# Searching, Simplifying, Proving

## A Tutorial on Modern SAT Solving


Armin Biere

JOHANNES KEPLER
UNIVERSITÄT LINZ


ETAPS'18

Thessaloniki, Greece

April 18, 2018

# Dress Code Tutorial Speaker as SAT Problem

- propositional logic:
  - variables **tie** **shirt**
  - negation $\neg$ (not)
  - disjunction $\vee$ (or)
  - conjunction $\wedge$ (and)
- clauses (conditions / constraints)

1. clearly one should not wear a **tie** without a **shirt** $\neg\textbf{tie} \vee \textbf{shirt}$
2. not wearing a **tie** nor a **shirt** is impolite $\textbf{tie} \vee \textbf{shirt}$
3. wearing a **tie** and a **shirt** is overkill $\neg(\textbf{tie} \wedge \textbf{shirt}) \equiv \neg\textbf{tie} \vee \neg\textbf{shirt}$

- Is this formula in conjunctive normal form (CNF) **satisfiable?**

$$(\neg\textbf{tie} \vee \textbf{shirt}) \wedge (\textbf{tie} \vee \textbf{shirt}) \wedge (\neg\textbf{tie} \vee \neg\textbf{shirt})$$


SAT-Race 2010 Award
"Plingeling"
by Armin Biere
is awarded the title of
Best Parallel Solver
Chair of SAT-Race Organizing Committee
SAT-Race 2010 Award
"Lingeling"
by Armin Biere
is awarded the title of
Second Prize Winn
Edinburgh, Scotland, July 14, 2010
Tenth International Conference
On Theory and Applications of Satisfiability Testing
Competition 2007
Gold medal
Awarded to Picosat 535 solver
written by Armin Biere
for best performance by a solver in the industrial category, satisfiable benchmarks specialty.
The SAT2007 competition judges and organiz
Twelfth International Conference
On Theory and Applications of Satisfiability Testing
Competition 2009
Gold medal
Eighteenth International Conference on Theory and Applications of Satisfiability Testing
SAT-Race 2015
Main Track
Special Prize: Most Innovative Solver
Awarded to: Lingeling ar15bal
Written by: Armin Biere
Parallel Track
3rd Prize
Treengeling sr15baq
2nd Prize
SAT-Race 2010
Configurable SAT Solver Challenge 2014
Best Parallel Solver 2010
SAT COMPETITION 2013
Competition 200
Silver meda
International Symposium
Delft
2002

Special thanks are due to Armin Biere, Randy Bryant, Sam Buss, Niklas Eén, Ian Gent, Marijn Heule, Holger Hoos, Svante Janson, Peter Jeavons, Daniel Kroening, Oliver Kullmann, Massimo Lauria, Wes Pegden, Will Shortz, Carsten Sinz, Niklas Sörensson, Udo Wermuth, Ryan Williams, and . . . for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections. Thanks also to Stanford's Information Systems Laboratory for providing extra computer power when my laptop machine was inadequate.

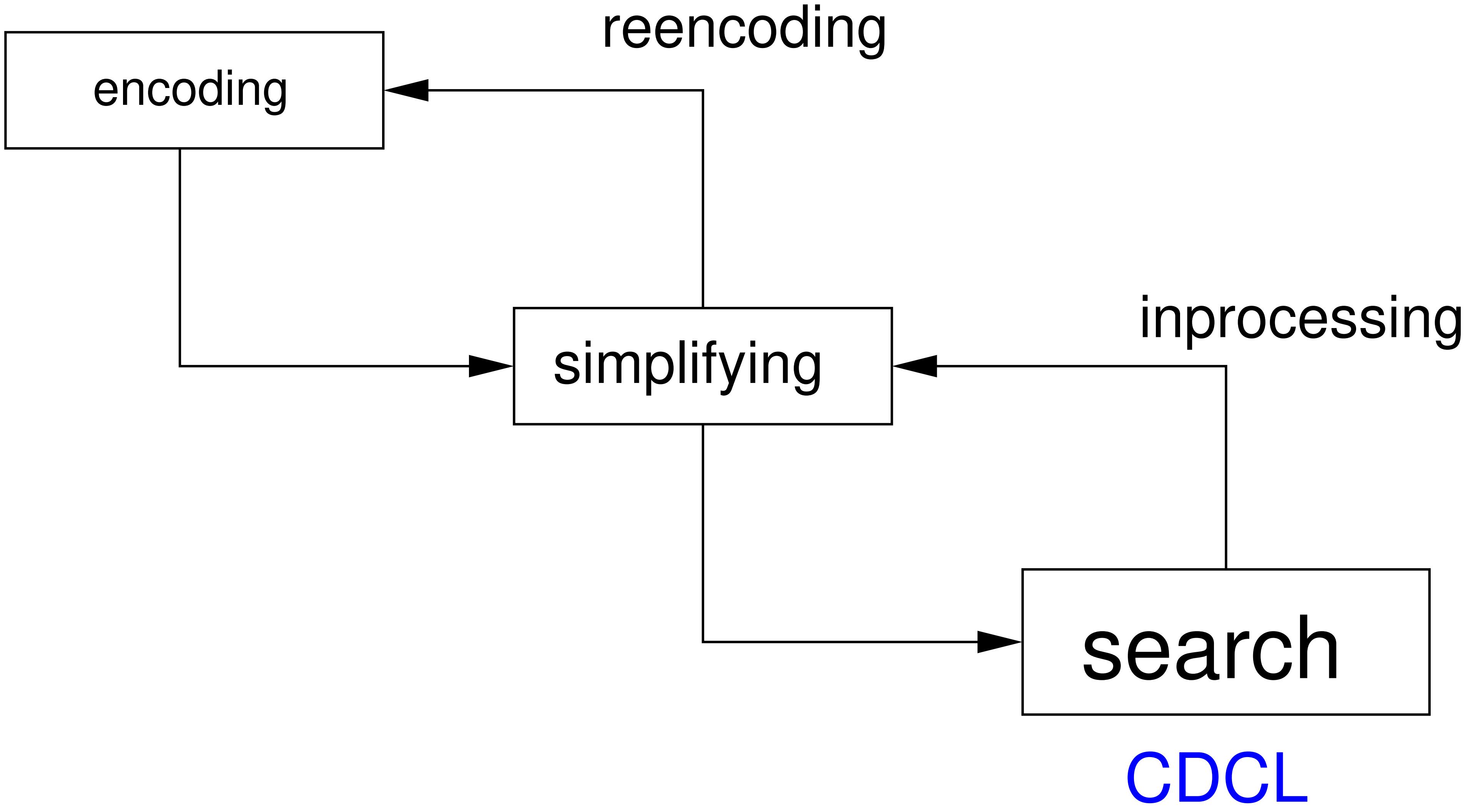
Biere
Bryant
Buss
Eén
Gent
Heule
Hoos
Janson
Jeavons
Kroening
Kullmann
Lauria
Pegden
Shortz
Sinz
Sörensson
Wermuth
Williams

* * *

Wow — Section 7.2.2.2 has turned out to be the longest section, by far, in *The Art of Computer Programming*. The SAT problem is evidently a "killer app," because it is key to the solution of so many other problems. Consequently I can only hope that my lengthy treatment does not also kill off my faithful readers! As I wrote this material, one topic always seemed to flow naturally into another, so there was no neat way to break this section up into separate subsections. (And anyway the format of *TAOCP* doesn't allow for a Section 7.2.2.2.1.)

Internet
MPR
Internet

# What is Practical SAT Solving?



# Equivalence Checking 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?

# Compilation

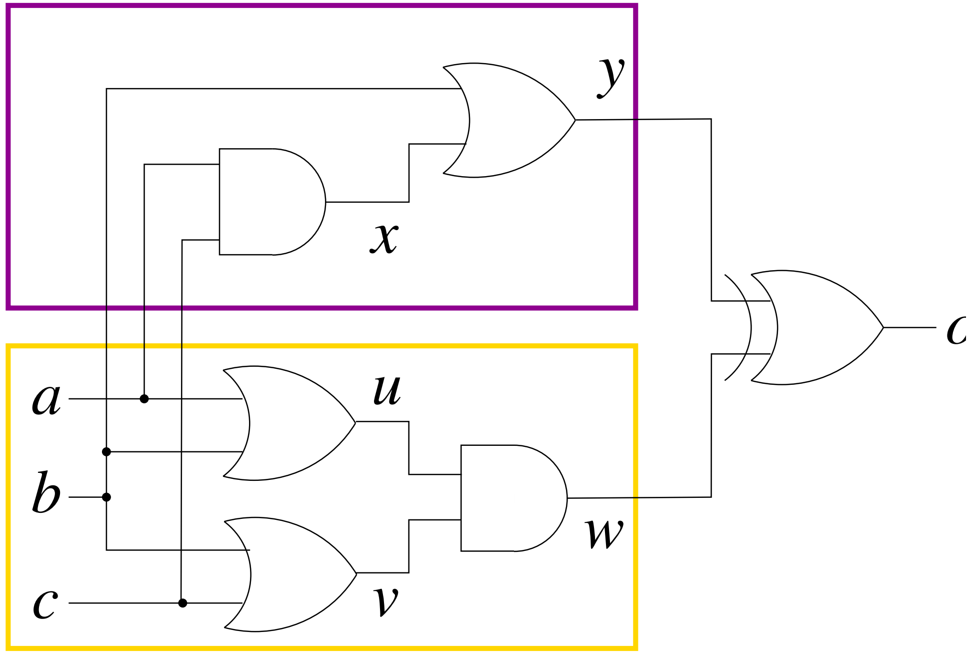
$$\begin{aligned}
\textit{original} &\equiv \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge \textbf{if } \neg a \textbf{ then } g \textbf{ else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

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

$$(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \quad \not\Leftrightarrow \quad a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)$$

satisfying assignment gives counter-example to equivalence

# Tseitin Transformation: Circuit to CNF




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

$$o \land (x \to a) \land (x \to c) \land (x \leftarrow a \land c) \land \ \ldots$$

$$\boxed{o \land (\overline{x} \lor a) \land (\overline{x} \lor c) \land (x \lor \overline{a} \lor \overline{c}) \land \ \ldots}$$

# Tseitin Transformation: Gate Constraints

$$
\begin{array}{rrcl}
\text{Negation:} & x \leftrightarrow \overline{y} & \Leftrightarrow & (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\
 & & \Leftrightarrow & (\overline{x} \vee \overline{y}) \wedge (y \vee x) \\[1em]
\text{Disjunction:} & x \leftrightarrow (y \vee z) & \Leftrightarrow & (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\
 & & \Leftrightarrow & (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z) \\[1em]
\text{Conjunction:} & x \leftrightarrow (y \wedge z) & \Leftrightarrow & (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\
 & & \Leftrightarrow & (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\
 & & \Leftrightarrow & (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x) \\[1em]
\text{Equivalence:} & x \leftrightarrow (y \leftrightarrow z) & \Leftrightarrow & (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
 & & \Leftrightarrow & (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y))) \wedge ((y \leftrightarrow z) \rightarrow x) \\
 & & \Leftrightarrow & (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\
 & & \Leftrightarrow & (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\
 & & \Leftrightarrow & (\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 & (\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 & (\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{array}
$$

# Bit-Blasting of Bit-Vector Addition

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

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

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

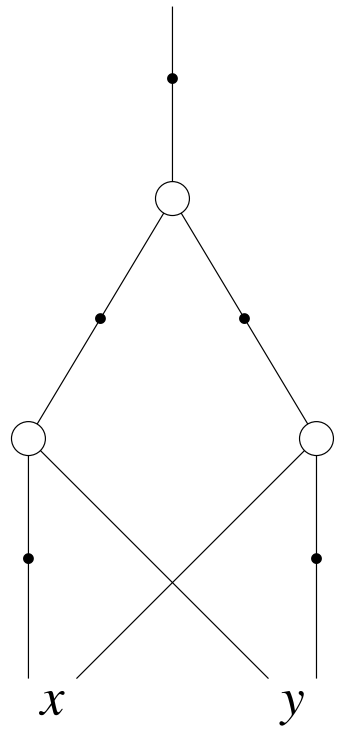
where

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

# Intermediate Representations

- encoding directly into CNF is hard, so we use intermediate levels:
  1. application level
  2. bit-precise semantics world-level operations (bit-vectors)
  3. bit-level representations such as And-Inverter Graphs (AIGs)
  4. conjunctive normal form (CNF)
- encoding “logical” constraints is another story

# XOR as AIG

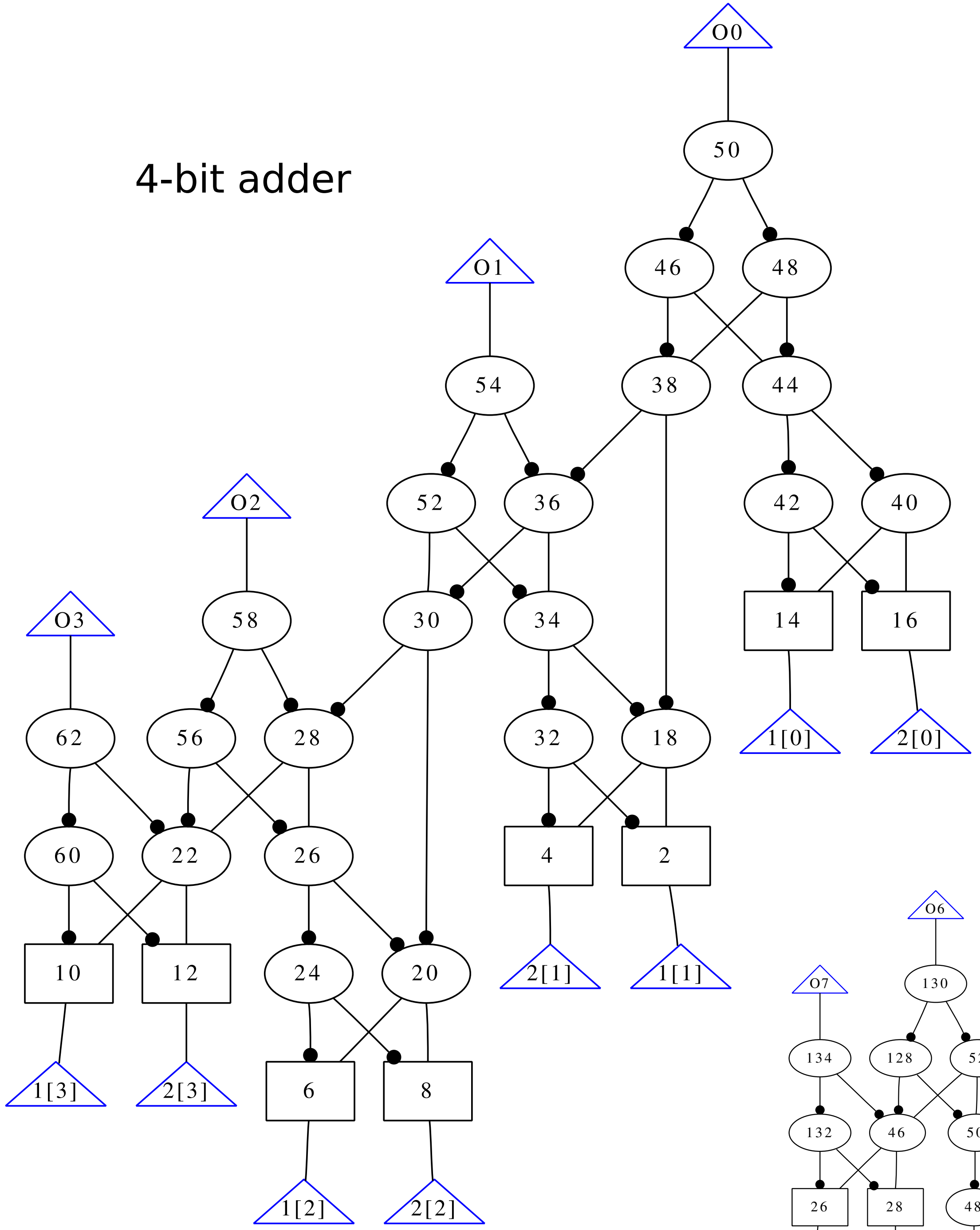


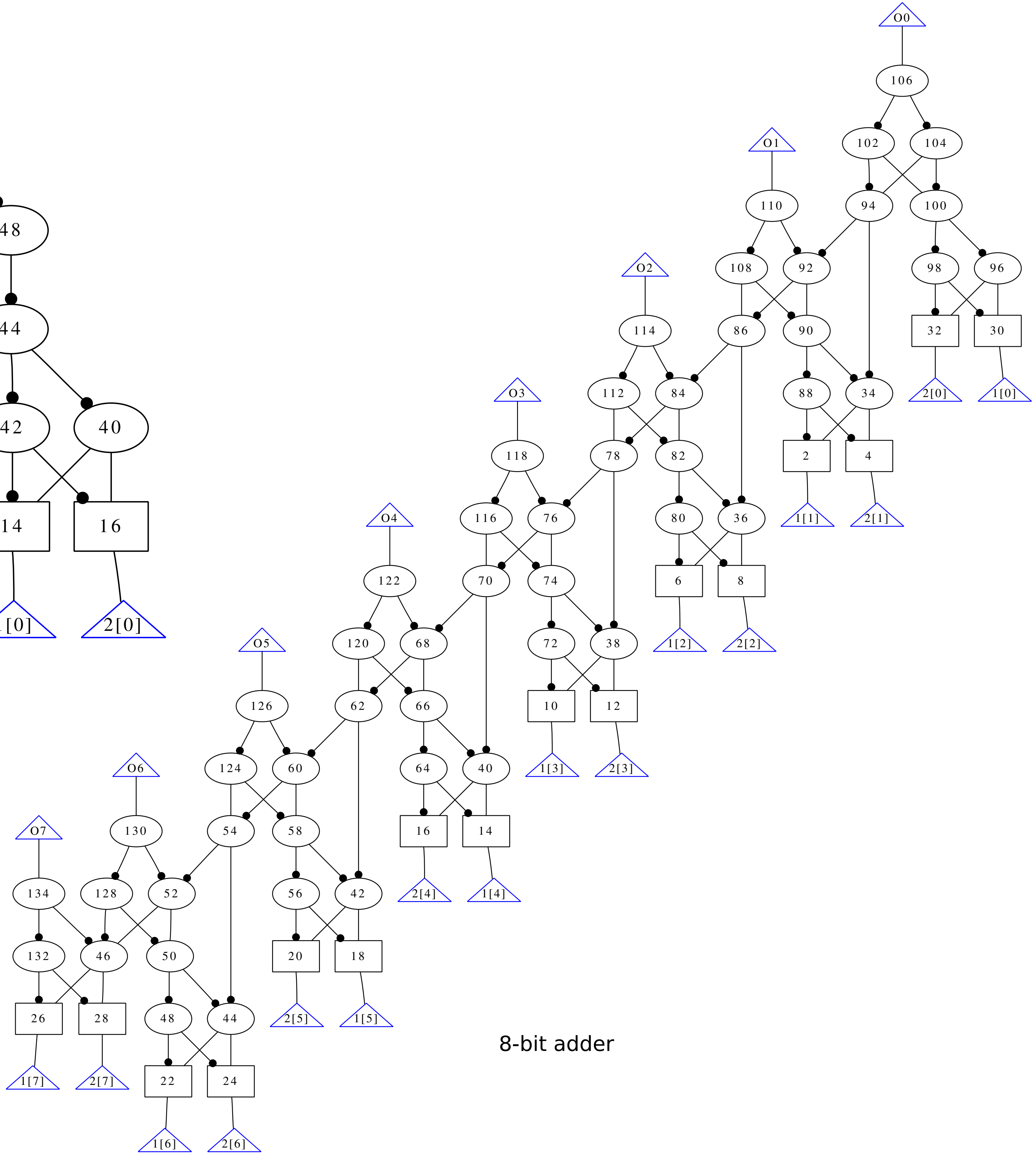

negation/sign are edge attributes

not part of node

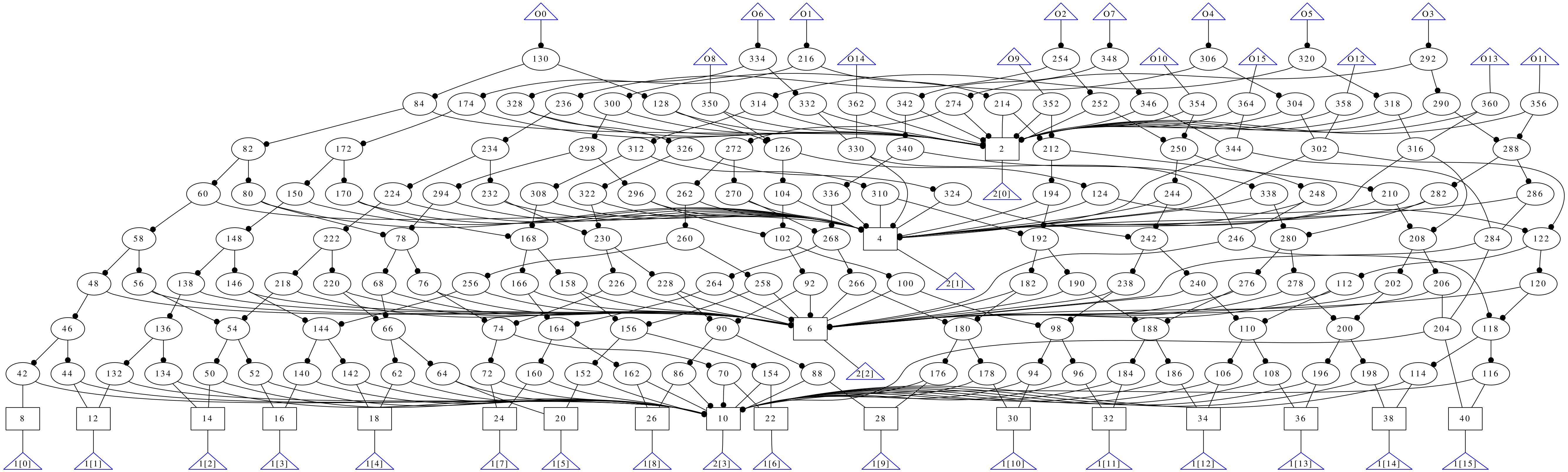
$$x \text{ xor } y \equiv (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \equiv \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

4-bit adder

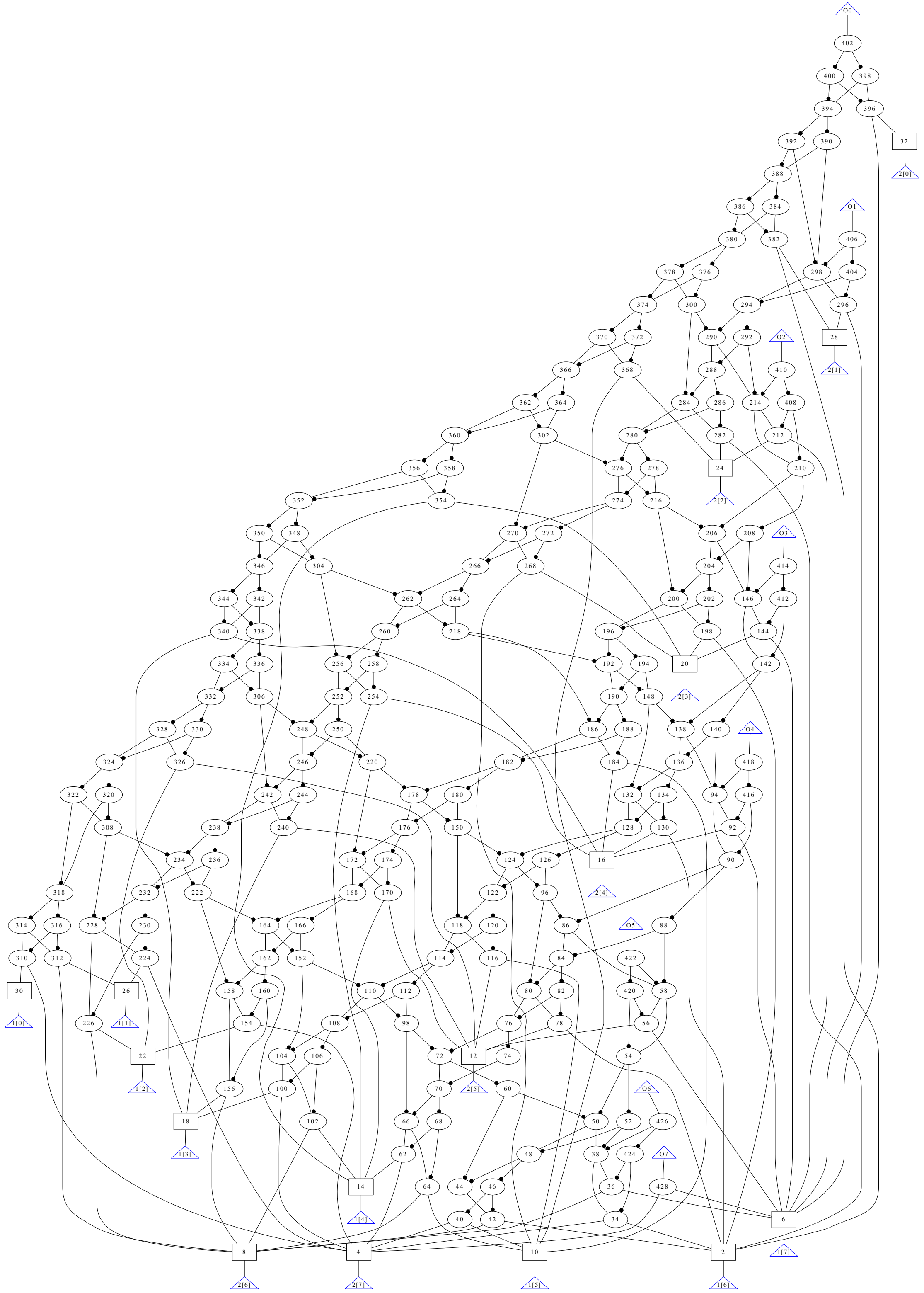


8-bit adder




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


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

# Encoding Logical Constraints

- Tseitin 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 stable models / logical programming / ASP

# Example of Logical Constraints: Cardinality Constraints

- given a set of literals $\{l_1, \dots l_n\}$
  - constraint the number of literals assigned to *true*
  - $l_1 + \cdots + l_n \geq k$ or $l_1 + \cdots + l_n \leq k$ or $l_1 + \cdots + l_n = k$
  - combined make up exactly all fully symmetric boolean functions

- multiple encodings of cardinality constraints
  - naïve encoding exponential: at-most-one quadratic, at-most-two cubic, etc.
  - quadratic $O(k \cdot n)$ encoding goes back to Shannon
  - linear $O(n)$ parallel counter encoding [Sinz'05]

- many variants even for at-most-one constraints
  - for an $O(n \cdot \log n)$ encoding see Prestwich's chapter in Handbook of SAT

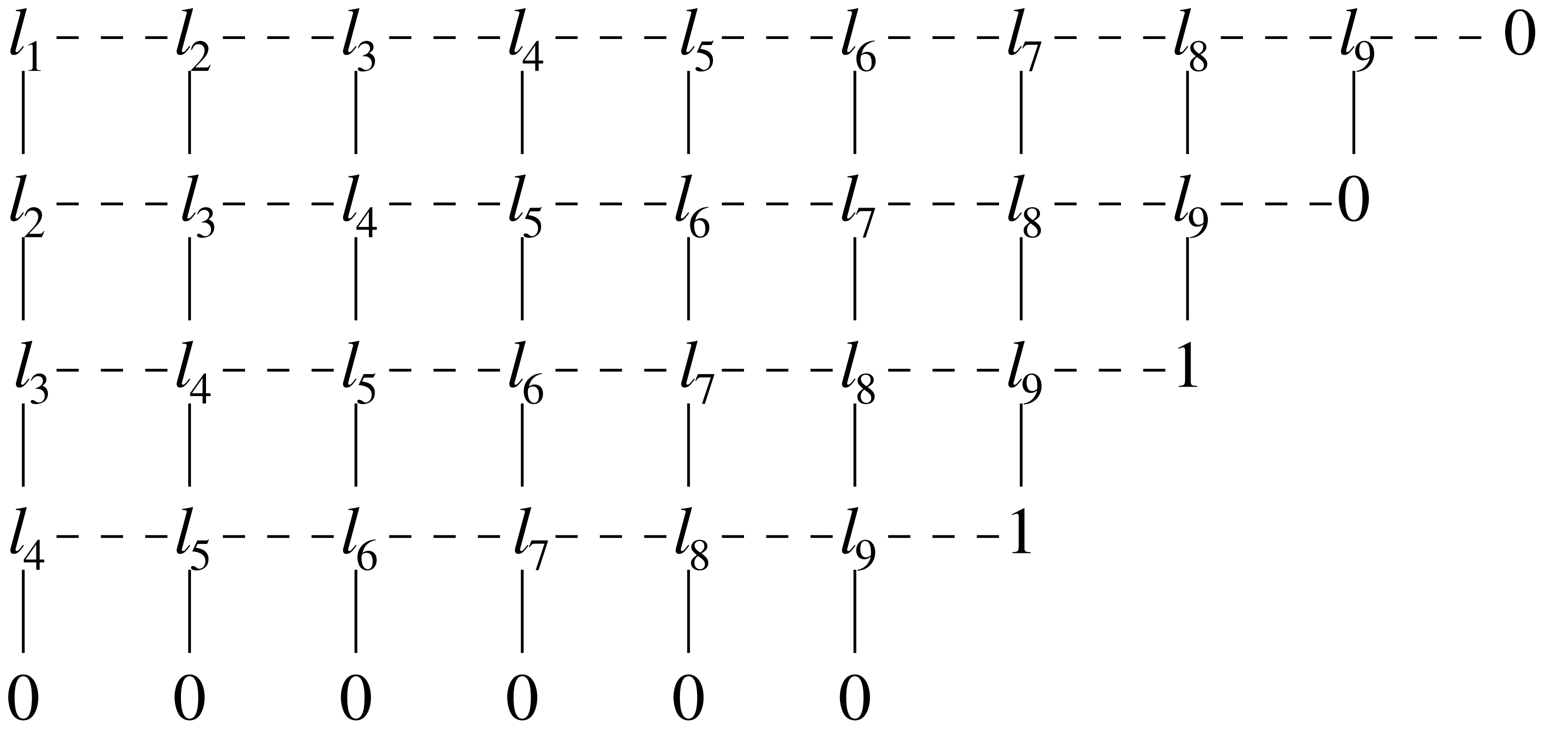
- Pseudo-Boolean constraints (PB) or 0/1 ILP constraints have many encodings too

$$2 \cdot \overline{a} + \overline{b} + c + \overline{d} + 2 \cdot e \ \geq \ 3$$

actually used to handle MaxSAT in SAT4J for configuration in Eclipse

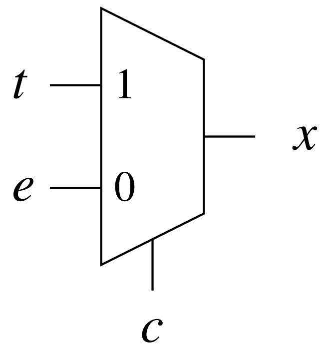
# BDD-Based Encoding of Cardinality Constraints

$$2 \leq l_1 + \cdots l_9 \leq 3$$




If-Then-Else gates (MUX) with “then” edge downward, dashed “else” edge to the right

# Tseitin Encoding of If-Then-Else Gate




$$x \leftrightarrow (c \; ? \; t : e) \;\Leftrightarrow\; (x \rightarrow (c \rightarrow t)) \wedge (x \rightarrow (\bar{c} \rightarrow e)) \wedge (\bar{x} \rightarrow (c \rightarrow \bar{t})) \wedge (\bar{x} \rightarrow (\bar{c} \rightarrow \bar{e}))$$

$$\Leftrightarrow\; (\bar{x} \vee \bar{c} \vee t) \wedge (\bar{x} \vee c \vee e) \wedge (x \vee \bar{c} \vee \bar{t}) \wedge (x \vee c \vee \bar{e})$$

minimal but not arc consistent:

- if $t$ and $e$ have the same value then $x$ needs to have that too
- possible additional clauses

$$(\bar{t} \wedge \bar{e} \rightarrow \bar{x}) \;\equiv\; (t \vee e \vee \bar{x}) \qquad\qquad (t \wedge e \rightarrow x) \;\equiv\; (\bar{t} \vee \bar{e} \vee x)$$

- but can be learned or derived through preprocessing (ternary resolution)
  keeping those clauses redundant is better in practice

# DIMACS Format

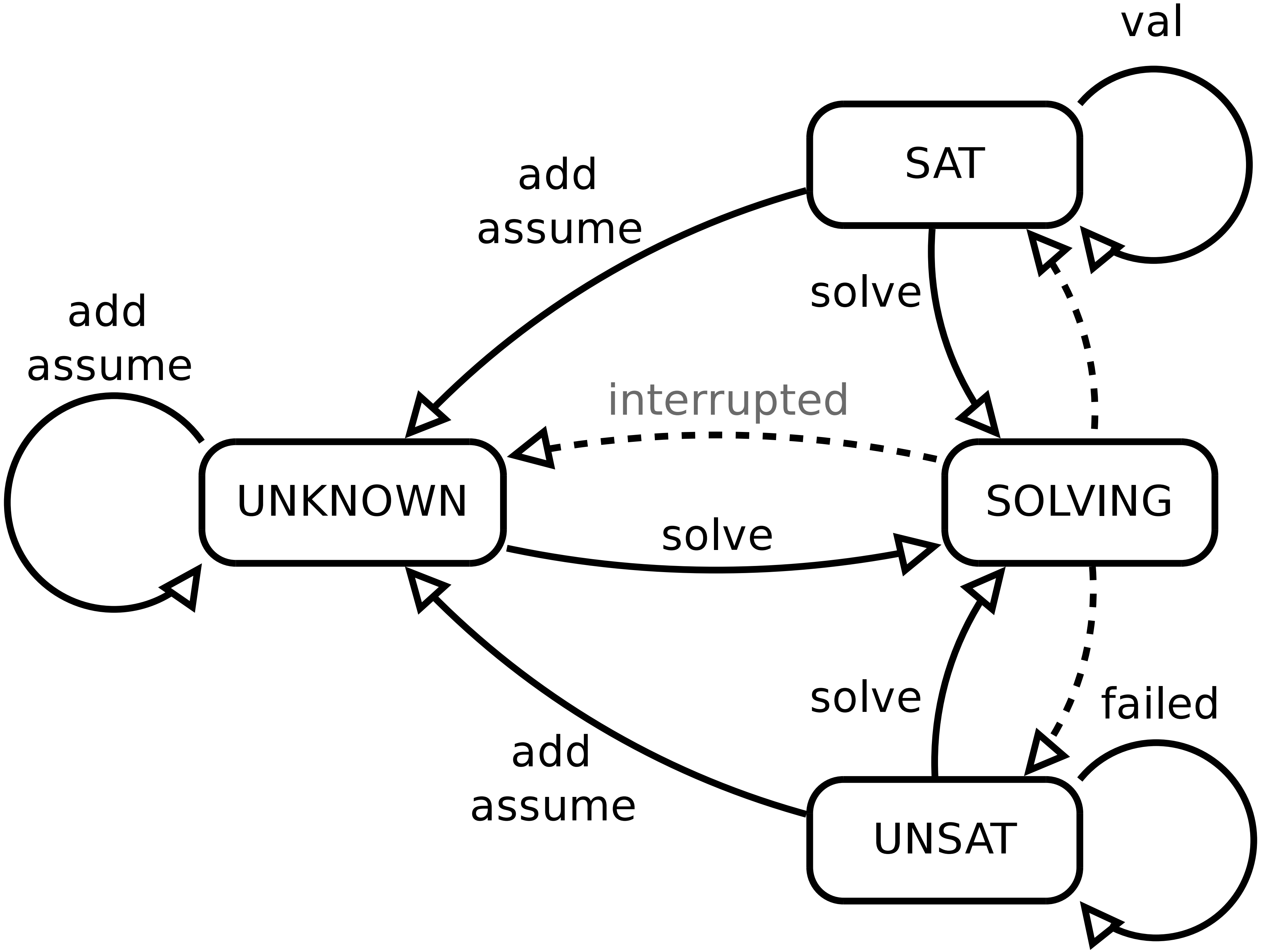
```
$ cat example.cnf
c comments start with 'c' and extend until the end of the line
c
c variables are encoded as integers:
c
c   'tie'   becomes '1'
c   'shirt' becomes '2'
c
c header 'p cnf <variables> <clauses>'
c
p cnf 2 3
-1  2 0          c  !tie  or  shirt
 1  2 0          c   tie  or  shirt
-1 -2 0          c  !tie  or !shirt
```

```
$ picosat example.cnf
s SATISFIABLE
v -1 2 0
```

# SAT Application Programmatic Interface (API)

- incremental usage of SAT solvers
  - add facts such as clauses incrementally
  - call SAT solver and get satisfying assignments
  - optionally retract facts
- retracting facts
  - remove clauses explicitly: complex to implement
  - push / pop: stack like activation, no sharing of learned facts
  - MiniSAT assumptions [EénSörensson’03]
- assumptions
  - unit assumptions: assumed for the next SAT call
  - easy to implement: force SAT solver to decide on assumptions first
  - shares learned clauses across SAT calls
- IPASIR: Reentrant Incremental SAT API
  - used in the SAT competition / race since 2015 [BalyoBiereIserSinz’16]

# IPASIR Model



```c
#include "ipasir.h"
#include <assert.h>
#include <stdio.h>
#define ADD(LIT) ipasir_add (solver, LIT)
#define PRINT(LIT) \
  printf (ipasir_val (solver, LIT) < 0 ?  " -" #LIT : " " #LIT)
int main () {
  void * solver = ipasir_init ();
  enum { tie = 1, shirt = 2 };
  ADD (-tie); ADD ( shirt); ADD (0);
  ADD ( tie); ADD ( shirt); ADD (0);
  ADD (-tie); ADD (-shirt); ADD (0);
  int res = ipasir_solve (solver);
  assert (res == 10);
  printf ("satisfiable:"); PRINT (shirt); PRINT (tie); printf ("\n");
  printf ("assuming now: tie shirt\n");
  ipasir_assume (solver, tie); ipasir_assume (solver, shirt);
  res = ipasir_solve (solver);
  assert (res == 20);
  printf ("unsatisfiable, failed:");
  if (ipasir_failed (solver, tie)) printf (" tie");
  if (ipasir_failed (solver, shirt)) printf (" shirt");
  printf ("\n");
  ipasir_release (solver);
  return res;
}
```

```
$ ./example
satisfiable: shirt -tie
assuming now: tie shirt
unsatisfiable, failed: tie
```

# IPASIR Functions

```
const char * ipasir_signature ();

void * ipasir_init ();

void ipasir_release (void * solver);

void ipasir_add (void * solver, int lit_or_zero);

void ipasir_assume (void * solver, int lit);

int ipasir_solve (void * solver);

int ipasir_val (void * solver, int lit);

int ipasir_failed (void * solver, int lit);

void ipasir_set_terminate (void * solver, void * state,
                           int (*terminate)(void * state));
```

# DP / DPLL

- dates back to the 50'ies:

  1st version DP is <u>resolution based</u> $\Rightarrow$ preprocessing

  2nd version D(P)LL splits space for time $\Rightarrow$ $\boxed{\text{CDCL}}$

- **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 ($\leq$ 20 years) optimizations:

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

  (we will have a look at each of them)

# DP Procedure

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 $\neg x$

$\Rightarrow$ Bounded Variable Elimination

# Bounded Variable Elimination

[EénBiere-SAT'05]

$$\text{Replace} \quad \begin{array}{cc} (\bar{x} \vee a)_1 & (\bar{x} \vee c)_4 \\ (\bar{x} \vee b)_2 & (x \vee d)_5 \\ (x \vee \bar{a} \vee \bar{b})_3 & \end{array} \quad \text{by} \quad \begin{array}{ccc} \cancel{(a \vee \bar{a} \vee \bar{b})}_{13} & (a \vee d)_{15} & (c \vee d)_{45} \\ \cancel{(b \vee \bar{a} \vee \bar{b})}_{23} & (b \vee d)_{25} & \\ (c \vee \bar{a} \vee \bar{b})_{34} & & \end{array}$$

- number of clauses not increasing
- strengthen and remove subsumed clauses too
- most important and most effective preproessing we have

# Bounded Variable Addition

[MantheyHeuleBiere-HVC'12]

$$\text{Replace} \quad \begin{array}{cc} (a \vee d) & (a \vee e) \\ (b \vee d) & (b \vee e) \\ (c \vee d) & (c \vee e) \end{array} \quad \text{by} \quad \begin{array}{ccc} (\bar{x} \vee a) & (\bar{x} \vee b) & (\bar{x} \vee c) \\ (x \vee d) & (x \vee e) & \end{array}$$

- number of clauses has to decrease strictly
- reencodes for instance naive at-most-one constraint encodings

# D(P)LL Procedure

$DPLL(F)$

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

if $F = \top$ **return** satisfiable

if $\perp \in F$ **return** unsatisfiable

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

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

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

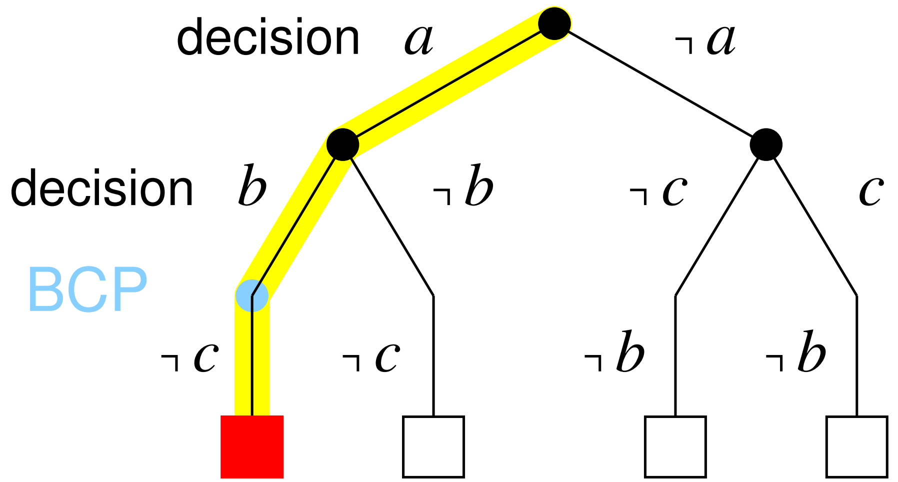
$\Rightarrow$ CDCL

# DPLL Example

$a = 1$

$b = 1$

$c = 0$




clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

# Conflict Driven Clause Learning (CDCL)

[MarqueSilvaSakallah'96]

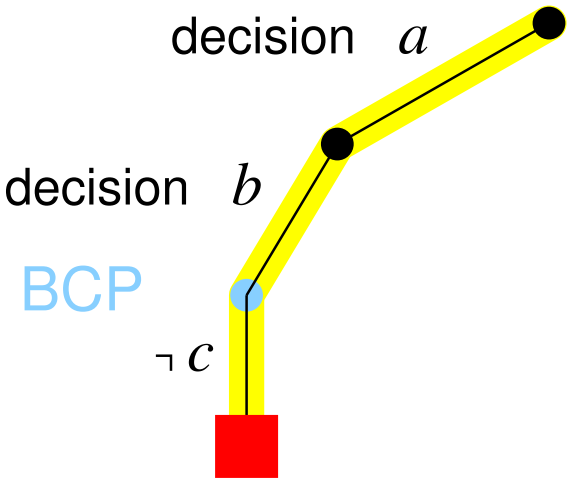
- first implemented in the context of GRASP SAT solver
  - name given later to distinguish it from DPLL
  - not recursive anymore
- essential for SMT
- learning clauses as no-goods
- notion of implication graph
- (first) unique implication points

# Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 1$

$c = 0$




clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

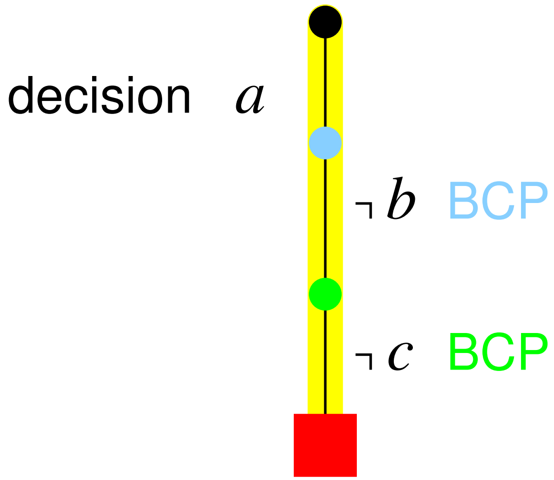
learn $\neg a \vee \neg b$

# Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$




clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

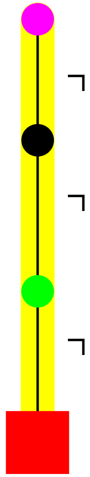
learn $\neg a$

# Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



$\neg a$ BCP

$\neg c$ decision

$\neg b$ BCP

clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

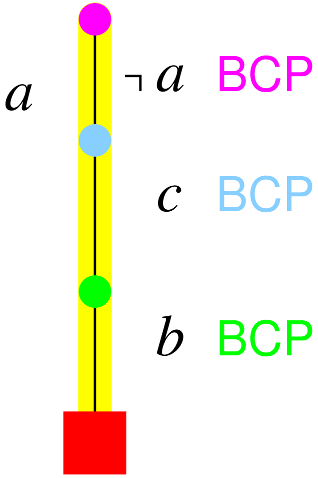
learn $c$

# Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$




clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

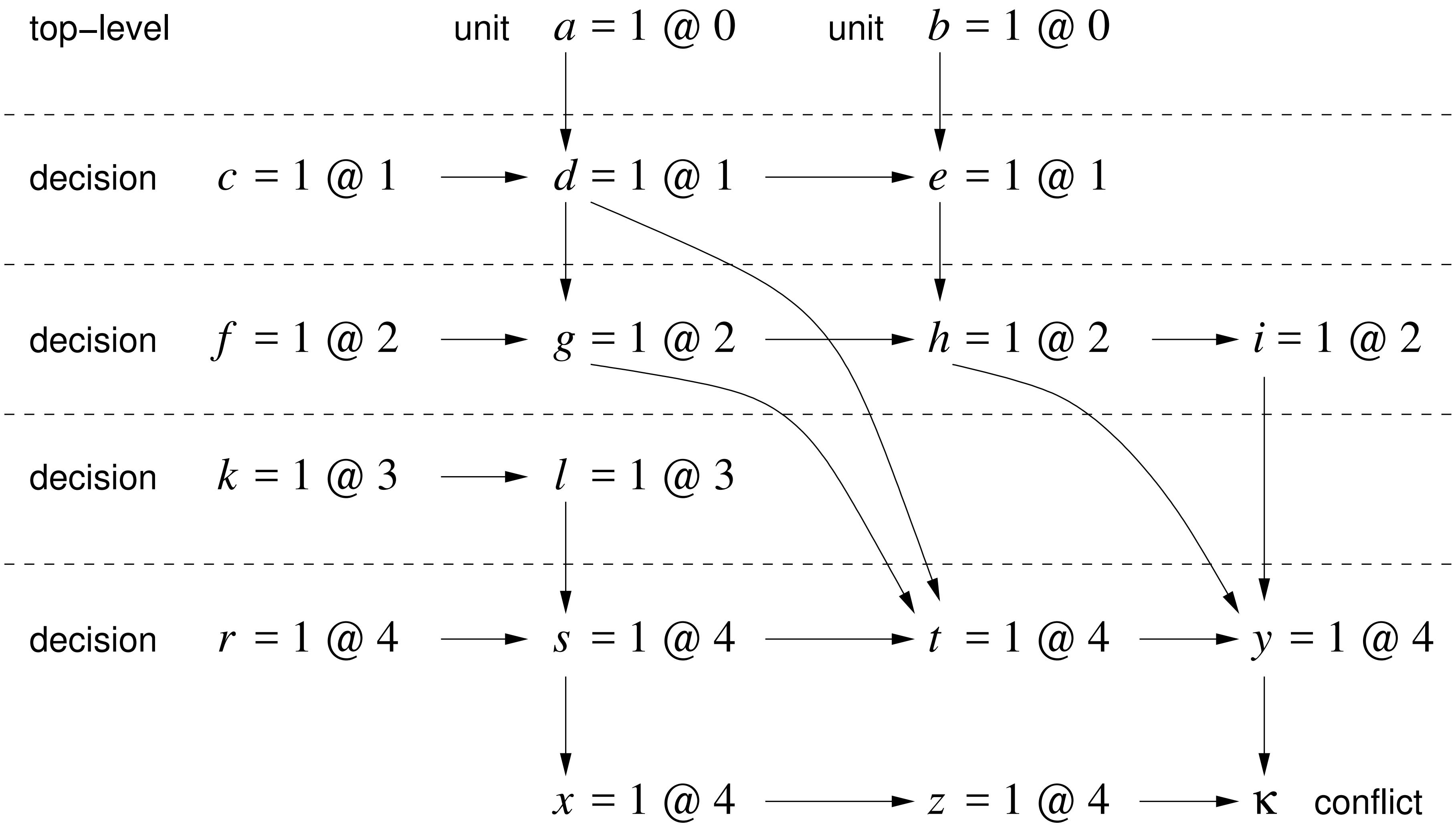
$\neg a$

$c$

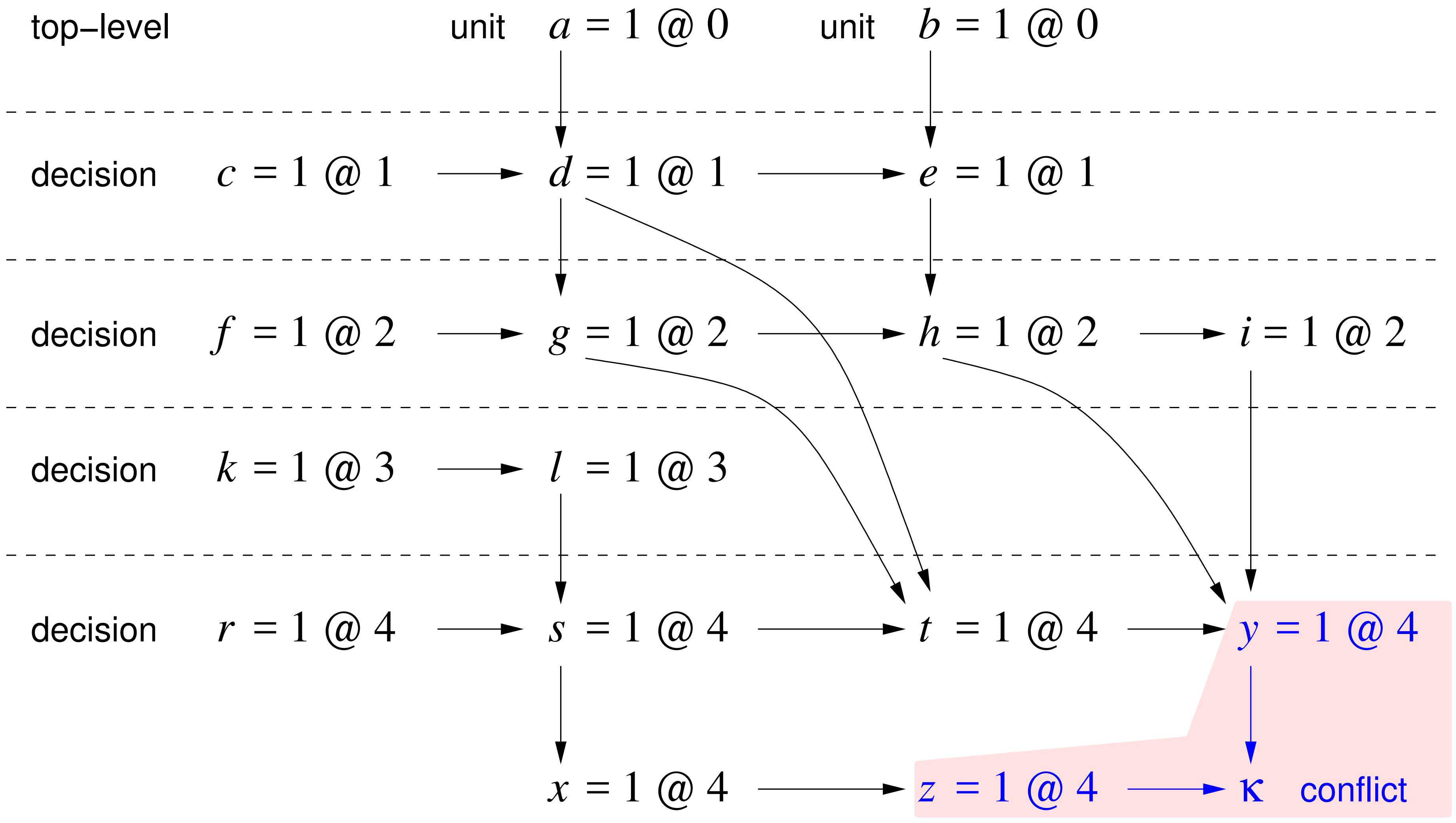
learn $\perp$

empty clause

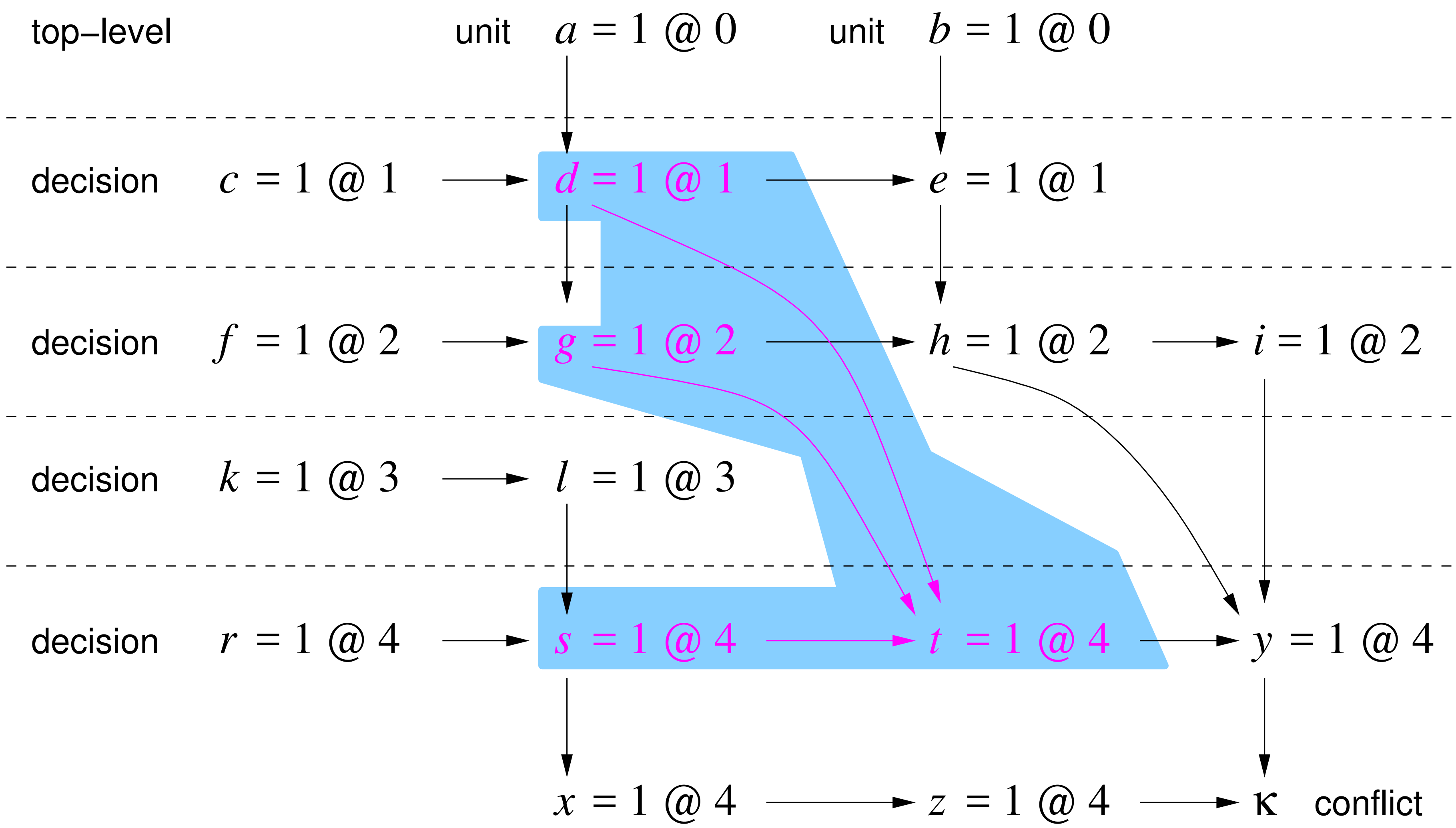
# Implication Graph



# Conflict

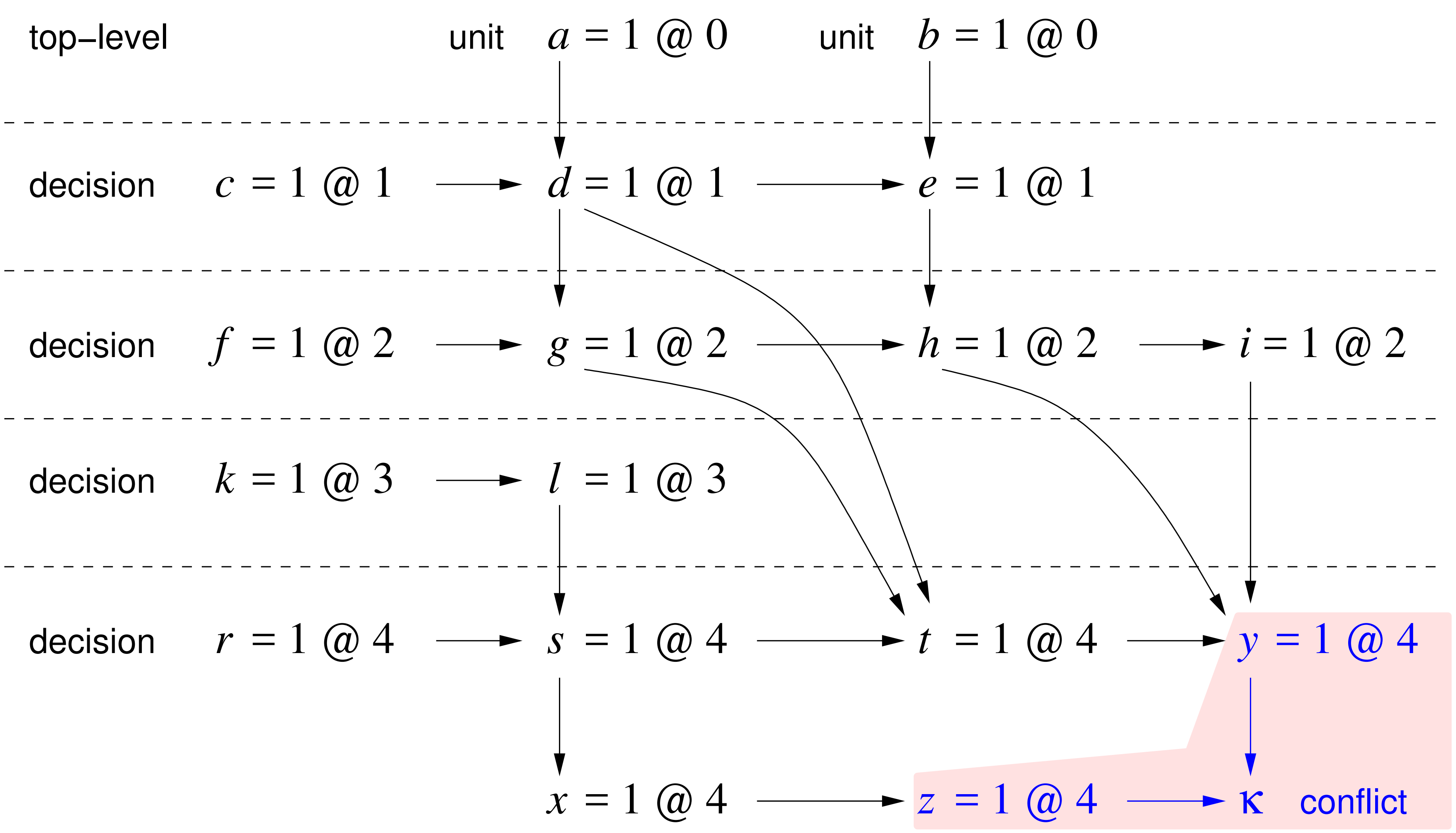

top−level
unit $a = 1 @ 0$
unit $b = 1 @ 0$
decision $c = 1 @ 1$
$d = 1 @ 1$
$e = 1 @ 1$
decision $f = 1 @ 2$
$g = 1 @ 2$
$h = 1 @ 2$
$i = 1 @ 2$
decision $k = 1 @ 3$
$l = 1 @ 3$
decision $r = 1 @ 4$
$s = 1 @ 4$
$t = 1 @ 4$
$y = 1 @ 4$
$x = 1 @ 4$
$z = 1 @ 4$
$\kappa$ conflict

# Antecedents / Reasons



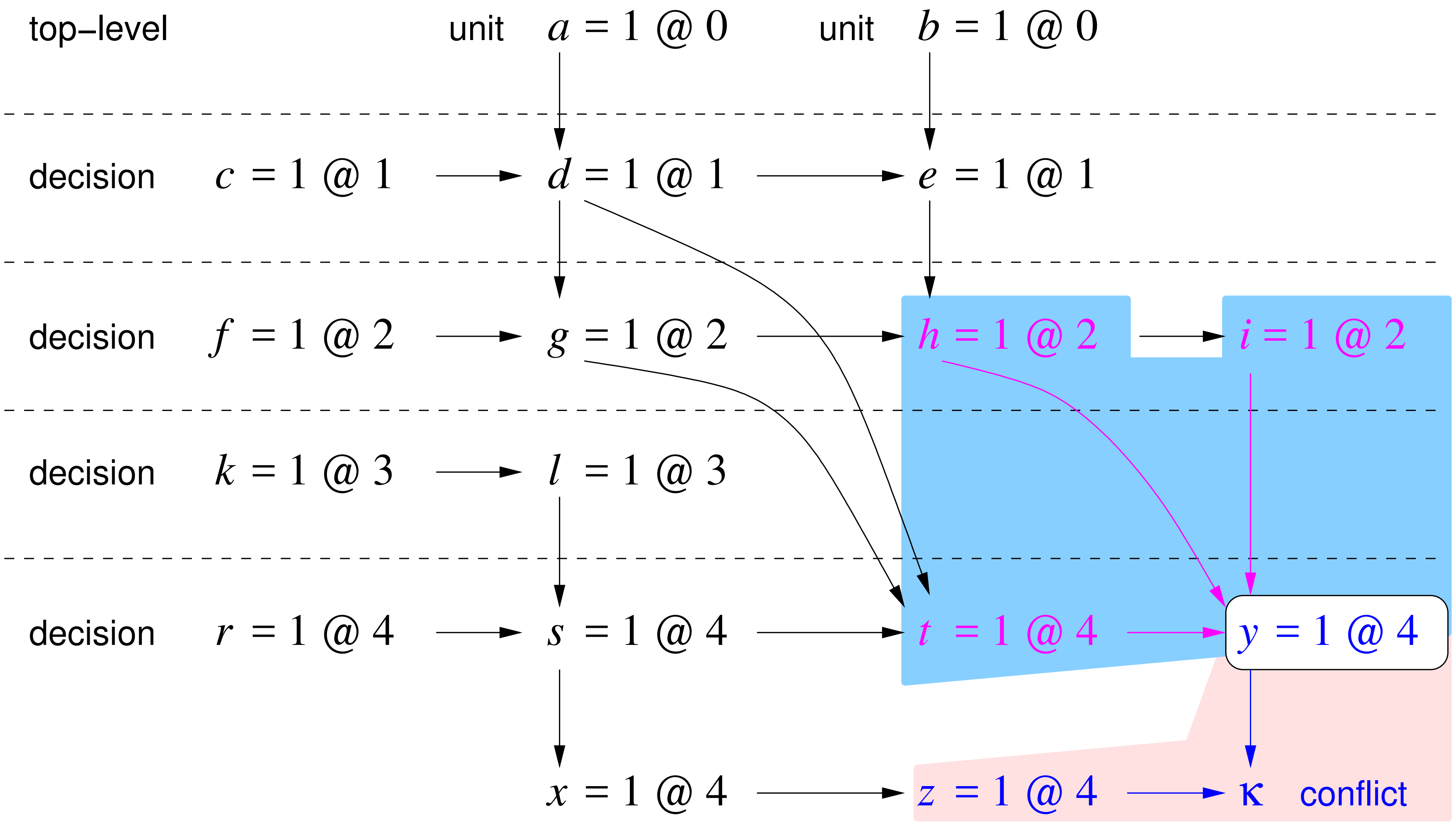

$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee t)$$

# Conflicting Clauses



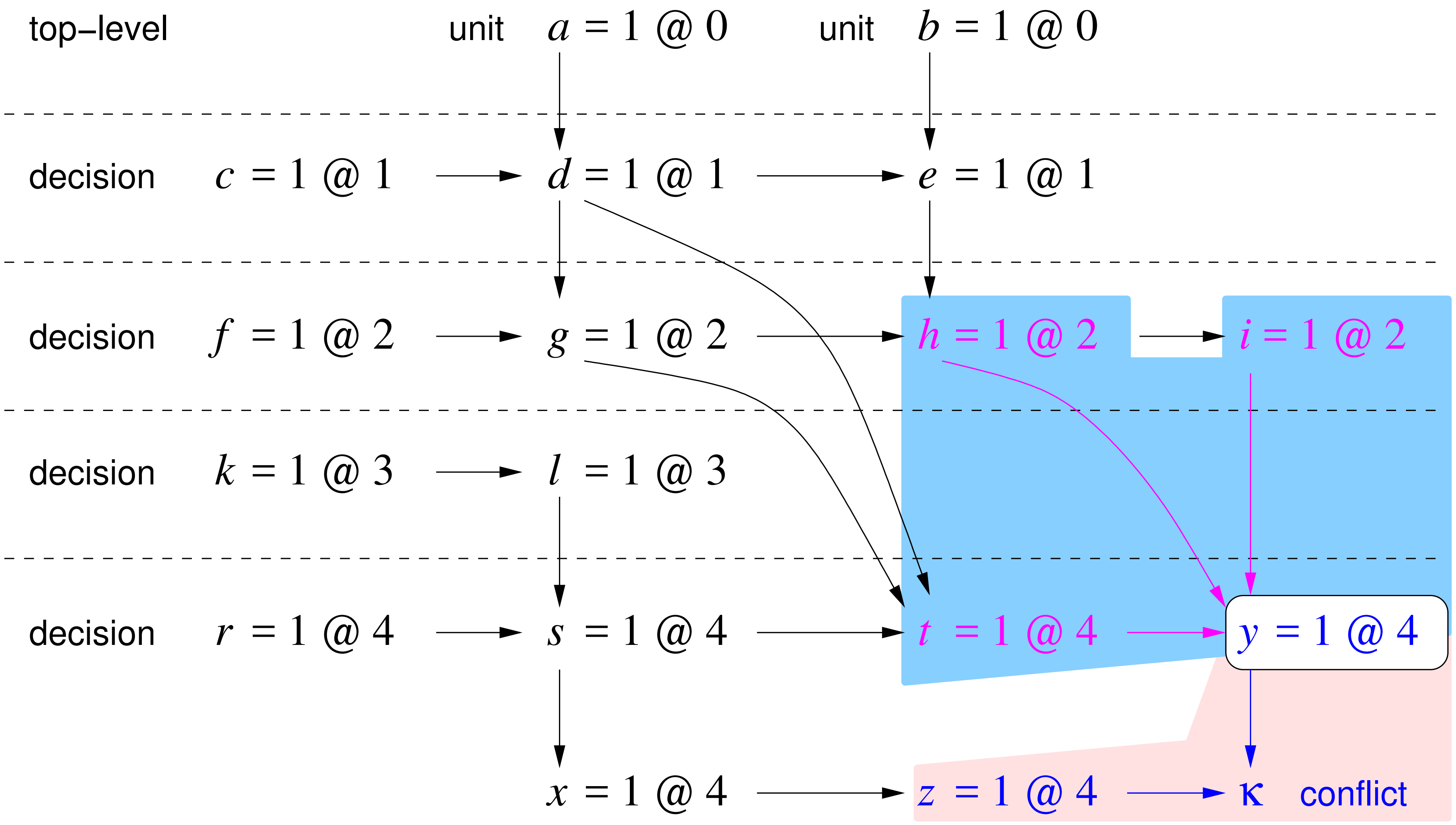

$$\neg(y \wedge z) \qquad \equiv \qquad (\bar{y} \vee \bar{z})$$

# Resolving Antecedents 1st Time



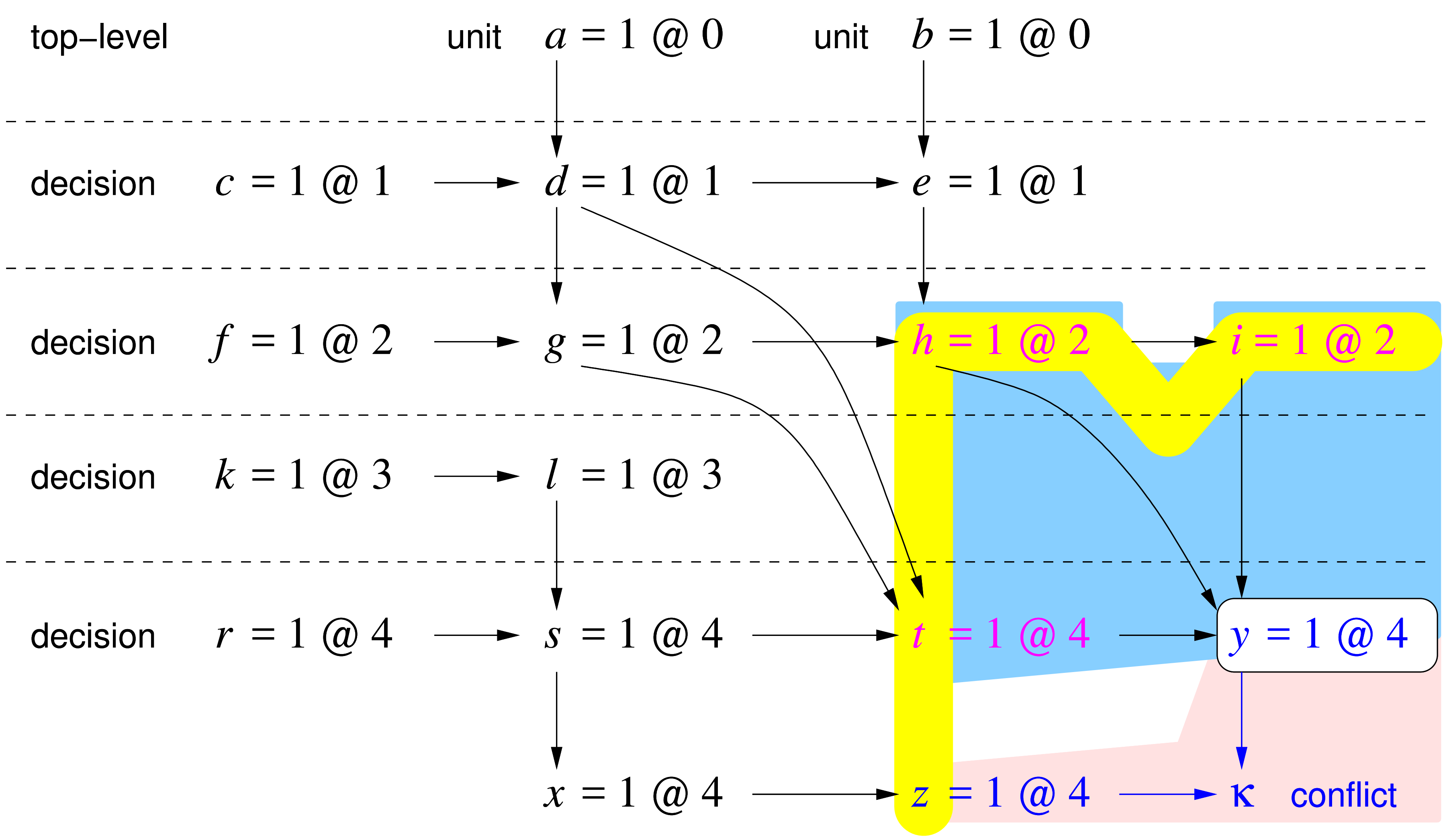

$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

# Resolving Antecedents 1st Time



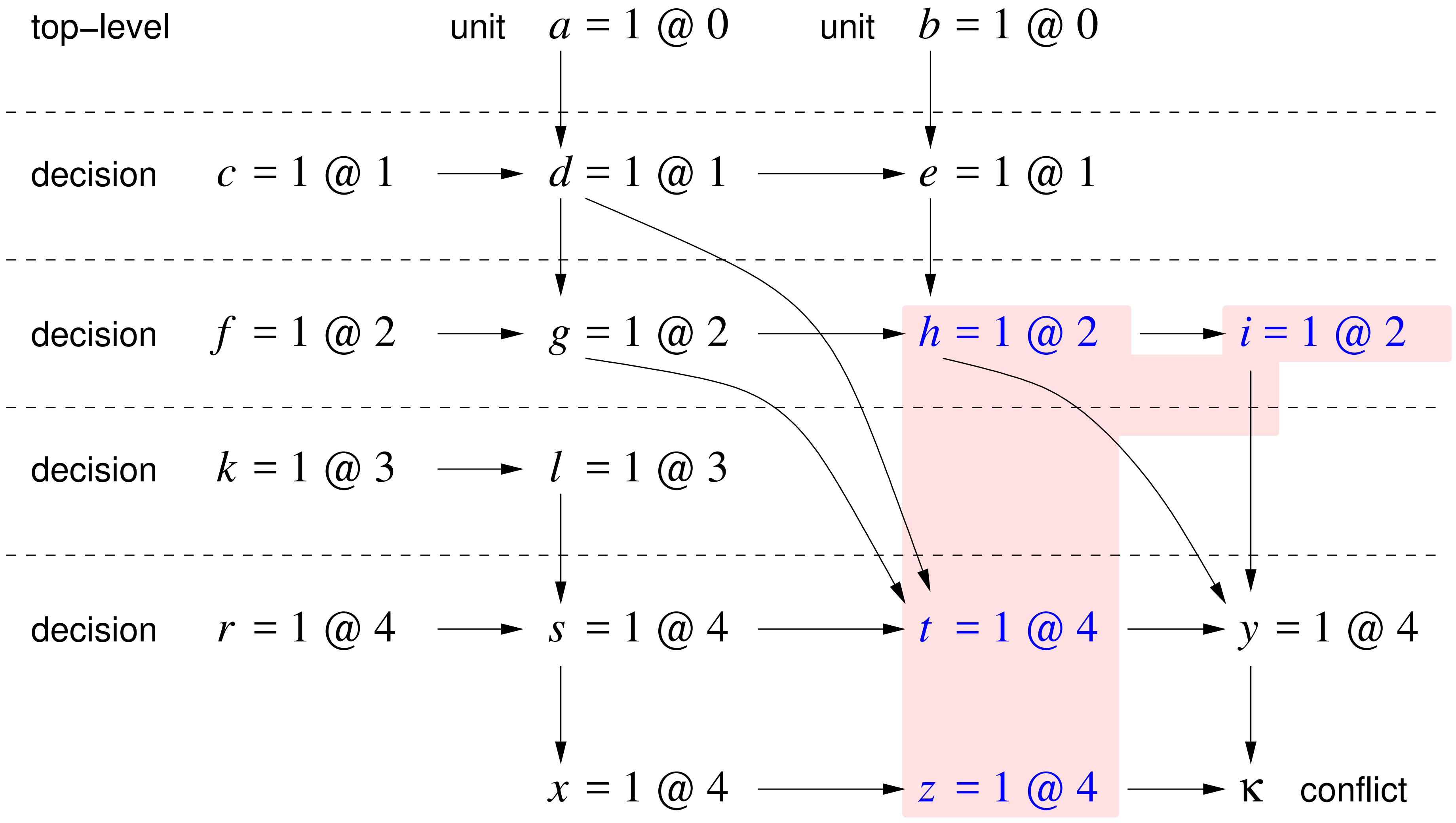

$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

# Resolvents = Cuts = Potential Learned Clauses



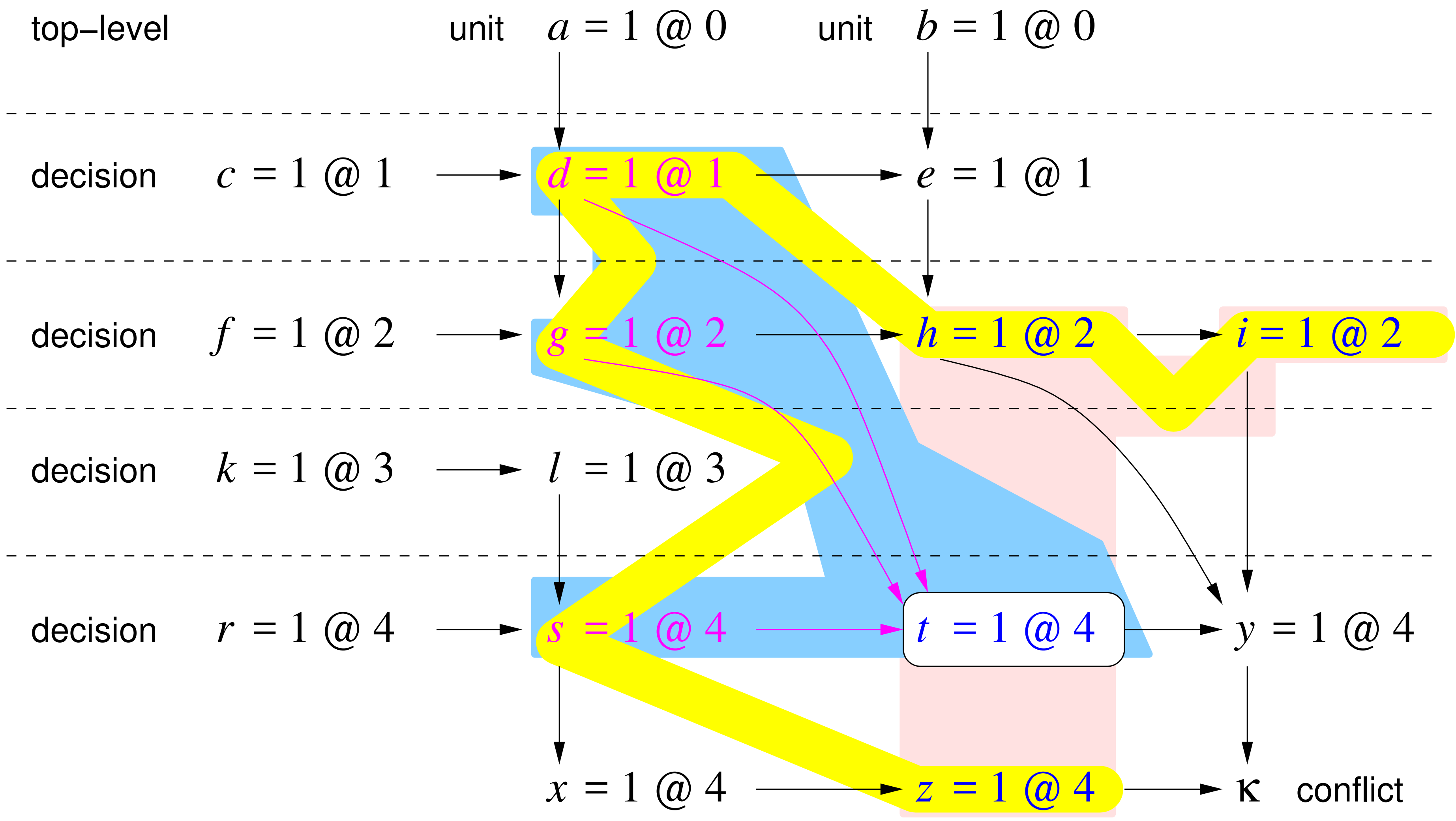

$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

# Potential Learned Clause After 1 Resolution



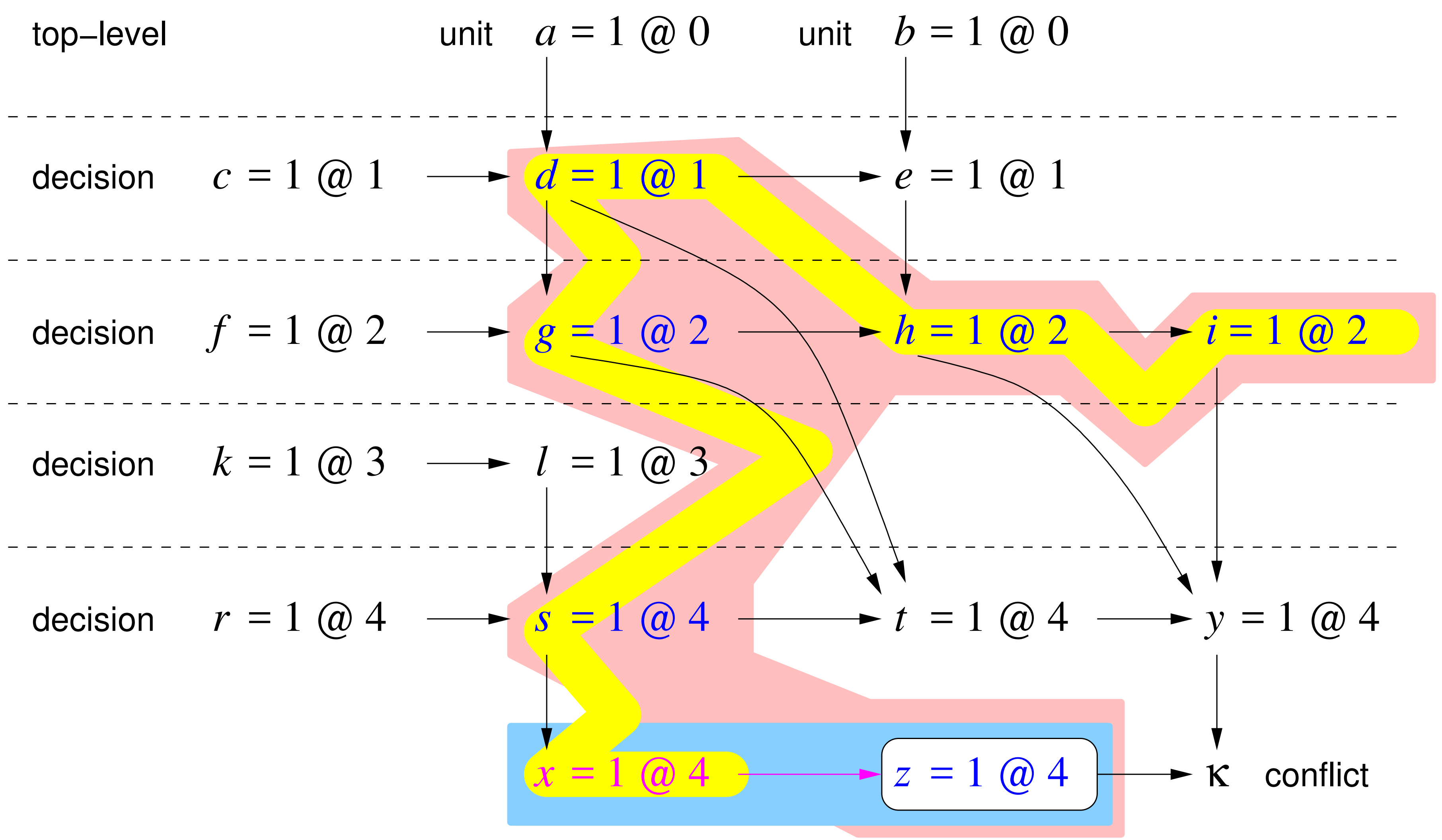

$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$

# Resolving Antecedents 2nd Time



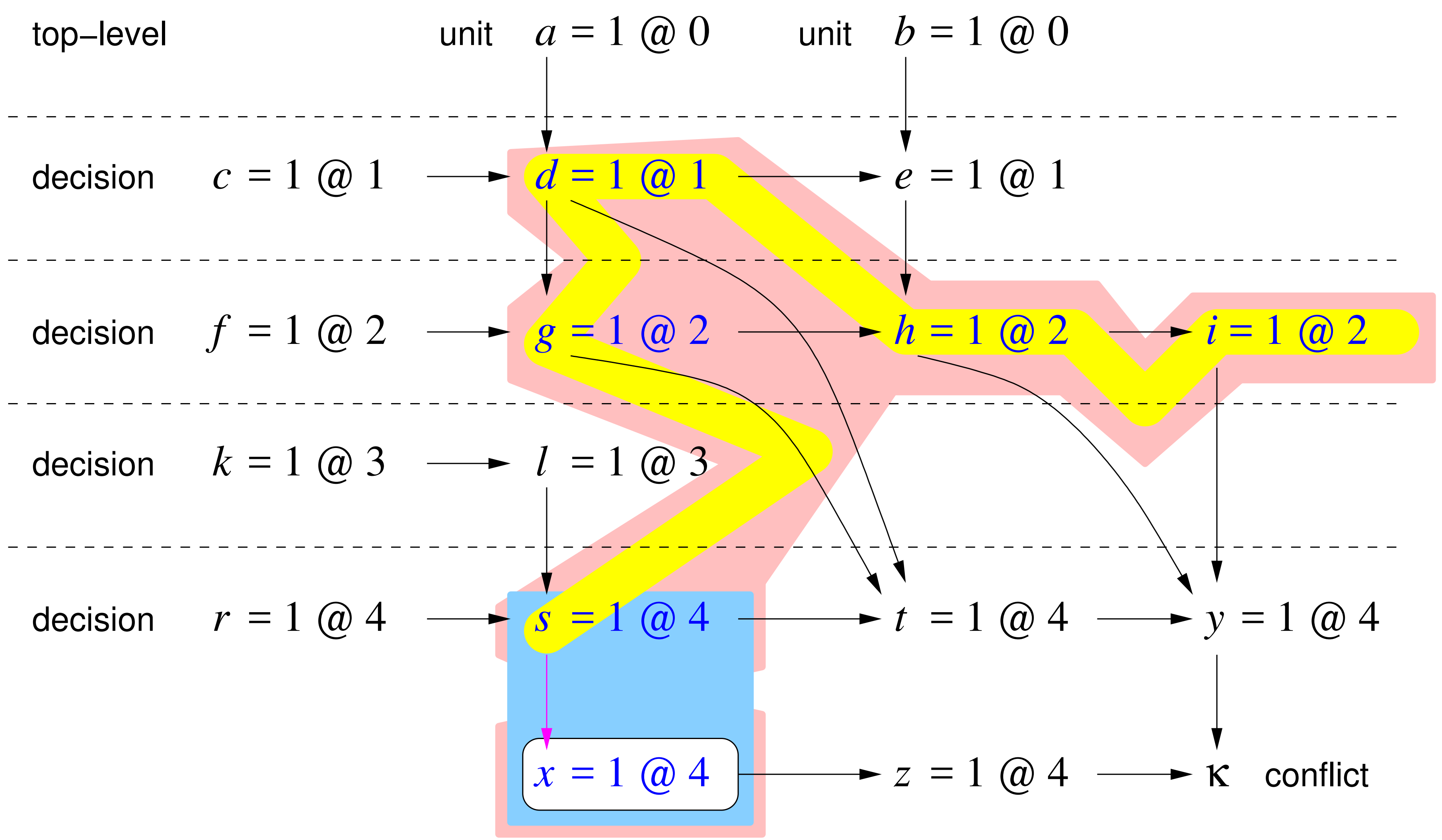

$$\frac{(\overline{d} \vee \overline{g} \vee \overline{s} \vee t) \qquad (\overline{h} \vee \overline{i} \vee \overline{t} \vee \overline{z})}{(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i} \vee \overline{z})}$$

# Resolving Antecedents 3rd Time




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

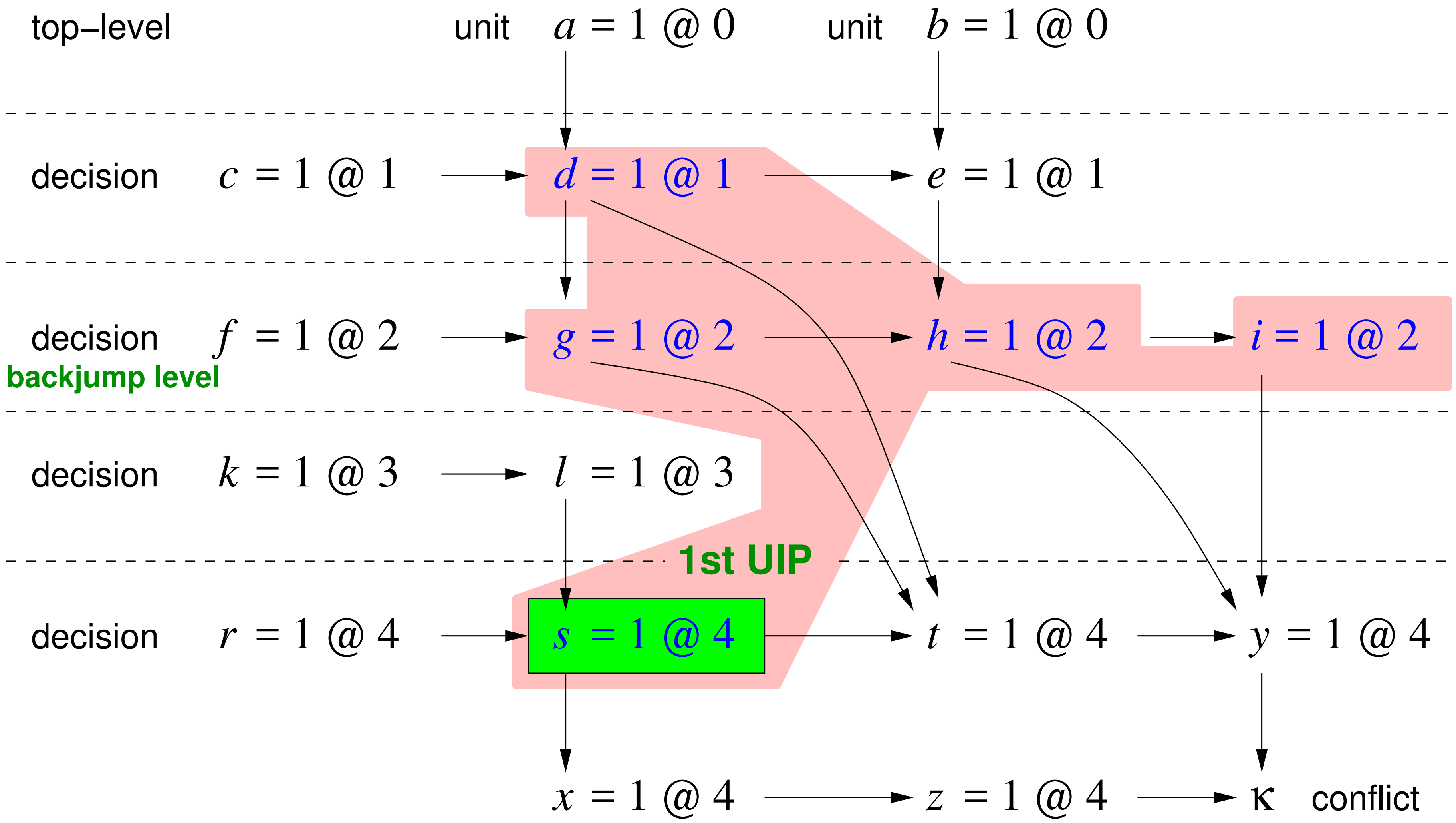
# Resolving Antecedents 4th Time




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

self subsuming resolution

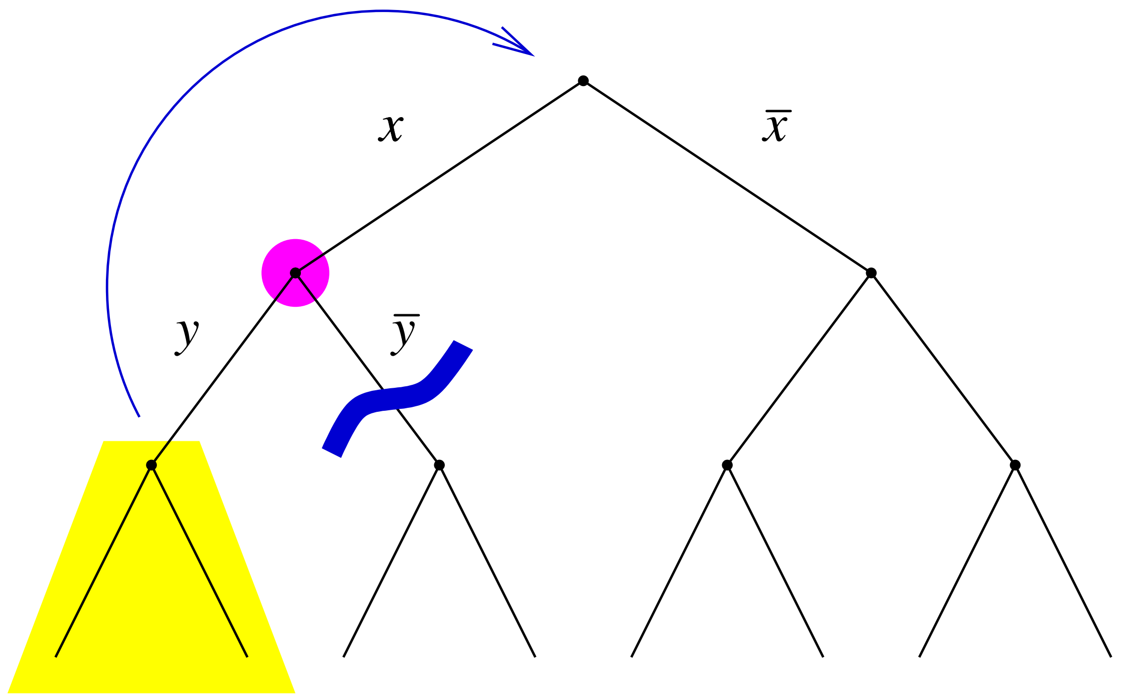
# 1st UIP Clause after 4 Resolutions




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

UIP = <u>unique implication point</u> dominates conflict on the last level

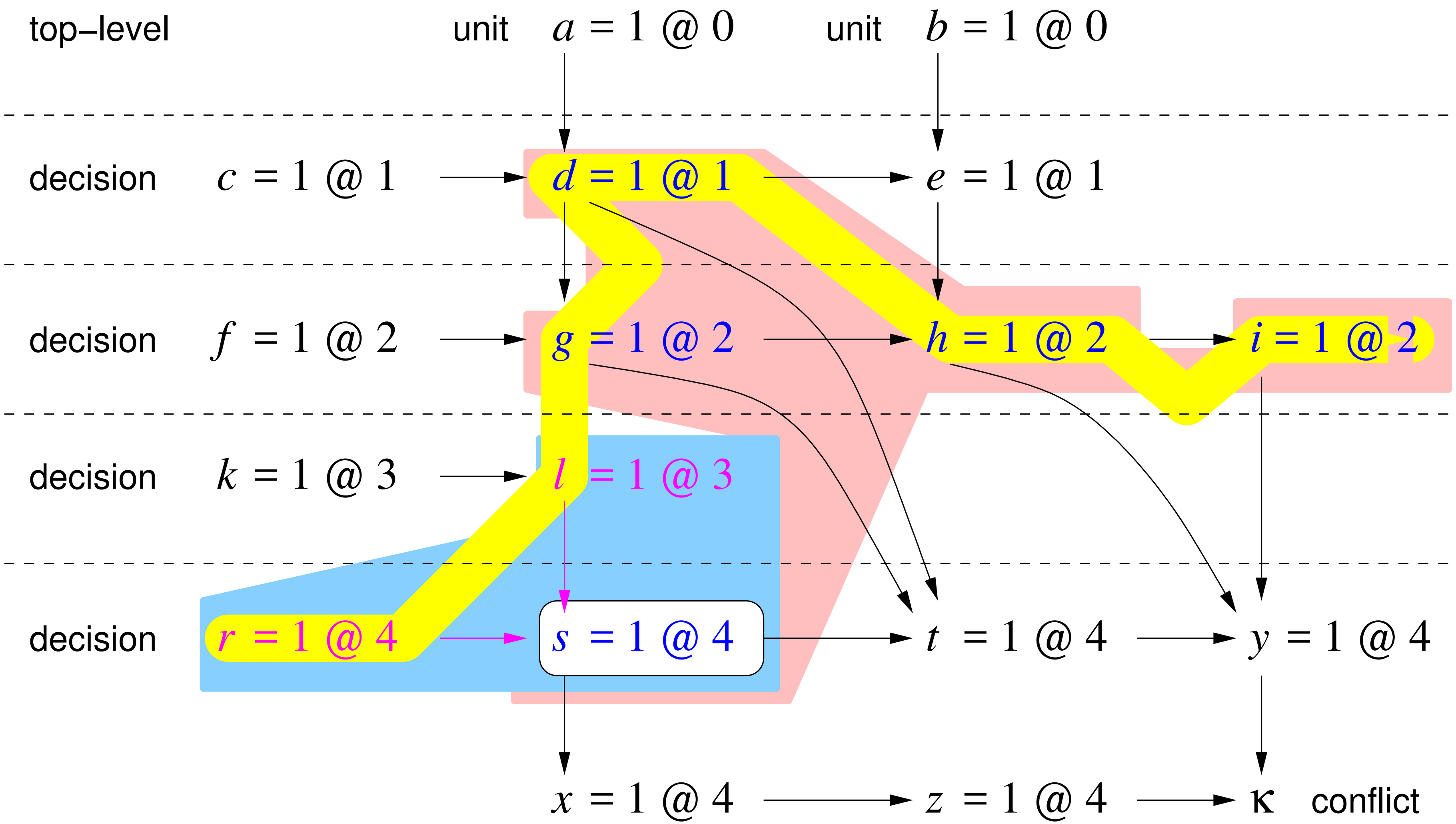
# Backjumping




If $y$ has never been used to derive a conflict, then skip $\overline{y}$ case.

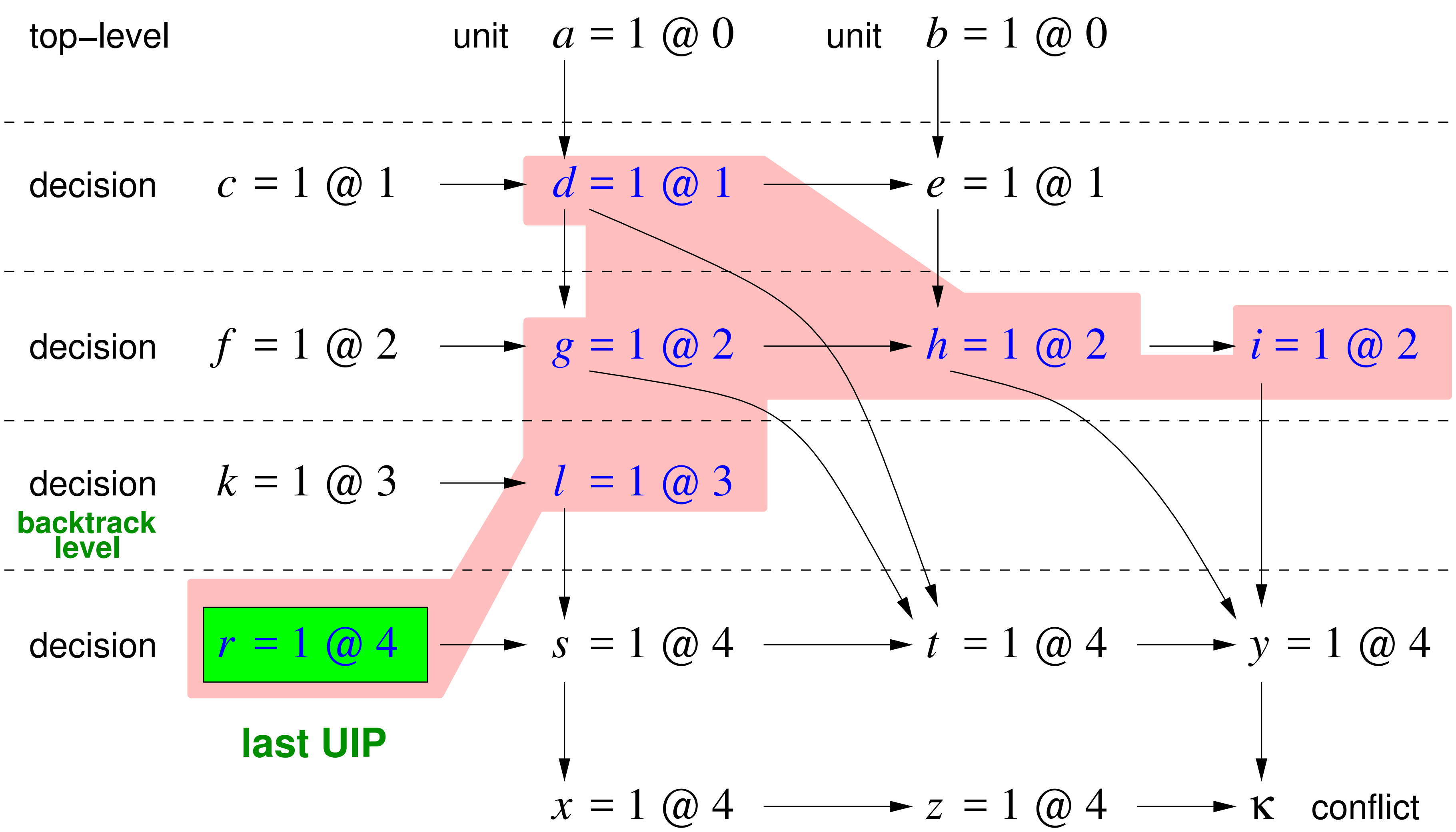
Immediately <u>jump back</u> to the $\overline{x}$ case – assuming $x$ was used.

# Resolving Antecedents 5th Time



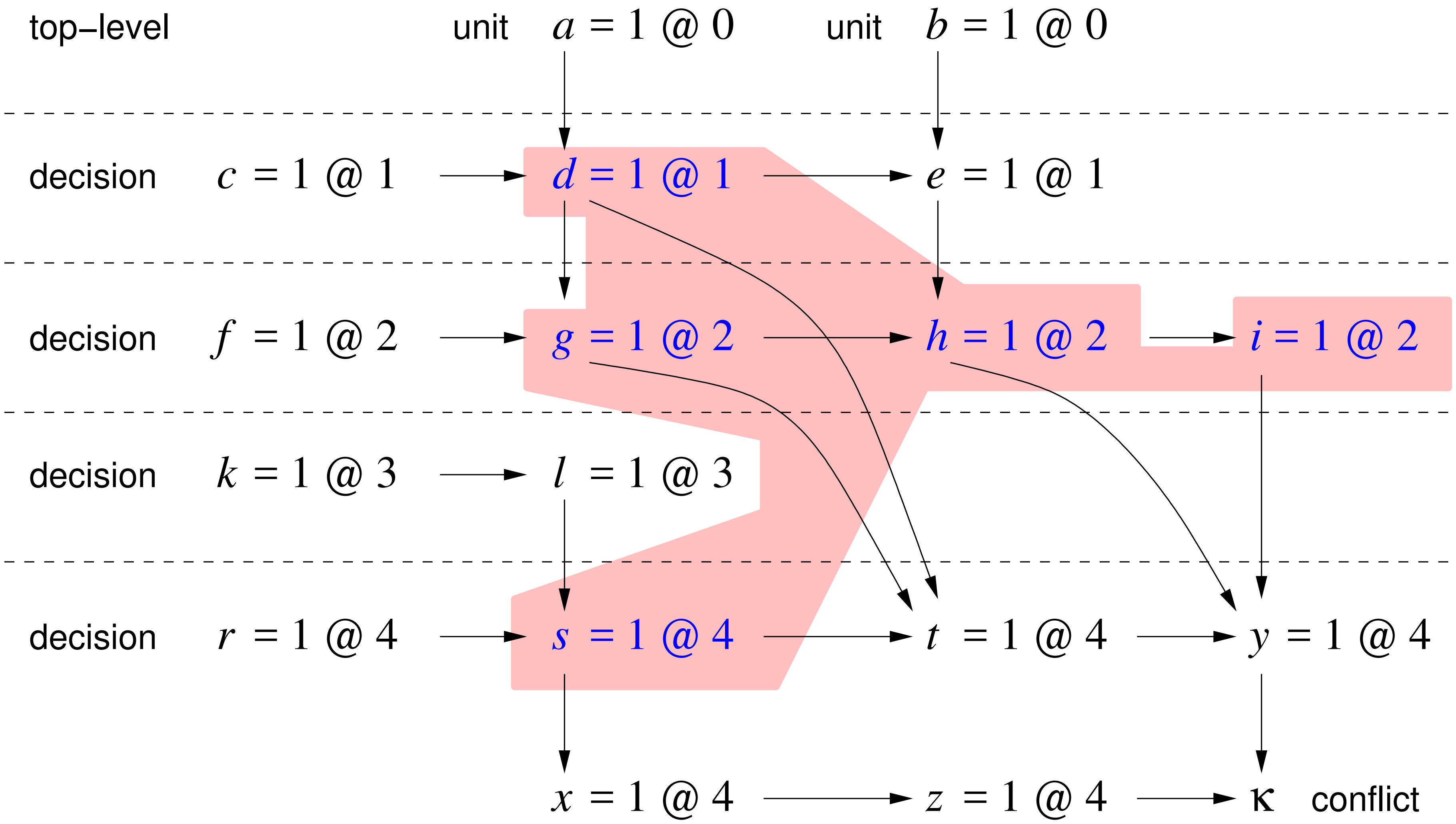

$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

# Decision Learned Clause



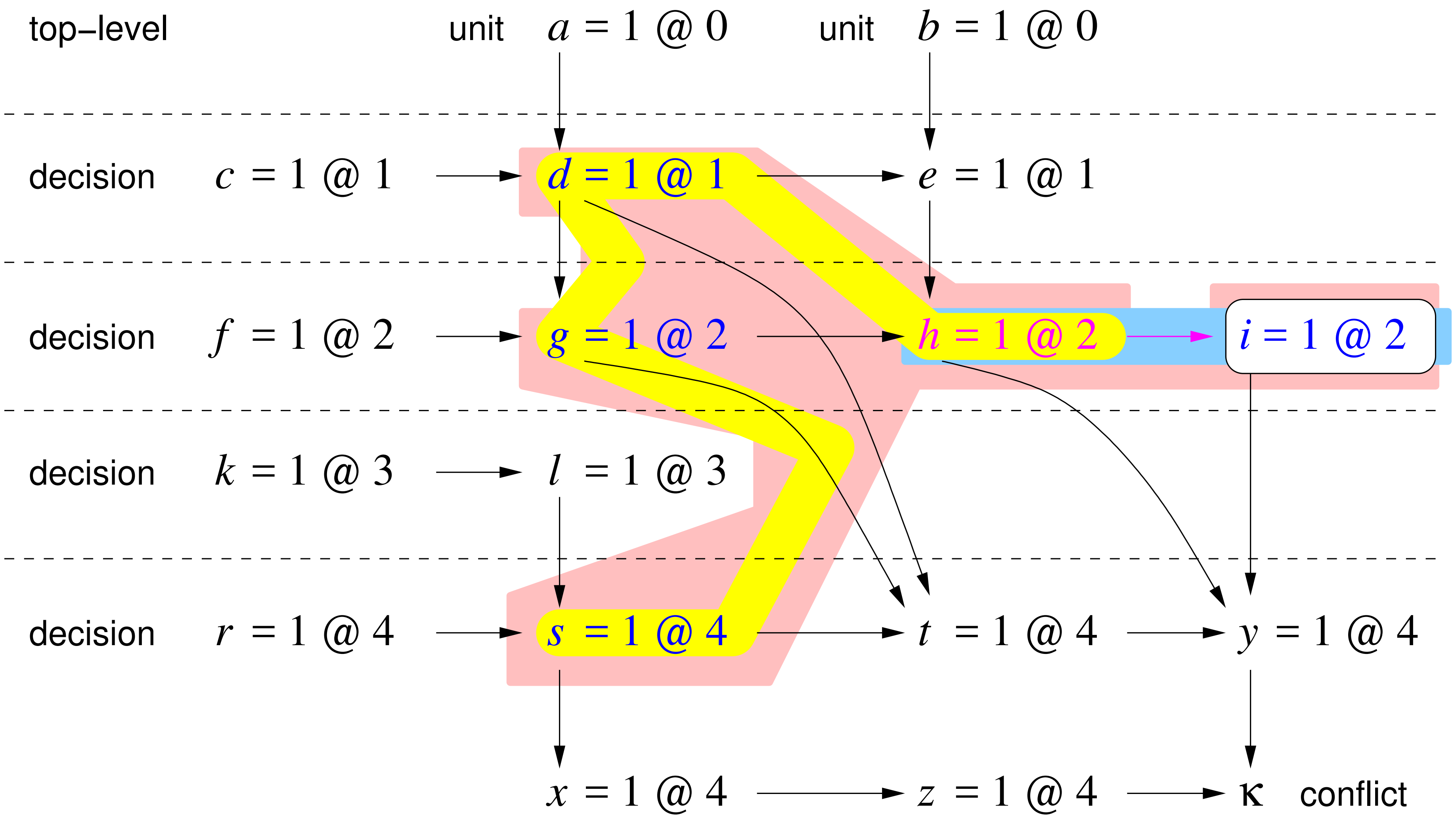

$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

# 1st UIP Clause after 4 Resolutions




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

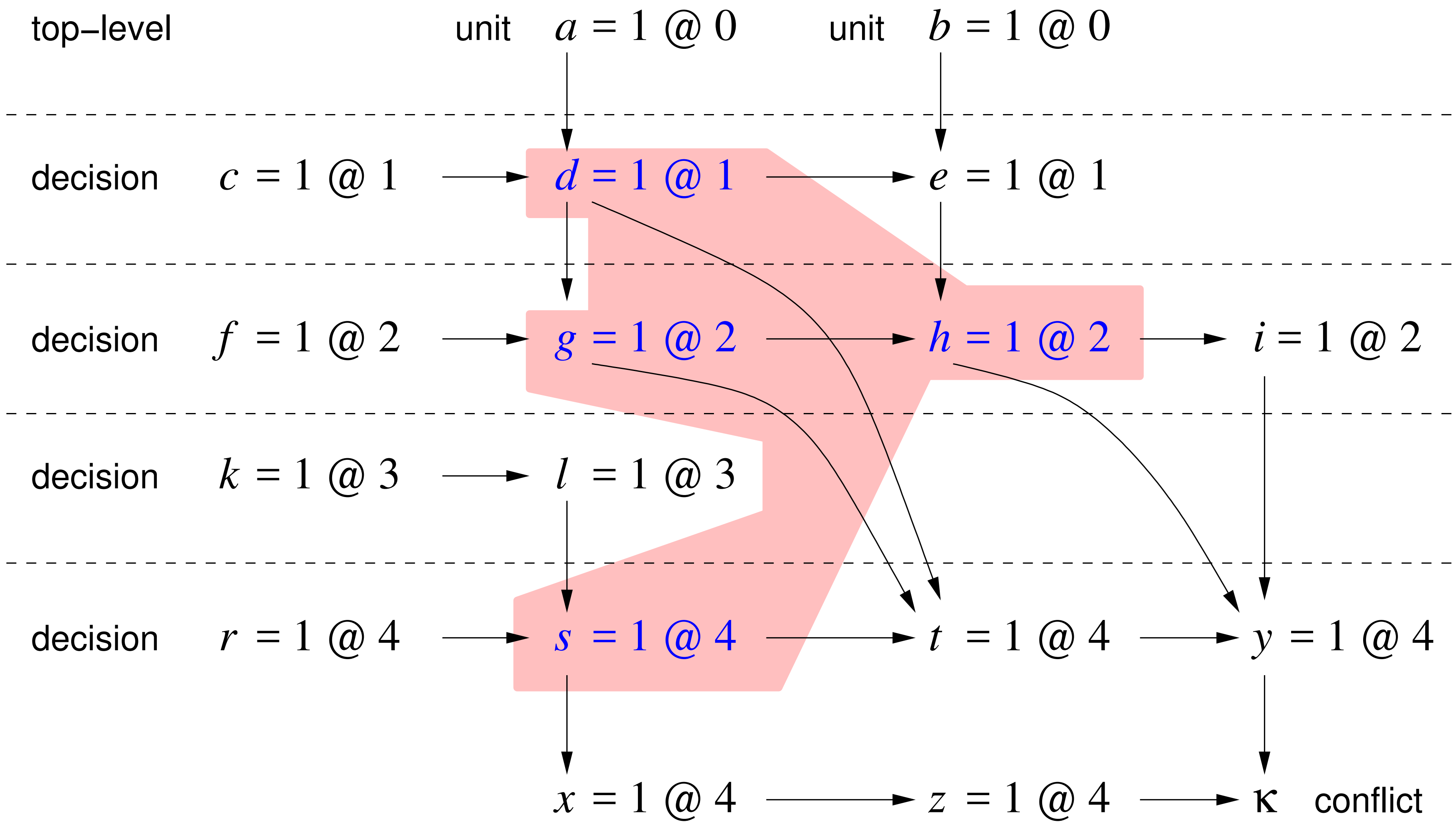
# Locally Minimizing 1st UIP Clause




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

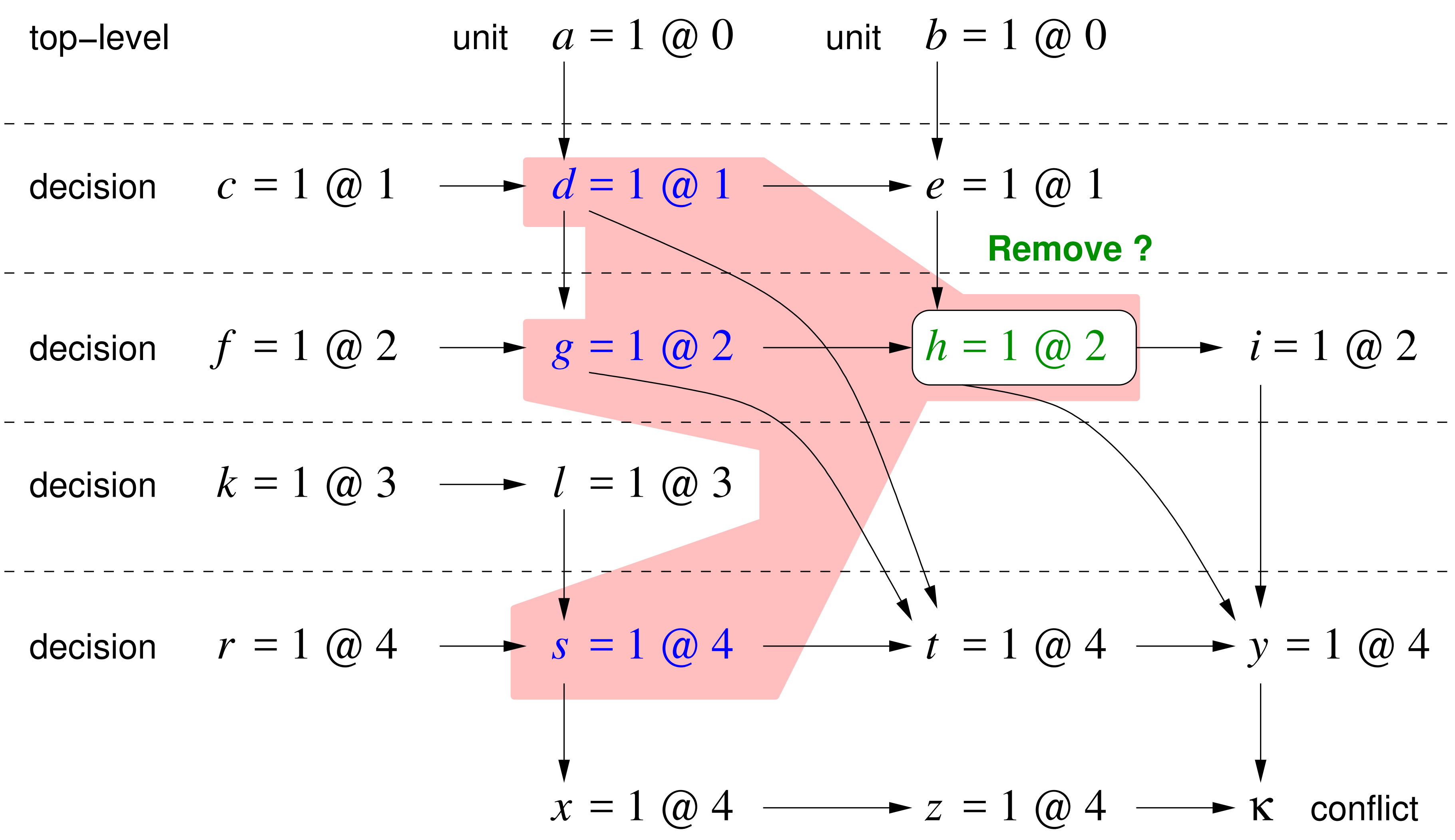
self subsuming resolution

# Locally Minimized Learned Clause



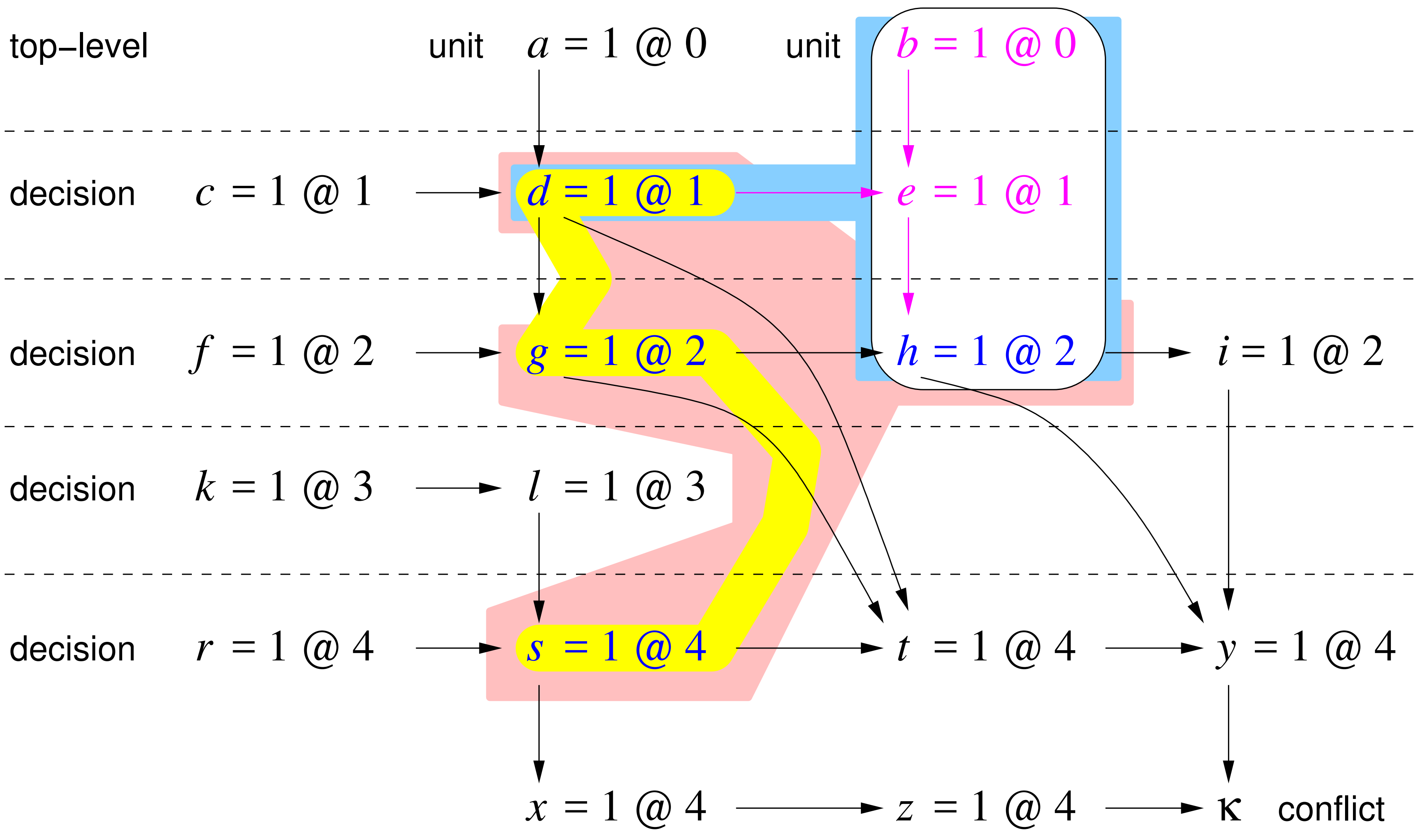

$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$

# Minimizing Locally Minimized Learned Clause Further?



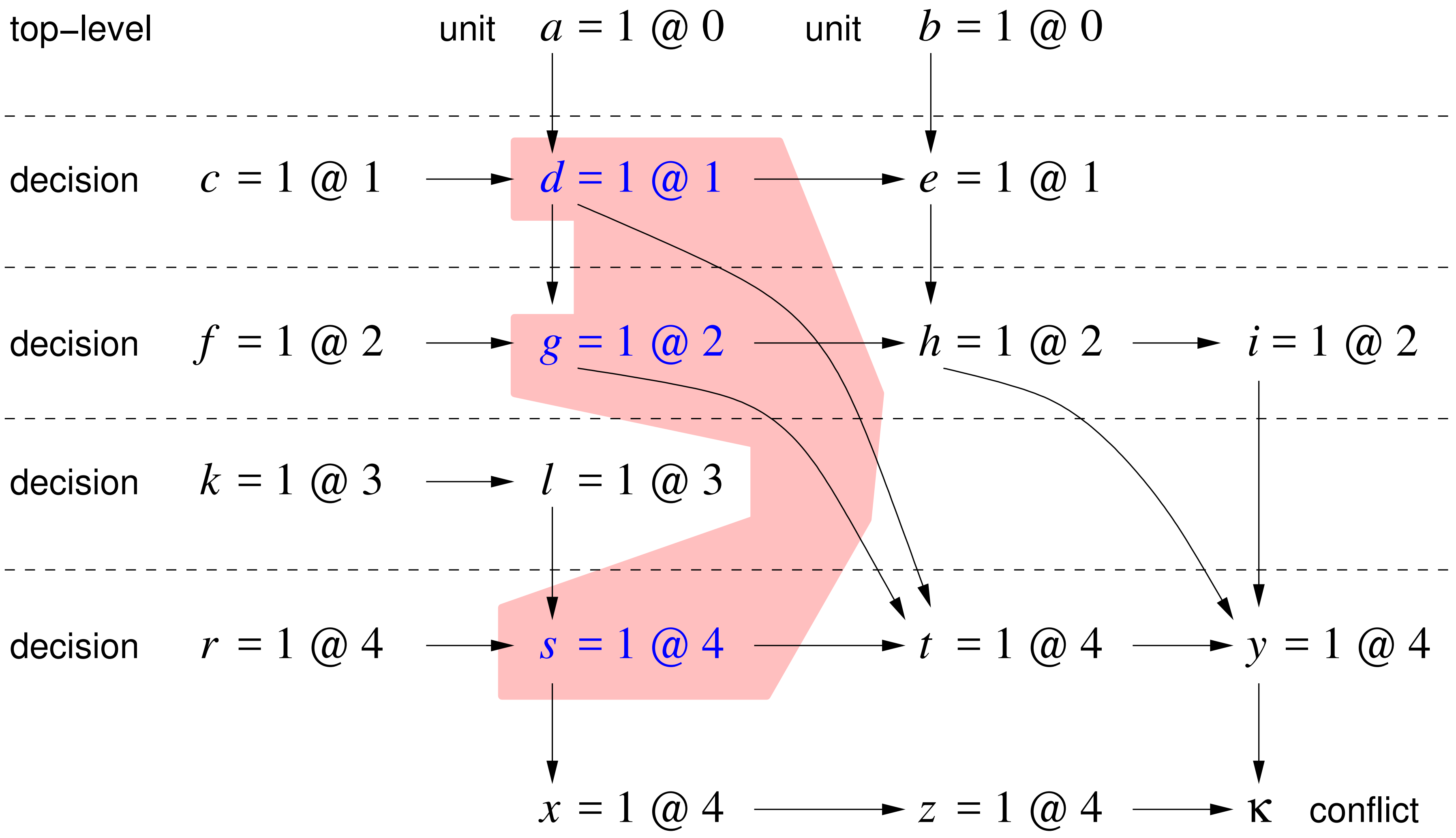

$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

# Recursively Minimizing Learned Clause




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

# Recursively Minimized Learned Clause




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

# Decision Heuristics

- number of variable occurrences in (remaining unsatisfied) clauses (LIS)
  - eagerly satisfy many clauses
  - many variations were studied in the 90ies
  - actually expensive to compute
- dynamic heuristics
  - **focus on variables which were usefull recently in deriving learned clauses**
  - can be interpreted as reinforcement learning
  - started with the VSIDS heuristic [MoskewiczMadiganZhaoZhangMalik'01]
  - most solvers rely on the exponential variant in MiniSAT (EVSIDS)
  - recently showed VMTF as effective as VSIDS [BiereFröhlich-SAT'15] survey
- look-ahead
  - spent more time in selecting good variables (and simplification)
  - related to our Cube & Conquer paper [HeuleKullmanWieringaBiere-HVC'11]
  - “The Science of Brute Force” [Heule & Kullman CACM August 2017]

# Variable Scoring Schemes

[BiereFröhlich-SAT'15]

$s$ old score $s'$ new score

| | variable score $s'$ after $i$ conflicts | | |
|---|---|---|---|
| | bumped | not-bumped | |
| STATIC | $s$ | $s$ | static decision order |
| INC | $s+1$ | $s$ | increment scores |
| SUM | $s+i$ | $s$ | sum of conflict-indices |
| VSIDS | $h_i^{256} \cdot s + 1$ | $h_i^{256} \cdot s$ | original implementation in Chaff |
| NVSIDS | $f \cdot s + (1-f)$ | $f \cdot s$ | normalized variant of VSIDS |
| EVSIDS | $s + g^i$ | $s$ | exponential MiniSAT dual of NVSIDS |
| ACIDS | $(s+i)/2$ | $s$ | average conflict-index decision scheme |
| VMTF | $i$ | $s$ | variable move-to-front |
| VMTF' | $b$ | $s$ | variable move-to-front variant |

$0 < f < 1$ $g = 1/f$ $h_i^m = 0.5$ if $m$ divides $i$ $h_i^m = 1$ otherwise

$i$ conflict index $b$ bumped counter

# Basic CDCL Loop

```
int basic_cdcl_loop () {
  int res = 0;

  while (!res)
         if (unsat) res = 20;
    else if (!propagate ()) analyze ();    // analyze propagated conflict
    else if (satisfied ()) res = 10;       // all variables satisfied
    else decide ();                        // otherwise pick next decision

  return res;
}
```

# Reducing Learned Clauses

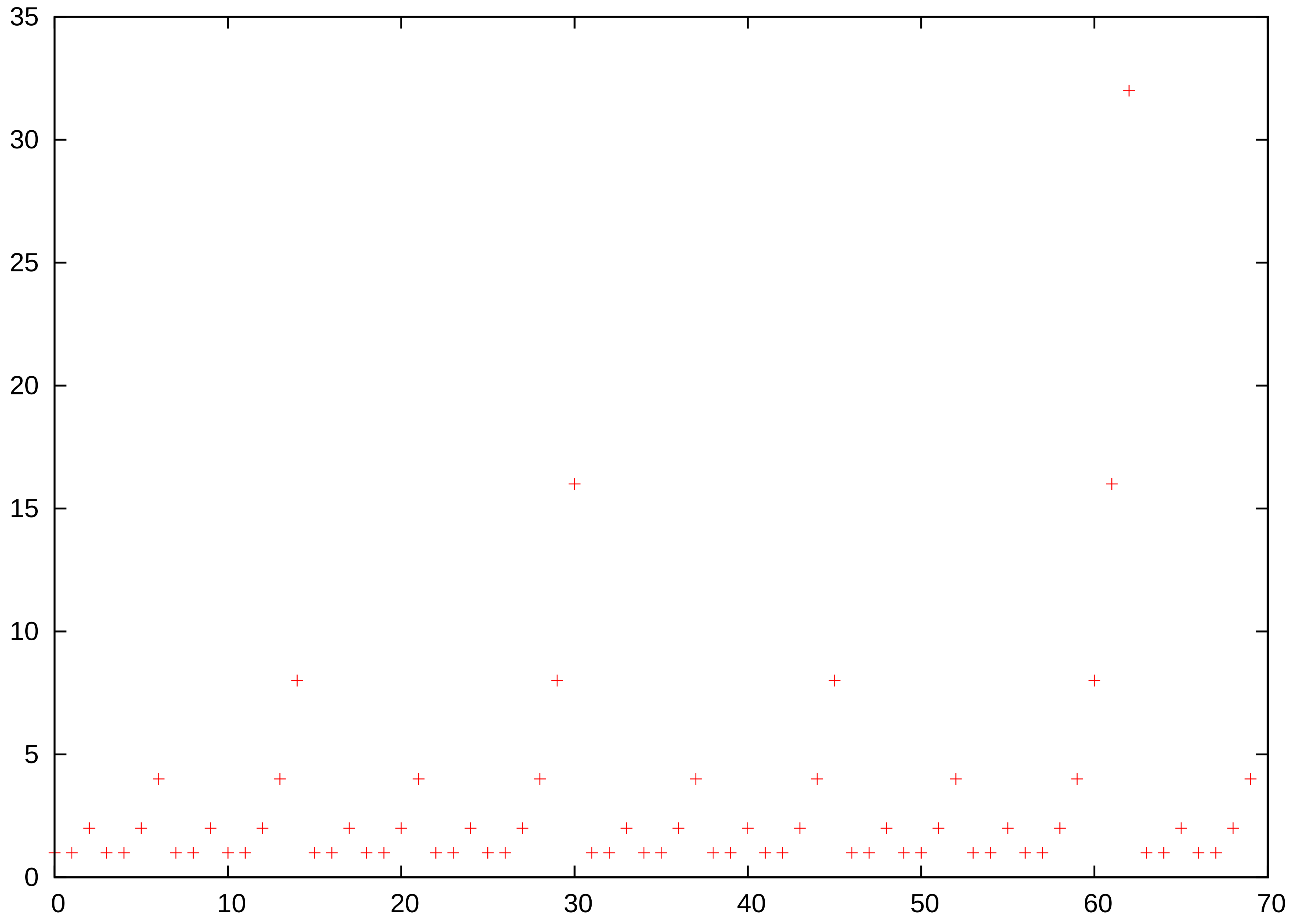
- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only "short" clauses
- better periodically delete "useless" learned clauses
  - keep a certain number of learned clauses — "search cache"
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - then maximum number kept learned clauses is increased $\boxed{\text{geometrically}}$
- LBD (glucose level / glue) prediction for usefulness — [AudemardSimon-IJCAI'09]
  - LBD = number of decision-levels in the learned clause
  - allows $\boxed{\text{arithmetic}}$ increase of number of kept learned clauses
  - keep clauses with small LBD forever ( $\leq 2 \ldots 5$)
  - large fixed cache usesful for hard satisfiable instances (crypto) — [Chanseok Oh]

# Restarts

- often it is a good strategy to abandon what you do and restart
  - for satisfiable instances the solver may get stuck in the unsatisfiable part
  - for unsatisfiable instances focusing on one part might miss short proofs
  - restart after the number of conflicts 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 during restart
- for completeness dynamically increase restart limit
  - arithmetically, geometrically, Luby, Inner/Outer
- Glucose restarts [AudemardSimon-CP'12]
  - short vs. large window exponential moving average (EMA) over LBD
  - if recent LBD values are larger than long time average then restart

# 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

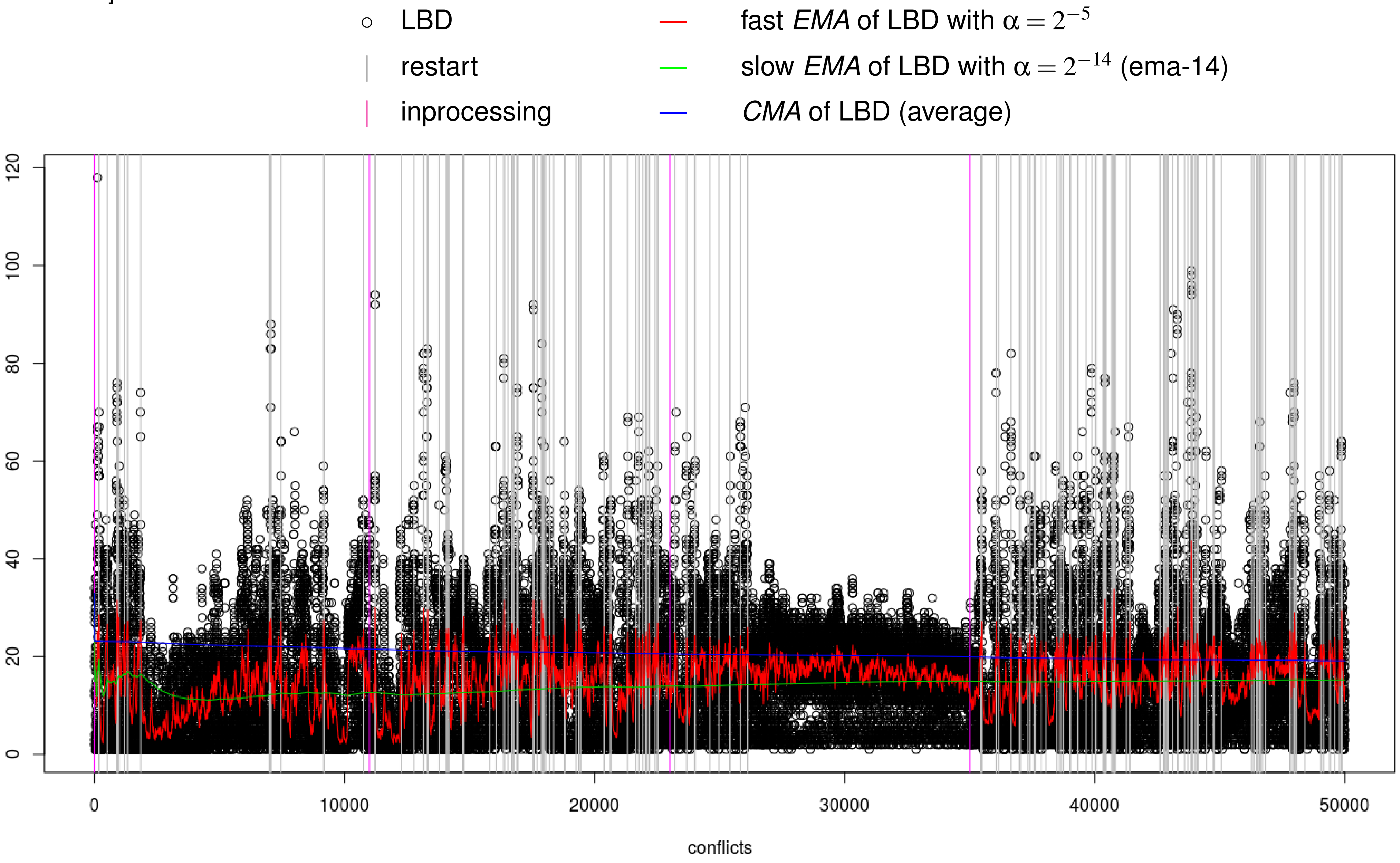
[Knuth'12]

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

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

# Restart Scheduling with Exponential Moving Averages

[BiereFröhlich-POS'15]



# Phase Saving and Rapid Restarts

- phase assignment:
  - assign decision variable to 0 or 1?
  - only thing that matters in satisfiable instances
- "phase saving" as in RSat [PipatsrisawatDarwiche'07]
  - 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**
  - and thus makes CDCL actually a rather "local" search procedure
- rapid restarts
  - varying restart interval with bursts of restarts
  - not only theoretically avoids local minima
  - works nicely together with phase saving
- reusing the trail can reduce the cost of restarts [RamosVanDerTakHeule-JSAT'11]

# CDCL Loop with Reduce and Restart

```
int basic_cdcl_loop_with_reduce_and_restart () {
  int res = 0;
  while (!res)
         if (unsat) res = 20;
    else if (!propagate ()) analyze ();    // analyze propagated conflict
    else if (satisfied ()) res = 10;       // all variables satisfied
    else if (restarting ()) restart ();    // restart by backtracking
    else if (reducing ()) reduce ();       // collect useless learned clauses
    else decide ();                        // otherwise pick next decision
  return res;
}
```

# Code from our SAT Solver CaDiCaL

```
int Internal::search () {
  int res = 0;
  START (search);
  while (!res)
         if (unsat) res = 20;
    else if (!propagate ()) analyze ();    // analyze propagated conflict
    else if (iterating) iterate ();        // report learned unit
    else if (satisfied ()) res = 10;       // all variables satisfied
    else if (terminating ()) break;        // limit hit or asynchronous abort
    else if (restarting ()) restart ();    // restart by backtracking
    else if (reducing ()) reduce ();       // collect useless learned clauses
    else if (probing ()) probe ();         // failed literal probing
    else if (subsuming ()) subsume ();     // subsumption algorithm
    else if (eliminating ()) elim ();      // bounded variable elimination
    else if (compactifying ()) compact (); // collect internal variables
    else decide ();                        // otherwise pick next decision
  STOP (search);
  return res;
}
```

https://github.com/arminbiere/cadical

https://fmv.jku.at/cadical

# Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - 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, tail 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
- blocking literals [ChuHarwoodStuckey'09]
- special treatment of short clauses (binary [PilarskiHu'02] or ternary [Ryan'04])
- cache start of search for replacement [Gent-JAIR'13]

# Proofs / RUP / DRUP

- original idea for proofs: proof traces / sequence consisting of “learned clauses”
- can be checked clause by clause through unit propagation
- reverse unit implied clauses (RUP) [GoldbergNovikov’03] [VanGelder’12]
- deletion information (DRUP): proof trace of added and deleted clauses
- RUP in SAT competition 2007, 2009, 2011, DRUP since 2013 to certify UNSAT

# Blocked Clauses

[Kullman-DAM’99] [JärvisaloHeuleBiere-JAR’12]

- clause $\overbrace{(a \vee l)}^{C}$ “blocked” on $l$ w.r.t. CNF $\overbrace{(\bar{a} \vee b) \wedge (l \vee c) \wedge \underbrace{(\bar{l} \vee \bar{a})}_{D}}^{F}$
  - all resolvents of $C$ on $l$ with clauses $D$ in $F$ are tautological
- blocked clauses are “redundant” too
  - adding or removing blocked clauses does not change satisfiability status
  - however it might change the set of models

# Resolution Asymmetric Tautologies (RAT)

"Inprocessing Rules" [JärvisaloHeuleBiere-IJCAR'12]

- justify complex preprocessing algorithms in Lingeling
  - examples are adding blocked clauses or variable elimination
  - interleaved with research (forgetting learned clauses = reduce)
- need more general notion of redundancy criteria
  - simply replace "resolvents are tautological" by "resolvents on $l$ are RUP"

$$(a \vee l) \qquad \text{RAT on } l \quad \text{w.r.t.} \qquad (\bar{a} \vee b) \wedge (l \vee c) \wedge \underbrace{(\bar{l} \vee b)}_{D}$$

  - deletion information is again essential (DRAT)
  - now mandatory in the main track of the last two SAT competitions
  - pretty powerful: can for instance also cover symmetry breaking

# Propagation Redundant (PR)

"Short Proofs Without New Variables" [HeuleKieslBiere-CADE'17] best paper

- more general than RAT: short proofs for pigeon hole formulas without new variables
- $C$ propagation redundant if $\exists$ (partial) assignment $\omega$ satisfying $C$ with $F|\overline{C} \vdash_1 F|\omega$
- Satisfaction Driven Clause Learning (SDCL) [HeuleKieslSeidlBiere-HVC'17]
  - first automatically generated PR proofs
  - prune paths for which we have other at least as satisfiable paths
- translate PR to DRAT [HeuleBiere-TACAS'18]
  - only one additional variable needed
  - shortest proofs for pigeon hole formulas
  - in general quadratic

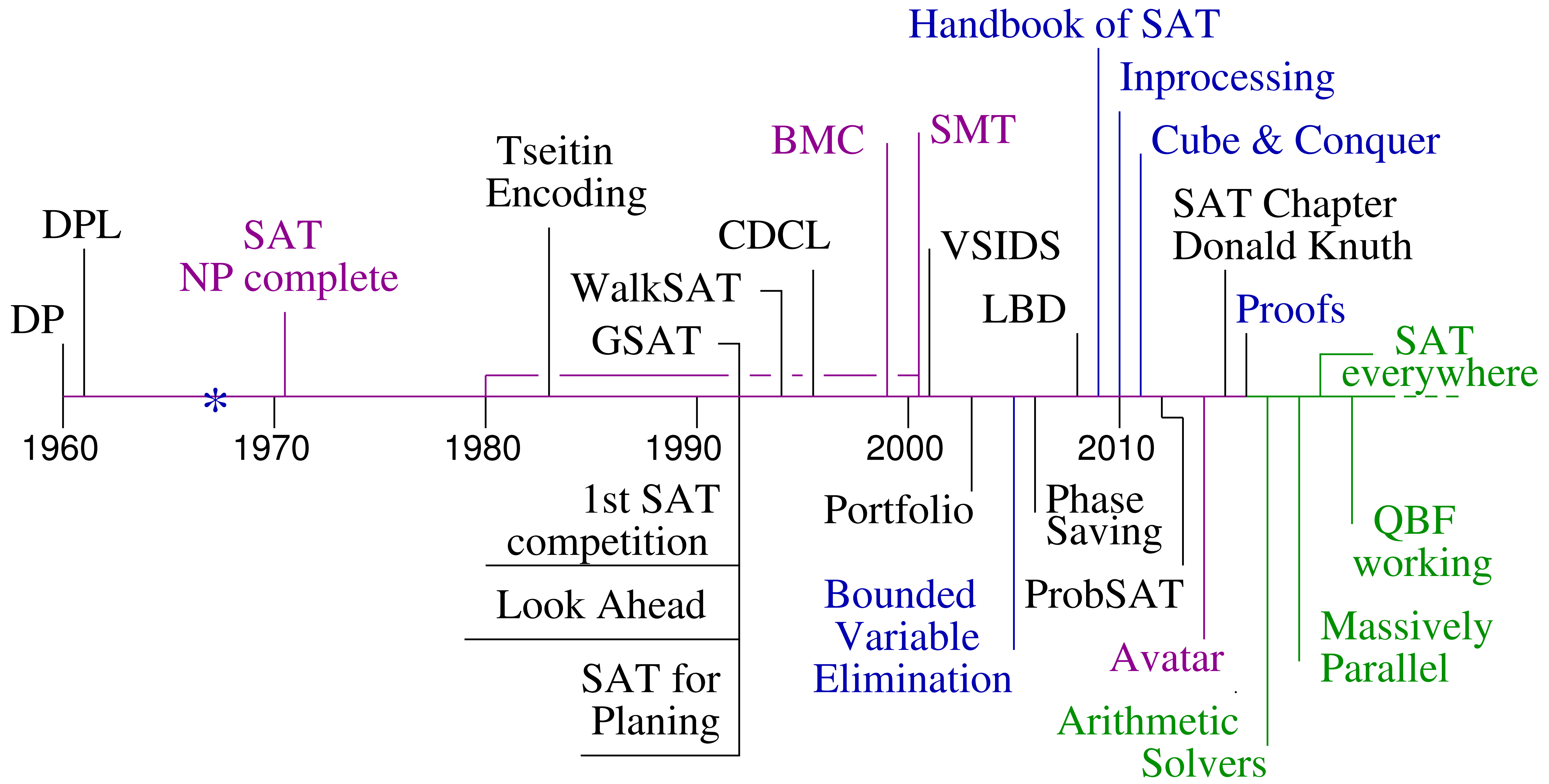
# Parallel SAT

- application level parallelism
- guiding path principle
- portfolio (with sharing)
- (concurrent) cube & conquer

$\Rightarrow$ Handbook of Parallel Constraint Reasoning

$\Rightarrow$ still many low-level programming issues left

Personal SAT Solver History



# SAT/SMT/AR Summer School 2018

International Summer School on Satisfiability, Satisfiability Modulo Theories, and Automated Reasoning



## Home

Satisfiability (SAT), Satisfiability Modulo Theories (SMT), and Automated Reasoning (AR) continue to make rapid advances and find novel uses in a wide variety of applications, both in computer science and beyond. The SAT/SMT/AR Summer School aims to bring a select group of students up to speed quickly in this exciting research area. The school continues the successful line of Summer Schools that ran from 2011 to 2015 as SAT/SMT Summer Schools and added AR in 2016.

The summer school will be taking place in the School of Computer Science at the University of Manchester. The school will take place on 3-6 July 2018 (preceding FLoC).