Understanding Modern SAT Solvers

VTSA'12

Summer School 2012

Verification Technology, Systems & Applications

September 2012

Max Planck Institut Informatik

Saarbrücken, Germany

Armin Biere

Institute for Formal Models and Verification

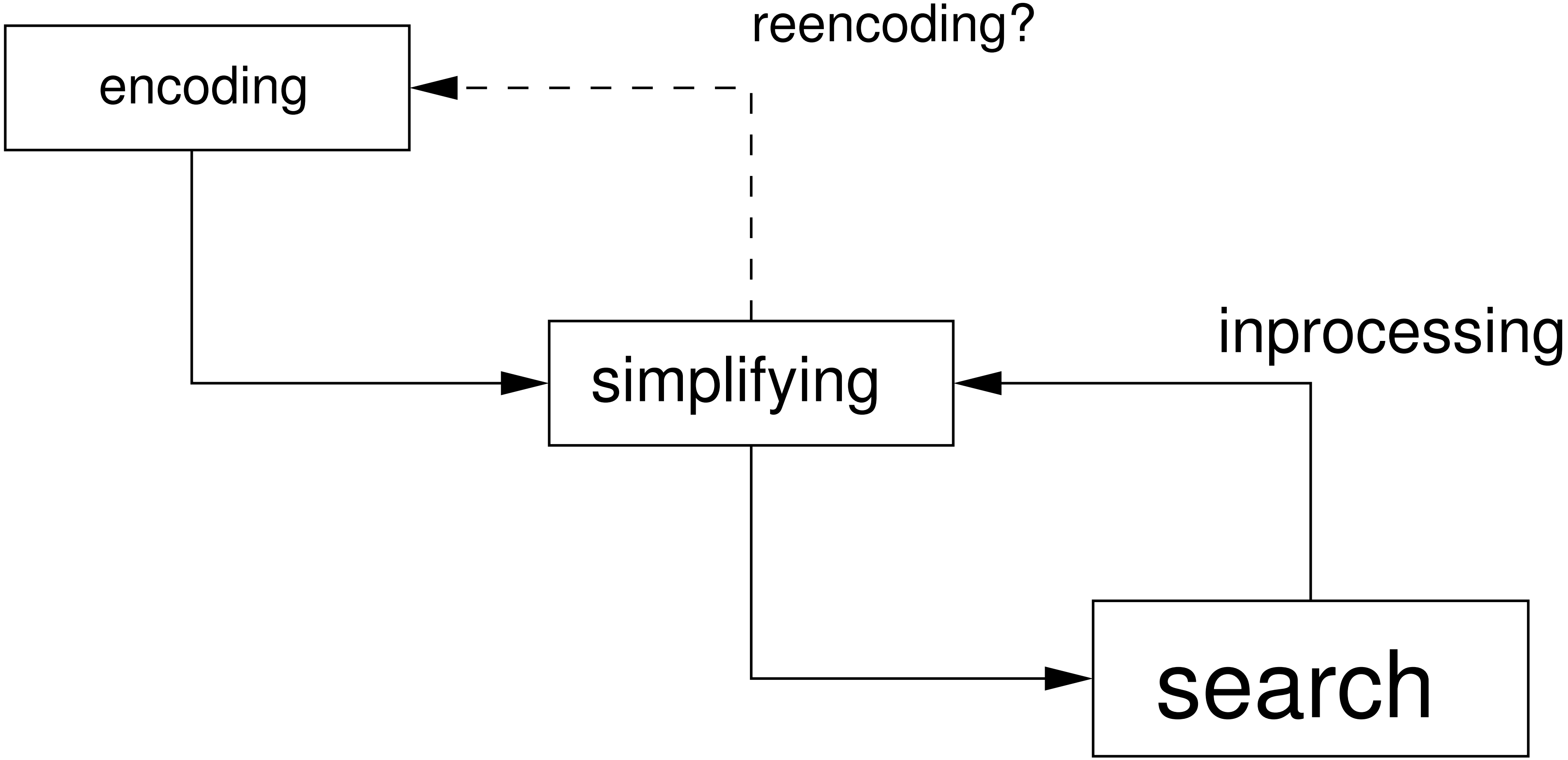
Johannes Kepler University, Linz, Austria

http://fmv.jku.at/picosat

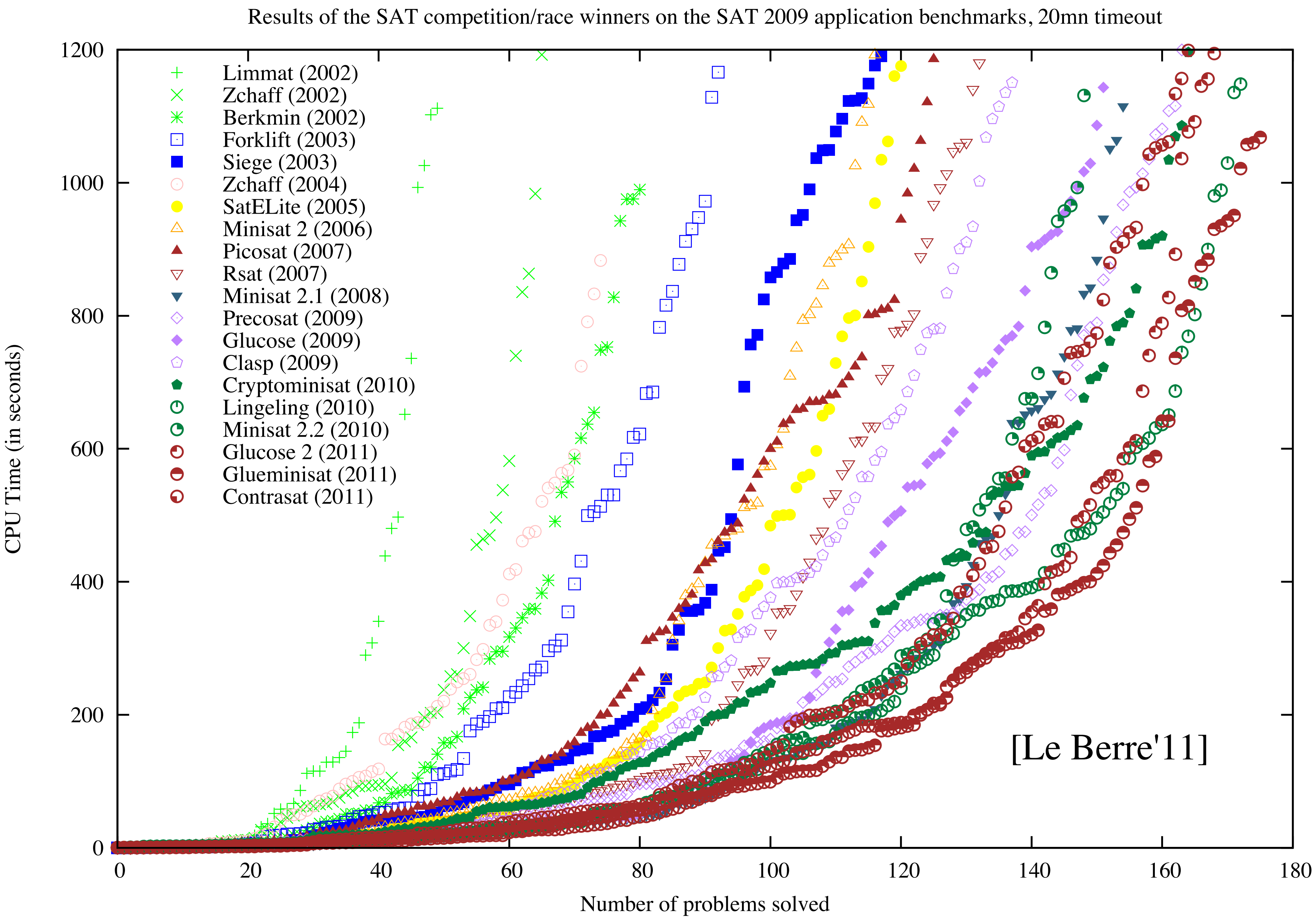
http://fmv.jku.at/biere/talks/Biere-VTSA12-talk.pdf

http://fmv.jku.at/cleaneling/cleaneling00f.zip

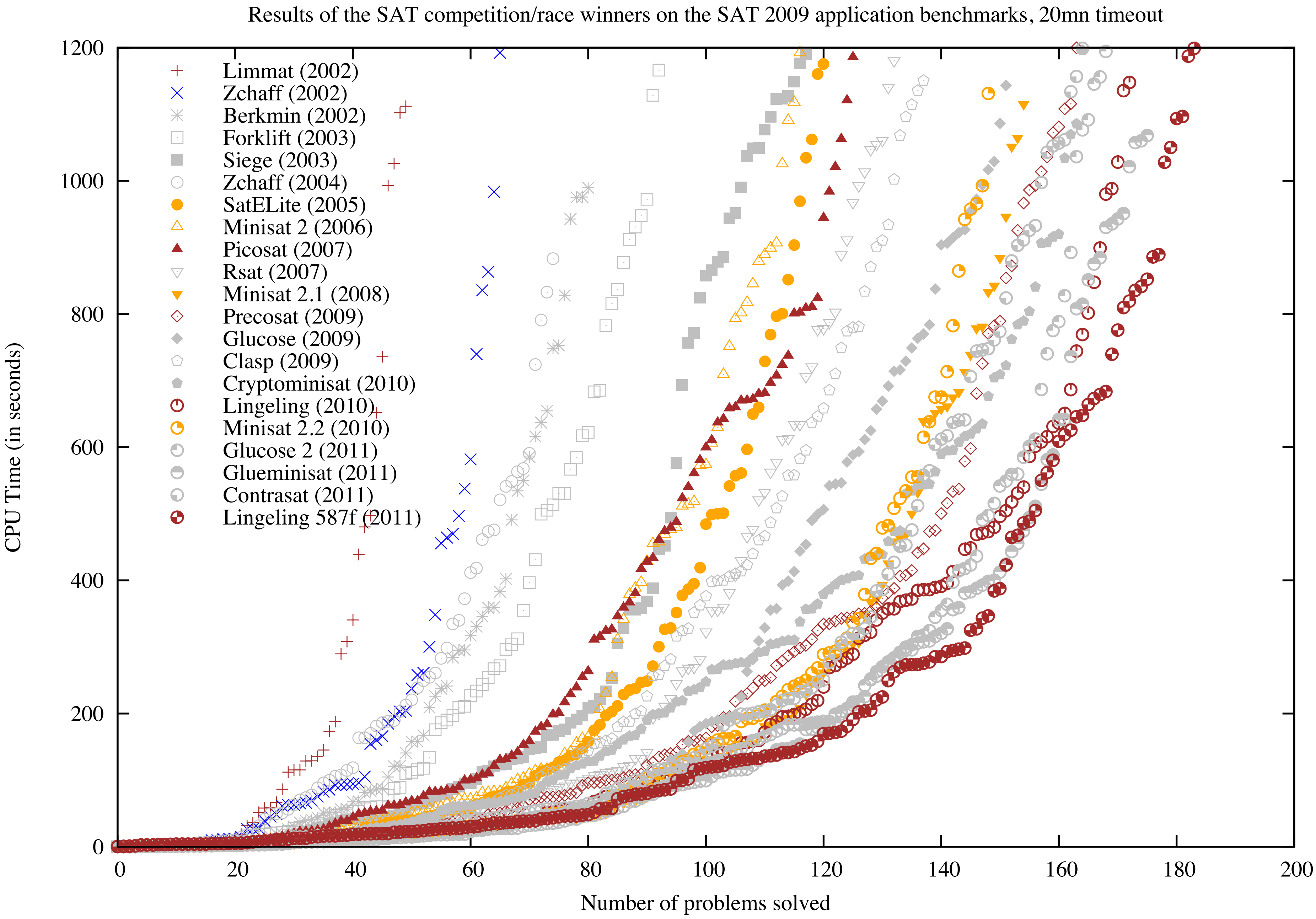
What is Practical SAT Solving?



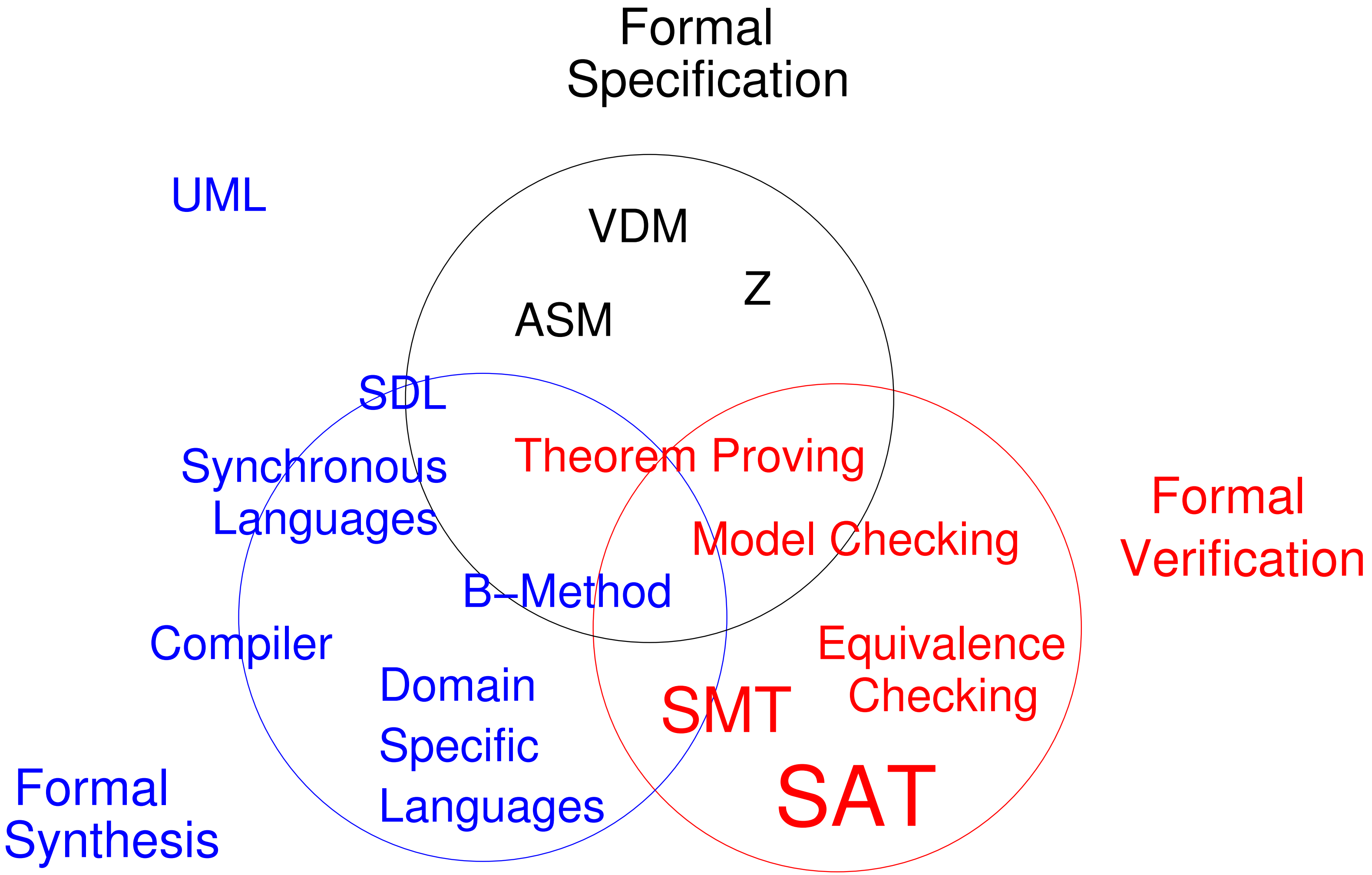
SAT Competition / Race Winners on SC 2009 Application Benchmarks



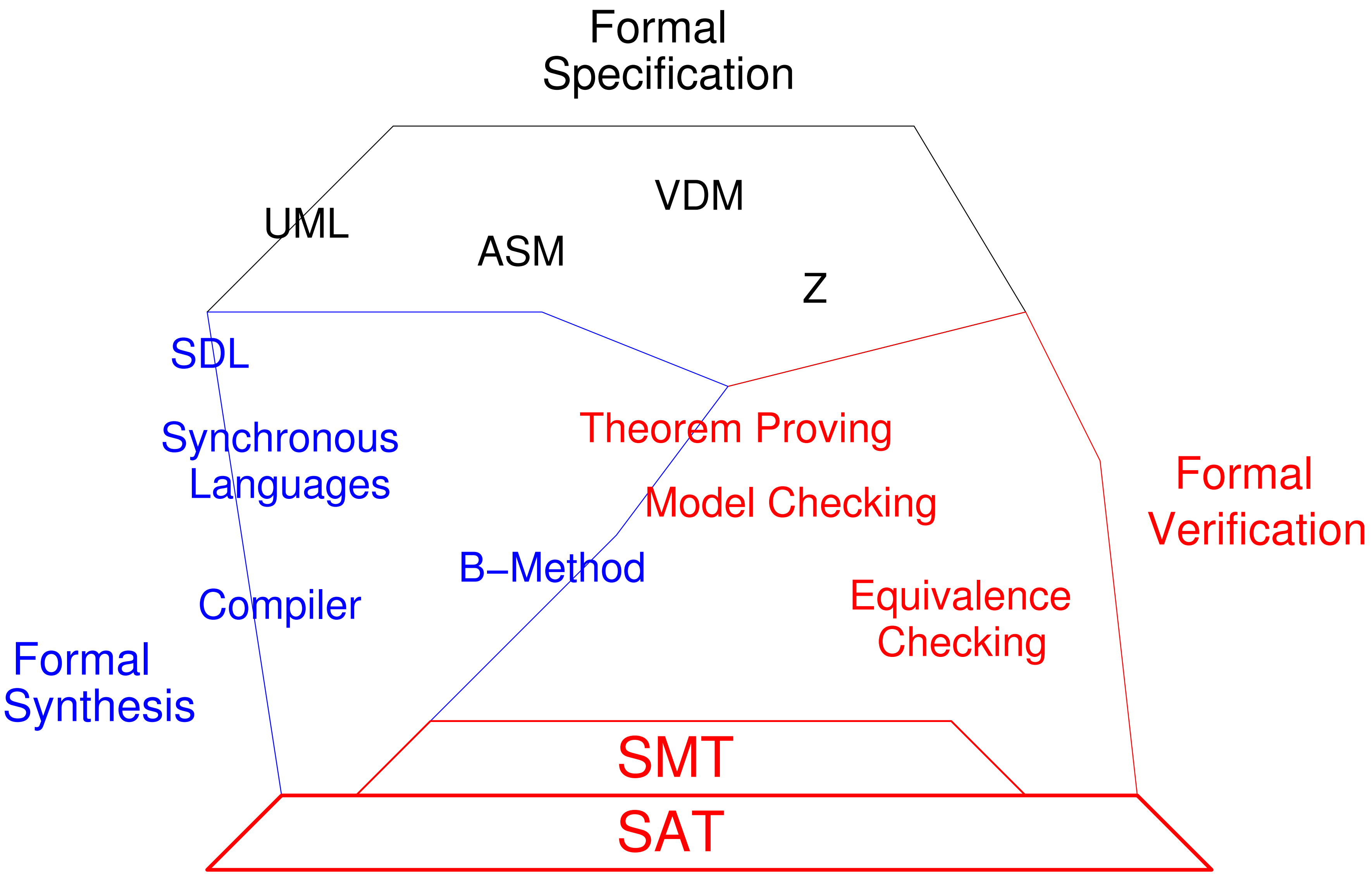
ZChaff, MiniSAT, My Solvers



Formal Methods and Satisfiability (SAT)



SAT as Core Technology



Dress Code of a Summer Shool Speaker

- 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 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 PrecoSAT, Lingeling, CryptoMiniSAT, ... [JärvisaloHeuleBiere'12]

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 with and without arrays for HW/SW verification
  - our SMT solver Boolector ranked #1 in these categories (SMT 2008/2009/2012)

Applications of SAT and SMT

- *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

$$\underset{\text{satisfiable}}{\forall \mathbf{tie}[\exists \mathbf{shirt}[\overbrace{(\mathbf{tie} \lor \mathbf{shirt}) \land (\neg \mathbf{tie} \lor \neg \mathbf{shirt})}^{\mathbf{tie} \neq \mathbf{shirt}}]]} \quad \not\equiv \quad \underset{\text{unsatisfiable}}{\exists \mathbf{tie}[\forall \mathbf{shirt}[\overbrace{(\mathbf{tie} \lor \mathbf{shirt}) \land (\neg \mathbf{tie} \lor \neg \mathbf{shirt})}^{\mathbf{tie} \neq \mathbf{shirt}}]]}$$

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

Tic-Tac-Toe

s_0

s_1

	O	

s_2

	O	
X		

s_3

O	O	
X		

s_4

O	O	X
X		

s_5

O	O	X
X		O

s_6

X		
O	O	X
X		O

s_7

X	O	
O	O	X
X		O

s_8

X	O	
O	O	X
X	X	O

s_9

X	O	O
O	O	X
X	X	O

No Winning Strategy for Tic-Tac-Toe

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

x_i, y_i plays (4 bits each)

s_i configurations

(9×3 bits each)

Example: Equivalence Checking If-then-else Chains

original code

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

⇓

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

⇒

optimized code

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

⇑

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

How to check that these two versions are equivalent?

SAT Example cont.

1. represent procedures as *independent* boolean variables

$$\textit{original} := \qquad\qquad \textit{optimized} :=$$

$$\begin{array}{ll}
\textbf{if } \neg a \wedge \neg b \textbf{ then } h & \textbf{if } a \textbf{ then } f \\
\textbf{else if } \neg a \textbf{ then } g & \textbf{else if } b \textbf{ then } g \\
\textbf{else } f & \textbf{else } h
\end{array}$$

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

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

Compilation

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

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

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

How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

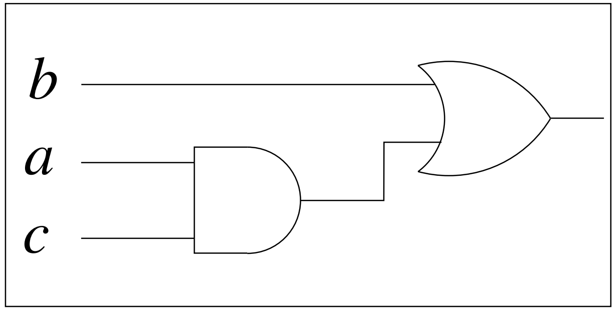
> Is there an assignment to a, b, f, g, h,
> which results in different evaluations of $\mathit{original}$ and $\mathit{optimized}$?

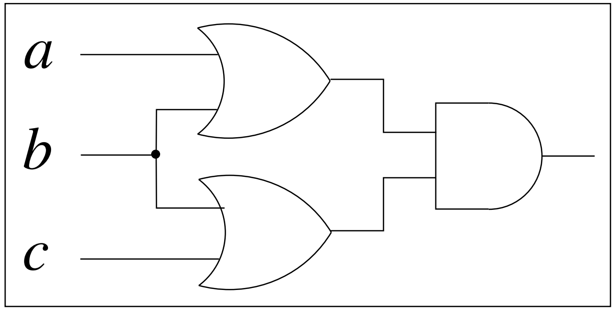
or equivalently:

> 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





$b \vee a \wedge c$

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

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

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

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

```
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
```

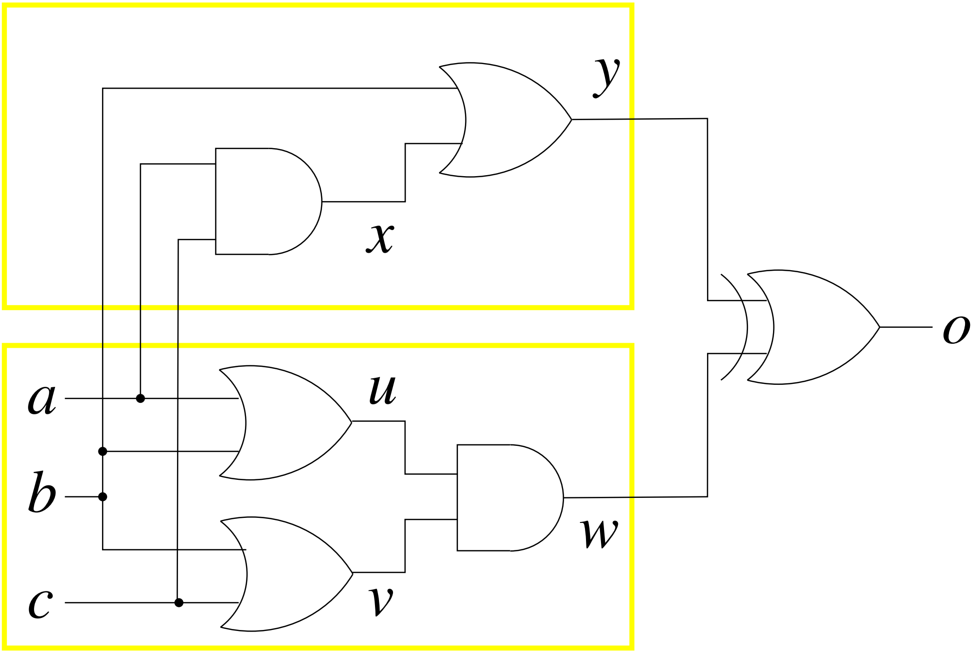
this is the *only one* since “assuming” the opposite values individually is UNSAT

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

Example of Tseitin Transformation: Circuit to CNF

[Tseitin'68]

CNF



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

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

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

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

Tseitin Transformation: Input / Output Constraints

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\Leftrightarrow (x \rightarrow \overline{y}) \land (\overline{y} \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor \overline{y}) \land (y \lor x) \end{aligned}$$

Disjunction:

$$\begin{aligned} x \leftrightarrow (y \lor z) &\Leftrightarrow (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \\ &\Leftrightarrow (\overline{y} \lor x) \land (\overline{z} \lor x) \land (\overline{x} \lor y \lor z) \end{aligned}$$

Conjunction:

$$\begin{aligned} x \leftrightarrow (y \land z) &\Leftrightarrow (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor y) \land (\overline{x} \lor z) \land (\overline{(y \land z)} \lor x) \\ &\Leftrightarrow (\overline{x} \lor y) \land (\overline{x} \lor z) \land (\overline{y} \lor \overline{z} \lor x) \end{aligned}$$

Equivalence:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow ((y \rightarrow z) \land (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow (y \rightarrow z)) \land (x \rightarrow (z \rightarrow y)) \land ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (((y \land z) \lor (\overline{y} \land \overline{z})) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land ((y \land z) \rightarrow x) \land ((\overline{y} \land \overline{z}) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor \overline{z} \lor y) \land (\overline{y} \lor \overline{z} \lor x) \land (y \lor z \lor x) \end{aligned}$$

Optimizations for Tseitin Transformation

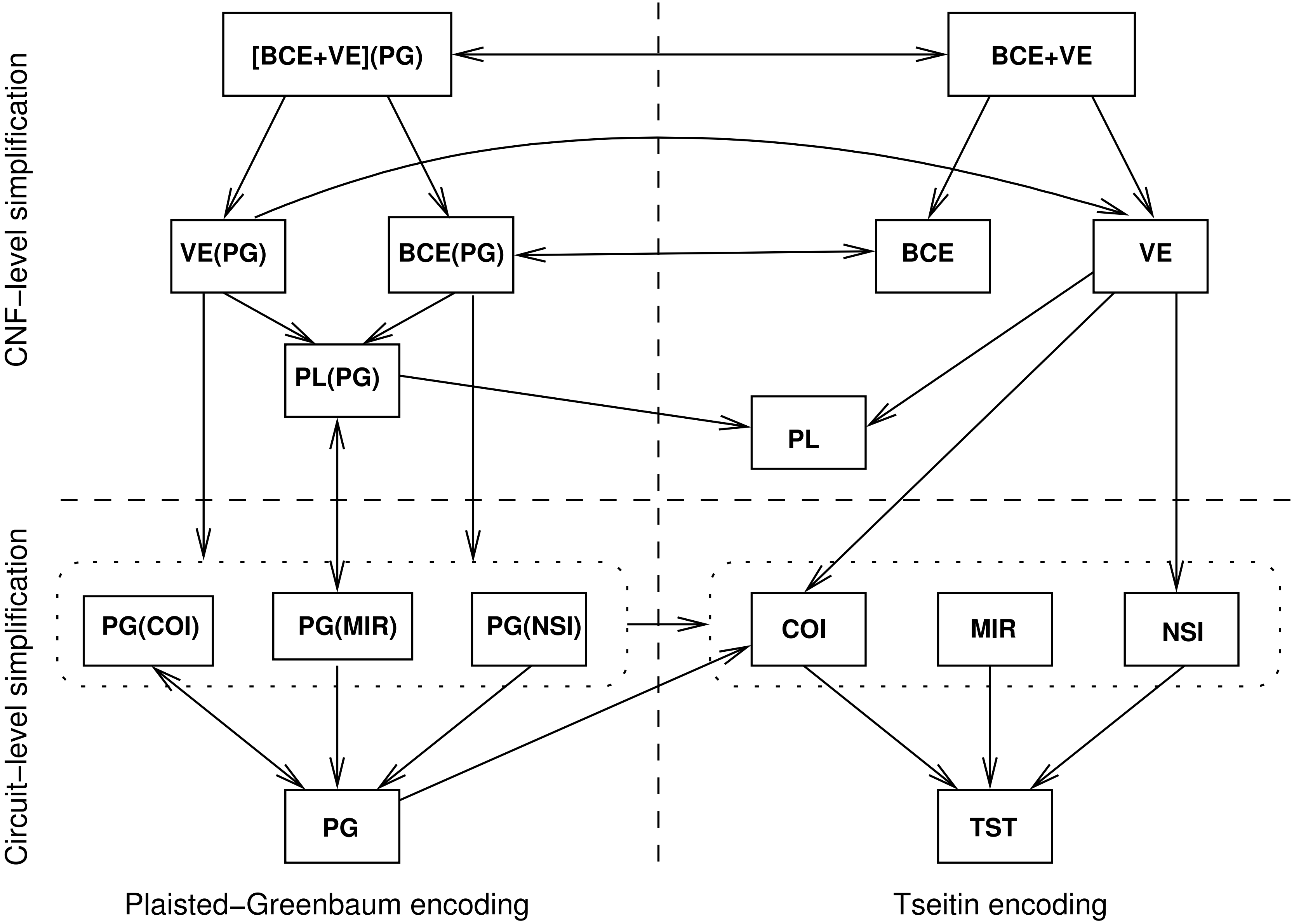
- goal is smaller CNF less variables, less clauses, so easier to solve (?!)

- 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: $s = x + y$

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

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

where

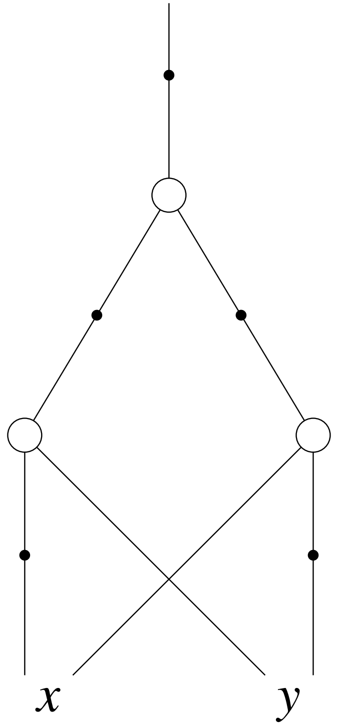
$$\begin{aligned}
[\, s \,, o \,]_2 &= \text{FullAdder}(x, y, i) \quad \text{with} \\
s &= x \text{ xor } y \text{ xor } i \\
o &= (x \wedge y) \vee (x \wedge i) \vee (y \wedge i) \;=\; ((x + y + i) \geq 2)
\end{aligned}$$

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

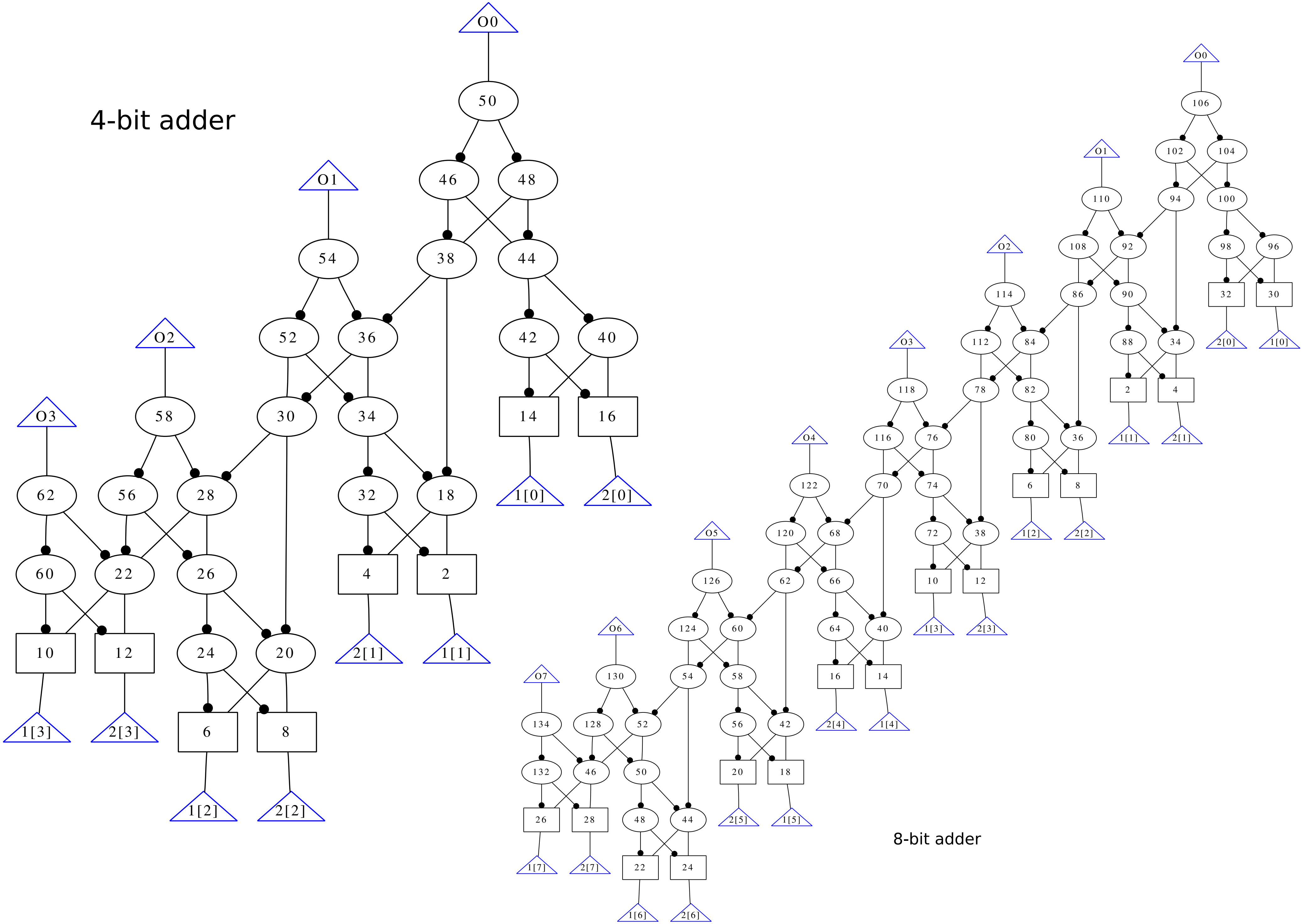
$$x \text{ xor } y \;\equiv\; (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \;\equiv\; \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

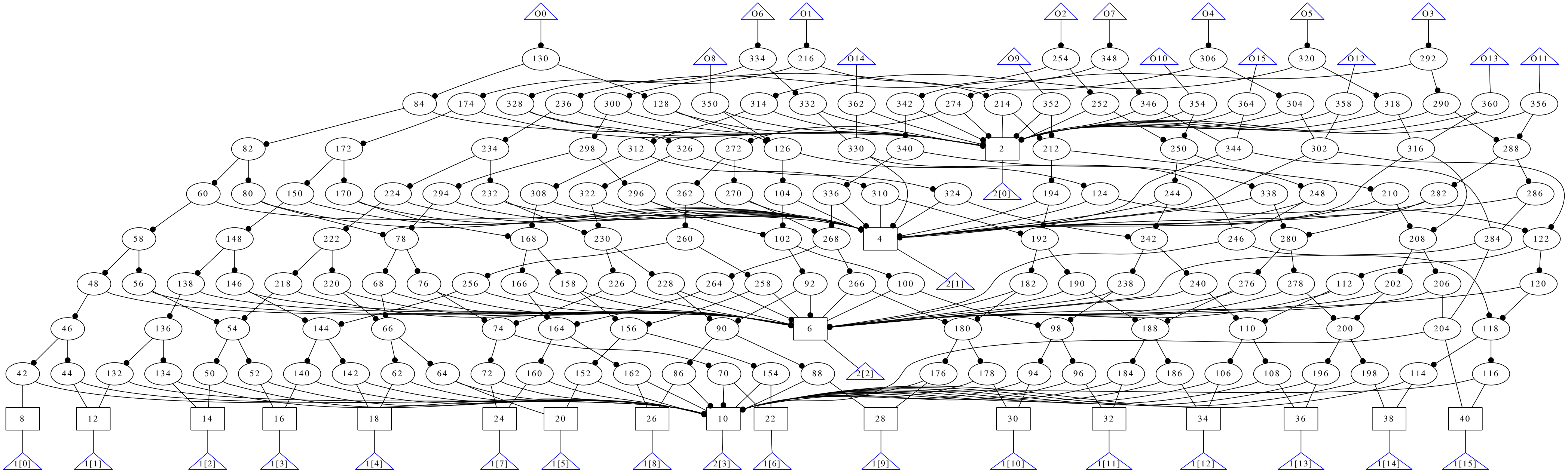
Bit-Stuffing Techniques for AIGs in C

```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)
```

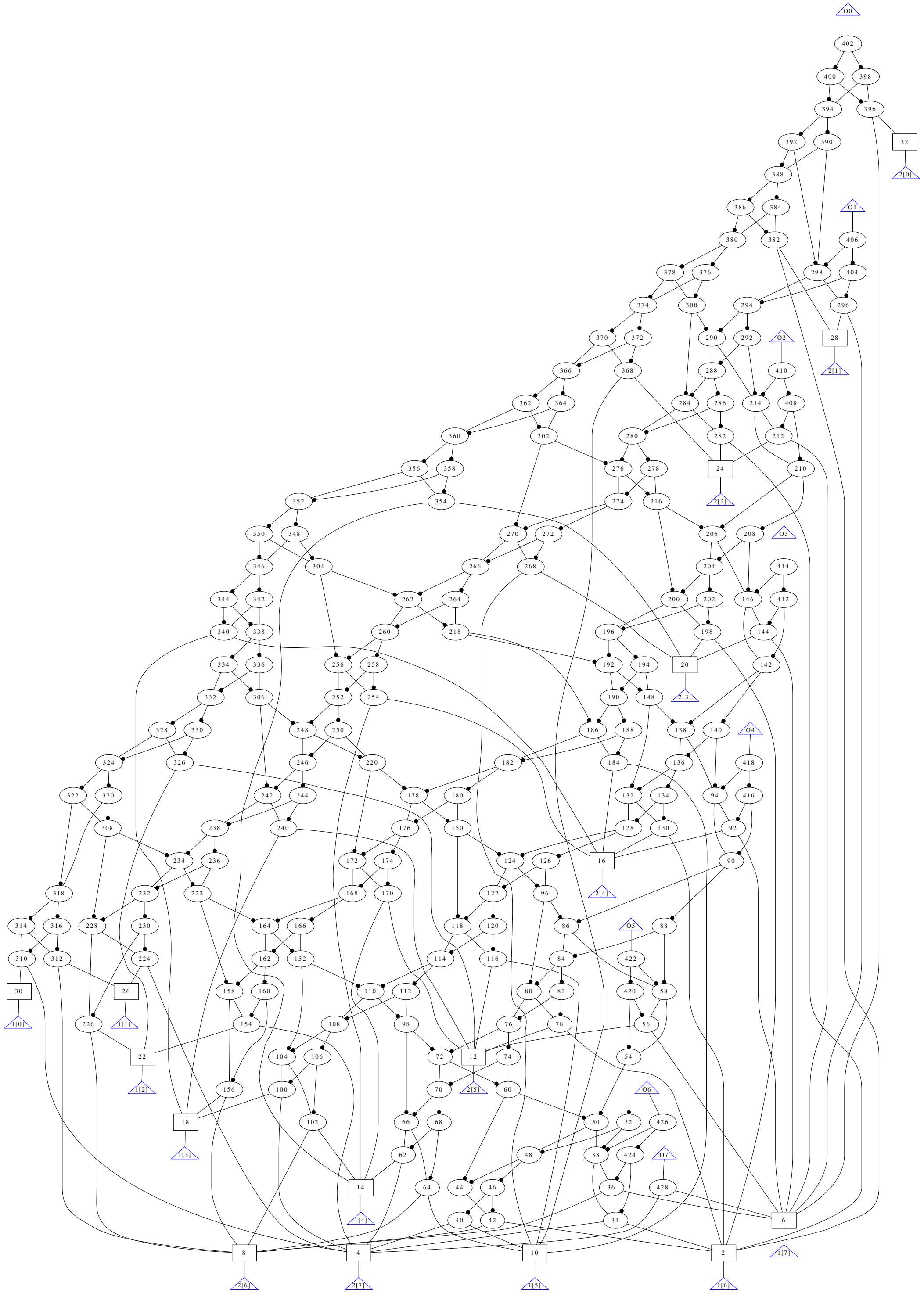
assumption for correctness:

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


4-bit adder
O0
50
46
48
O1
54
38
44
O2
52
36
42
40
O3
58
30
34
14
16
62
56
28
32
18
1[0]
2[0]
60
22
26
4
2
10
12
24
20
2[1]
1[1]
1[3]
2[3]
6
8
1[2]
2[2]
O0
106
102
104
O1
110
94
100
O2
108
92
98
96
114
86
90
32
30
O3
112
84
88
34
2[0]
1[0]
118
78
82
2
4
O4
116
76
80
36
1[1]
2[1]
122
70
74
6
8
O5
120
68
72
38
1[2]
2[2]
126
62
66
10
12
O6
124
60
64
40
1[3]
2[3]
O7
130
54
58
16
14
134
128
52
56
42
2[4]
1[4]
132
46
50
20
18
26
28
48
44
2[5]
1[5]
8-bit adder
1[7]
2[7]
22
24
1[6]
2[6]



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: $x = (a \vee y), \quad y = (b \vee x)$
  - 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

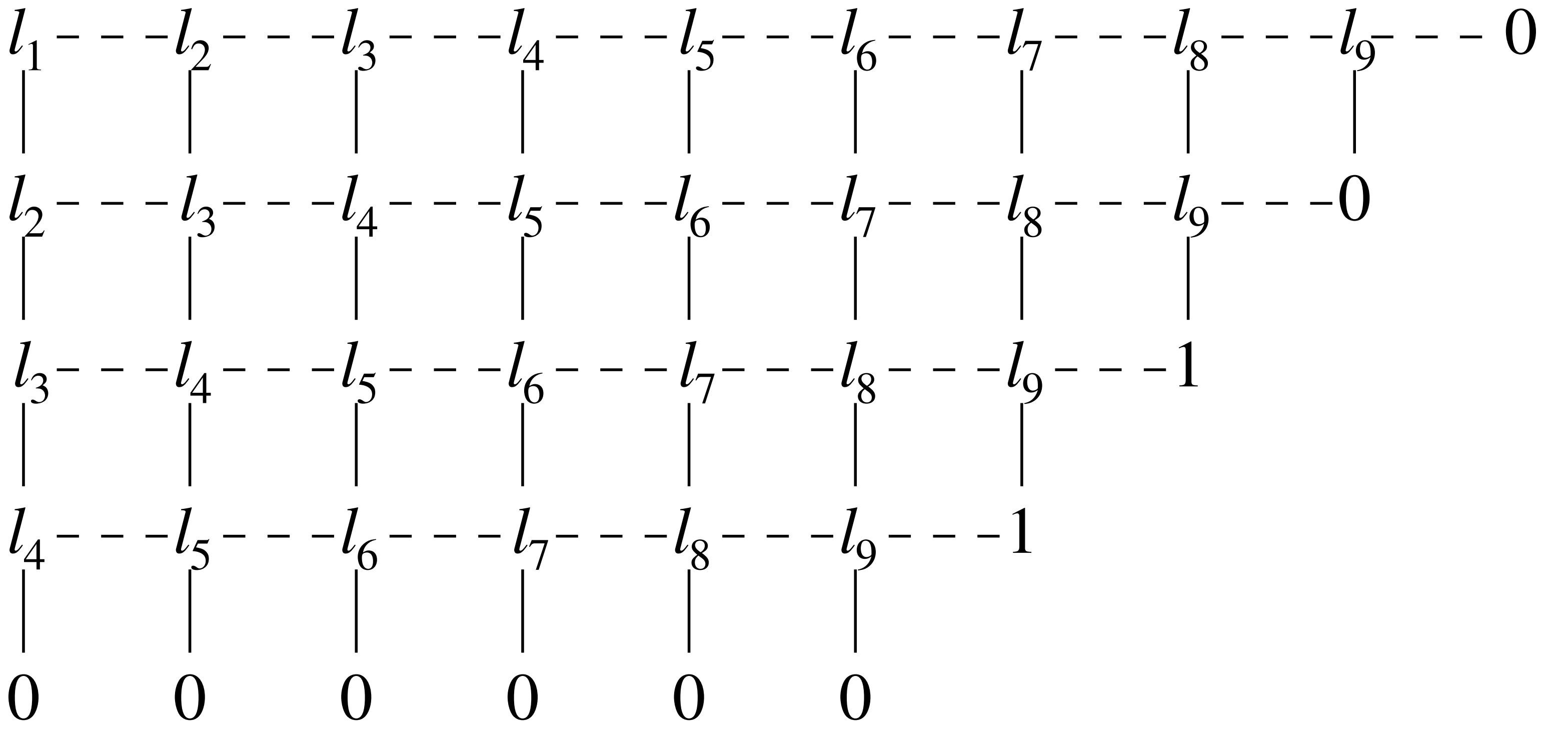
Example of Logical Constraints: Cardinality Constraints

- 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

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

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

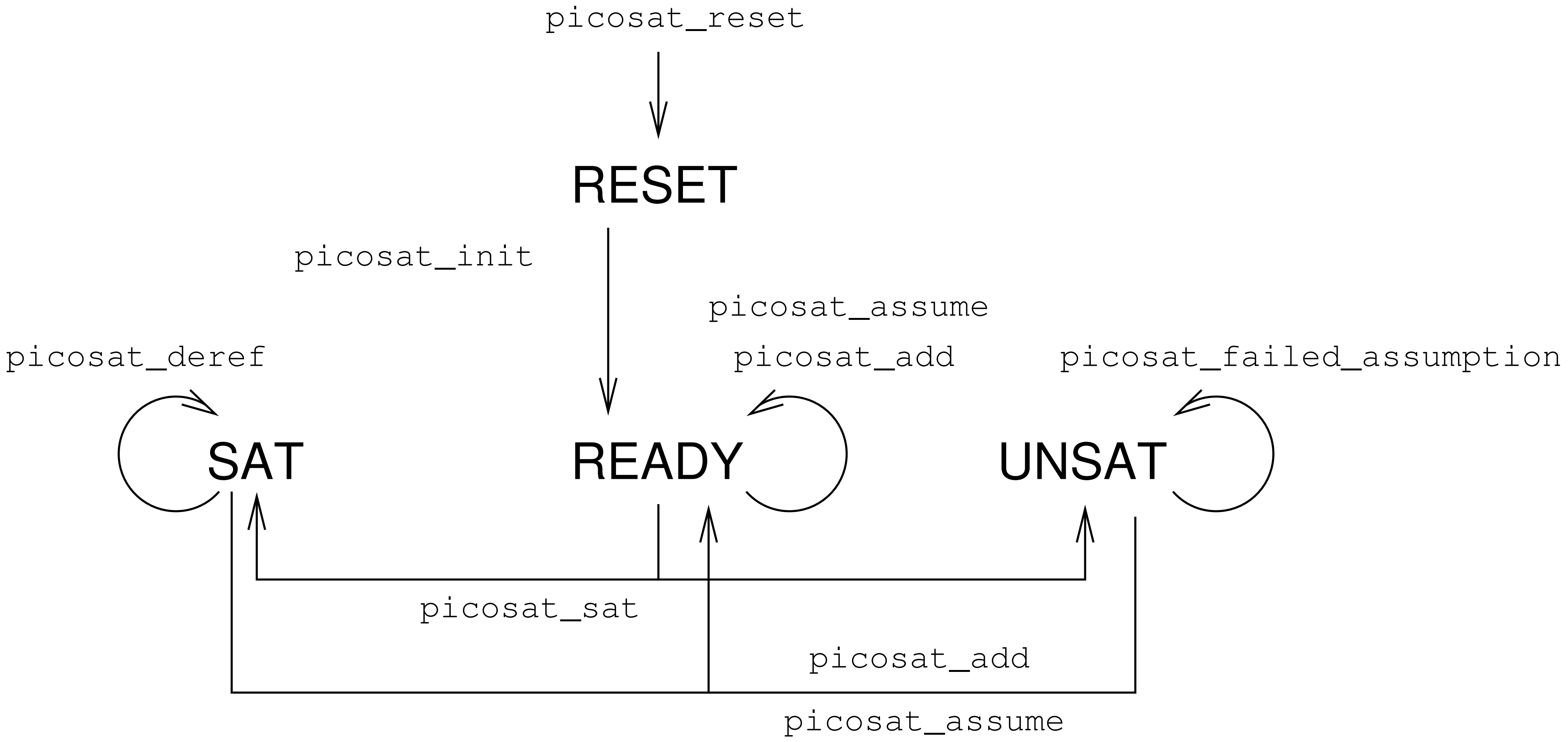
Example 2 with PicoSAT API

```
// 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

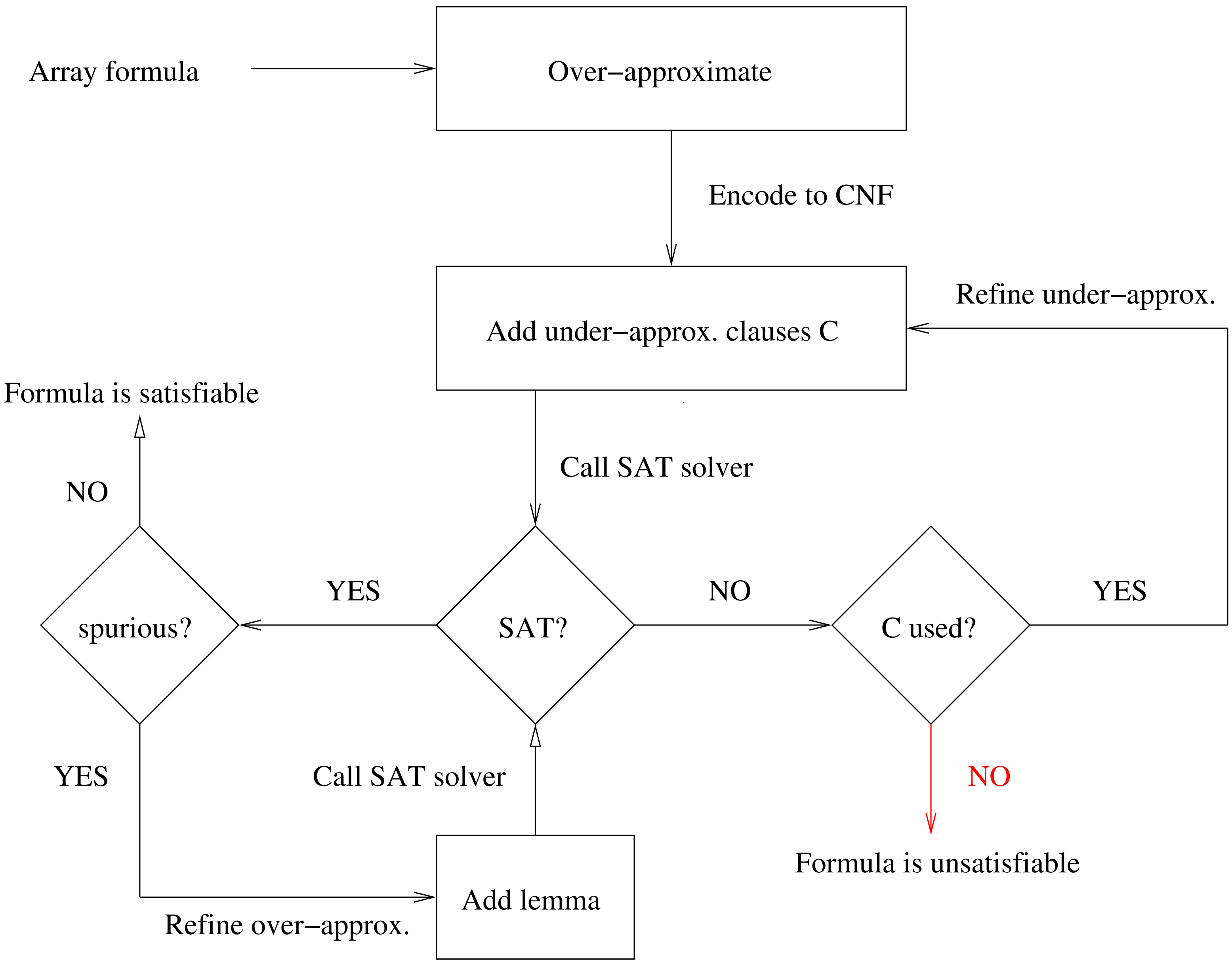
  otherwise increase eff. width of bit-vectors where it was used to derive UNSAT

  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

Boolector: Lemmas-on-Demand + Underapproximation

[BrummayerBiere'09]



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 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);
```

DP / DPLL

- dates back to the 50'ies:

  1^{st} version DP is *resolution based* ⇒ SatELite preprocessor [EénBiere05]

  2^{st} version D(P)LL splits space for time ⇒ $\boxed{\text{CDCL}}$

- **ideas:**

  - 1st version: eliminate the two cases of assigning a variable in space or

  - 2nd version: case analysis in time, e.g. try $x = 0, 1$ in turn and recurse

- most successful SAT solvers are based on variant (CDCL) of the second version

  works for very large instances

- recent ($\leq$ 15 years) optimizations:

  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures

  (we will have a look at each of them)

DP Procedure

[DavisPutnam'61]

```
forever
    if $F = \top$ **return** *satisfiable*
    if $\bot \in F$ **return** *unsatisfiable*
    pick remaining variable $x$
    add all resolvents on $x$
    remove all clauses with $x$ and $\neg x$
```

⇒ SatELite preprocessor [EénBiere05]

D(P)LL Procedure

[DavisLogemannLoveland'62]

```
$DPLL(F)$

    $F := BCP(F)$                                        boolean constraint propagation

    if $F = \top$ return satisfiable

    if $\bot \in F$ return unsatisfiable

    pick remaining variable $x$ and literal $l \in \{x, \neg x\}$

    if $DPLL(F \wedge \{l\})$ returns satisfiable return satisfiable

    return $DPLL(F \wedge \{\neg l\})$
```

⇒ CDCL

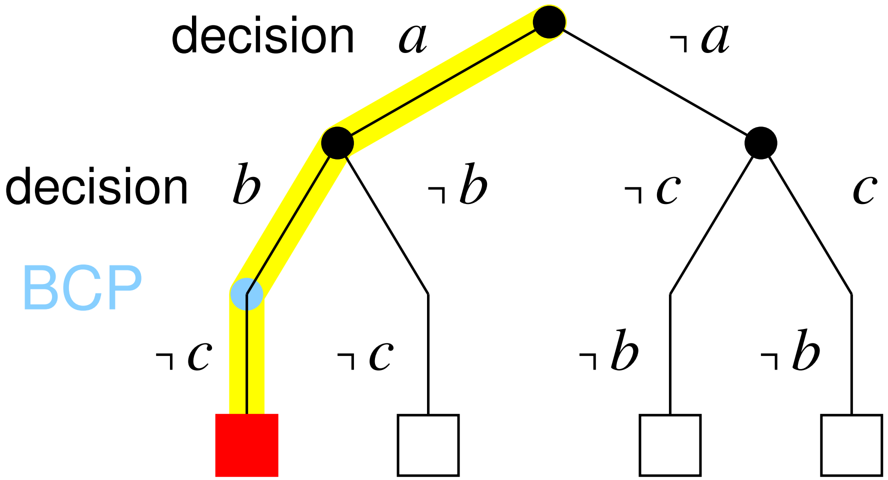
DPLL Example

[DavisLogemannLoveland'62]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

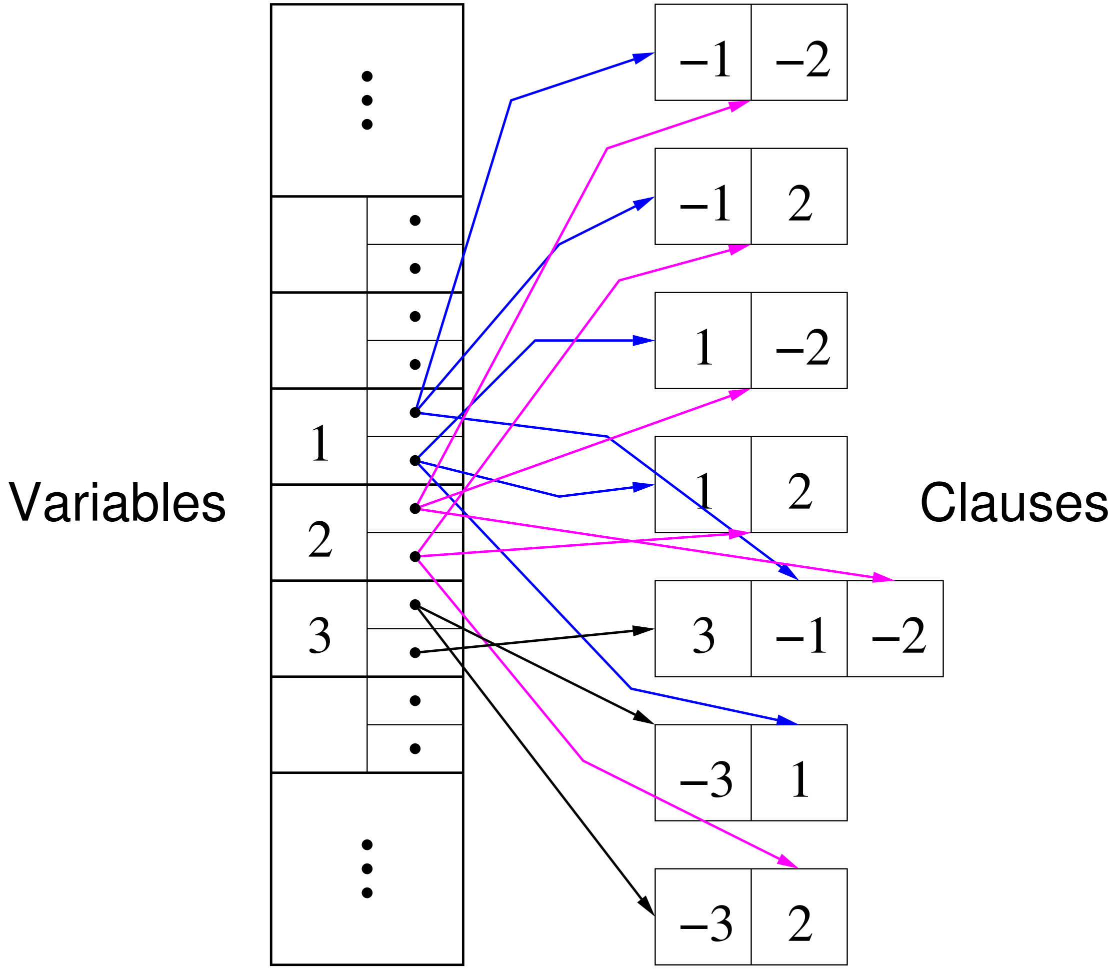
$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

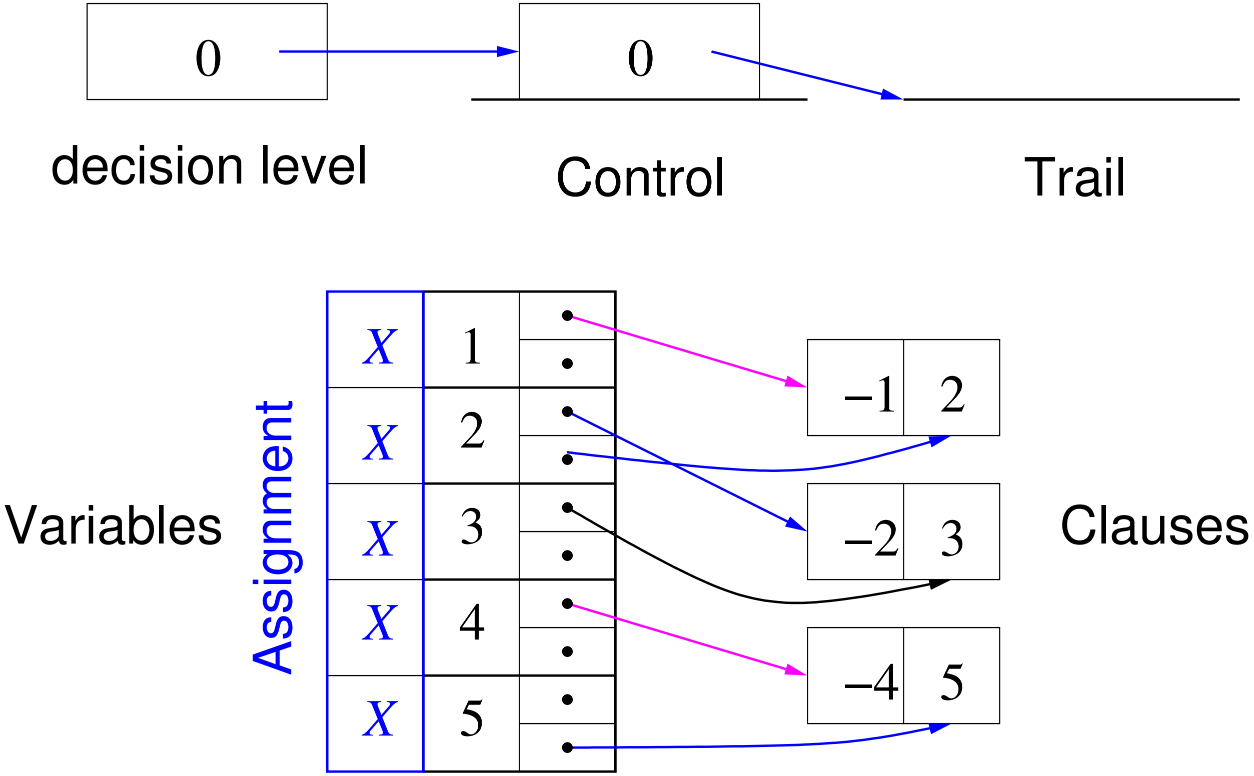
Simple Data Structures in DPLL Implementation

[DavisLogemannLoveland'62]



BCP Example

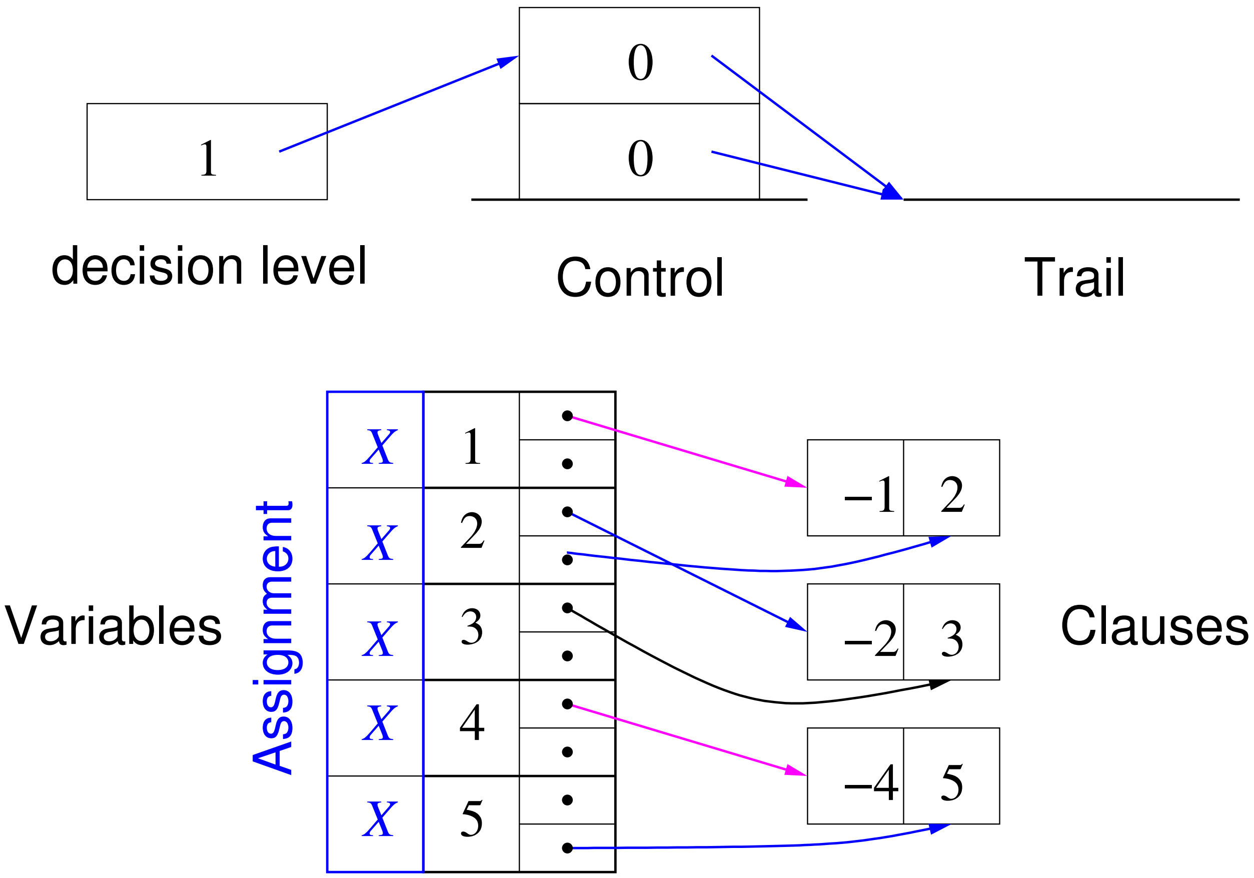
[DavisLogemannLoveland'62]



Example cont.

[DavisLogemannLoveland'62]

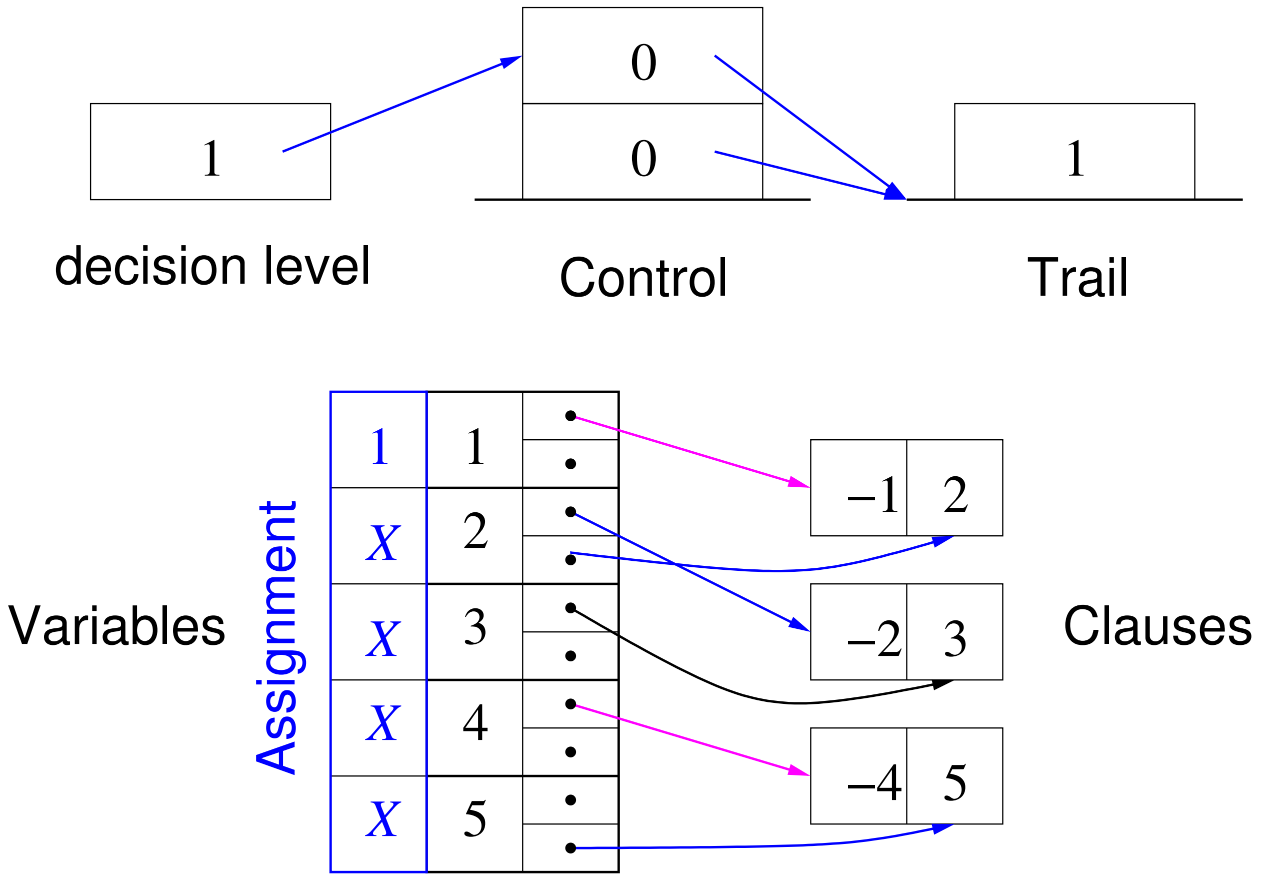
Decide



Example cont.

[DavisLogemannLoveland'62]

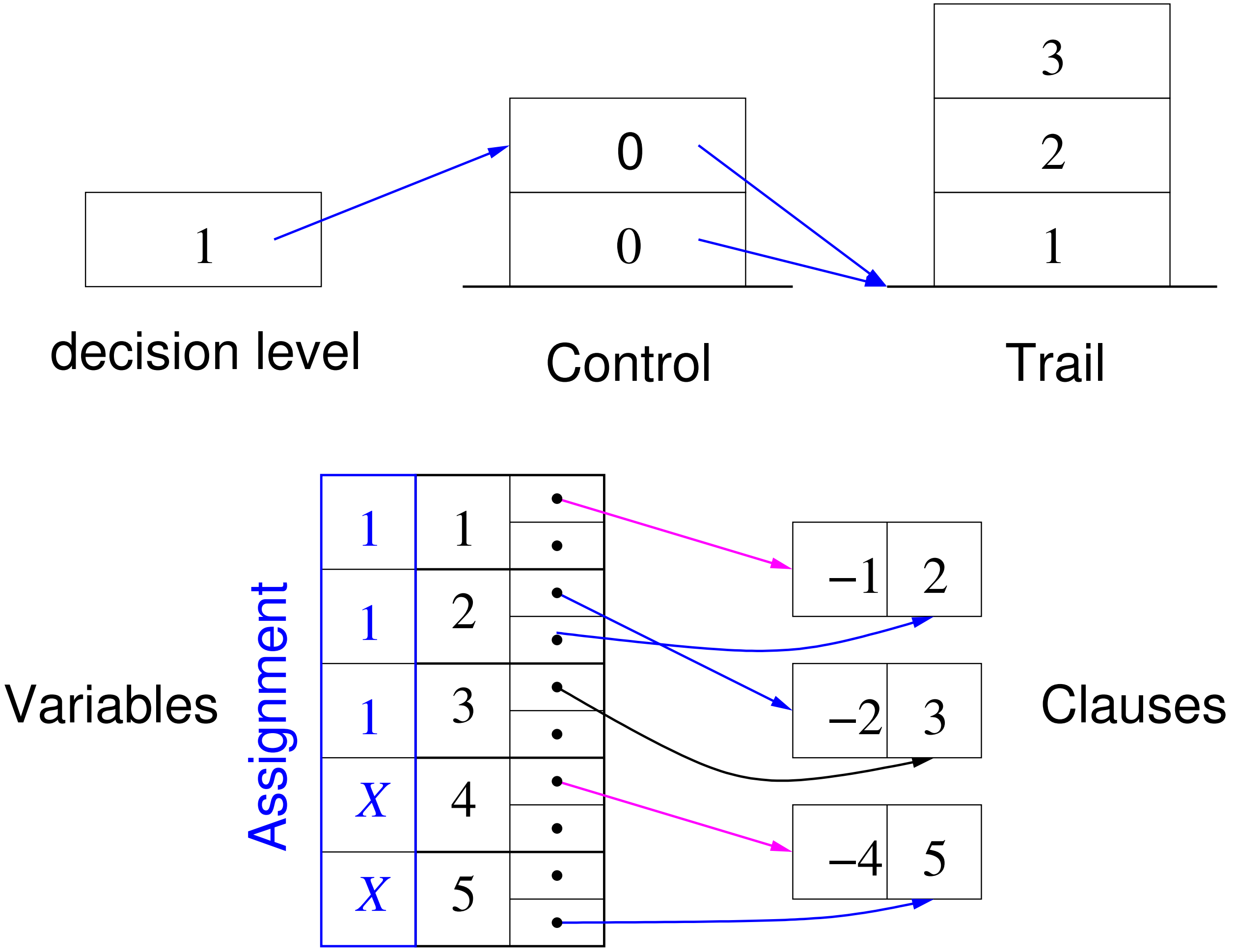
Assign



Example cont.

[DavisLogemannLoveland'62]

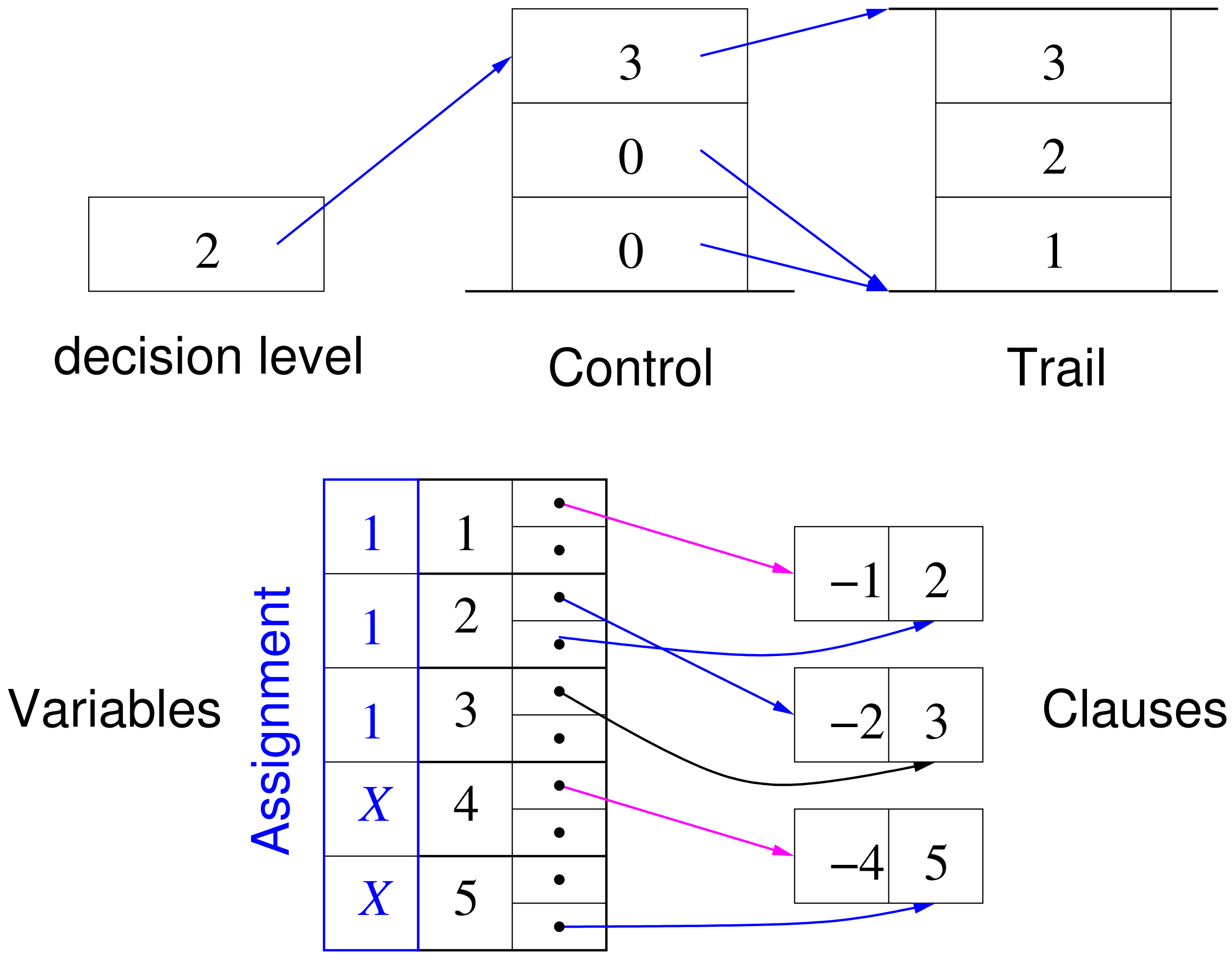
BCP



Example cont.

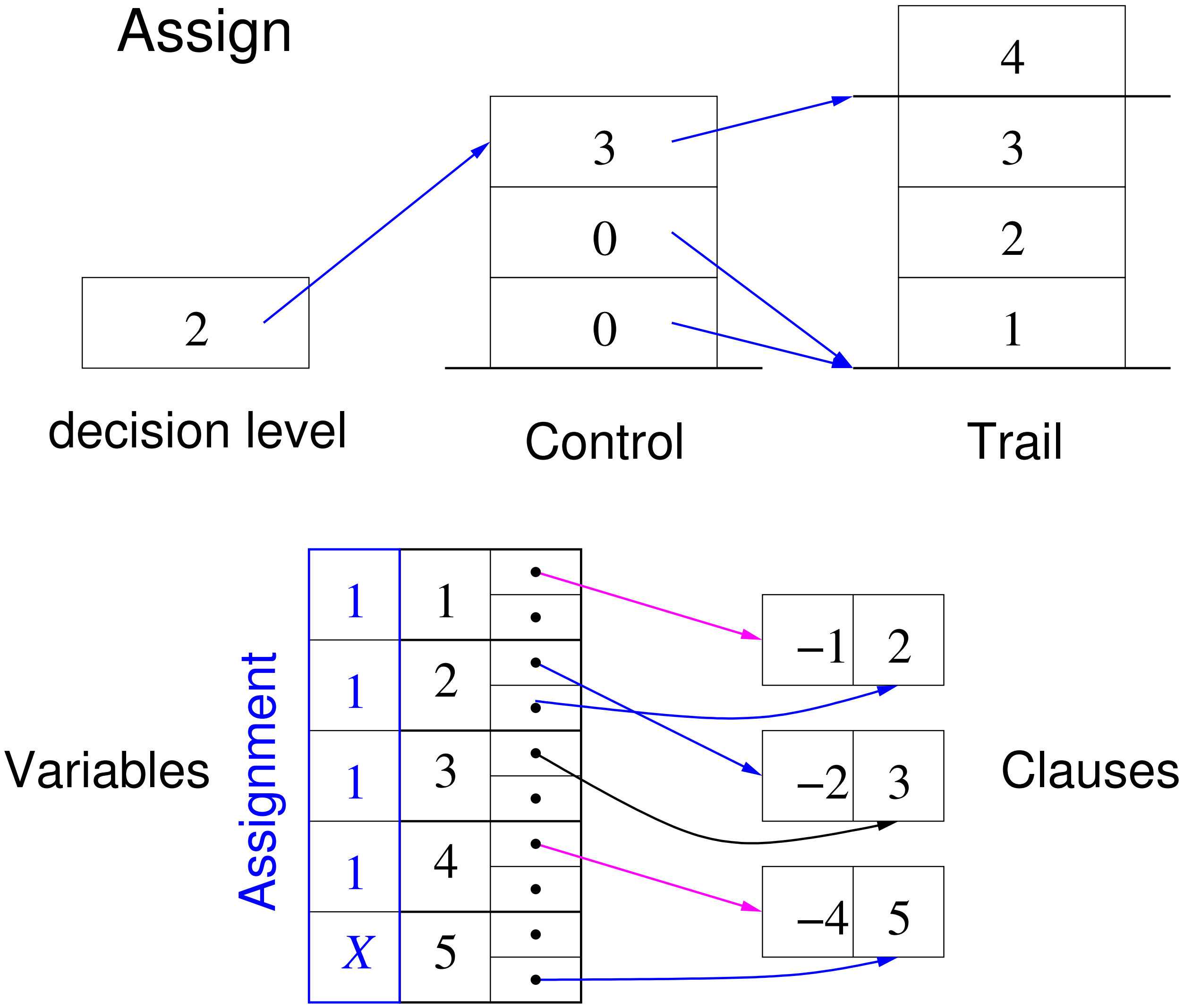
[DavisLogemannLoveland'62]

Decide



Example cont.

[DavisLogemannLoveland'62]



Assign

decision level

Control

Trail

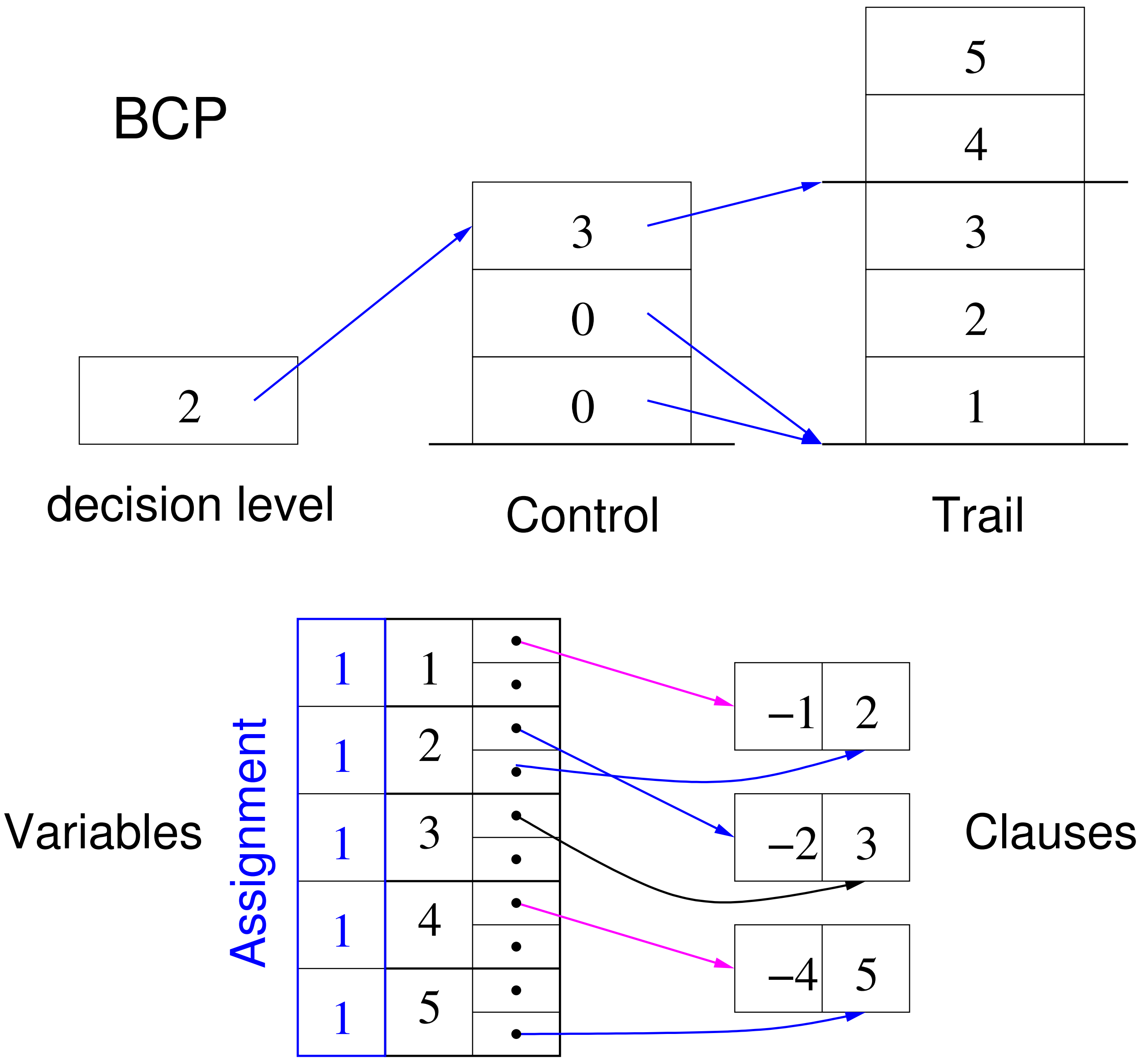
Variables

Assignment

Clauses

Example cont.

[DavisLogemannLoveland’62]



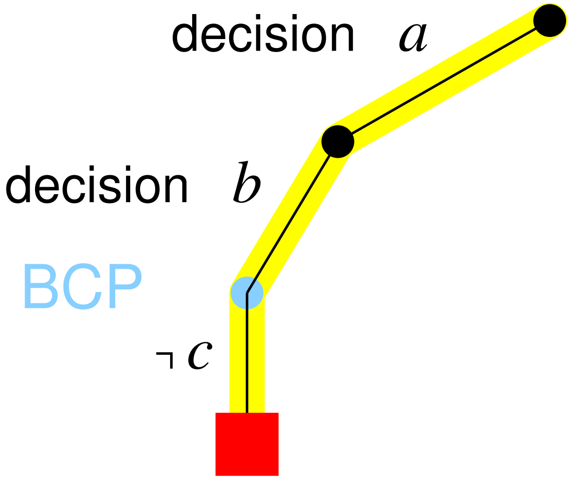
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah’96]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

learn $\neg a \vee \neg b$

Conflict Driven Clause Learning (CDCL)

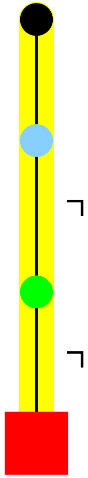
Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$

decision a



$\neg b$ BCP

$\neg c$ BCP

clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

learn $\neg a$

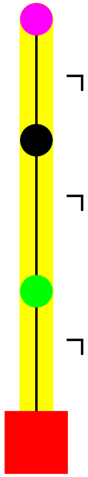
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah’96]

$a = 1$

$b = 0$

$c = 0$



$\neg a$ BCP

$\neg c$ decision

$\neg b$ BCP

clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

learn c

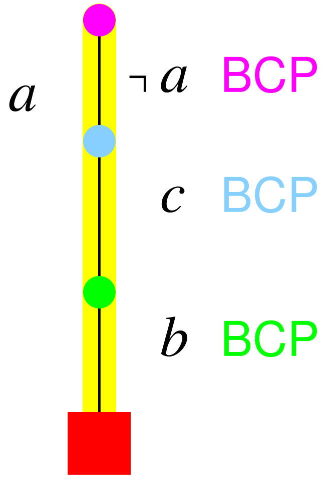
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah’96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

c

learn $\bot$

empty clause

Decision Heuristics

- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast to compute, since only calculated once
  - and thus can also use more expensive algorithms

- **dynamic heuristics**
  - typically calculated from number of occurences of literals (in unsatisfied clauses)
  - could be rather expensive, since it requires traversal of all clauses (or more expensive updates in BCP)
  - effective *second order* dynamic heuristics (e.g. VSIDS in Chaff)

Other popular Decision Heuristics

- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic *first order* heuristic (e.g. GRASP solver)
  - choose literal (variable + phase) which occurs most often (ignore satisfied clauses)
  - requires explicit traversal of CNF (or more expensive BCP)

- look-ahead heuristics (e.g. SATZ or MARCH solver) **failed literals, probing**
  - trial assignments and BCP for all/some unassigned variables (both phases)
  - if BCP leads to conflict, enforce toggled assignment of current trial decision
  - optionally learn binary clauses and perform equivalent literal substitution
  - decision: most balanced w.r.t. prop. assignments / sat. clauses / reduced clauses
  - related to our recent Cube & Conquer paper [HeuleKullmanWieringaBiere'11]

Exponential VSIDS (EVSIDS)

Chaff [MoskewiczMadiganZhaoZhangMalik’01]

- increment score of involved variables by 1
- decay score of all variables every 256’th conflict by halfing the score
- sort priority queue after decay and not at every conflict

MiniSAT uses EVSIDS [EénSörensson’03/’06]

- update score of involved variables as actually LIS would also do
- dynamically adjust increment: $\delta' = \delta \cdot \frac{1}{f}$ δ typically increment by $5\% - 11\%$
- use floating point representation of score
- “rescore” to avoid overflow in regular intervals
- EVSIDS linearly related to NVSIDS

Normalized VSIDS: NVSIDS

- VSIDS score can be normalized to the interval [0,1] as follows:
  - pick a decay factor f per conflict: typically $f = 0.9$
  - each variable is punished by this decay factor at every conflict
  - if a variable is involved in conflict, add $1 - f$ to score
  - s old score of one fixed variable before conflict, s' new score after conflict

$$\boxed{s, f \leq 1, \quad \text{then} \quad s' \leq \overbrace{s \cdot f}^{\text{decay in any case}} \underbrace{+1 - f}_{\text{increment if involved}} \leq f + 1 - f = 1}$$

- recomputing score of all variables at each conflict is costly
  - linear in the number of variables, e.g. millions
  - particularly, because number of involved variabels $<<$ number of variables

Relating EVSIDS and NVSIDS

consider again only one variable with score sequence s_n resp. S_n

$$\delta_k = \begin{cases} 1 & \text{if involved in } k\text{-th conflict} \\ 0 & \text{otherwise} \end{cases}$$

$$i_k = (1-f)\cdot\delta_k$$

$$s_n = \boxed{(\ldots(i_1\cdot f + i_2)\cdot f + i_3)\cdot f \cdots)\cdot f + i_n} = \sum_{k=1}^{n} i_k\cdot f^{n-k} = (1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} \qquad \text{(NVSIDS)}$$

$$S_n = \frac{f^{-n}}{(1-f)}\cdot s_n = \frac{f^{-n}}{(1-f)}\cdot(1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} = \sum_{k=1}^{n}\delta_k\cdot f^{-k} \qquad \text{(EVSIDS)}$$

BerkMin's Dynamic Second Order Heuristics

[GoldbergNovikov-DATE'02]

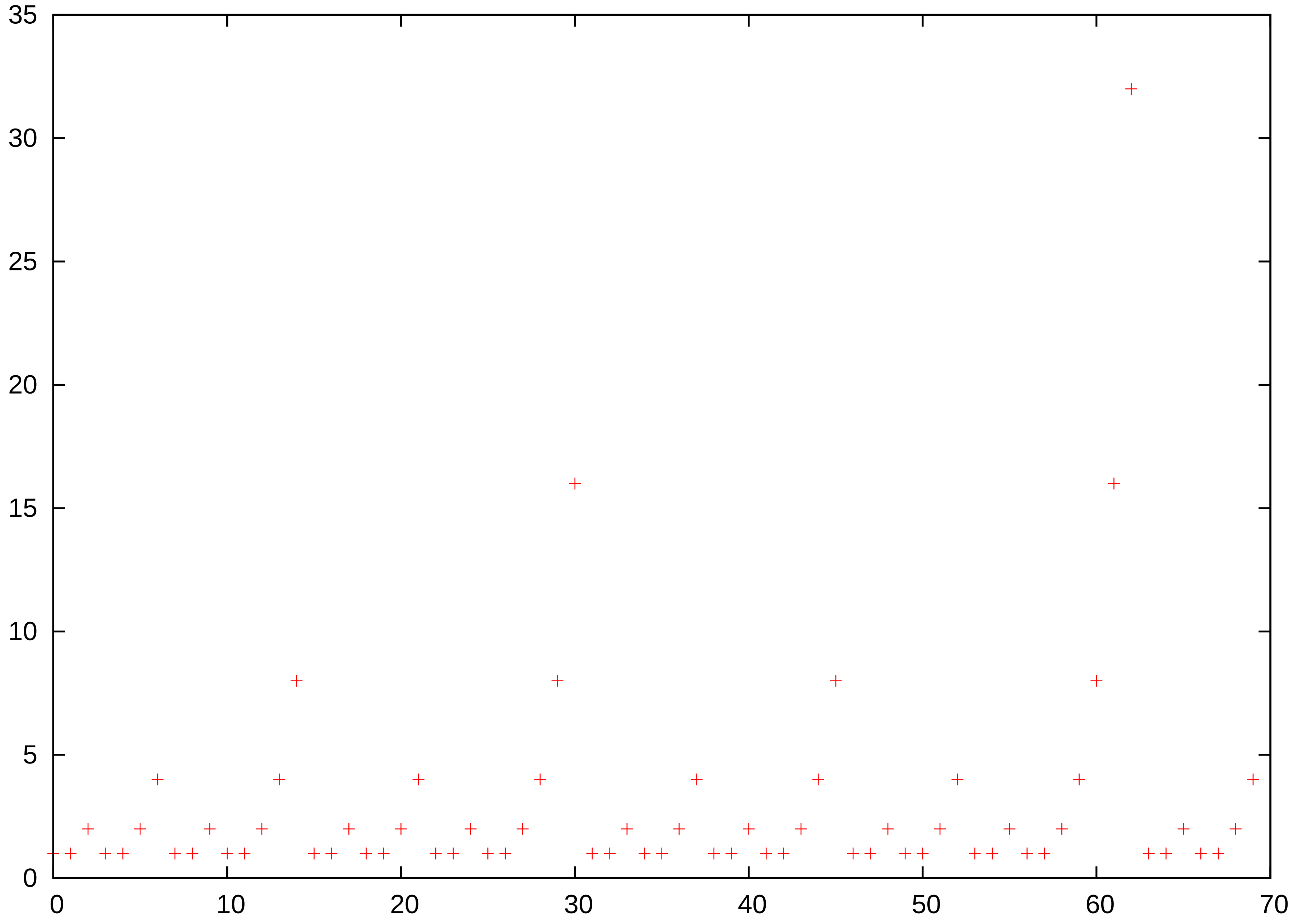
- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to choose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
- mixed results as other variants VMTF, CMTF (var/clause move to front)

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
  - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
  - restart after the number of decisions reached a *restart limit*
- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit

Luby’s Restart Intervals

70 restarts in 104448 conflicts



Luby Restart Scheduling

```
unsigned
luby (unsigned i)
{
  unsigned k;

  for (k = 1; k < 32; k++)
    if (i == (1 << k) - 1)
      return 1 << (k - 1);

  for (k = 1;; k++)
    if ((1 << (k - 1)) <= i && i < (1 << k) - 1)
      return luby (i - (1 << (k-1)) + 1);
}

limit = 512 * luby (++restarts);
...  // run SAT core loop for ’limit’ conflicts
```

Reluctant Doubling Sequence

[Knuth'12]

$$
\begin{aligned}
(u_1, v_1) &:= (1, 1) \\
(u_{n+1}, v_{n+1}) &:= (u_n \mathbin{\&} -u_n = v_n \mathrel{?} (u_n + 1, 1) : (u_n, 2v_n))
\end{aligned}
$$

$$(1,1),\ (2,1),\ (2,2),\ (3,1),\ (4,1),\ (4,2),\ (4,4),\ (5,1),\ \ldots$$

Phase Saving and Rapid Restarts

- phase assignment / direction heuristics:
  - assign decision variable to 0 or 1?
  - only thing that matters in *satisfiable* instances
- "phase saving" as in RSat:
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)
  - so can be seen to maintain a *global full assignment*
- rapid restarts: varying restart interval with bursts of restarts
  - not ony theoretically avoids local minima
  - empirically works nice together with phase saving

Reusing the Trail

[Van Der Tak, Heule, Ramos POS'11]

- in general *restarting does not much change much*: since phases and scores saved
- assignment after restart can only differ if
  - before restarting
  - there is a decision literal d assigned on the trail
  - with smaller score than the next decision n on the priority queue
- in this situation backtrack **only** to decision level of d
  - simple to compute, particularly if decisions are saved separately
  - allows to skip many redundant backtracks
  - allows much higer restart frequency,
    e.g. base interval 10 for reluctant doubling sequence (Luby)

Reducing Learned Clauses

- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only “short” clauses

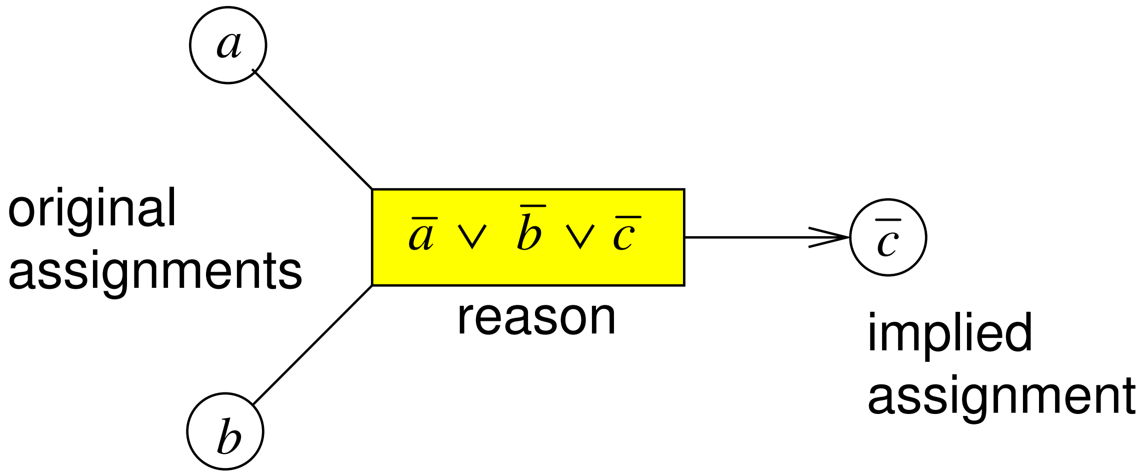
- better periodically delete “useless” learned clauses
  - keep a certain number of learned clauses — “search cache”
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - keep *most active*, then *shortest*, then *youngest* (FIFO) clauses
  - after reduction maximum number kept learned clauses is increased geometrically

- LBD (Glue) based (apriori!) prediction for usefullness — [AudemardSimon’09]
  - LBD (Glue) = number of decision-levels in the learned clause
  - allows arithmetic increase of number of kept learned clauses

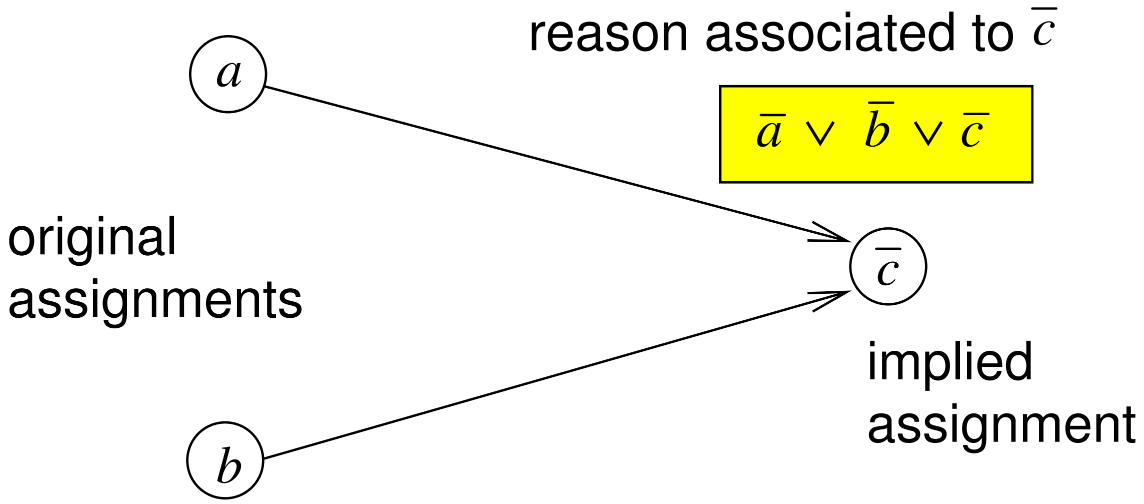
- freeze high PSM (dist. to phase assign.) clauses — [AudemardLagniezMazureSais’11]

General Implication Graph as Hyper-Graph

CDCL / Grasp [MarquesSilvaSakallah'96]



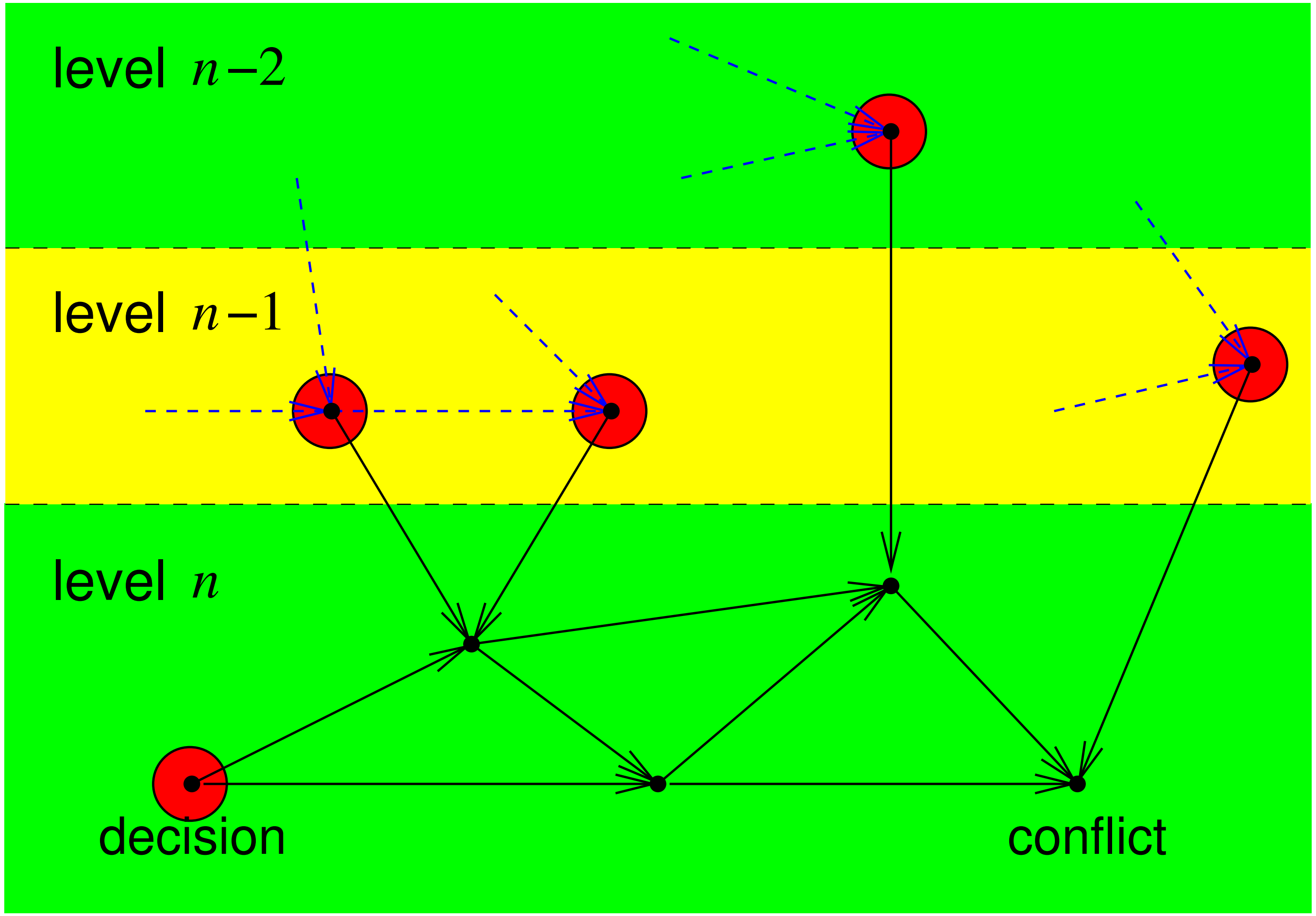
Implication Graph Standard Notation



reason associated to $\overline{c}$
$\overline{a} \vee \overline{b} \vee \overline{c}$
a
original assignments
b
$\overline{c}$
implied assignment

Conflict Clauses as Cuts in the Implication Graph

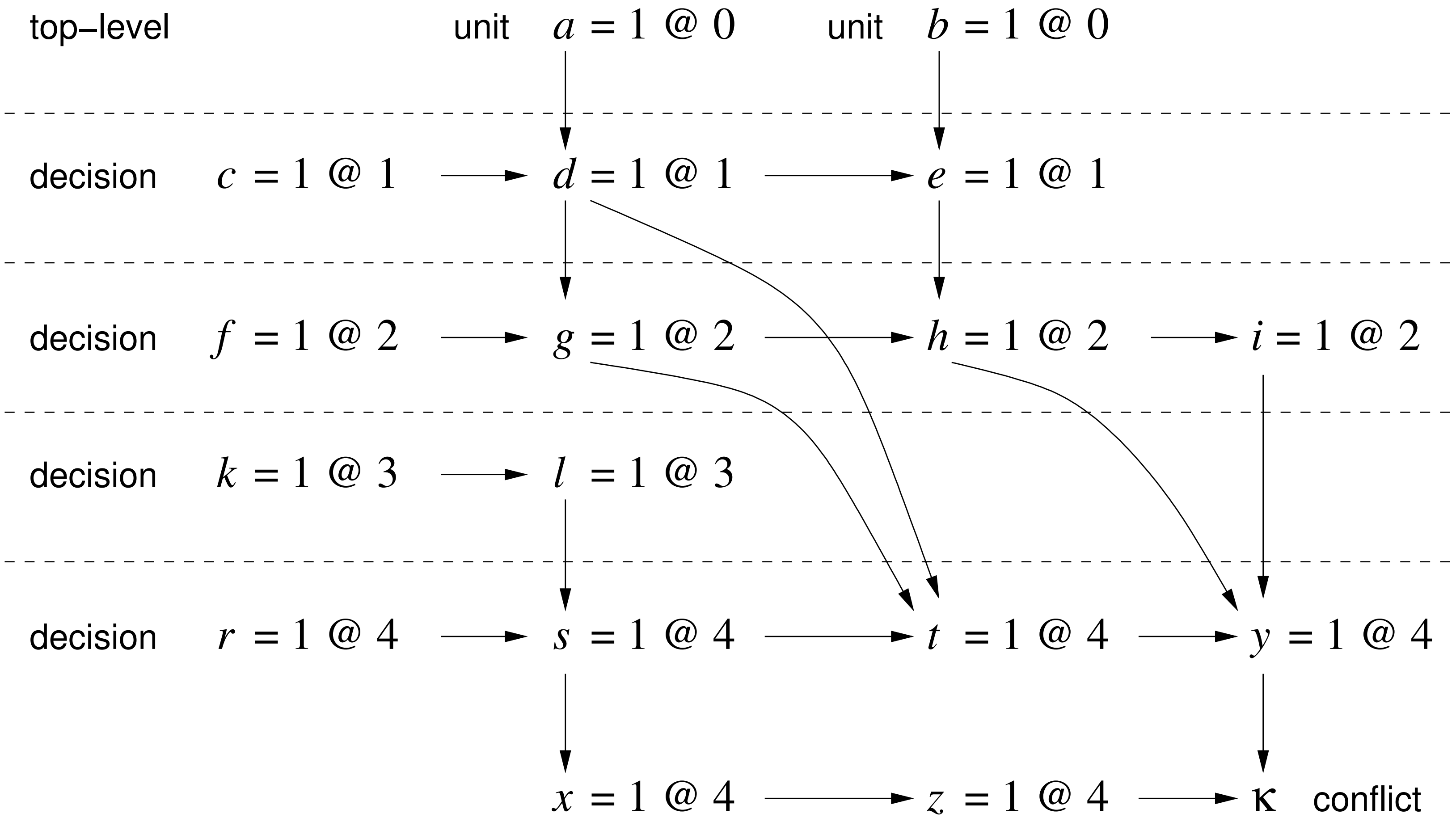
CDCL / Grasp [MarquesSilvaSakallah'96]



a simple cut always exists: set of roots (decisions) contributing to the conflict

Implication Graph

CDCL / Grasp [MarquesSilvaSakallah'96]



top−level unit $a = 1 @ 0$ unit $b = 1 @ 0$

decision $c = 1 @ 1$ $d = 1 @ 1$ $e = 1 @ 1$

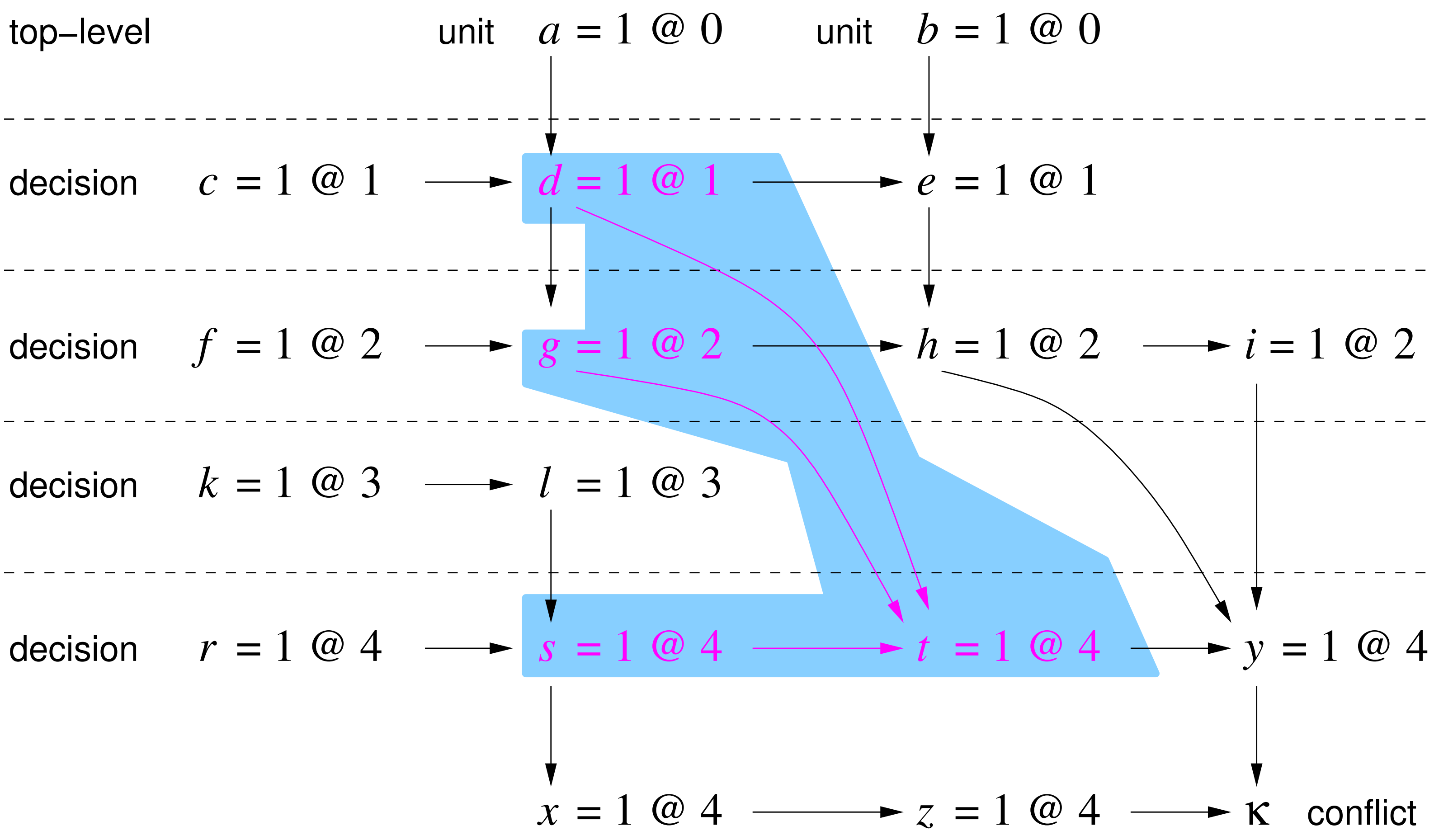
decision $f = 1 @ 2$ $g = 1 @ 2$ $h = 1 @ 2$ $i = 1 @ 2$

decision $k = 1 @ 3$ $l = 1 @ 3$

decision $r = 1 @ 4$ $s = 1 @ 4$ $t = 1 @ 4$ $y = 1 @ 4$

$x = 1 @ 4$ $z = 1 @ 4$ κ conflict

Antecedents / Reasons



top–level unit $a = 1 \ @ \ 0$ unit $b = 1 \ @ \ 0$

decision $c = 1 \ @ \ 1$ $d = 1 \ @ \ 1$ $e = 1 \ @ \ 1$

decision $f = 1 \ @ \ 2$ $g = 1 \ @ \ 2$ $h = 1 \ @ \ 2$ $i = 1 \ @ \ 2$

decision $k = 1 \ @ \ 3$ $l = 1 \ @ \ 3$

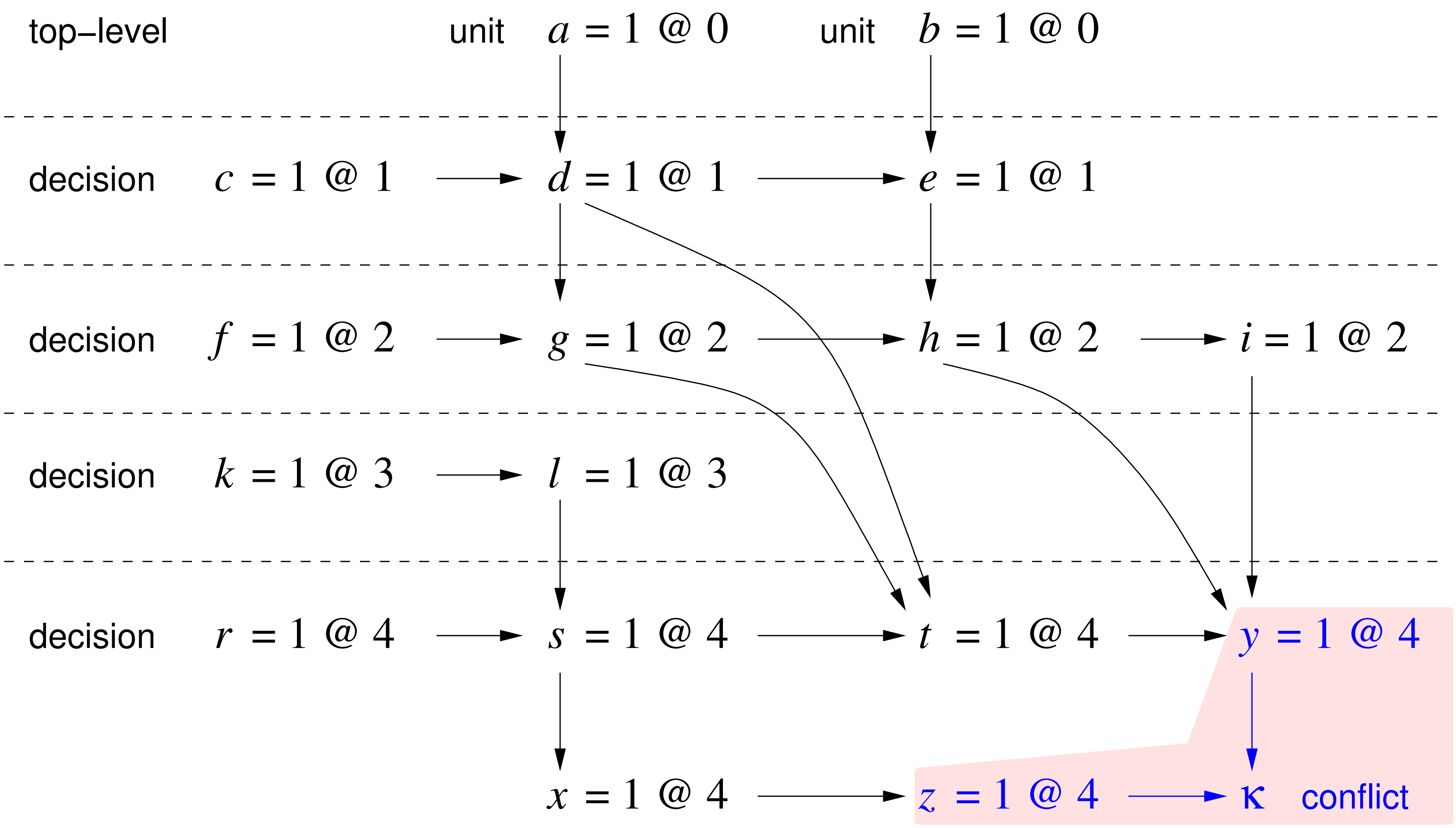
decision $r = 1 \ @ \ 4$ $s = 1 \ @ \ 4$ $t = 1 \ @ \ 4$ $y = 1 \ @ \ 4$

$x = 1 \ @ \ 4$ $z = 1 \ @ \ 4$ κ conflict

$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee t)$$

Conflicting Clauses

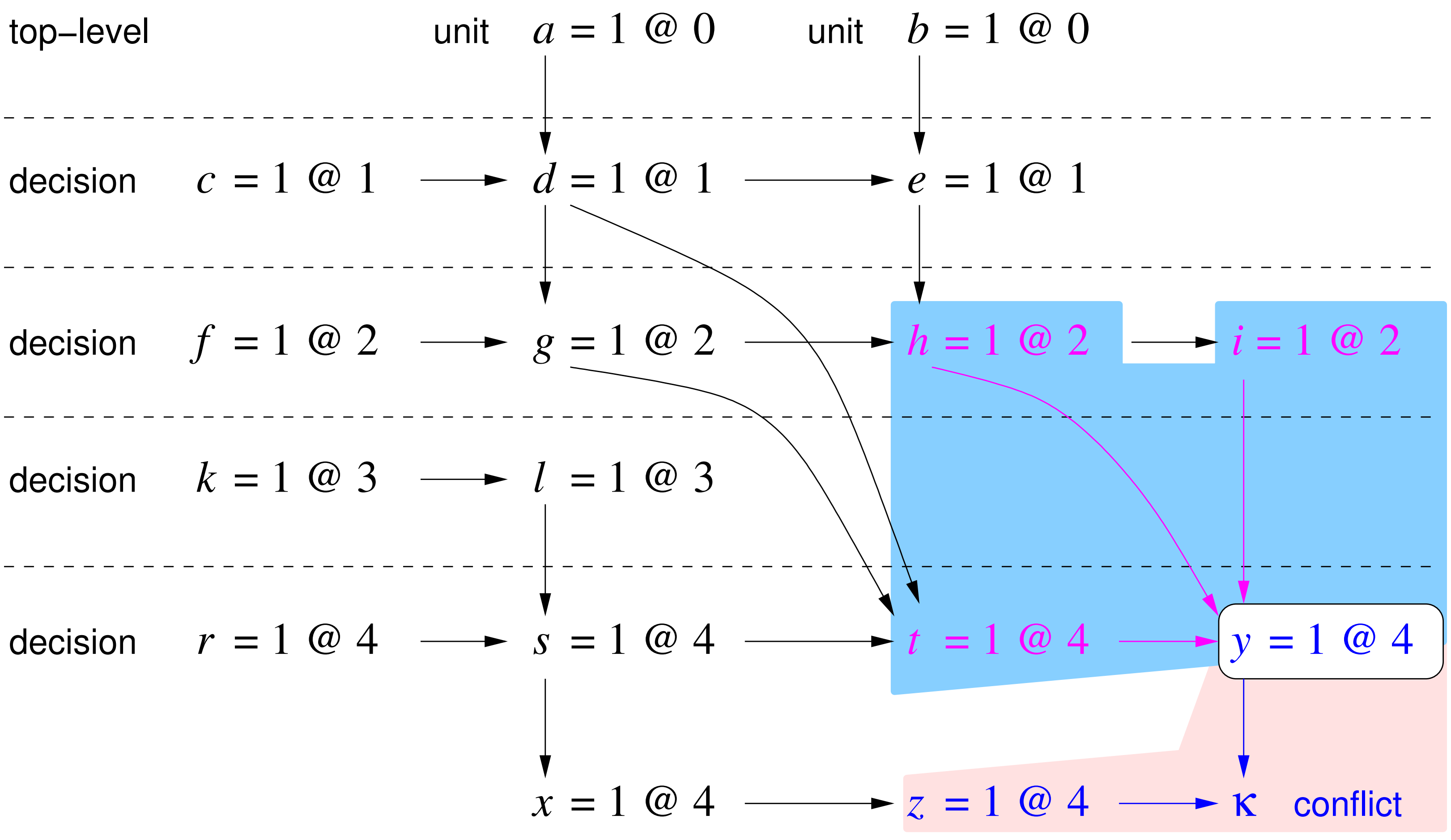
CDCL / Grasp [MarquesSilvaSakallah'96]



$$\neg(y \wedge z) \qquad \equiv \qquad (\bar{y} \vee \bar{z})$$

Resolving Antecedents 1st Time

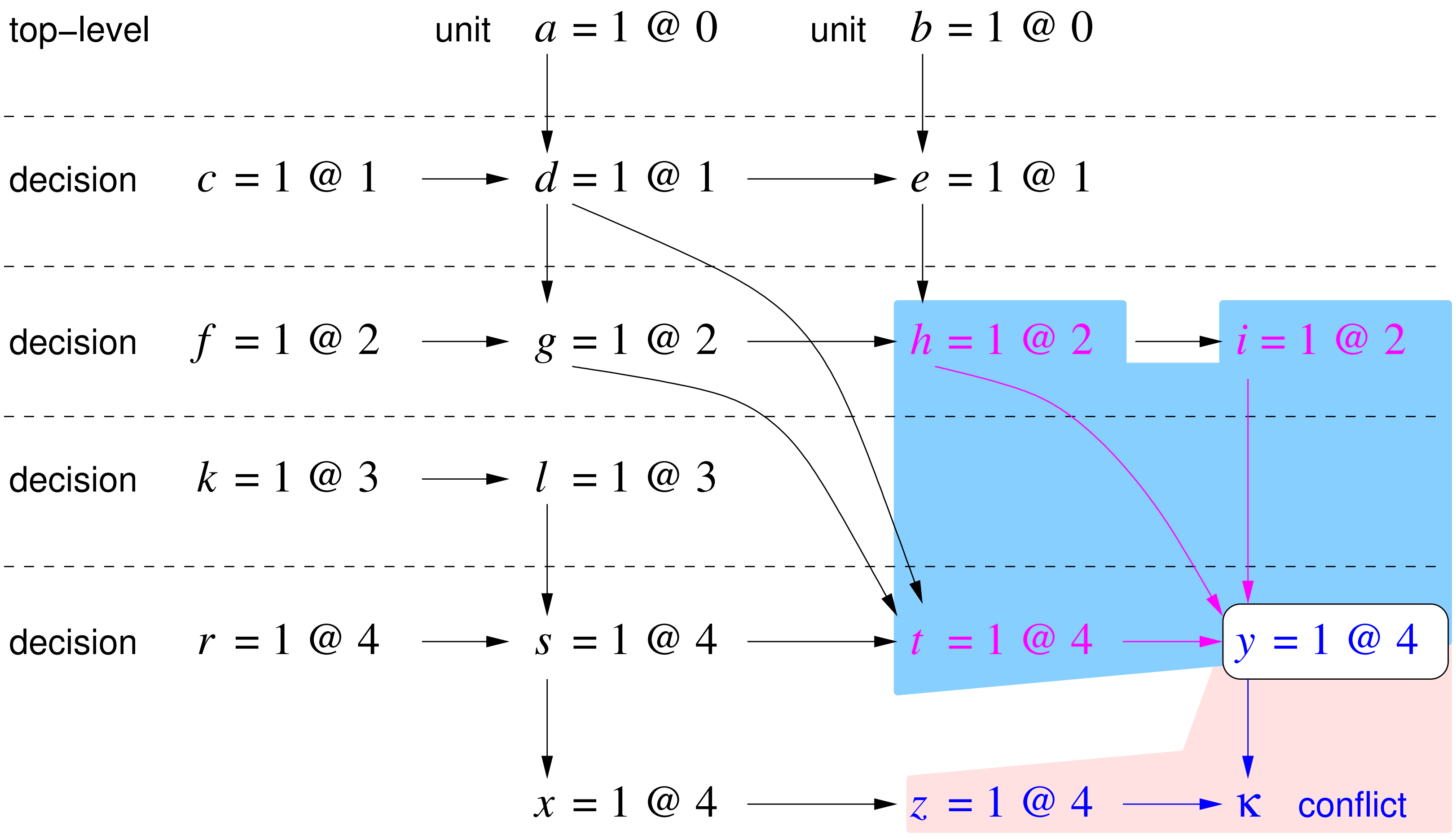
CDCL / Grasp [MarquesSilvaSakallah’96]



$(\overline{h} \vee \overline{i} \vee \overline{t} \vee y)$ $(\overline{y} \vee \overline{z})$

Resolving Antecedents 1st Time

CDCL / Grasp [MarquesSilvaSakallah'96]



top–level unit $a = 1 @ 0$ unit $b = 1 @ 0$

decision $c = 1 @ 1$ $d = 1 @ 1$ $e = 1 @ 1$

decision $f = 1 @ 2$ $g = 1 @ 2$ $h = 1 @ 2$ $i = 1 @ 2$

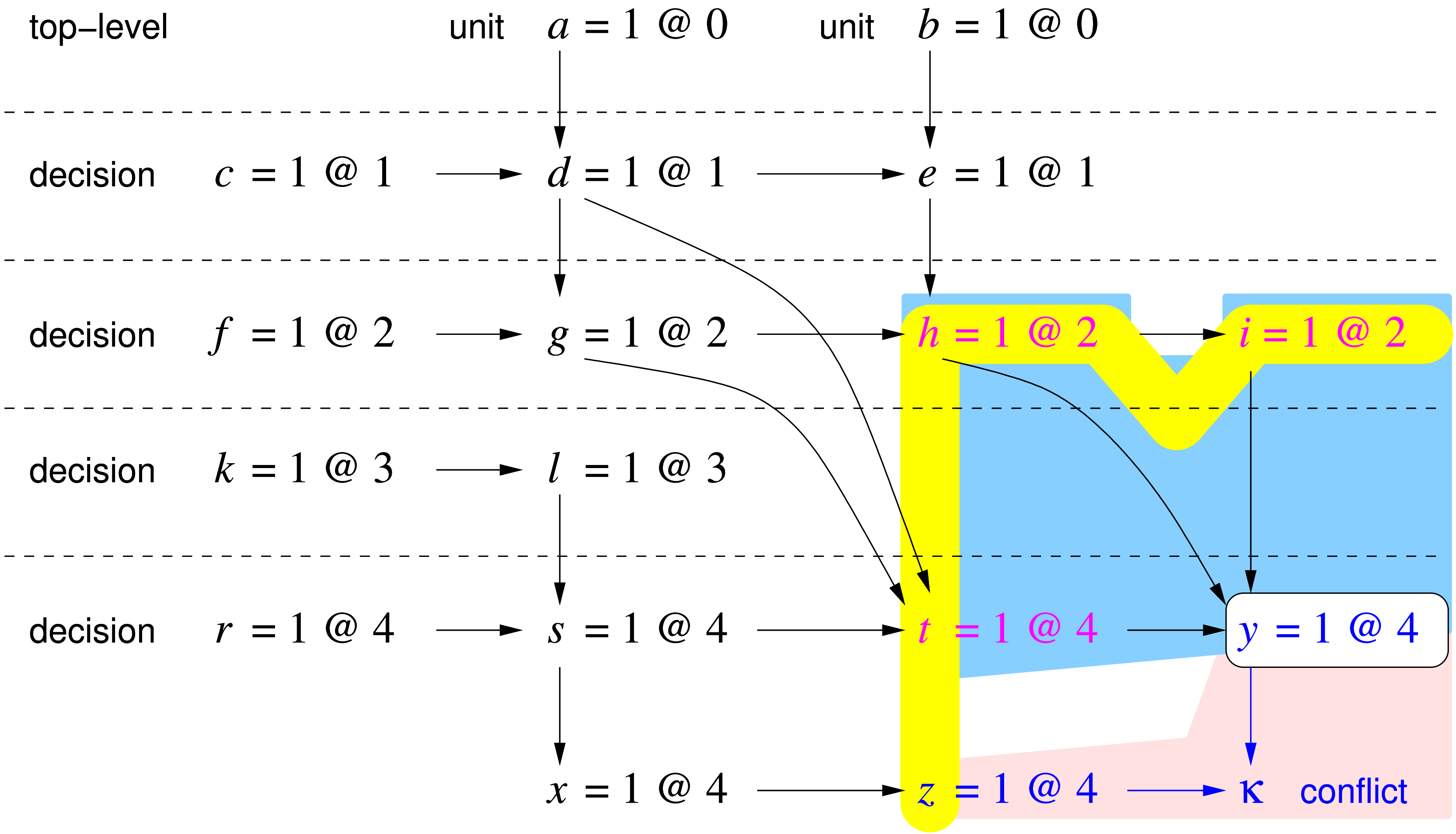
decision $k = 1 @ 3$ $l = 1 @ 3$

decision $r = 1 @ 4$ $s = 1 @ 4$ $t = 1 @ 4$ $y = 1 @ 4$

$x = 1 @ 4$ $z = 1 @ 4$ κ conflict

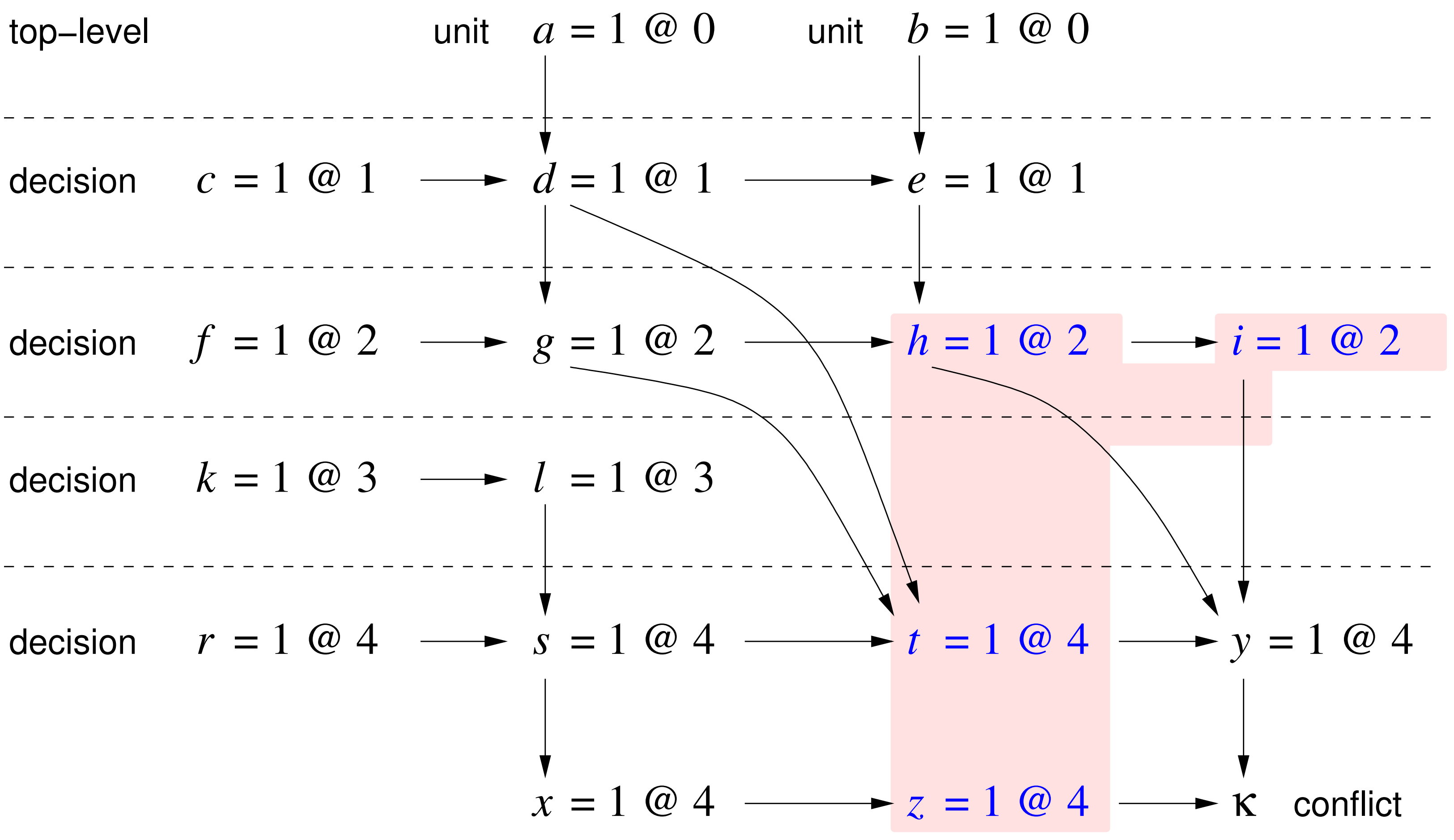
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Resolvents = Cuts = Potential Learned Clauses



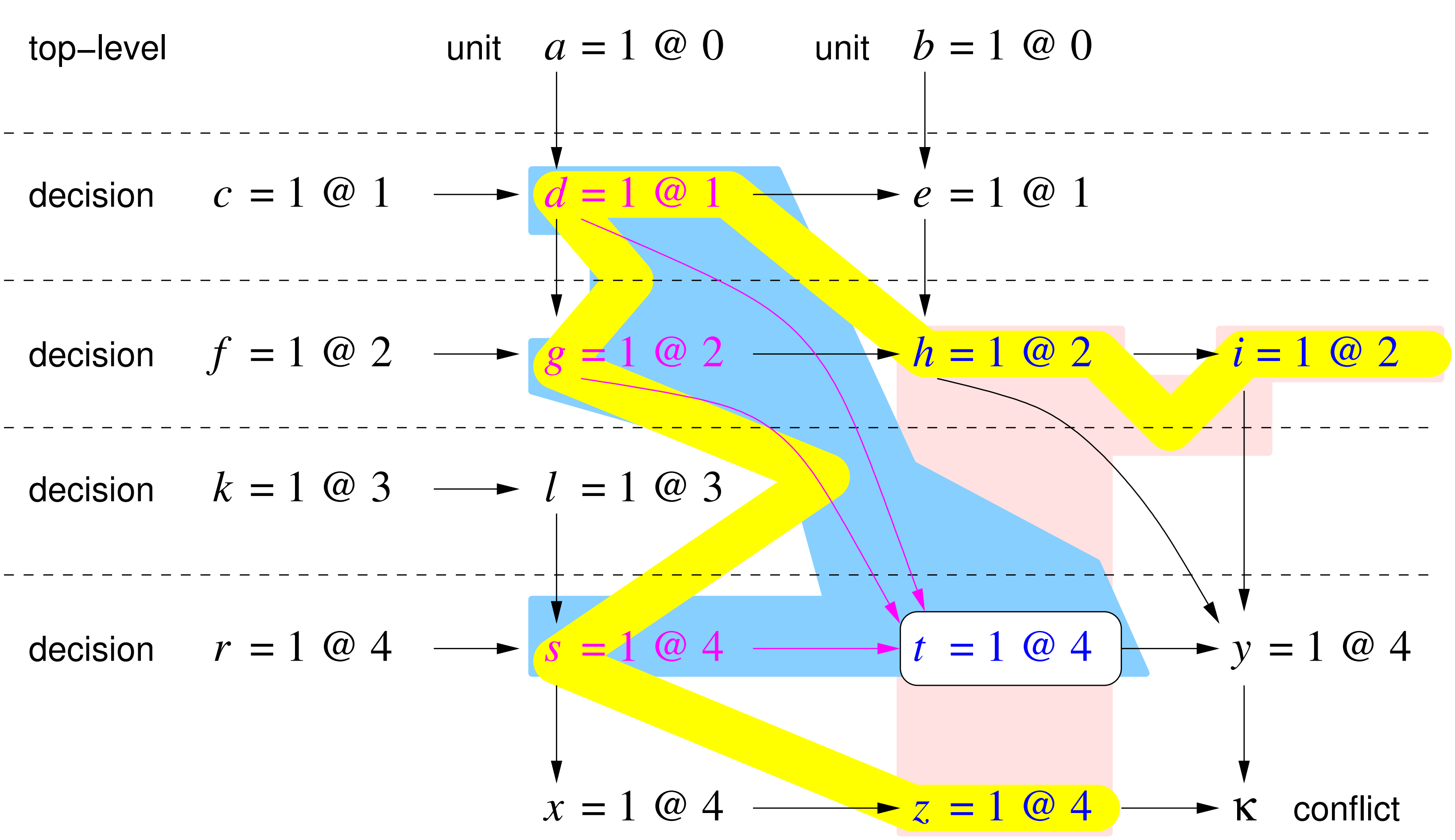
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Potential Learned Clause After 1 Resolution



$$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$$

Resolving Antecedents 2nd Time



top–level unit $a = 1 \,@\, 0$ unit $b = 1 \,@\, 0$

decision $c = 1 \,@\, 1$ $d = 1 \,@\, 1$ $e = 1 \,@\, 1$

decision $f = 1 \,@\, 2$ $g = 1 \,@\, 2$ $h = 1 \,@\, 2$ $i = 1 \,@\, 2$

decision $k = 1 \,@\, 3$ $l = 1 \,@\, 3$

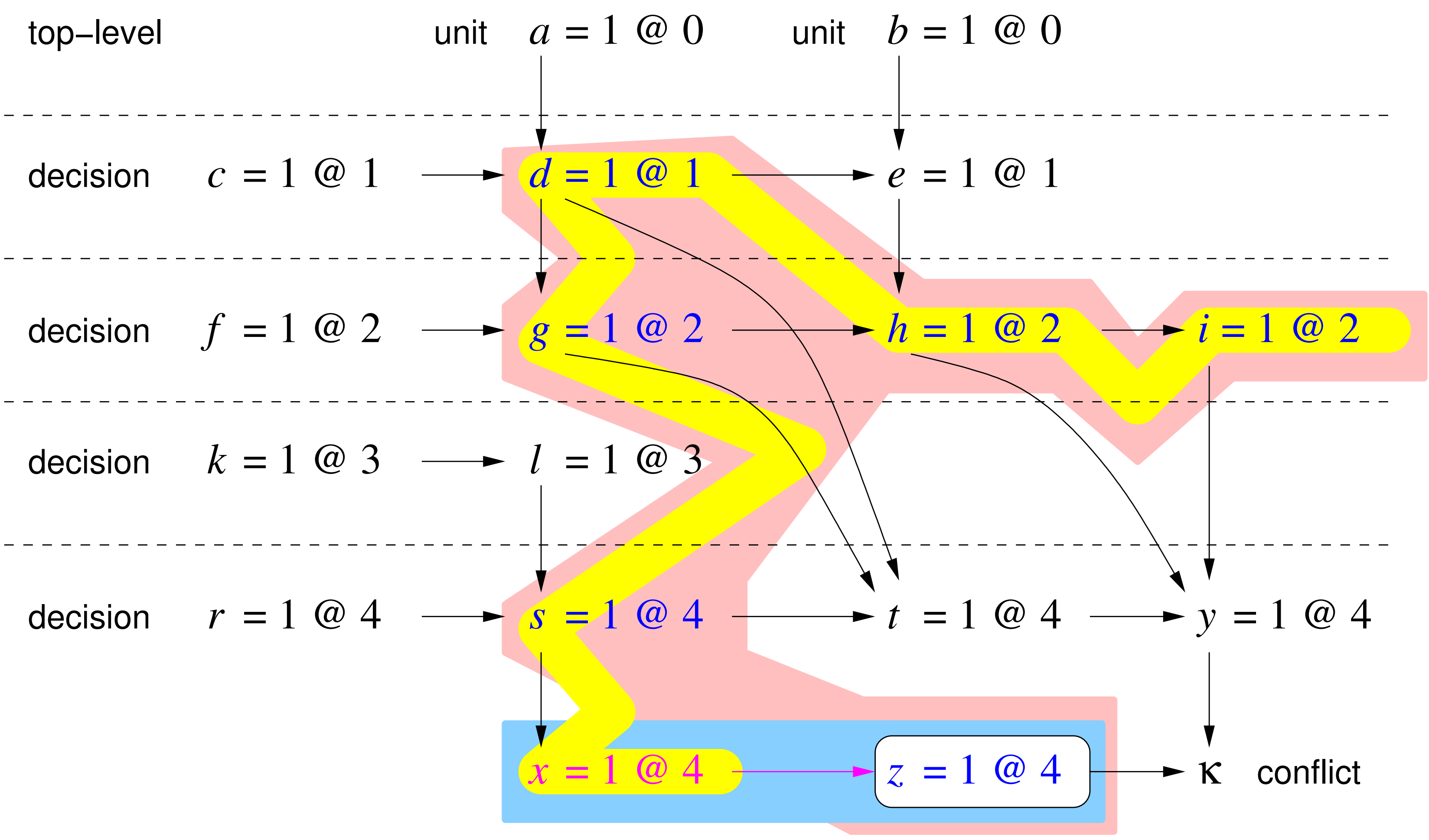
decision $r = 1 \,@\, 4$ $s = 1 \,@\, 4$ $t = 1 \,@\, 4$ $y = 1 \,@\, 4$

$x = 1 \,@\, 4$ $z = 1 \,@\, 4$ κ conflict

$$\frac{(\bar{d} \vee \bar{g} \vee \bar{s} \vee t) \qquad (\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}$$

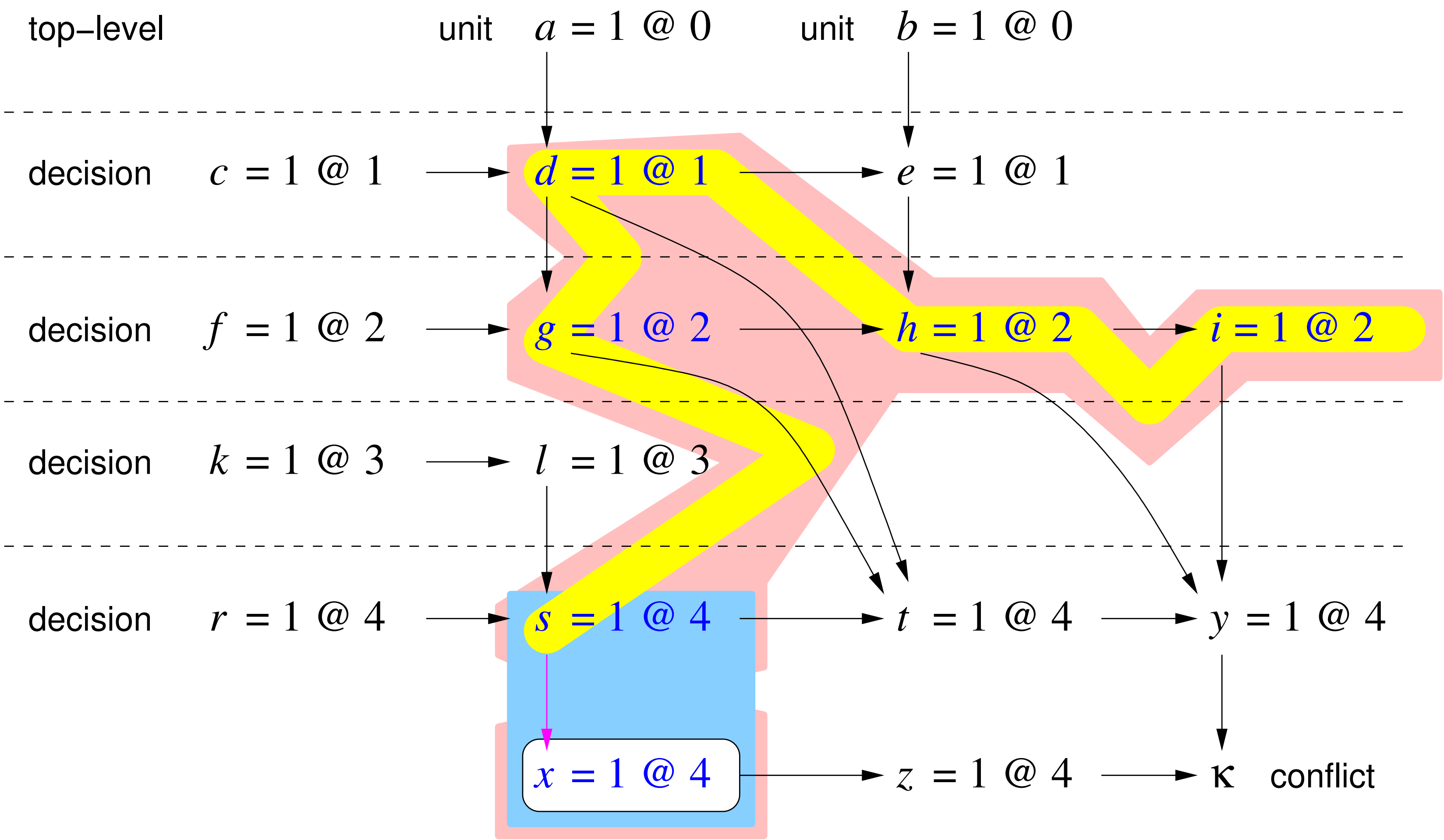
Resolving Antecedents 3rd Time

CDCL / Grasp [MarquesSilvaSakallah'96]



$$\frac{(\overline{x} \vee z) \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i} \vee \overline{z})}{(\overline{x} \vee \overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})}$$

Resolving Antecedents 4th Time

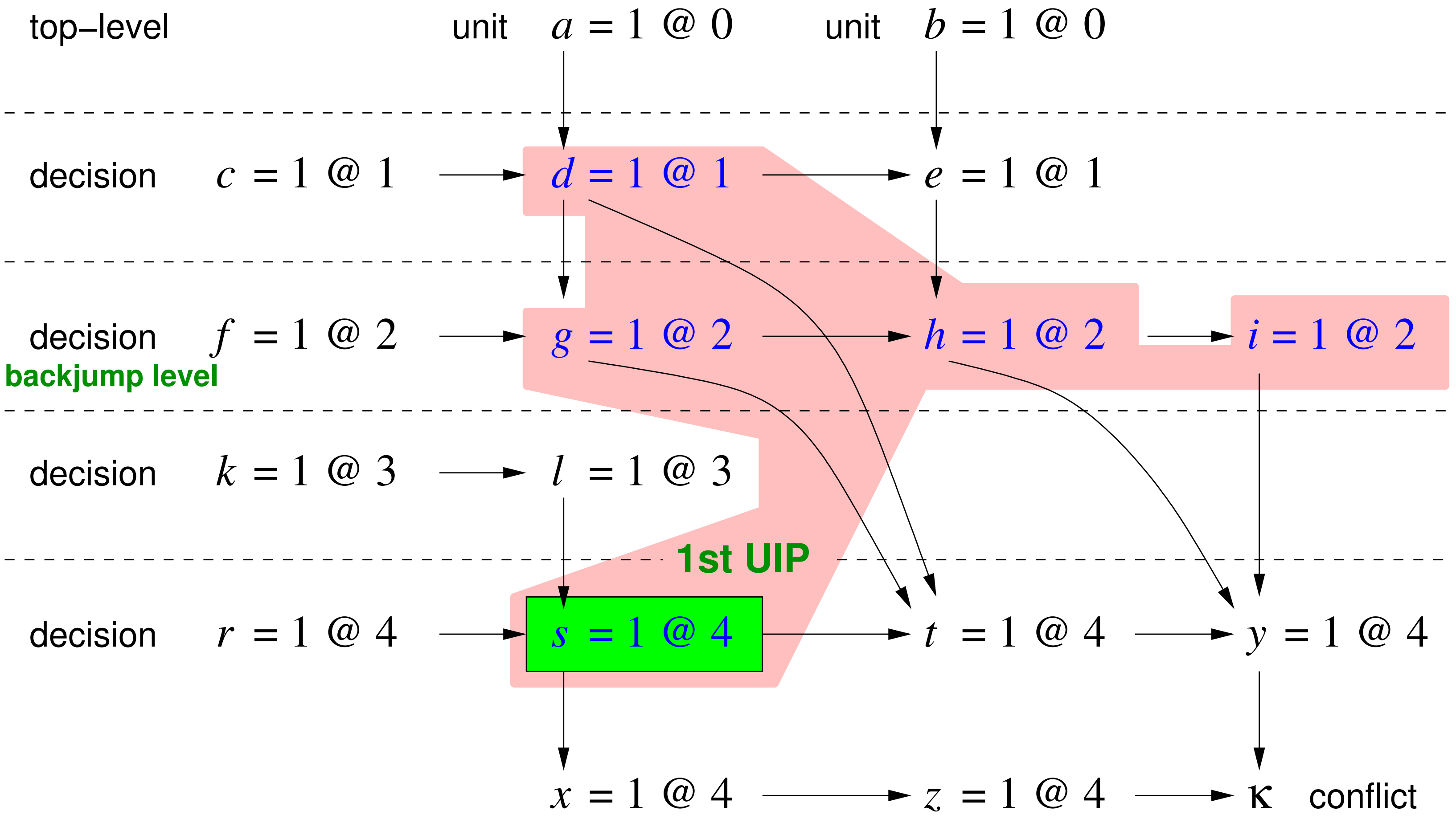


$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

1^{st} UIP Clause after 4 Resolutions

CDCL / Grasp [MarquesSilvaSakallah'96]



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

UIP = *unique implication point* dominates conflict on the last level

Simple Algorithm to Find First UIP Clause

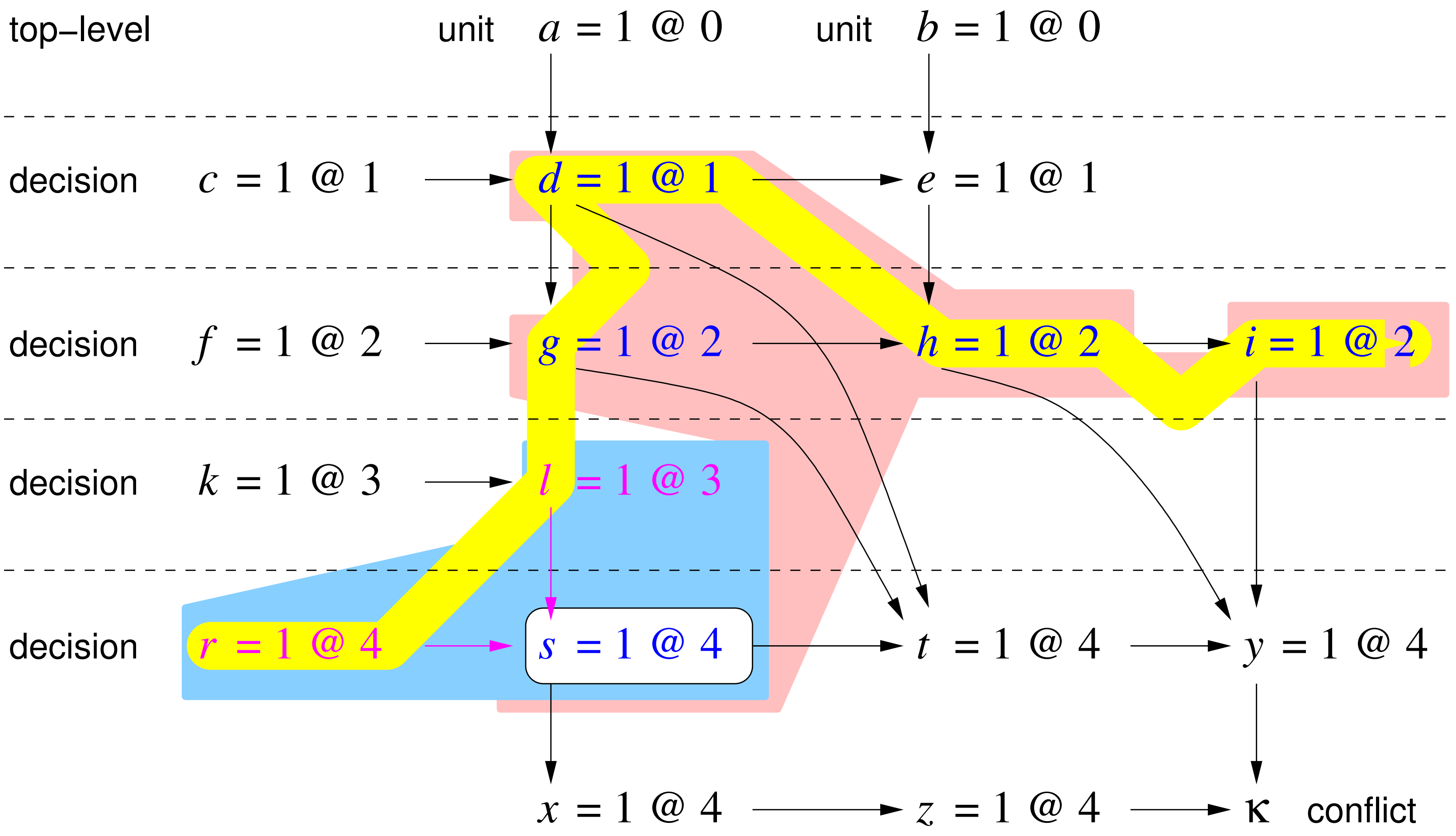
- can be found by *graph traversal* in the reverse order of made assignments
  - *trail* respects this order
  - mark literals in conflict
  - traverse reasons of marked variables on trail in reverse order
- count *number unresolved variables* on current decision level
  - decrease counter if new reason / antecedent clause resolved
  - if counter=1 (only one unresolved marked variable left) then this node is a UIP
  - note, decision of current decision level is a UIP and thus a *sentinel*

Modern CDCL Loop

actual Cleaneling code

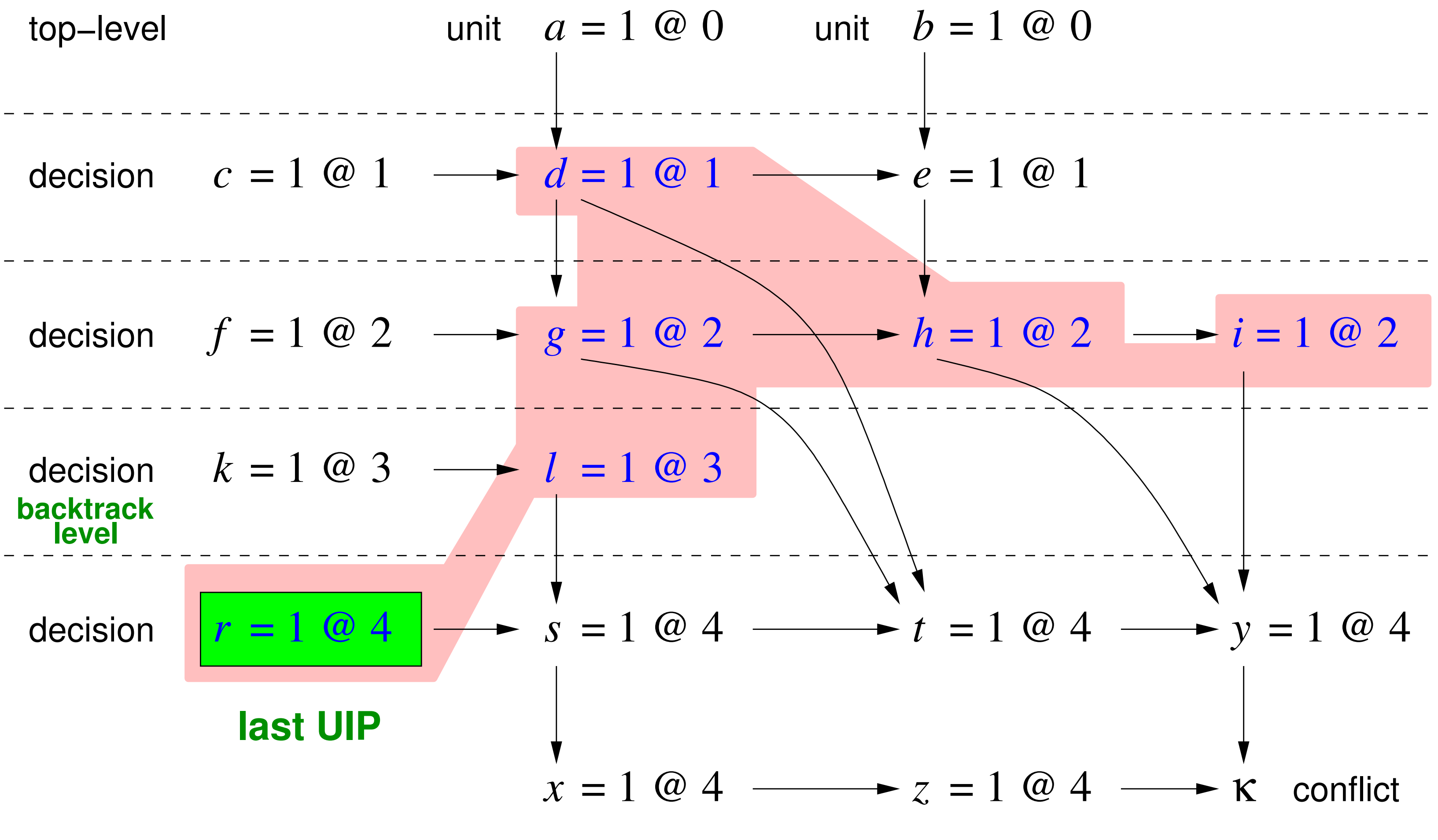
```
Status Solver::search (long limit) {
  long conflicts = 0; Clause * conflict; Status res = UNKNOWN;
  while (!res)
    if (empty) res = UNSATISFIABLE;
    else if ((conflict = bcp ())) analyze (conflict), conflicts++;
    else if (conflicts >= limit) break;
    else if (reducing ()) reduce ();
    else if (restarting ()) restart ();
    else if (!decide ()) res = SATISFIABLE;
  return res;
}

Status Solver::solve () {
  long conflicts = 0, steps = 1e6;
  Status res;
  for (;;)
    if ((res = search (conflicts))) break;
    else if ((res = simplify (steps))) break;
    else conflicts += 1e4, steps += 1e6;
  return res;
}
```

Resolving Antecedents 5th Time



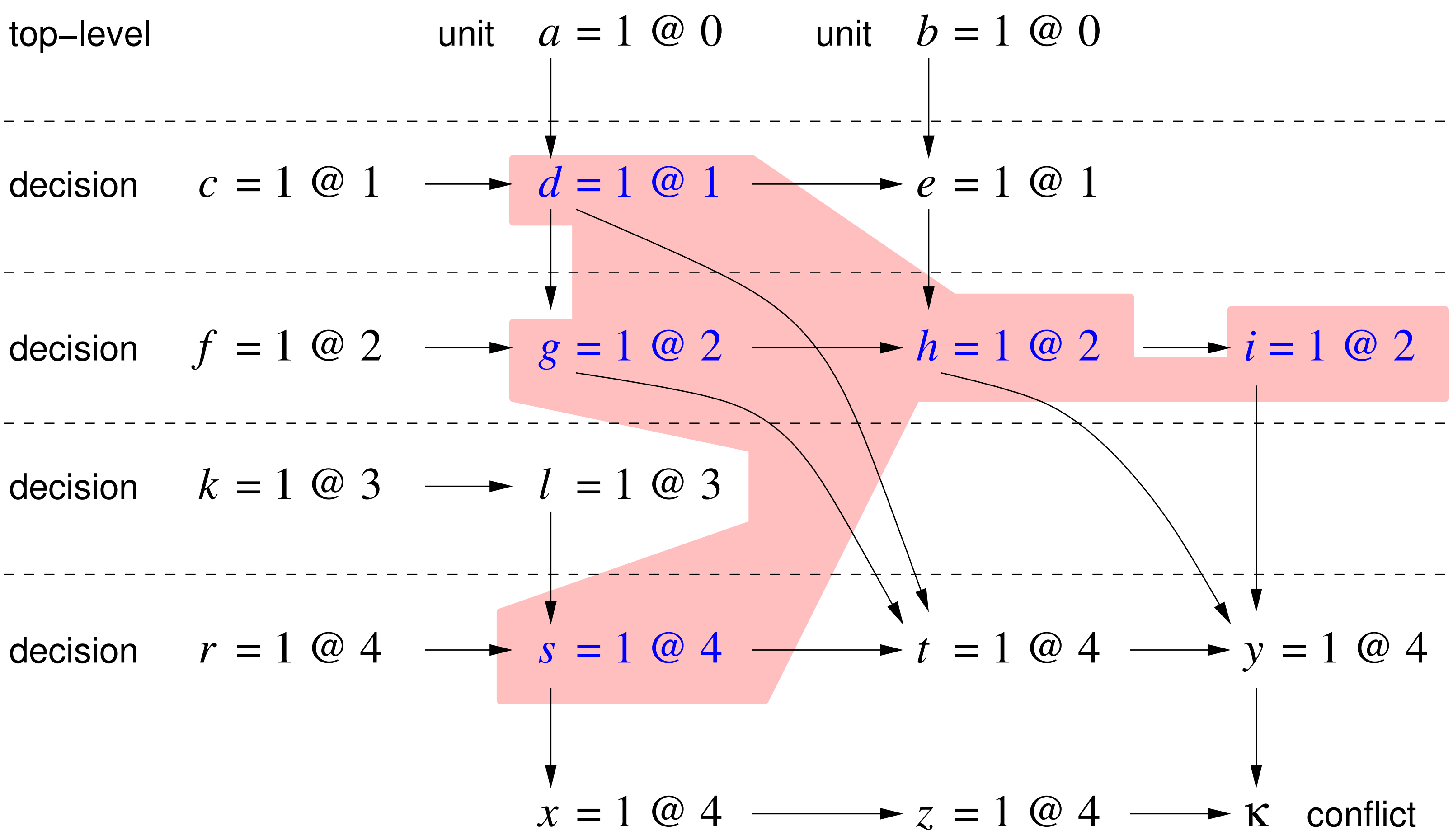
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

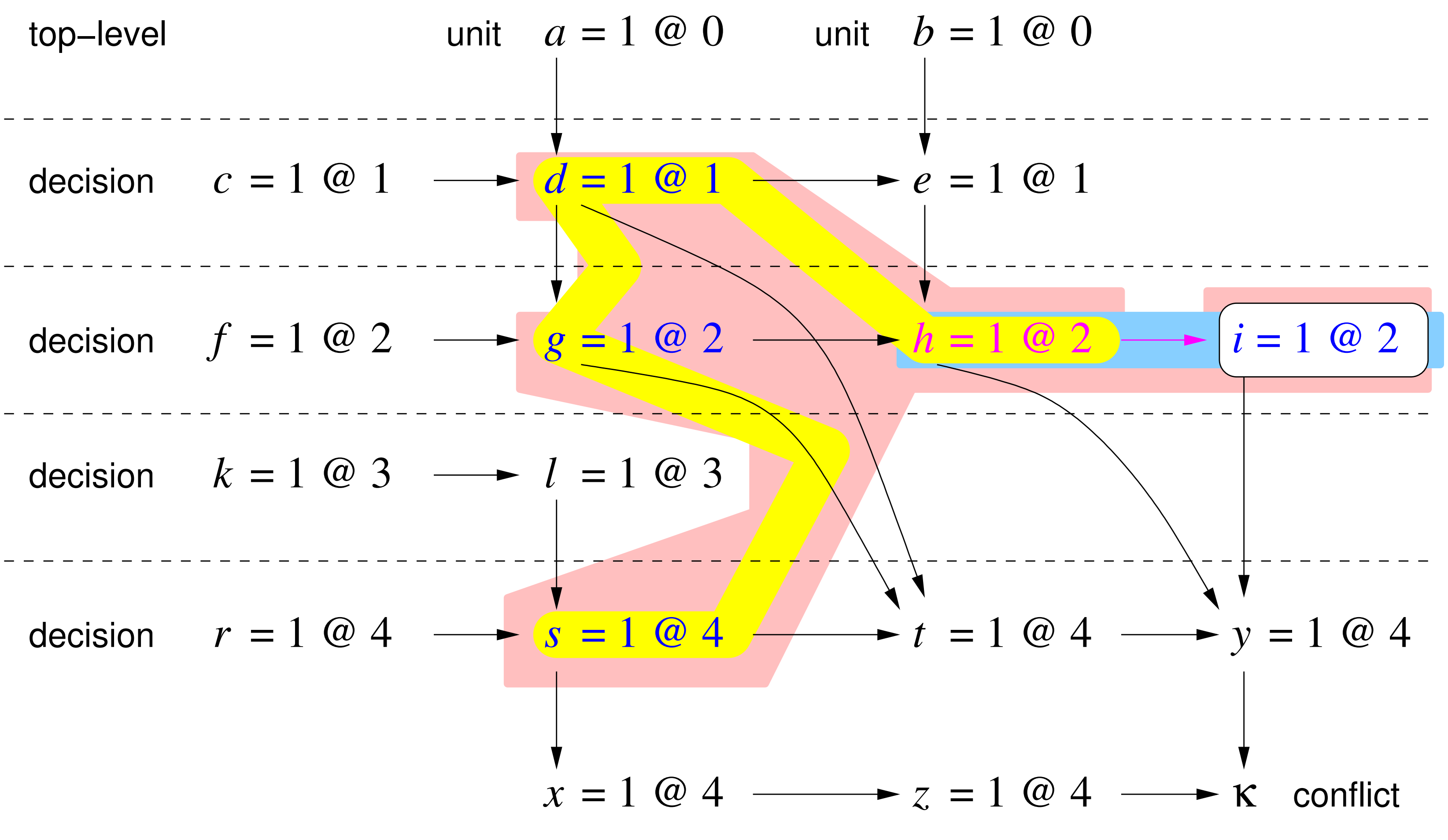
1st UIP Clause after 4 Resolutions



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

Locally Minimizing 1^{st} UIP Clause

Sörensson’06, BiereSörensson’09

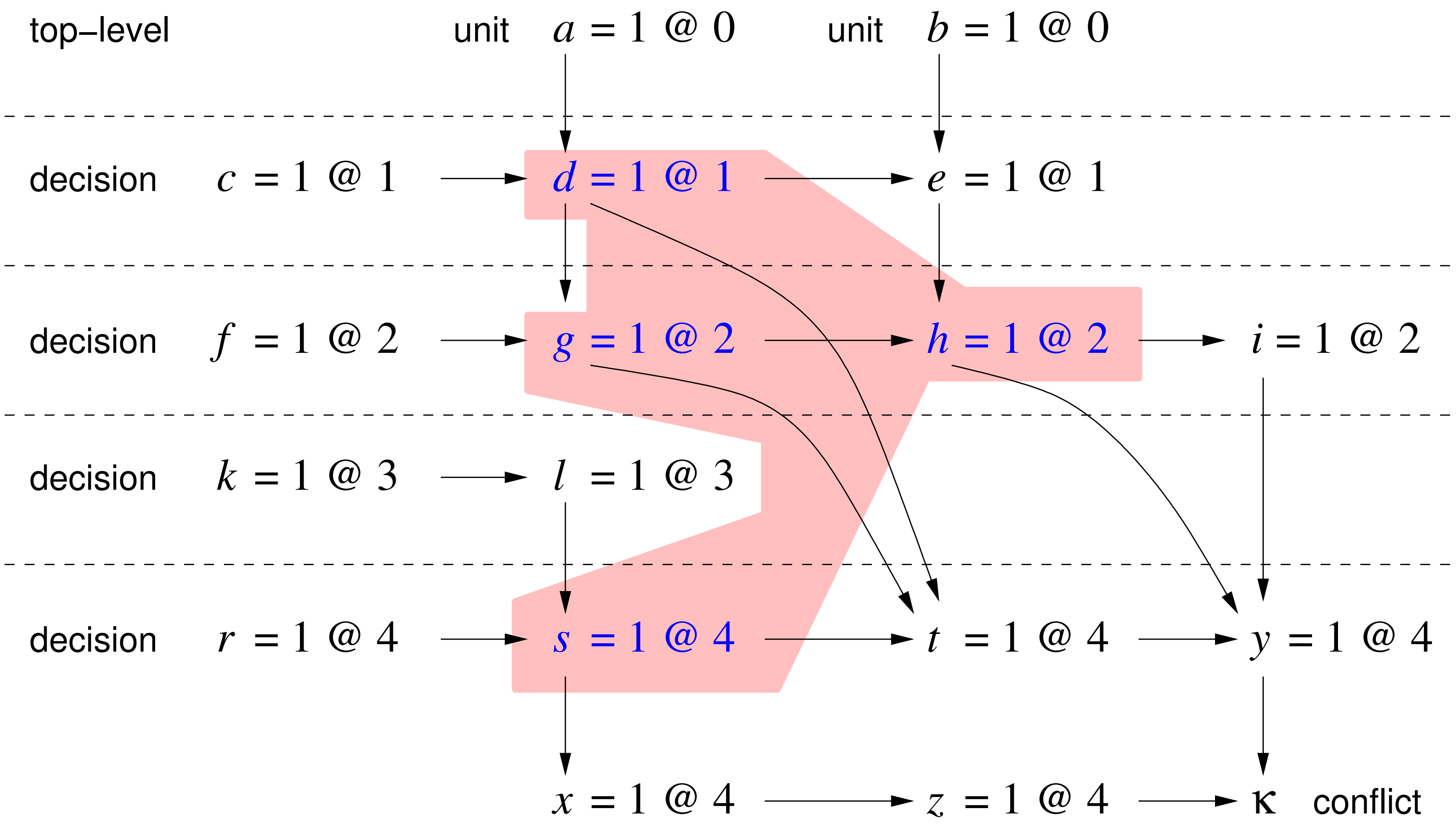


$$\frac{(\bar{h} \vee i) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h})}$$

self subsuming resolution

Locally Minimized Learned Clause

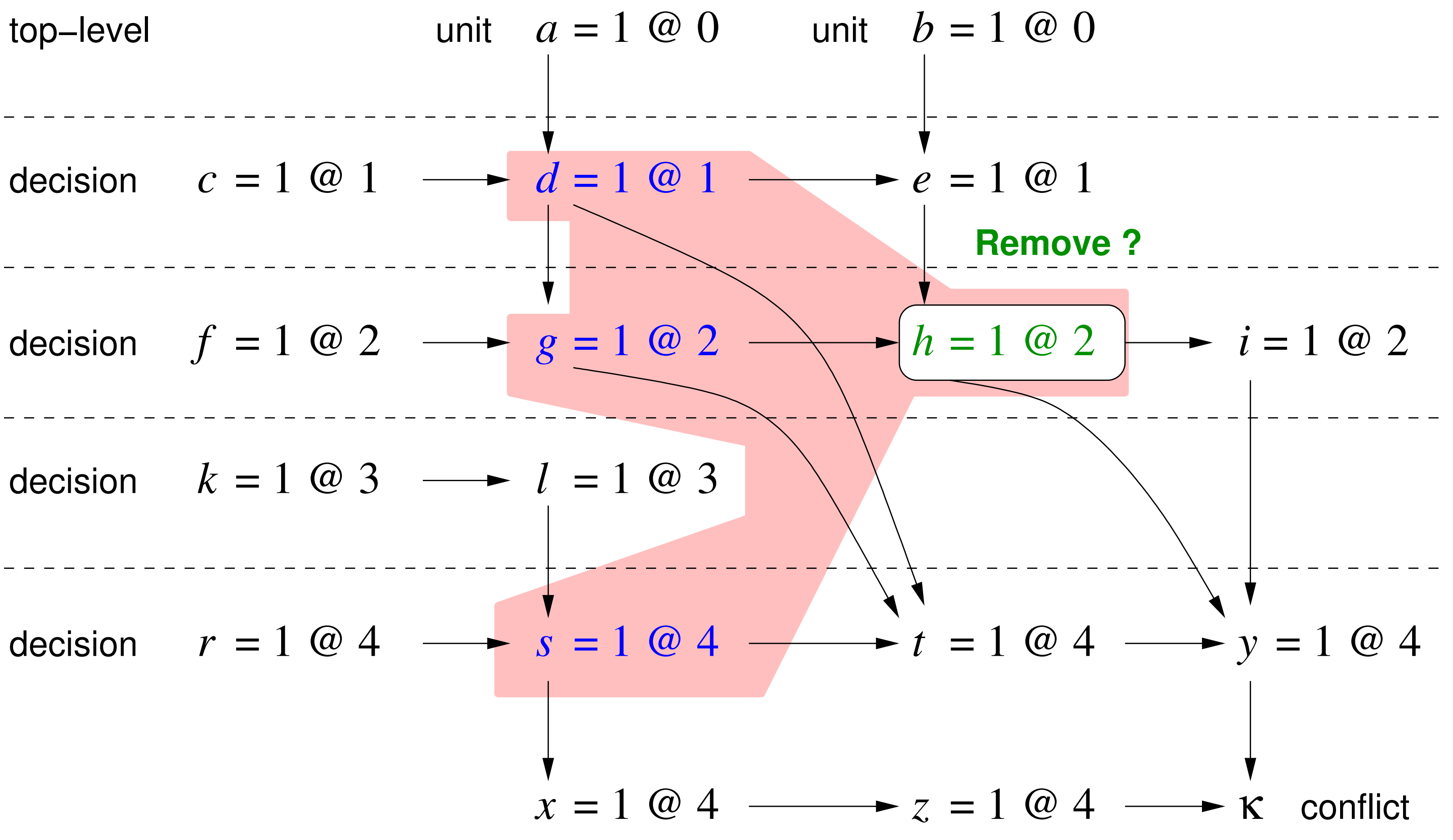
Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$$

Minimizing Locally Minimized Learned Clause Further?

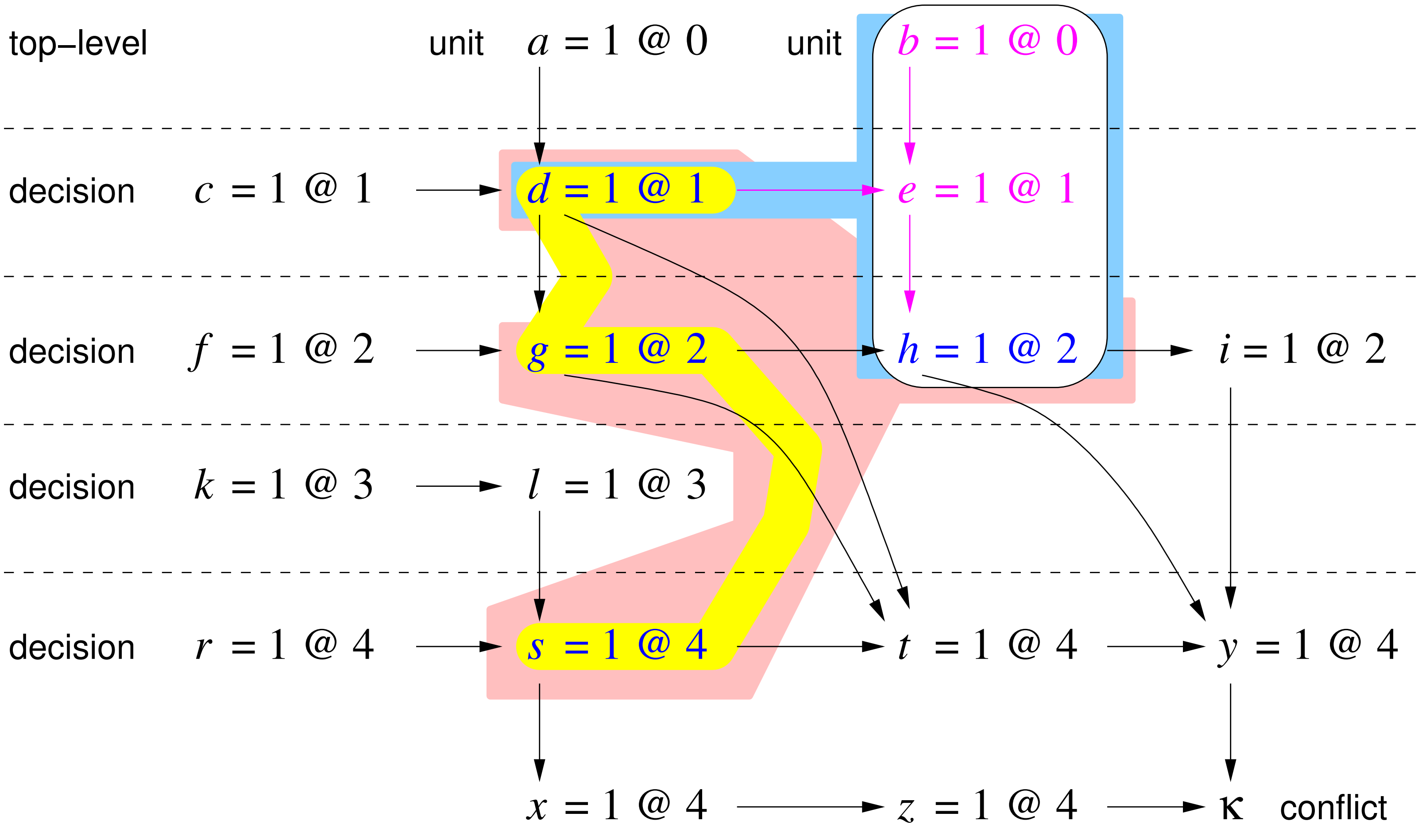
Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

Recursively Minimizing Learned Clause

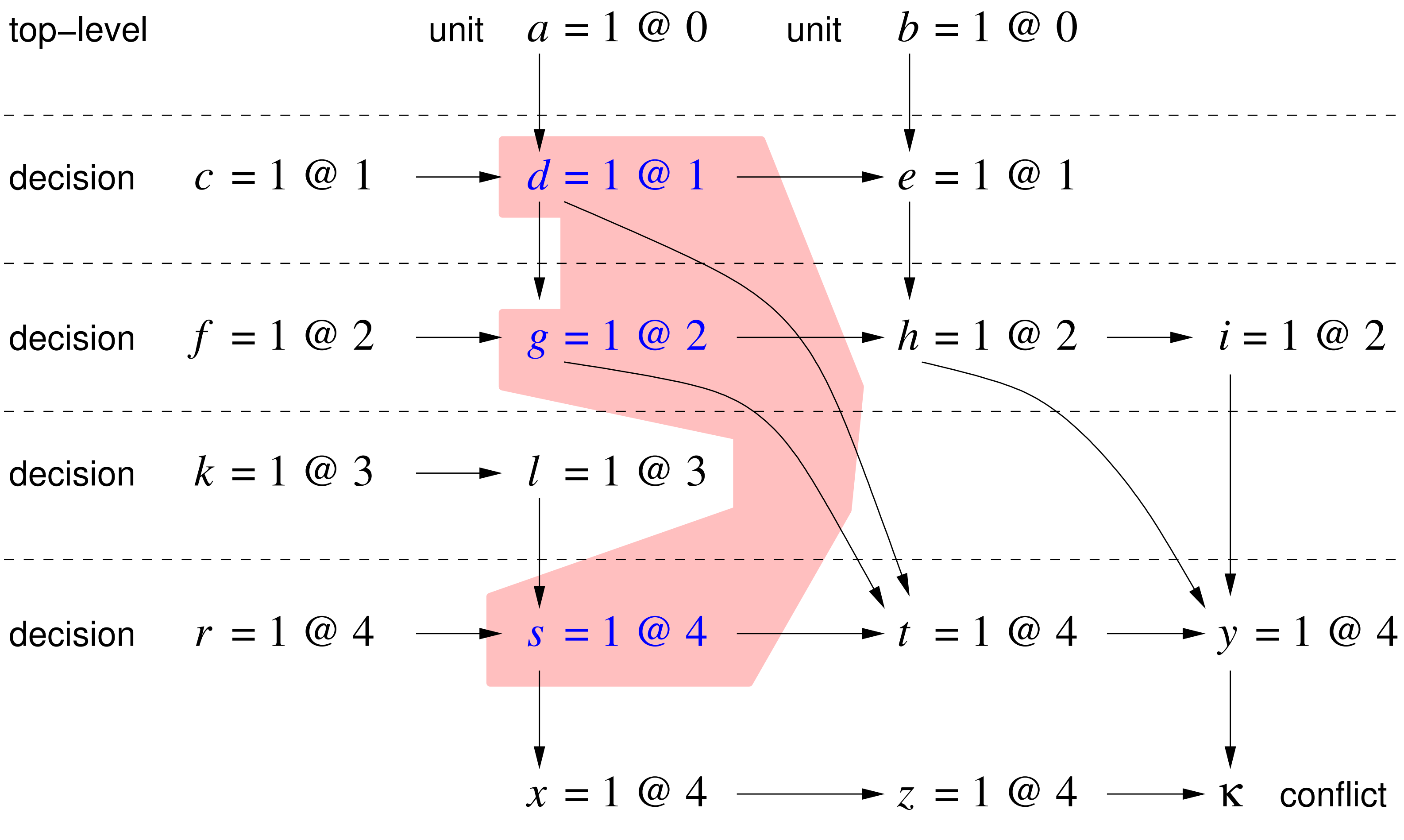
Sörensson’06, BiereSörensson’09



$$\dfrac{(b) \qquad \dfrac{(\bar{d} \vee \bar{b} \vee e) \qquad \dfrac{(\bar{e} \vee \bar{g} \vee h) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h})}{(\bar{e} \vee \bar{d} \vee \bar{g} \vee \bar{s})}}{(\bar{b} \vee \bar{d} \vee \bar{g} \vee \bar{s})}}{(\bar{d} \vee \bar{g} \vee \bar{s})}$$

Recursively Minimized Learned Clause

Sörensson'06, BiereSörensson'09



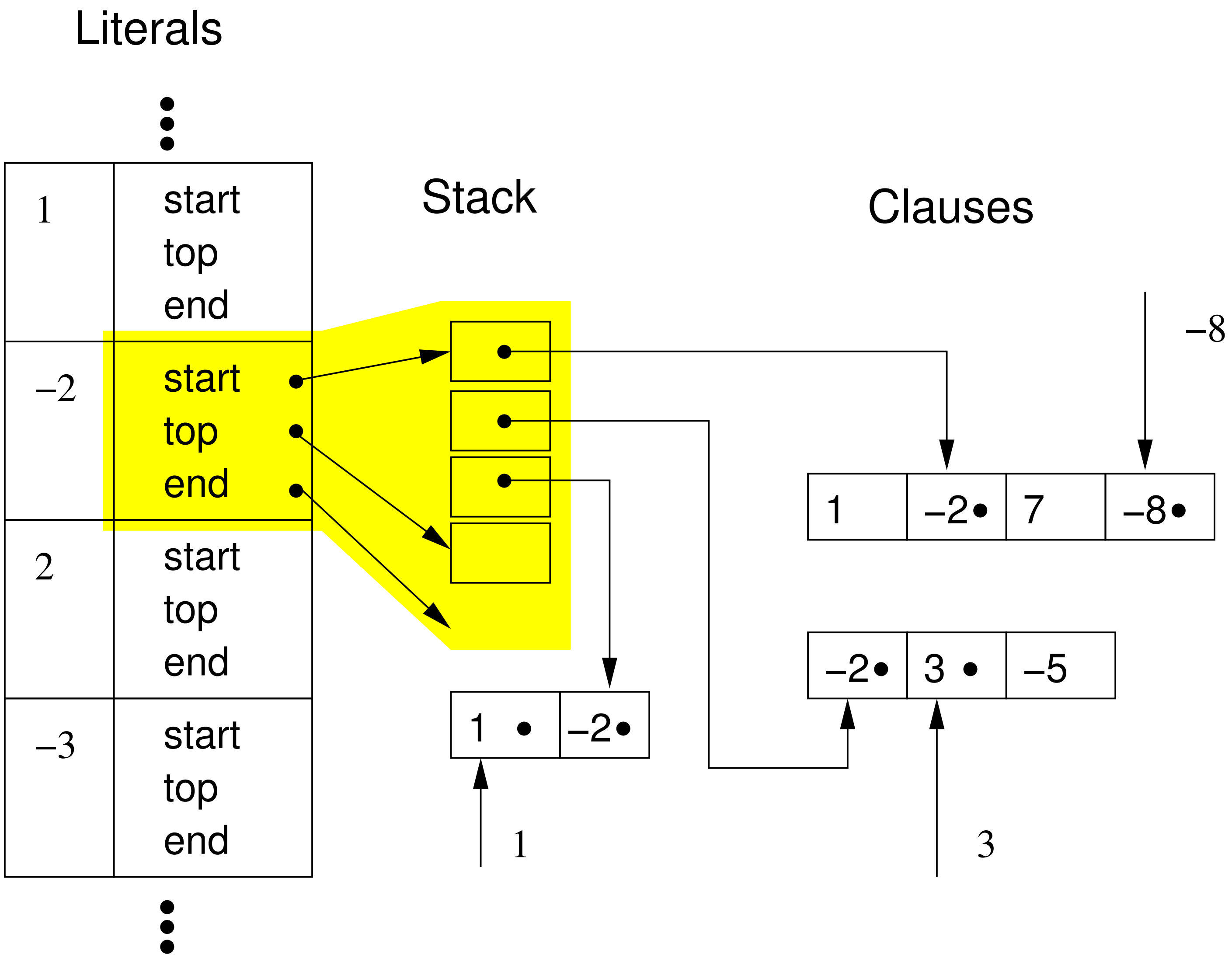
$(\overline{d} \vee \overline{g} \vee \overline{s})$

algorithm of Allen Van Gelder in SAT'09 produces regular input resolution proofs directly

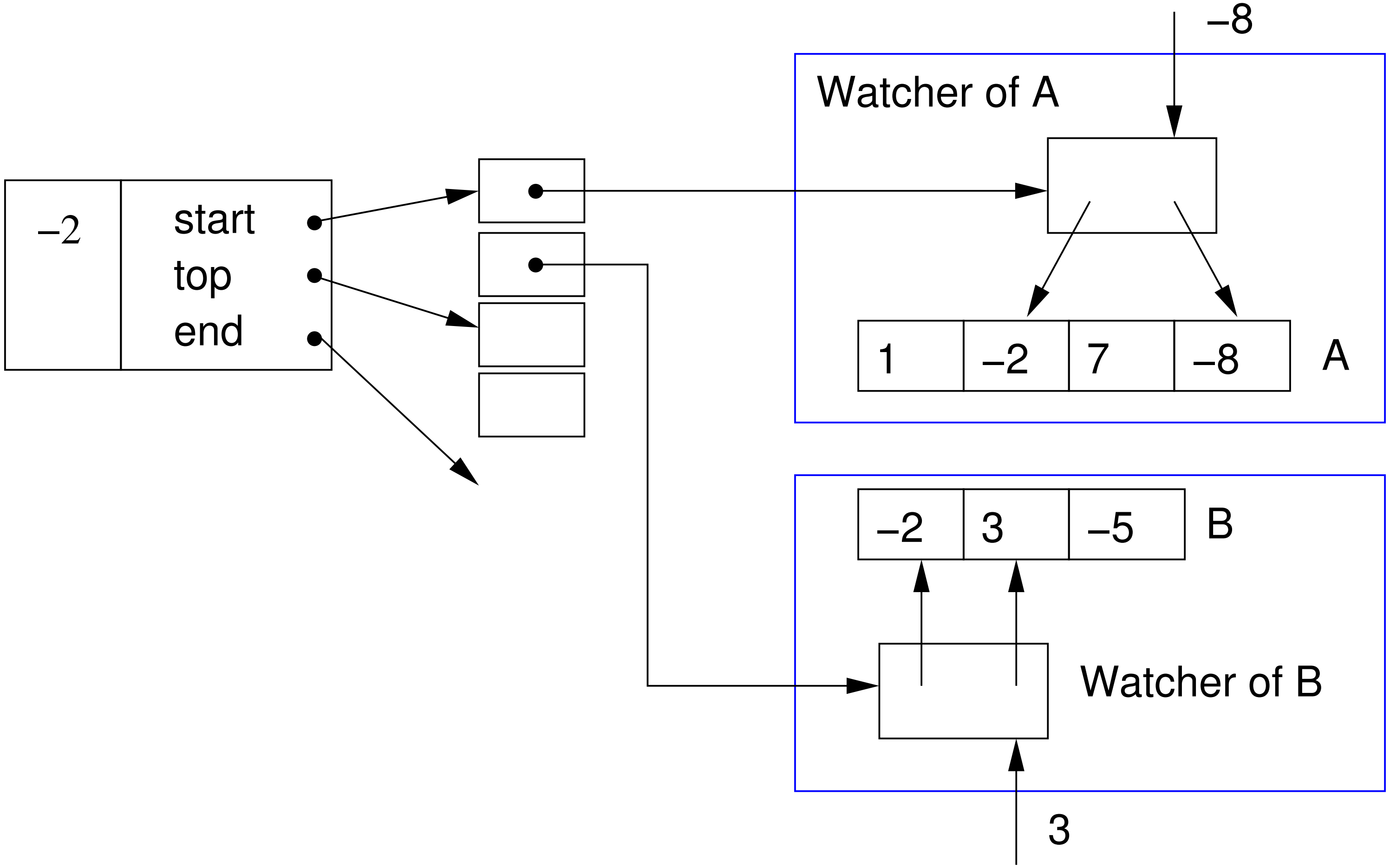
Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - maintain the invariant: $\boxed{\text{always watch two non-false literals}}$
  - if a watched literal becomes *false* replace it
  - if no replacement can be found clause is either unit or empty
  - original version used *head* and *tail* pointers on Tries
- improved variant from Chaff [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily SATO: *head* forward, *trail* backward
  - no update needed during backtracking
- *one* watch is enough to ensure correctness but looses *arc consistency*
- reduces *visiting* clauses by 10x, particularly useful for large and many learned clauses

ZChaff Occurrence Stacks

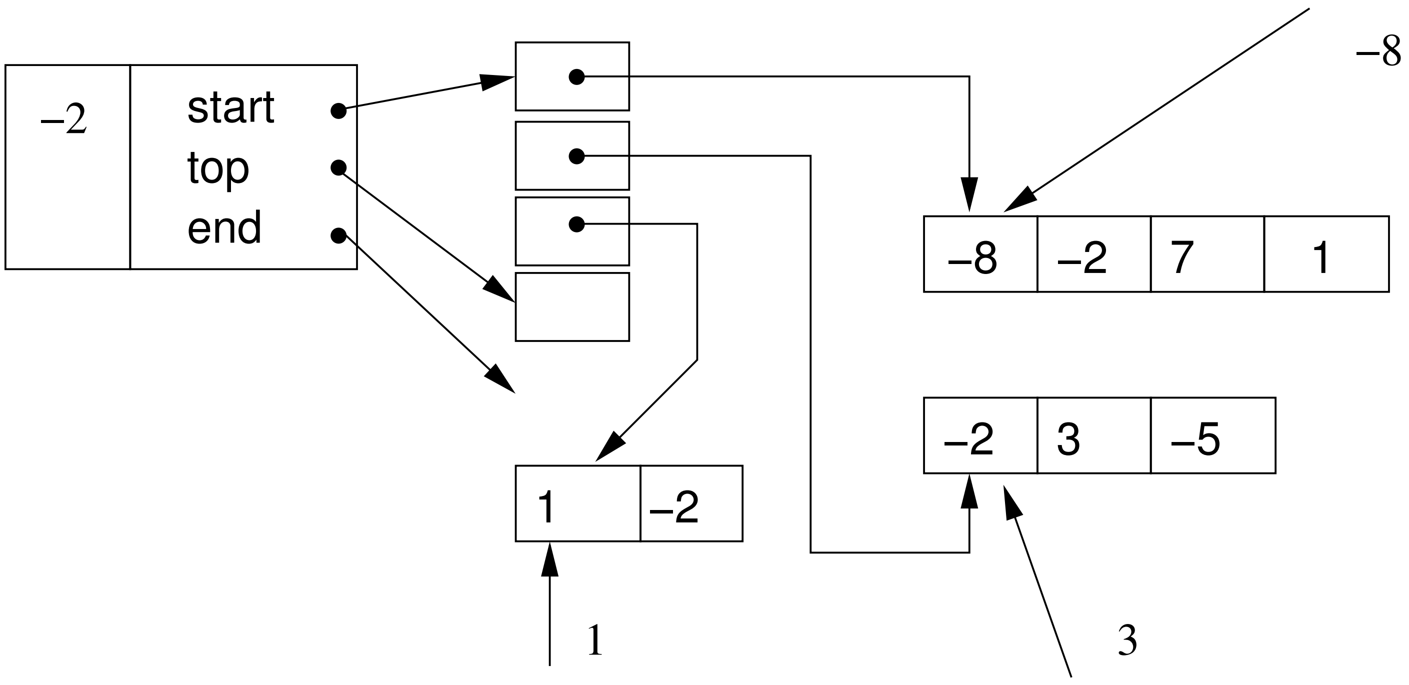


Limmat / FunEx Occurrence Stacks



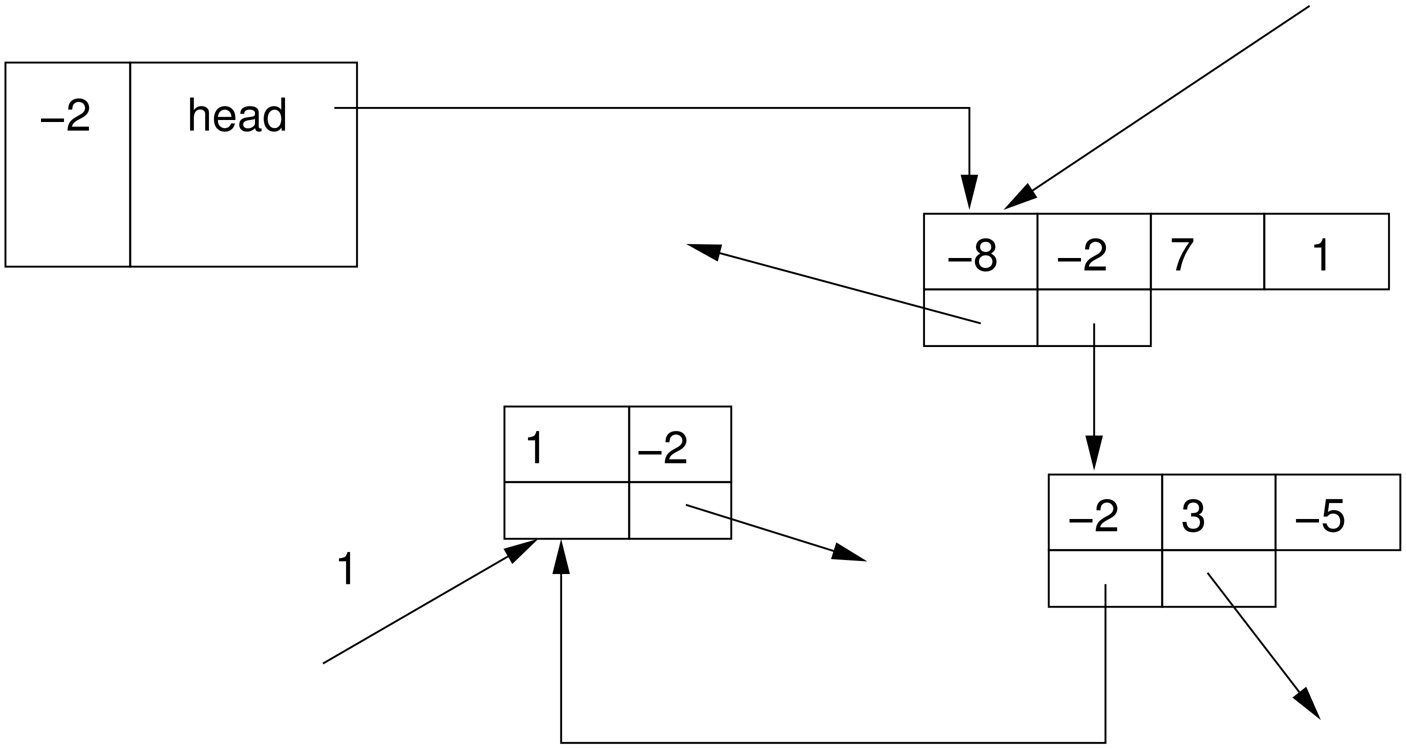
still seems to be best way for *real* sharing of clauses in multi-threaded solvers

CompSAT / MiniSAT Occurrence Stacks



invariant: first two literals are watched

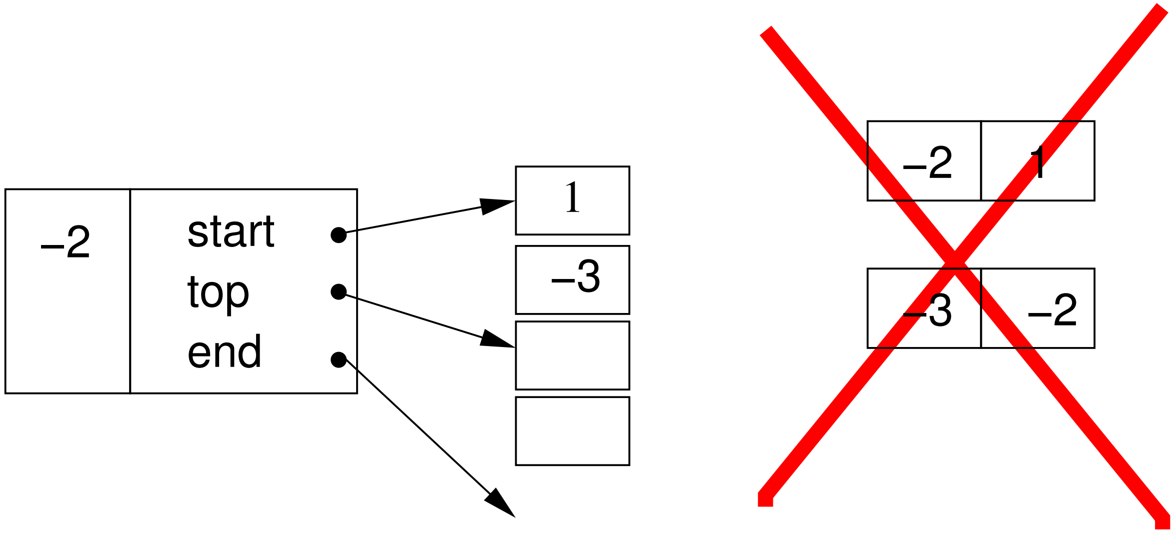
MChaff / PicoSAT Occurrence Lists



invariant: first two literals are watched

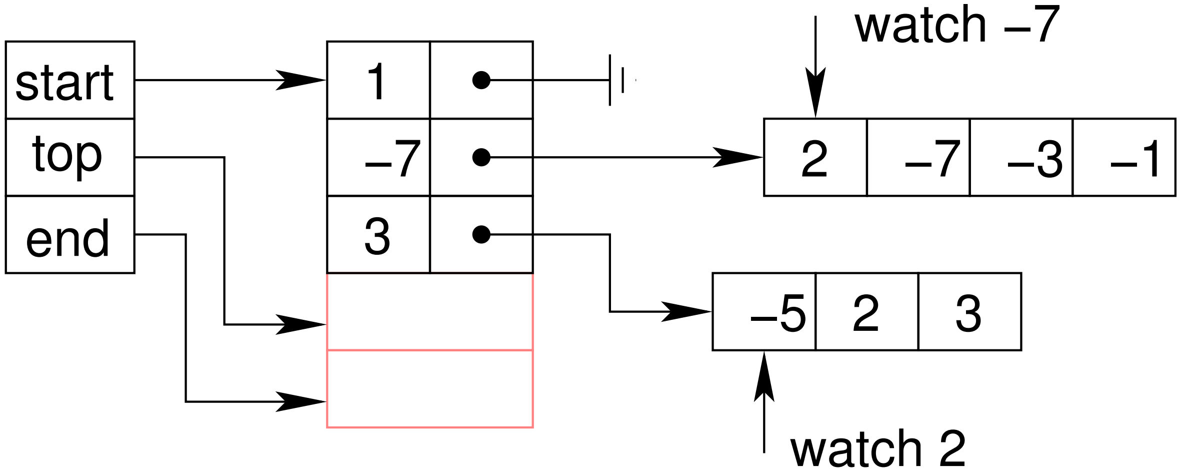
Occurrence Stacks for Binary Clauses

Additional Binary Clause Watcher Stack



Caching Potential Satisfied Literals (Blocking Literals)

ChuHarwoodStuckey'09



observation: often the *other* watched literal satisfies the clause

so cache this literals in watch list to avoid pointer dereference

for binary clause no need to store clause at all

can easily be adjusted for ternary clauses (with full occurrence lists)

LINGELING uses more compact pointer-less variant

Failed Literal Probing

we are still working on tracking down the origin before [Freeman'95] [LeBerre'01]

- key technique in look-ahead solvers such as Satz, OKSolver, March
  - failed literal probing at all search nodes
  - used to find the best decision variable and phase
- simple algorithm
  1. assume literal l, propagate (BCP), if this results in conflict, add unit clause $\neg l$
  2. continue with all literals l until *saturation* (nothing changes)
- quadratic to cubic complexity
  - BCP linear in the size of the formula — 1st linear factor
  - each variable needs to be tried — 2nd linear factor
  - and tried again if some unit has been derived — 3rd linear factor

Failed Literal Probing Extensions

- lifting
  - complete case split: literals implied in all cases become units
  - similar to Stålmark's method and Recursive Learning [PradhamKunz'94]
- asymmetric branching
  - assume all but one literal of a clause to be false
  - if BCP leads to conflict remove originally remaining unassigned literal
  - implemented for a long time in MiniSAT but switched off by default
- generalizations:
  - vivification [PietteHamadiSais ECAI'08]
  - distillation [JinSomenzi'05][HanSomenzi DAC'07] probably most general (+ tries)

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - can be applied recursively (however, is often too expensive)
- Stålmarck's Method
  - works on triplets (intermediate form of the Tseitin transformation):

    $x = (a \wedge b)$, $y = (c \vee d)$, $z = (e \oplus f)$ etc.
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable
- Recursive Learning (Kunz & Pradhan)
  - (originally) works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value

Bounded Variable Elimination (VE)

[DavisPutnam60][Biere SAT’04] [SubbarayanPradhan SAT’04] [EénBiere SAT’05]

- use DP to existentially quantify out variables as in [DavisPutnam60]

- only remove a variable if this does not add (too many) clauses

  - do not count tautological resolvents

  - detect units on-the-fly

- schedule removal attempts with a priority queue [Biere SAT’04] [EénBiere SAT’05]

  - variables ordered by the number of occurrences

- strengthen and remove subsumed clauses (on-the-fly)
  (SATeLite [EénBiere SAT’05] and Quantor [Biere SAT’04])

Fast (Self) Subsumption

- for each (new or strengthened) clause
  - traverse list of clauses of the least occuring literal in the clause
  - check whether traversed clauses are subsumed or
  - strengthen traversed clauses by self-subsumption [EénBiere SAT'05]
  - use Bloom Filters (as in "bit-state hashing"), aka signatures
- check old clauses being subsumed by new clause: backward (self) subsumption
  - new clause (self) subsumes existing clause
  - new clause smaller or equal in size
- check new clause to be subsumed by existing clauses forward (self) subsumption
  - can be made more efficient by one-watcher scheme [Zhang-SAT'05]

Variable Instantiation

[AnderssonBjesseCookHanna DAC’02] also in Oepir SAT solver, this is our reformulation

- for all iterals l
  - for all clauses c in which l occurs (with this particular phase)
    - $*$ assume the negation of all the other literals in c, assume l
    - $*$ if BCP does not lead to a conflict continue with next literal in outer loop
  - if all clauses produced a conflict permanently assign $\neg l$

Correctness: Let $c = l \vee d$, assume $\neg d \wedge l$.

If this leads to a conflict $d \vee \neg l$ could be learned (but is not added to the CNF).

Self subsuming resolution with c results in d and c is removed.

If all such cases lead to a conflict, $\neg l$ becomes a pure literal.

Autarkies

Generalization of pure literals.

Given a partial assignment σ.

A clause of a CNF is “touched” by σ if it contains a literal assigned by σ.

A clause of a CNF is “satisfied” by σ if it contains a literal assigned to true by σ.

If all touched clauses are satisfied then σ is an “autarky”.

All clauses touched by an autarky can be removed.

Example: $(-1\ 2)(-1\ 3)(1\ -2\ -3)(2\ 5)\cdots$ (more clauses without 1 and 3).

Then $\sigma = \{-1, -3\}$ is an autarky.

Blocked Clause Elimination (BCE)

one clause $C \in F$ with l

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

fix a CNF F

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

$a \vee b \vee l$

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

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

Proof assume assignment σ satisfies $F \backslash C$ but not C

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

Blocked Clauses

Kullmann'99

Definition A literal l in a clause C of a CNF F blocks C w.r.t. F if for every clause $C' \in F$ with $\bar{l} \in C'$, the resolvent $(C \setminus \{l\}) \cup (C' \setminus \{\bar{l}\})$ obtained from resolving C and C' on l is a tautology.

Definition [Blocked Clause] A clause is blocked if has a literal that blocks it.

Definition [Blocked Literal] A literal is blocked if it blocks a clause.

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

only first clause is not blocked.

second clause contains two blocked literals: a and $\bar{c}$.

literal c in the last clause is blocked.

after removing either $(a \vee \bar{b} \vee \bar{c})$ or $(\bar{a} \vee c)$, the clause $(a \vee b)$ becomes blocked
actually all clauses can be removed

Blocked Clauses and Encoding / Preprocessing Techniques

JärvisaloBiereHeule'10 + JAR Article

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

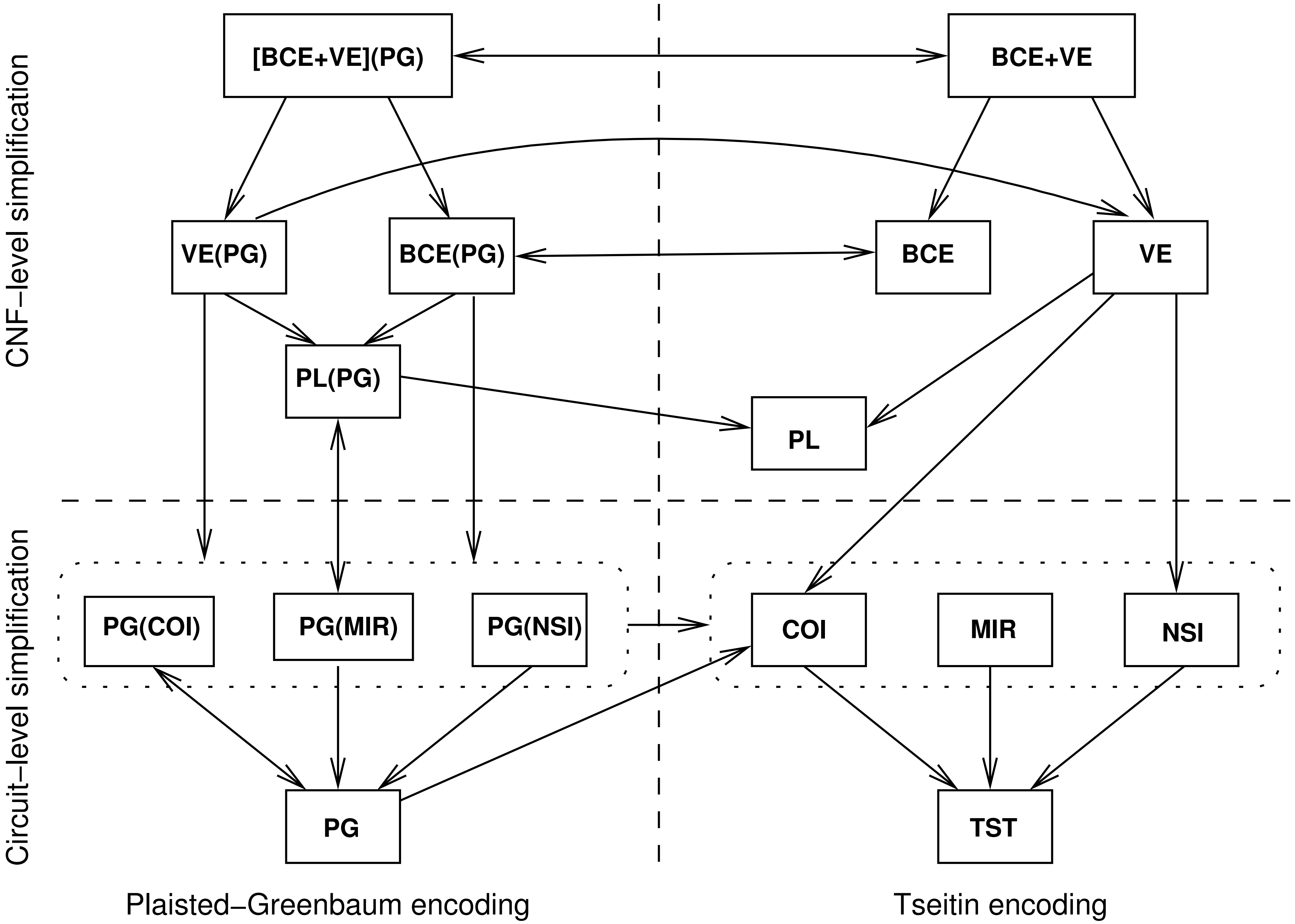
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination


CNF–level simplification
Circuit–level simplification
[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
Plaisted–Greenbaum encoding
Tseitin encoding

Inprocessing: Interleaving Preprocessing and Search

PrecoSAT [Biere'09], Lingeling [Biere'10], now also in CryptoMiniSAT (Mate Soos)

- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [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

Other Inprocessing / Preprocessing Techniques

JärvisaloHeuleBiere'12

equivalent literal substitution find strongly connected components in binary implication graph, replace equivalent literals by representatives

boolean ring reasoning extract XORs, then Gaussian elimination etc.

hyper-binary resolution focus on producing binary resolvents

hidden/asymmetric tautology elimination discover redundant clauses through probing

covered clause elimination use covered literals in probing for redundant clauses

unhiding randomized algorithm (one phase linear) for clause removal and strengthening

Benefits of Inprocessing

- 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 VE 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 get right! "Inprocessing Rules" [JärvisaloHeuleBiere'12]

Inprocessing Rules

JärvisaloHeuleBiere'12

$$\frac{\varphi\,[\,\rho \wedge C\,]\,\sigma}{\varphi \wedge C\,[\,\rho\,]\,\sigma}$$

STRENGTHEN

$$\frac{\varphi\,[\,\rho \wedge C\,]\,\sigma}{\varphi\,[\,\rho\,]\,\sigma}$$

FORGET

$$\frac{\varphi\,[\,\rho\,]\,\sigma}{\varphi\,[\,\rho \wedge C\,]\,\sigma}\;\mathcal{L}$$

LEARN

$$\frac{\varphi \wedge C\,[\,\rho\,]\,\sigma}{\varphi\,[\,\rho \wedge C\,]\,\sigma, l{:}C}\;\mathcal{W}$$

WEAKEN

$\mathcal{L}$ is that $\boxed{\varphi \wedge \rho}$ and $\boxed{\varphi \wedge \rho \wedge C}$ are satisfiability-equivalent.

$\mathcal{W}$ is that $\boxed{\varphi}$ and $\boxed{\varphi \wedge C}$ are satisfiability-equivalent.

RAT – Resolution Asymmetric Tautology

JärvisaloHeuleBiere'12

"*resolution look-ahead*"

Clause R *asymmetric tautology* (AT) w.r.t. G iff $G \wedge \neg C$ refuted by BCP.

Given clause $C \in F$, $l \in C$.

Assume all resolvents R of C on l with clauses in F are AT w.r.t. $F \backslash \{C\}$.

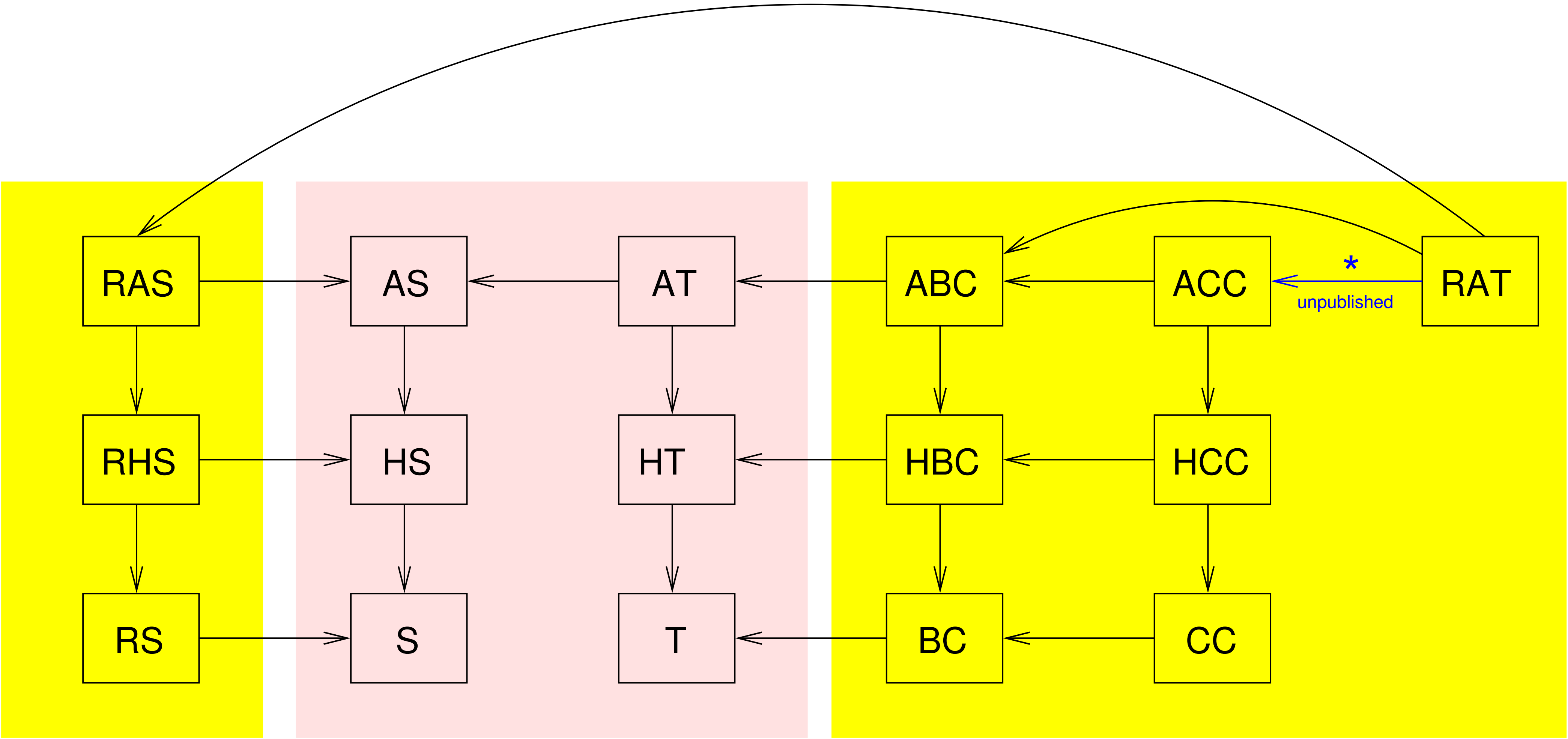
Then C is called *resolution asymmetric tautology* (RAT) w.r.t. F on l.

In this case F is satisfiability equivalent to $F \backslash \{C\}$.

Inprocessing Rules with RAT simulate all techniques in current SAT solvers

Clause Elimination Procedures

JärvisaloHeuleBiere’12



logically equivalent

satisfiability equivalent

Asymmetric, **B**locked **C**lause, **C**overed **C**lause,
Hidden, **R**esolution, **S**ubsumed

Parallel SAT Solving

- application level parallelism
  - run multiple “properties” at the same time
  - run multiple “engines” at the same time (streaming)
- portfolio solving
  - predict best solver through machine learning techniques SATzilla
  - run multiple solvers in parallel or sequentially (with/without “sharing”)
    ManySAT, Plingeling, ppfolio . . .
- split search space
  - guiding path principle [ZhangBonacinaHsiang’96]
  - cube & conquer [HeuleKullmanWieringaBiere’11]
- low-level parallelsim: parellize BCP (threads, FPGA, GPU, . . .) P complete

Cube & Conquer

HeuleKullmanWieringaBiere'11

- use Look-Ahead at the top of the search tree, CDLC at the bottom
  - Look-Ahead solvers provide "good" global decisions but are slow
  - CDCL solvers are extremely fast based on local heuristics
- **when to switch from Look-Ahead to CDCL?**
  - limit decision height of Look-Ahead solver, e.g. 20 maximum tree height
  - avoid having too many branches closed by (slow) Look-Ahead
  - Concurrent Cube & Conquer runs CDCL and Look-Ahead in parallel
- open branches = cubes $\Rightarrow$ solve in parallel
- solves hard instances, which none of the other approaches can