq z \land x > y \rightarrow m = y) \land \\
(y \geq z \land x \leq y \land x > z \rightarrow m = x) \land \\
(y \geq z \land x \leq y \land x \leq z \rightarrow m = z) \land \\
a[i] = x \land a[j] = y \land a[k] = z \land \\
a[0] \leq a[1] \land a[1] \leq a[2] \land \\
i \neq j \land i \neq k \land j \neq k \land \\
m \neq a[1]
\]
(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) #b01100101100011110100011000011001)
  (define-fun y () (_ BitVec 32) #b01100001101010111000011000010101)
  (define-fun z () (_ BitVec 32) #b111010111101110111100011000010110)
  (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 011001011000011101000011000011001 x
3 01100001101010111000011000010101 y
4 1110101111011101110001100010110 z
5 01100001101010111000011000010101 m
28 01 i
29 00 j
30 10 k
31[00] 01100001101010111000011000010101 a
31[01] 01100101100011101000011000011001 a
31[10] 1110101111011101110001100010110 a
$ boolector middle32-fixed.smt2
unsat
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
equality check of 4-bit numbers $x, y$ with one bit result $e$

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

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

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

$$e_0 \leftrightarrow ((x_3 \leftrightarrow y_3) \land (x_2 \leftrightarrow y_2) \land (x_1 \leftrightarrow y_1) \land (x_0 \leftrightarrow y_0))$$
(strict unsigned) inequality check of 4-bit numbers $x, y$ with one bit result $c$

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

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

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

with

$$\text{LessThan}(-1, x, y) = \perp$$

$$\text{LessThan}(i, x, y) = (\neg x_i \land y_i) \lor ((x_i \leftrightarrow y_i) \land \text{LessThan}(i - 1, x, y)) \quad \text{if } i \leq 0$$

$$c_0 \leftrightarrow \bar{x}_3 y_3 \lor (x_3 = y_3)(\bar{x}_2 y_2 \lor (x_2 = y_2)(\bar{x}_1 y_1 \lor (x_1 = y_1)\bar{x}_1 y_1))$$
addition of 4-bit numbers $x, y$ with result $s$ also 4-bit

$$s = x + y$$

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

$$[s_3, \cdot]_2 = \text{FullAdder}(x_3, y_3, c_2)$$

$$[s_2, c_2]_2 = \text{FullAdder}(x_2, y_2, c_1)$$

$$[s_1, c_1]_2 = \text{FullAdder}(x_1, y_1, c_0)$$

$$[s_0, c_0]_2 = \text{FullAdder}(x_0, y_0, \text{false})$$

where

$$[s, o]_2 = \text{FullAdder}(x, y, i) \quad \text{with}$$

$$s \leftrightarrow x \text{ xor } y \text{ xor } i$$

$$o \leftrightarrow (x \land y) \lor (x \land i) \lor (y \land i) = ((x + y + i) \geq 2)$$
And-Inverter-Graphs (AIG)

- widely adopted bit-level intermediate representation
  - see for instance our AIGER format [http://fmv.jku.at/aiger](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

---

Translating into SAT @ Industrial Day SAT’16
XOR as AIG

\[ x \text{ xor } y \equiv (\bar{x} \wedge y) \lor (x \wedge \bar{y}) \equiv (\bar{x} \wedge y) \wedge (x \wedge \bar{y}) \]

negation/sign are edge attributes
not part of node
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
Structural Hashing and Hyper-Binary Resolution

[HeuleJärvisaloBiere-CPAIOR’13]

\[ a \leftrightarrow x \land y \quad b \leftrightarrow x \land y \]
\[ a \leftrightarrow b \]

\((\overline{a} \lor x)(\overline{a} \lor y)(a \lor \overline{x} \lor \overline{y})(\overline{b} \lor x)(\overline{b} \lor y)(b \lor \overline{x} \lor \overline{y})\)

**hyper-binary resolve** in multiple binary clauses in “parallel”:

\[
\begin{array}{ccc}
\overline{a} \lor x & \overline{a} \lor y & b \lor \overline{x} \lor \overline{y} \\
\overline{a} \lor b & & \\
\end{array}
\quad
\begin{array}{ccc}
\overline{b} \lor x & \overline{b} \lor y & a \lor \overline{x} \lor \overline{y} \\
& a \lor \overline{b} & \\
\end{array}
\]

thus “in principle” hyper-binary resolution can simulate structural hashing, however …
Lingeling versus Splatz

The diagram compares Lingeling and Splatz, showing the performance of various benchmarks. The x-axis represents Lingeling's performance, and the y-axis represents Splatz's performance. Each point on the graph represents a benchmark, with different symbols and colors indicating different categories.

Categories include:
- 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

The data suggests that in most categories, Lingeling outperforms Splatz, with a few exceptions where Splatz performs slightly better.
Boolector Architecture

O1 = bottom up simplification
O2 = global but almost linear
O3 = normalizing (often non-linear) [default]

Lingeling / PicoSAT / MiniSAT
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: \( x = (a \lor y), \quad y = (b \lor x) \)
  - has unique least fix-point \( x = y = (a \lor b) \)
  - and unique largest fix-point \( x = y = \text{true} \)
  - 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.
  \[ 2 \cdot a + b + c + d + 2 \cdot e \geq 3 \]
  actually used to handle MaxSAT in SAT4J for configuration in Eclipse

Translating into SAT @ Industrial Day SAT’16
BDD based Encoding of Cardinality Constraints

\[ 2 \leq |\{l_1, \ldots, l_9\}| \leq 3 \]

```
  l_1  l_2  l_3  l_4  l_5  l_6  l_7  l_8  l_9  0
  l_2  l_3  l_4  l_5  l_6  l_7  l_8  l_9  0
  l_3  l_4  l_5  l_6  l_7  l_8  l_9  1
  l_4  l_5  l_6  l_7  l_8  l_9  1
  0  0  0  0  0  0  0  0
```

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

Translating into SAT @ Industrial Day SAT’16
Bounded Variable Elimination (BVE)

- 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

\[
\bigwedge_i (x \lor C_i) \land \bigwedge_j (\neg x \lor D_j)
\]

by

\[
\bigwedge_{i,j} (C_i \lor D_j)
\]

- ignore tautological \(C_i \lor D_j\)
- 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** $F$

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

$(a \lor b \lor l)$

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

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

**Proof**

assignment $\sigma$ satisfying $F \setminus C$ but not $C$

can be extended to a satisfying assignment of $F$ by flipping value of $l$
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COI</td>
<td>Cone-of-Influence reduction</td>
</tr>
<tr>
<td>MIR</td>
<td>Monontone-Input-Reduction</td>
</tr>
<tr>
<td>NSI</td>
<td>Non-Shared Inputs reduction</td>
</tr>
<tr>
<td>PG</td>
<td>Plaisted-Greenbaum polarity based encoding</td>
</tr>
<tr>
<td>TST</td>
<td>standard Tseitin encoding</td>
</tr>
<tr>
<td>(B)VE</td>
<td>(Bounded) Variable-Elimination</td>
</tr>
<tr>
<td>BCE</td>
<td>Blocked-Clause-Elimination</td>
</tr>
</tbody>
</table>

Translating into SAT @ Industrial Day SAT'16
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, cannot 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` …

Translating into SAT @ Industrial Day SAT’16