Conflict-Driven Clause Learning

Armin Biere

JOHANNES KEPLER
UNIVERSITÄT LINZ

ATVA'2019 Tutorial

Academia Sinica, Taipei, Taiwan

October 28, 2019

Dress Code Tutorial Speaker as SAT Problem

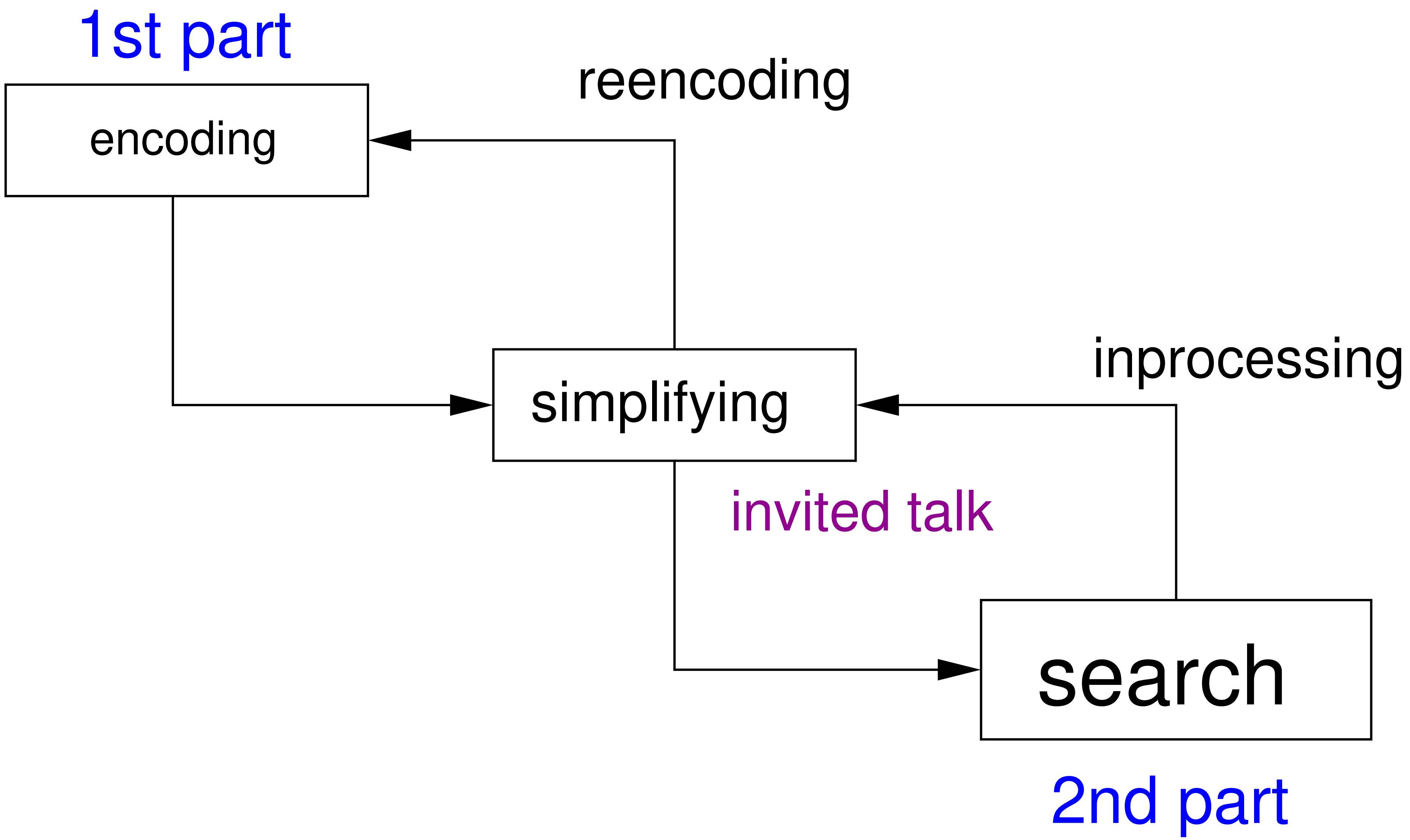
- propositional logic:
  - variables **tie** **shirt**
  - negation $\neg$ (not)
  - disjunction $\vee$ (or)
  - conjunction $\wedge$ (and)
- clauses (conditions / constraints)

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?**

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

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

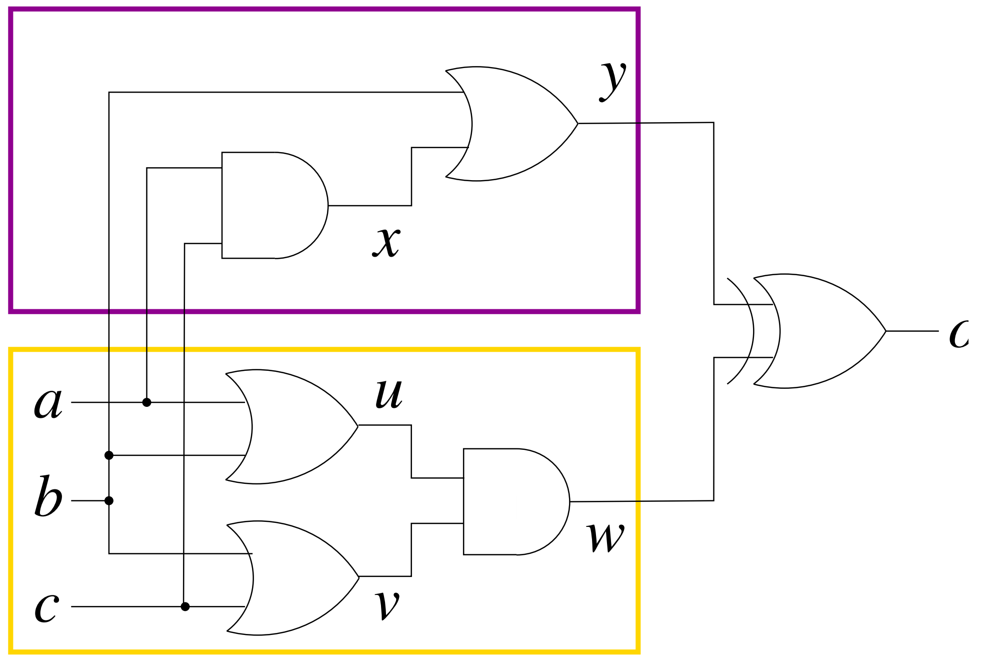
$$\begin{aligned} \textit{original} &\equiv \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\ &\equiv (\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge \textbf{ if } \neg a \textbf{ then } g \textbf{ else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

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

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

satisfying assignment gives counter-example to equivalence

Tseitin Transformation: Circuit to CNF



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

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

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

Tseitin Transformation: Gate Constraints

Negation:

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

Disjunction:

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

Conjunction:

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

Equivalence:

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

Bit-Blasting of Bit-Vector Addition

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

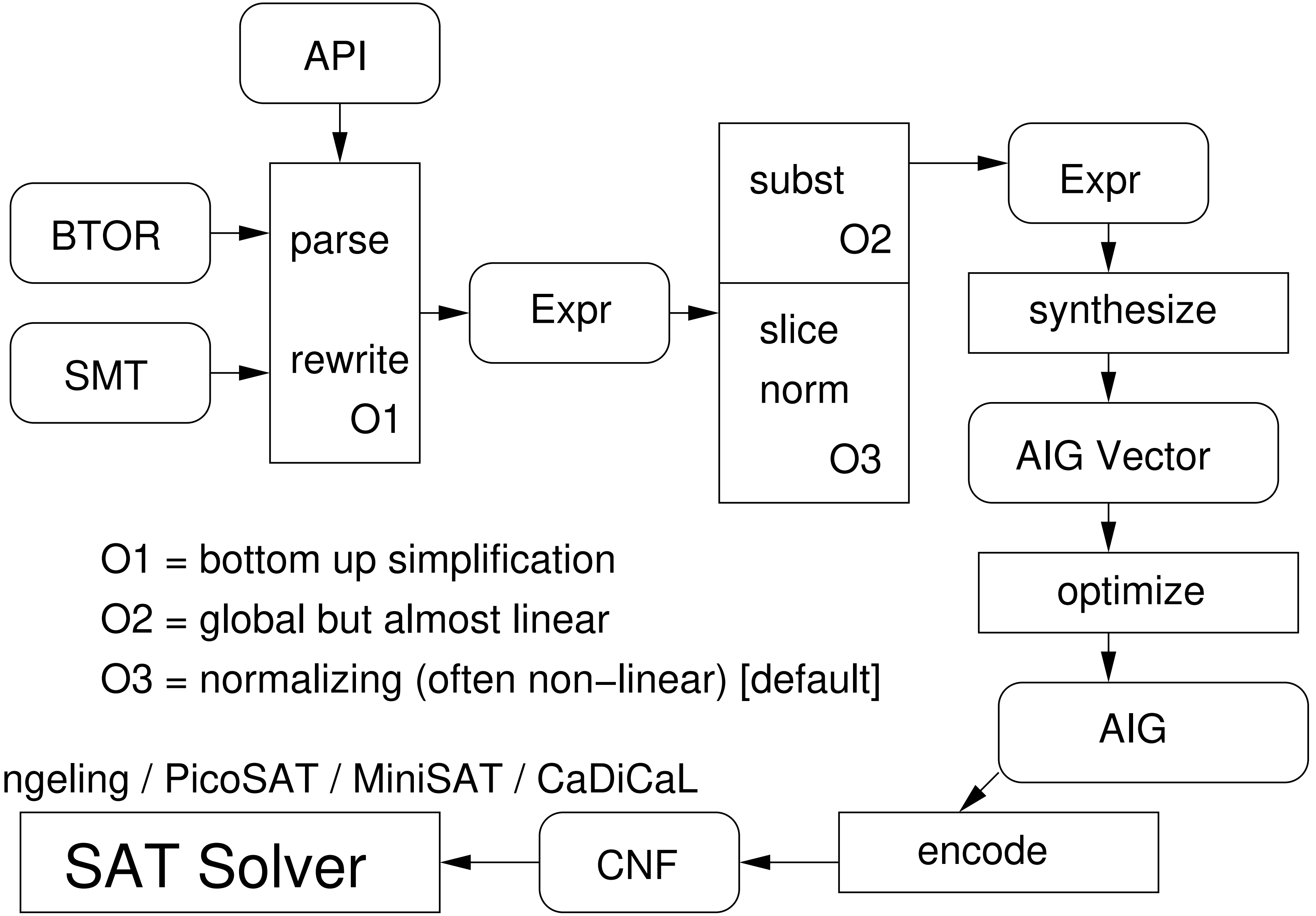
$$[s_3, s_2, s_1, s_0]_4 \;=\; [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

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

where

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

Boolector Architecture



O1 = bottom up simplification

O2 = global but almost linear

O3 = normalizing (often non–linear) [default]

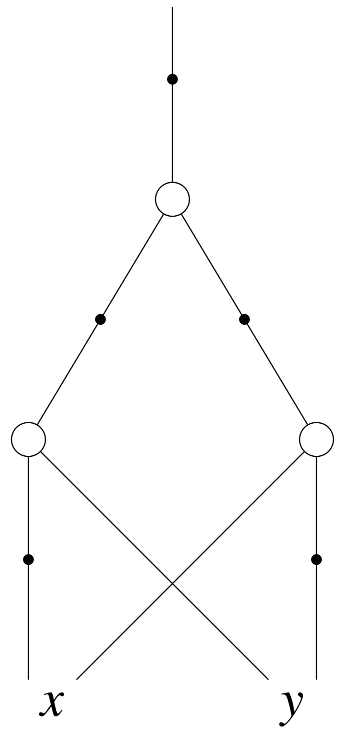
Intermediate Representations

- 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

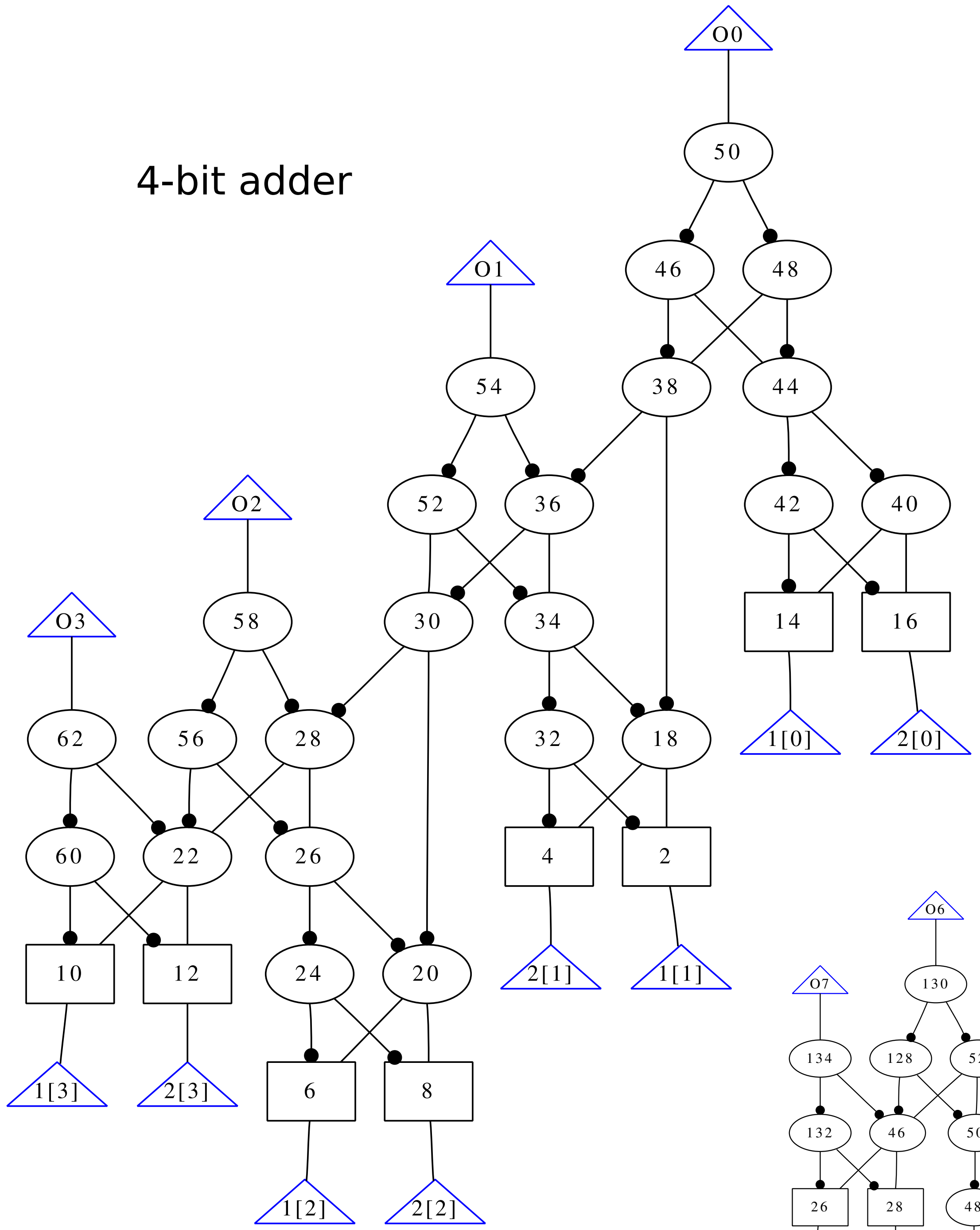


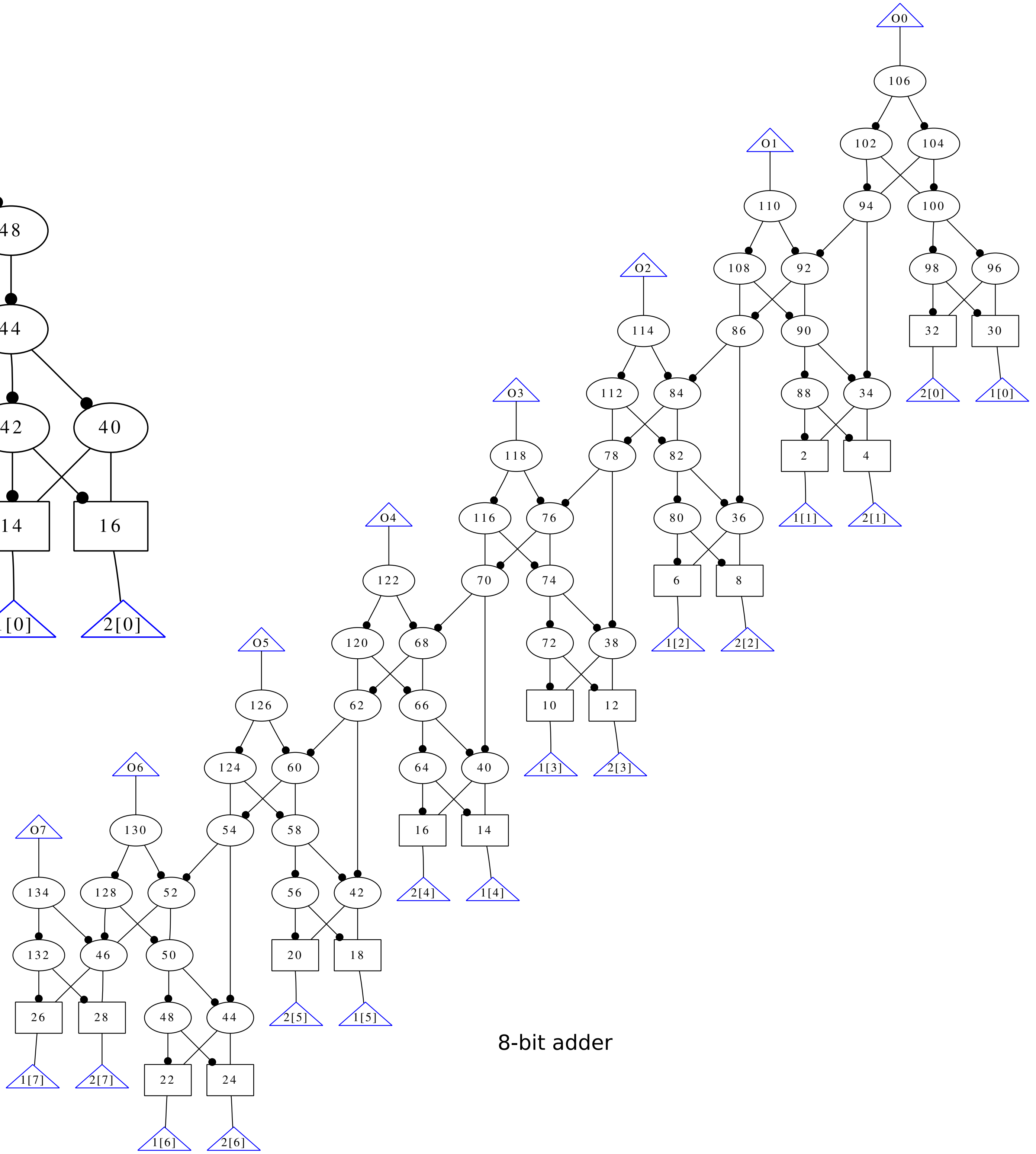
negation/sign are edge attributes

not part of node

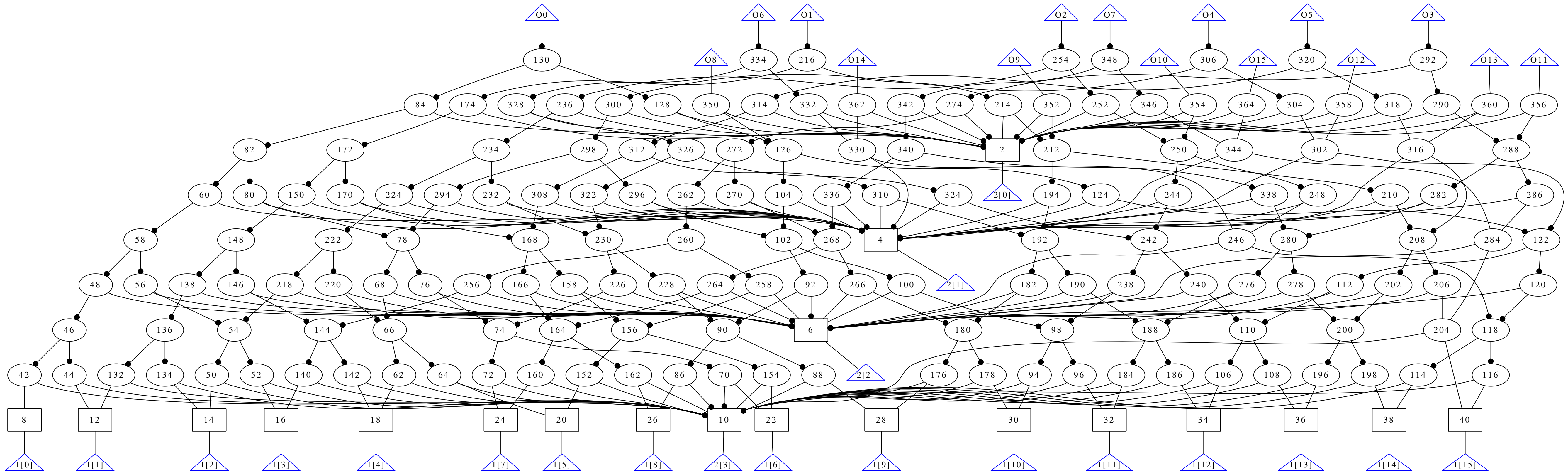
$$x \text{ xor } y \equiv (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \equiv \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

4-bit adder

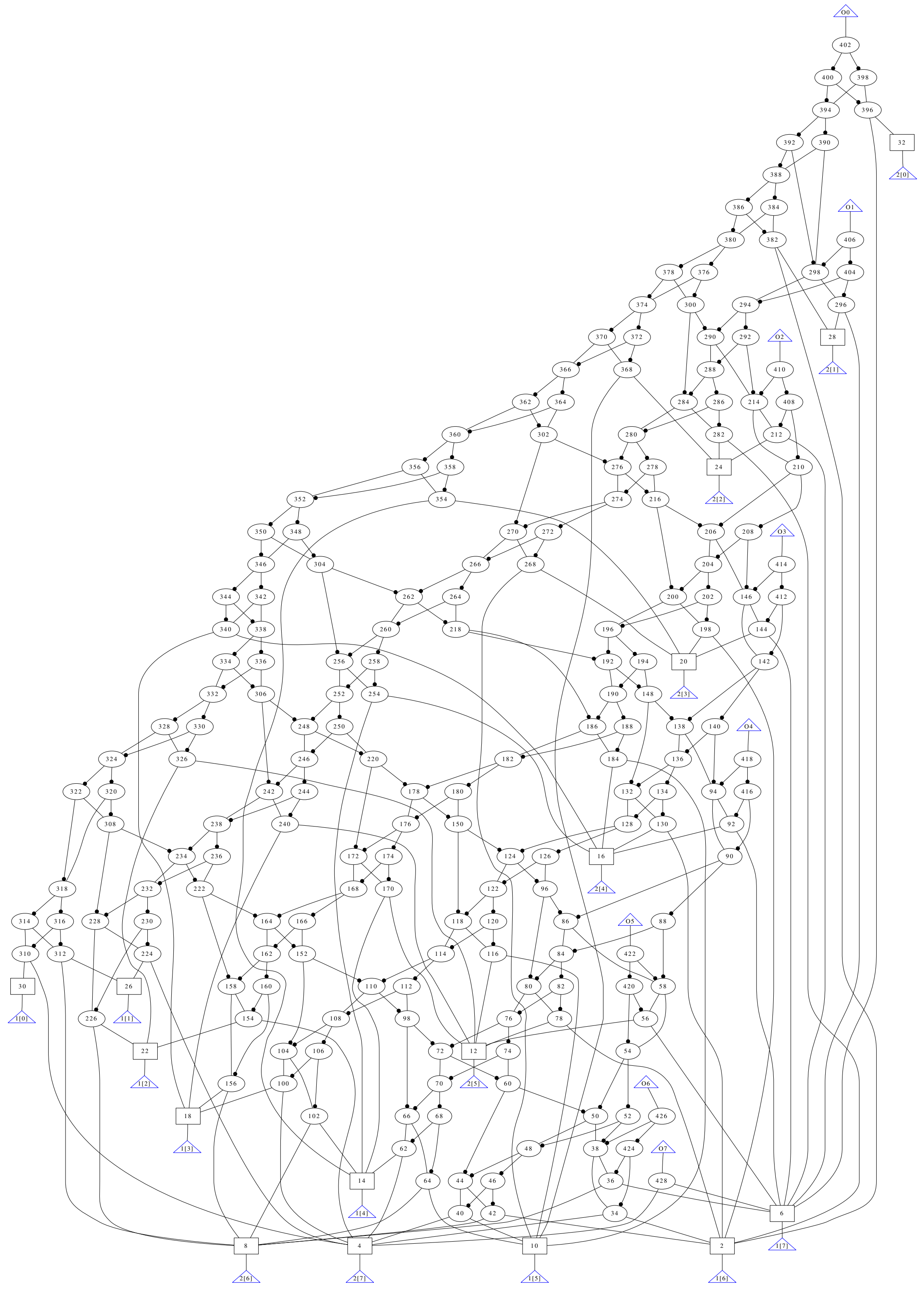




8-bit adder



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



Encoding Logical Constraints

- Tseitin construction suitable for most kinds of “model constraints”
  - assuming simple operational semantics: encode an interpreter
  - small domains: <u>one-hot encoding</u> large domains: <u>binary encoding</u>

- harder to encode <u>properties</u> or additional <u>constraints</u>
  - temporal logic / fix-points
  - environment constraints

- example for fix-points / recursive equations: $x = (a \vee y), \quad y = (b \vee x)$
  - has unique <u>least</u> fix-point $x = y = (a \vee b)$
  - and unique <u>largest</u> fix-point $x = y = true$ 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*
  - $l_1 + \dots + l_n \geq k$ or $l_1 + \dots + l_n \leq k$ or $l_1 + \dots + l_n = k$
  - 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

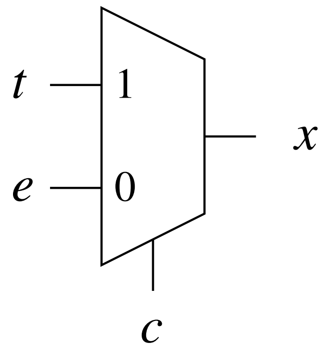
BDD-Based Encoding of Cardinality Constraints

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

$$\begin{tikzcd}
l_1 \arrow[r, dashed, dash] \arrow[d, dash] & l_2 \arrow[r, dashed, dash] \arrow[d, dash] & l_3 \arrow[r, dashed, dash] \arrow[d, dash] & l_4 \arrow[r, dashed, dash] \arrow[d, dash] & l_5 \arrow[r, dashed, dash] \arrow[d, dash] & l_6 \arrow[r, dashed, dash] \arrow[d, dash] & l_7 \arrow[r, dashed, dash] \arrow[d, dash] & l_8 \arrow[r, dashed, dash] \arrow[d, dash] & l_9 \arrow[r, dashed, dash] \arrow[d, dash] & 0 \\
l_2 \arrow[r, dashed, dash] \arrow[d, dash] & l_3 \arrow[r, dashed, dash] \arrow[d, dash] & l_4 \arrow[r, dashed, dash] \arrow[d, dash] & l_5 \arrow[r, dashed, dash] \arrow[d, dash] & l_6 \arrow[r, dashed, dash] \arrow[d, dash] & l_7 \arrow[r, dashed, dash] \arrow[d, dash] & l_8 \arrow[r, dashed, dash] \arrow[d, dash] & l_9 \arrow[r, dashed, dash] \arrow[d, dash] & 0 & \\
l_3 \arrow[r, dashed, dash] \arrow[d, dash] & l_4 \arrow[r, dashed, dash] \arrow[d, dash] & l_5 \arrow[r, dashed, dash] \arrow[d, dash] & l_6 \arrow[r, dashed, dash] \arrow[d, dash] & l_7 \arrow[r, dashed, dash] \arrow[d, dash] & l_8 \arrow[r, dashed, dash] \arrow[d, dash] & l_9 \arrow[r, dashed, dash] \arrow[d, dash] & 1 & & \\
l_4 \arrow[r, dashed, dash] \arrow[d, dash] & l_5 \arrow[r, dashed, dash] \arrow[d, dash] & l_6 \arrow[r, dashed, dash] \arrow[d, dash] & l_7 \arrow[r, dashed, dash] \arrow[d, dash] & l_8 \arrow[r, dashed, dash] \arrow[d, dash] & l_9 \arrow[r, dashed, dash] \arrow[d, dash] & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & &
\end{tikzcd}$$

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

Tseitin Encoding of If-Then-Else Gate



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

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

minimal but <u>not</u> arc consistent:

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

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

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

DIMACS Format

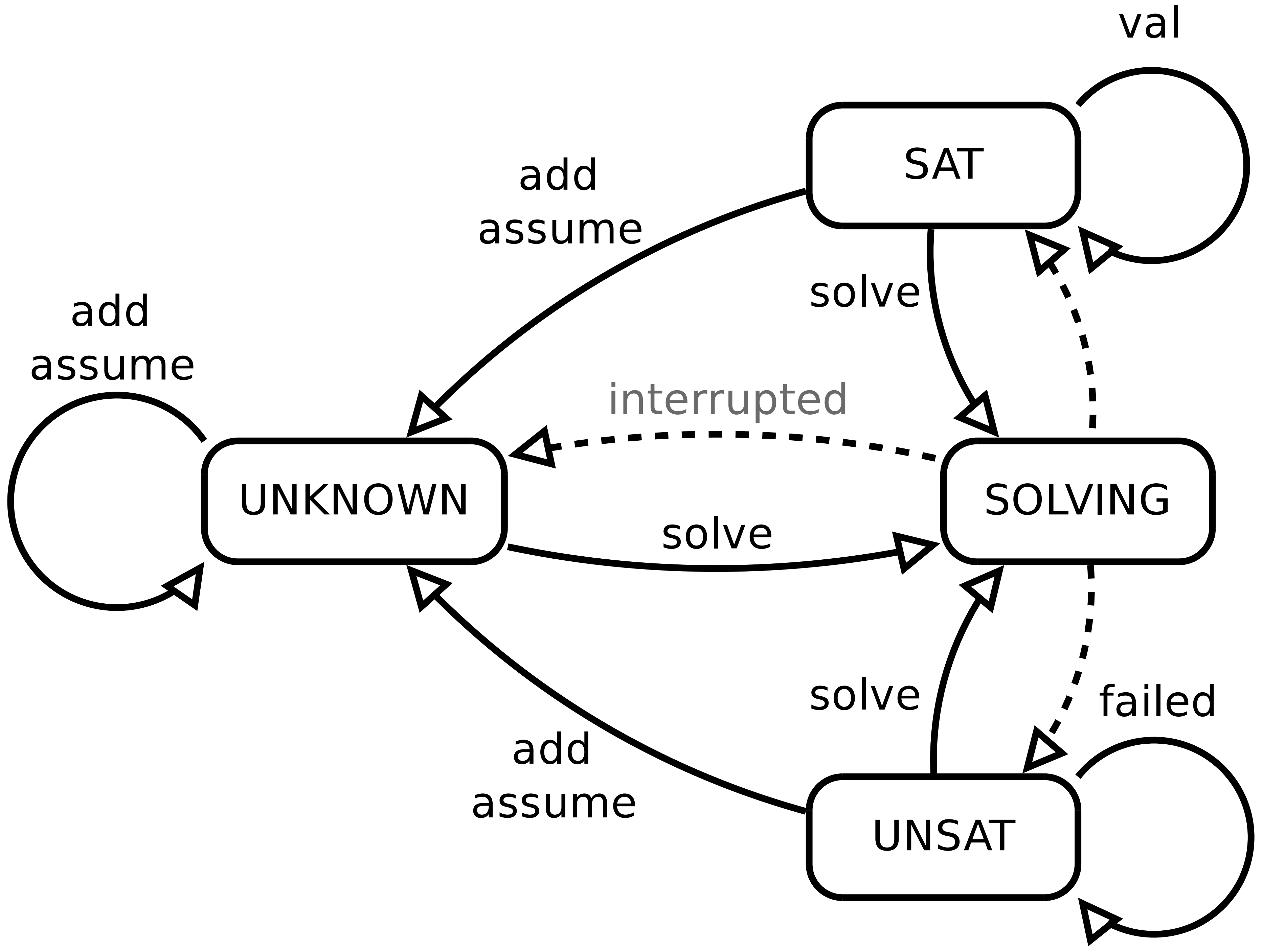
```
$ cat example.cnf
c comments start with 'c' and extend until the end of the line
c
c variables are encoded as integers:
c
c   'tie'   becomes '1'
c   'shirt' becomes '2'
c
c header 'p cnf <variables> <clauses>'
c
p cnf 2 3
-1  2 0           c  !tie  or  shirt
 1  2 0           c   tie  or  shirt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

```
#include "cadical.hpp"
#include <cassert>
#include <iostream>
using namespace std;
#define ADD(LIT) solver.add (LIT)
#define PRINT(LIT) \
  (solver.val (LIT) < 0 ? " -" #LIT : " " #LIT)
int main () {
  CaDiCaL::Solver solver; solver.set ("quiet", 1);
  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 = solver.solve ();
  assert (res == 10);
  cout << "satisfiable:" << PRINT (shirt) << PRINT (tie) << endl;
  cout << "assuming now: tie shirt" << endl;
  solver.assume (tie), solver.assume (shirt);
  res = solver.solve ();
  assert (res == 20);
  cout << "unsatisfiable, failed:";
  if (solver.failed (tie)) cout << " tie";
  if (solver.failed (shirt)) cout << " shirt";
  cout << endl;
  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 ($\leq$ 25 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

D(P)LL Procedure

```
$DPLL(F)$
    $F := BCP(F)$                                              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** $DPLL(F \wedge \{\neg l\})$
```

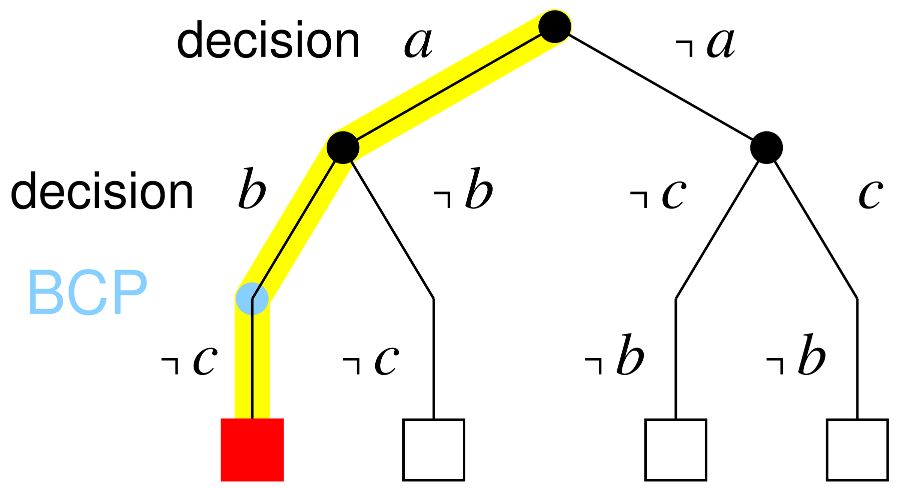
$\Rightarrow$ CDCL

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]

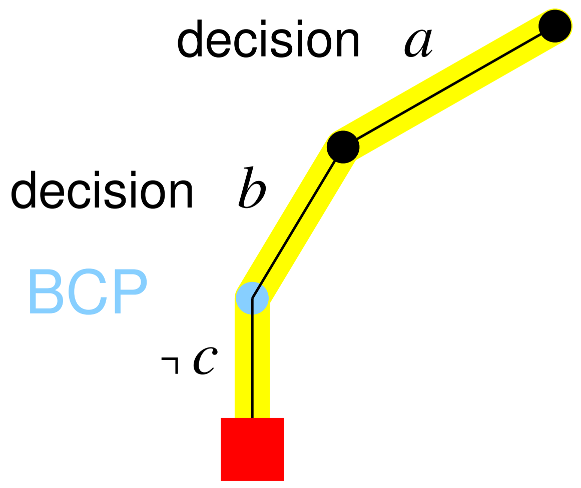
- first implemented in the context of GRASP SAT solver
  - name given later to distinguish it from DPLL
  - not recursive anymore
- essential for SMT
- learning clauses as no-goods
- notion of implication graph
- (first) unique implication points

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 1$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

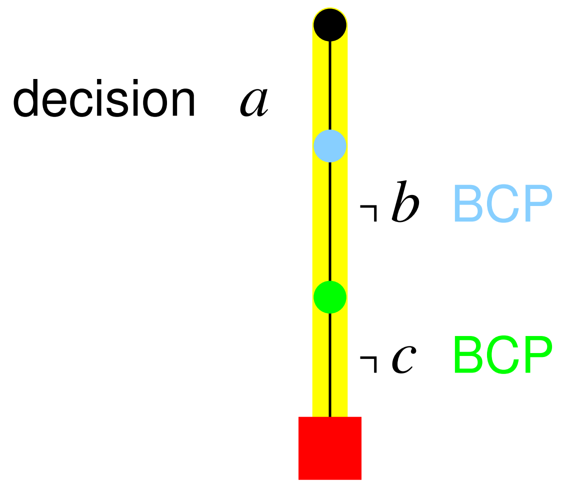
learn $\neg a \vee \neg b$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

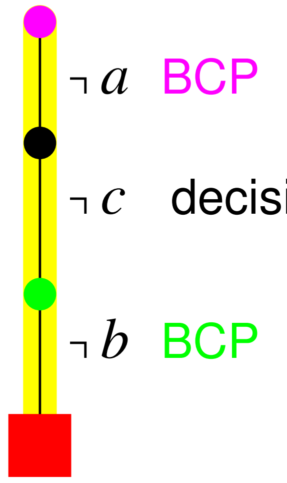
learn $\neg a$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

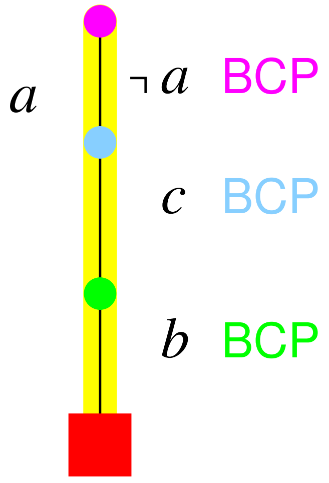
learn c

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 $\bot$

empty clause

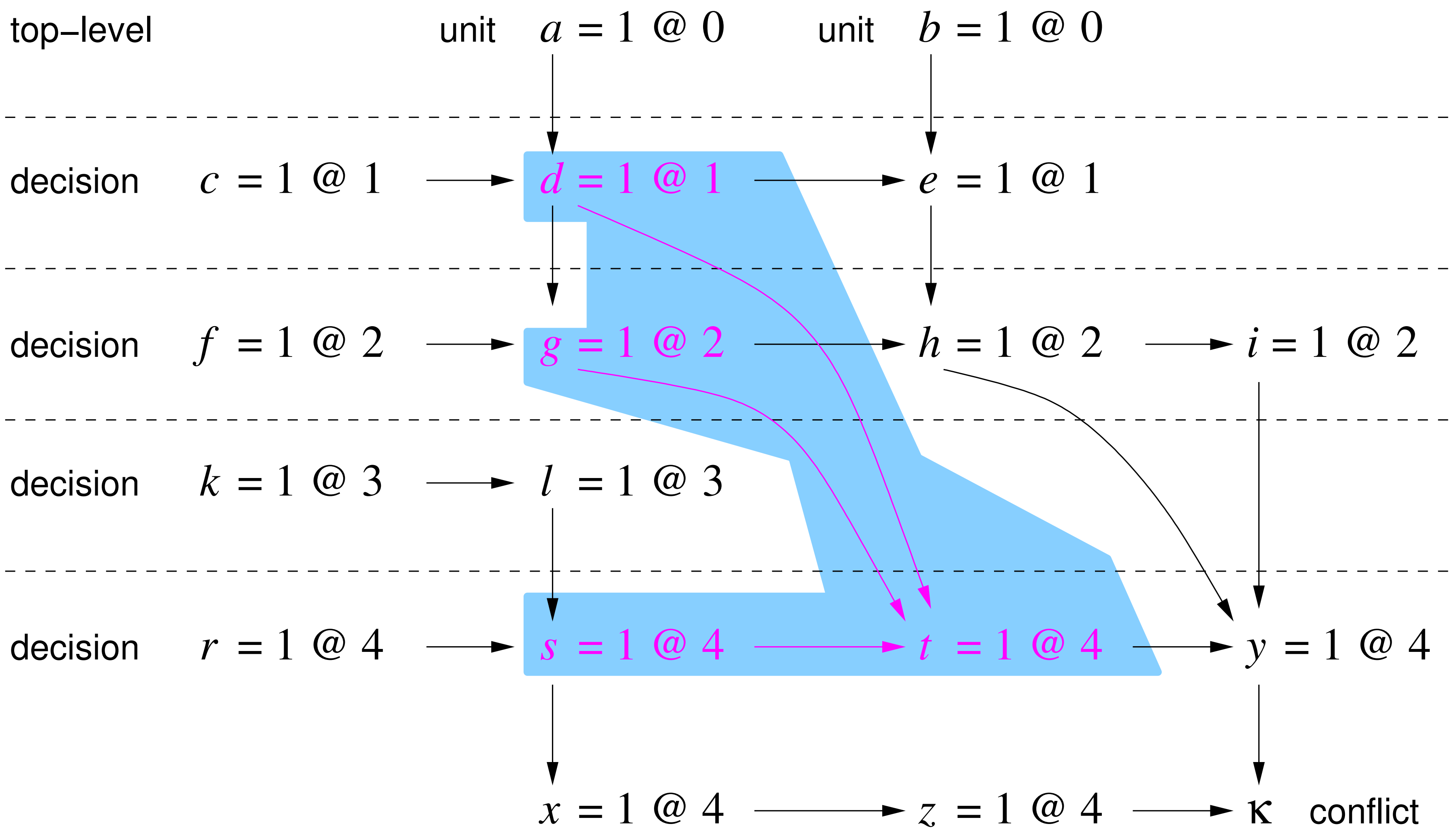
Implication Graph

$$\begin{tikzcd}
\text{top-level} & & \text{unit}\ a = 1 \ @\ 0 \arrow[d] & \text{unit}\ b = 1 \ @\ 0 \arrow[d] & \\
\text{decision} & c = 1 \ @\ 1 \arrow[r] & d = 1 \ @\ 1 \arrow[r] \arrow[d] \arrow[dddr, bend left] & e = 1 \ @\ 1 \arrow[d] & \\
\text{decision} & f = 1 \ @\ 2 \arrow[r] & g = 1 \ @\ 2 \arrow[r] \arrow[ddr, bend left] & h = 1 \ @\ 2 \arrow[r] \arrow[ddr, bend left] & i = 1 \ @\ 2 \arrow[dd] \\
\text{decision} & k = 1 \ @\ 3 \arrow[r] & l = 1 \ @\ 3 \arrow[d] & & \\
\text{decision} & r = 1 \ @\ 4 \arrow[r] & s = 1 \ @\ 4 \arrow[r] \arrow[d] & t = 1 \ @\ 4 \arrow[r] & y = 1 \ @\ 4 \arrow[d] \\
& & x = 1 \ @\ 4 \arrow[r] & z = 1 \ @\ 4 \arrow[r] & \kappa \ \text{conflict}
\end{tikzcd}$$

Conflict

$$\begin{tikzcd}
\text{top-level} & & \text{unit } a = 1 @ 0 \arrow[d] & \text{unit } b = 1 @ 0 \arrow[d] & \\
\text{decision} & c = 1 @ 1 \arrow[r] & d = 1 @ 1 \arrow[r] \arrow[d] \arrow[dddr, bend left] & e = 1 @ 1 \arrow[d] & \\
\text{decision} & f = 1 @ 2 \arrow[r] & g = 1 @ 2 \arrow[r] \arrow[ddr, bend left] & h = 1 @ 2 \arrow[r] \arrow[ddr, bend left] & i = 1 @ 2 \arrow[dd] \\
\text{decision} & k = 1 @ 3 \arrow[r] & l = 1 @ 3 \arrow[d] & & \\
\text{decision} & r = 1 @ 4 \arrow[r] & s = 1 @ 4 \arrow[r] \arrow[d] & t = 1 @ 4 \arrow[r] & y = 1 @ 4 \arrow[d] \\
& & x = 1 @ 4 \arrow[r] & z = 1 @ 4 \arrow[r] & \kappa \ \text{conflict}
\end{tikzcd}$$

Antecedents / Reasons



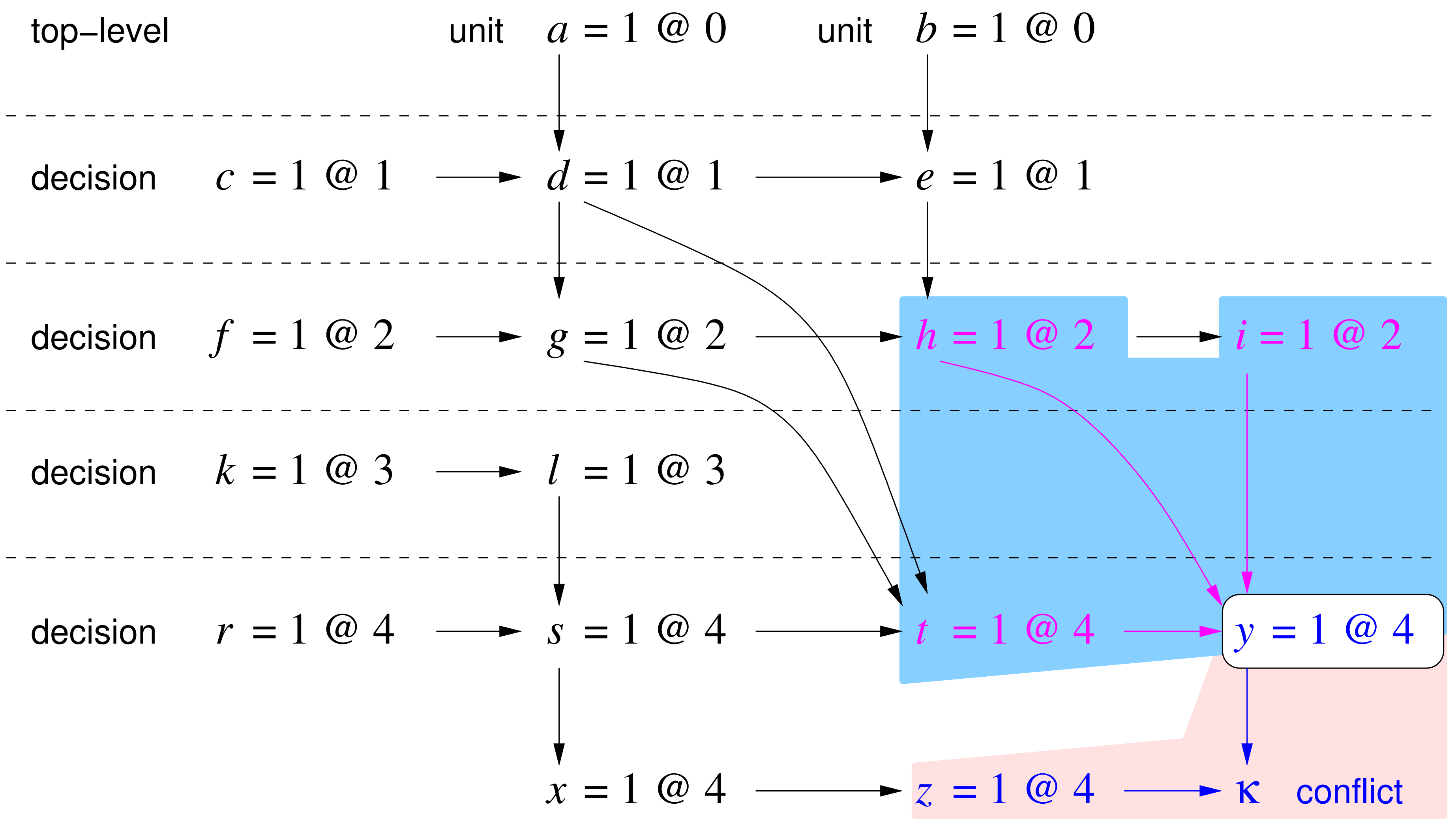
$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee t)$$

Conflicting Clauses

$$\begin{tikzcd}
\text{top-level} & & \text{unit}\ a = 1\ @\ 0 \arrow[d] & \text{unit}\ b = 1\ @\ 0 \arrow[d] & \\
\text{decision} & c = 1\ @\ 1 \arrow[r] & d = 1\ @\ 1 \arrow[r] \arrow[d] \arrow[dddr, bend left] & e = 1\ @\ 1 \arrow[d] & \\
\text{decision} & f = 1\ @\ 2 \arrow[r] & g = 1\ @\ 2 \arrow[r] \arrow[ddr, bend left] & h = 1\ @\ 2 \arrow[r] \arrow[ddr, bend left] & i = 1\ @\ 2 \arrow[dd] \\
\text{decision} & k = 1\ @\ 3 \arrow[r] & l = 1\ @\ 3 \arrow[d] & & \\
\text{decision} & r = 1\ @\ 4 \arrow[r] & s = 1\ @\ 4 \arrow[r] \arrow[d] & t = 1\ @\ 4 \arrow[r] & y = 1\ @\ 4 \arrow[d] \\
 & & x = 1\ @\ 4 \arrow[r] & z = 1\ @\ 4 \arrow[r] & \kappa\ \text{conflict}
\end{tikzcd}$$

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

Resolving Antecedents 1st Time



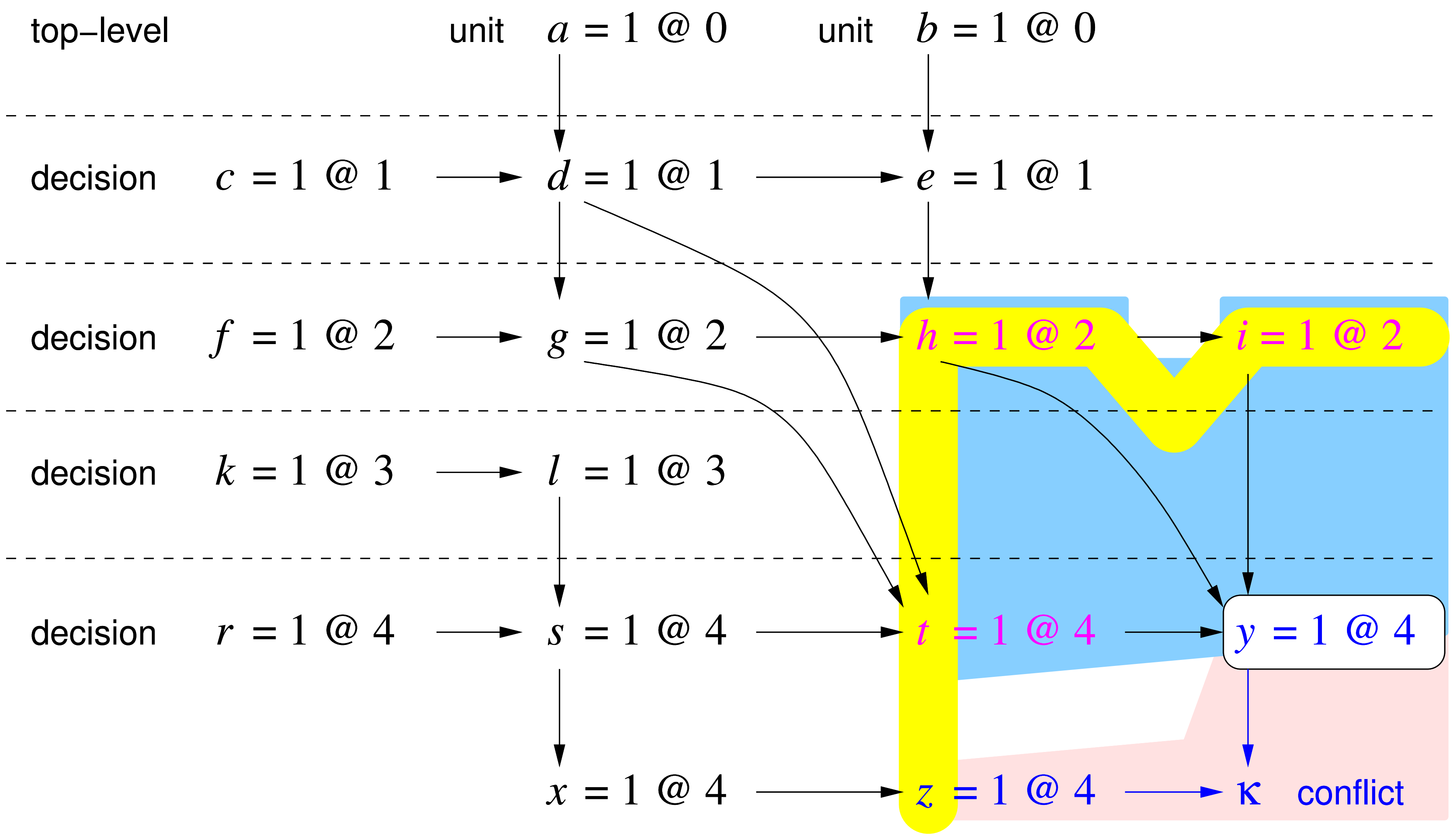
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

Resolving Antecedents 1st Time

$$\begin{tikzcd}
\text{top-level} & & \text{unit } a = 1 \; @ \; 0 \arrow[d] & \text{unit } b = 1 \; @ \; 0 \arrow[d] & \\
\text{decision} & c = 1 \; @ \; 1 \arrow[r] & d = 1 \; @ \; 1 \arrow[r] \arrow[d] \arrow[dddr, bend left] & e = 1 \; @ \; 1 \arrow[d] & \\
\text{decision} & f = 1 \; @ \; 2 \arrow[r] & g = 1 \; @ \; 2 \arrow[r] \arrow[ddr, bend left] & h = 1 \; @ \; 2 \arrow[r] \arrow[ddr, bend left] & i = 1 \; @ \; 2 \arrow[dd] \\
\text{decision} & k = 1 \; @ \; 3 \arrow[r] & l = 1 \; @ \; 3 \arrow[d] & & \\
\text{decision} & r = 1 \; @ \; 4 \arrow[r] & s = 1 \; @ \; 4 \arrow[r] \arrow[d] & t = 1 \; @ \; 4 \arrow[r] & \boxed{y = 1 \; @ \; 4} \arrow[d] \\
& & x = 1 \; @ \; 4 \arrow[r] & z = 1 \; @ \; 4 \arrow[r] & \kappa \quad \text{conflict}
\end{tikzcd}$$

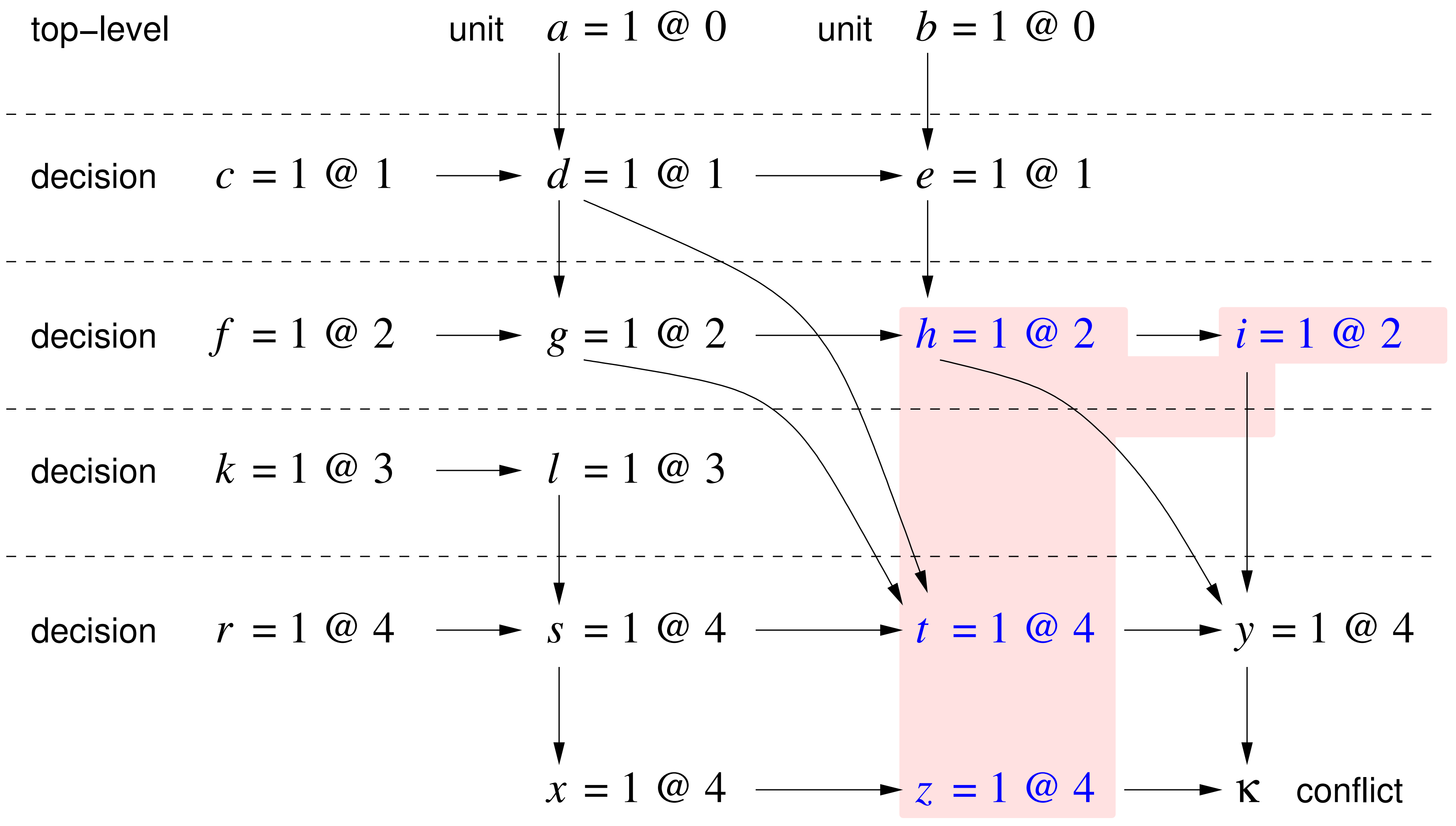
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Resolvents = Cuts = Potential Learned Clauses



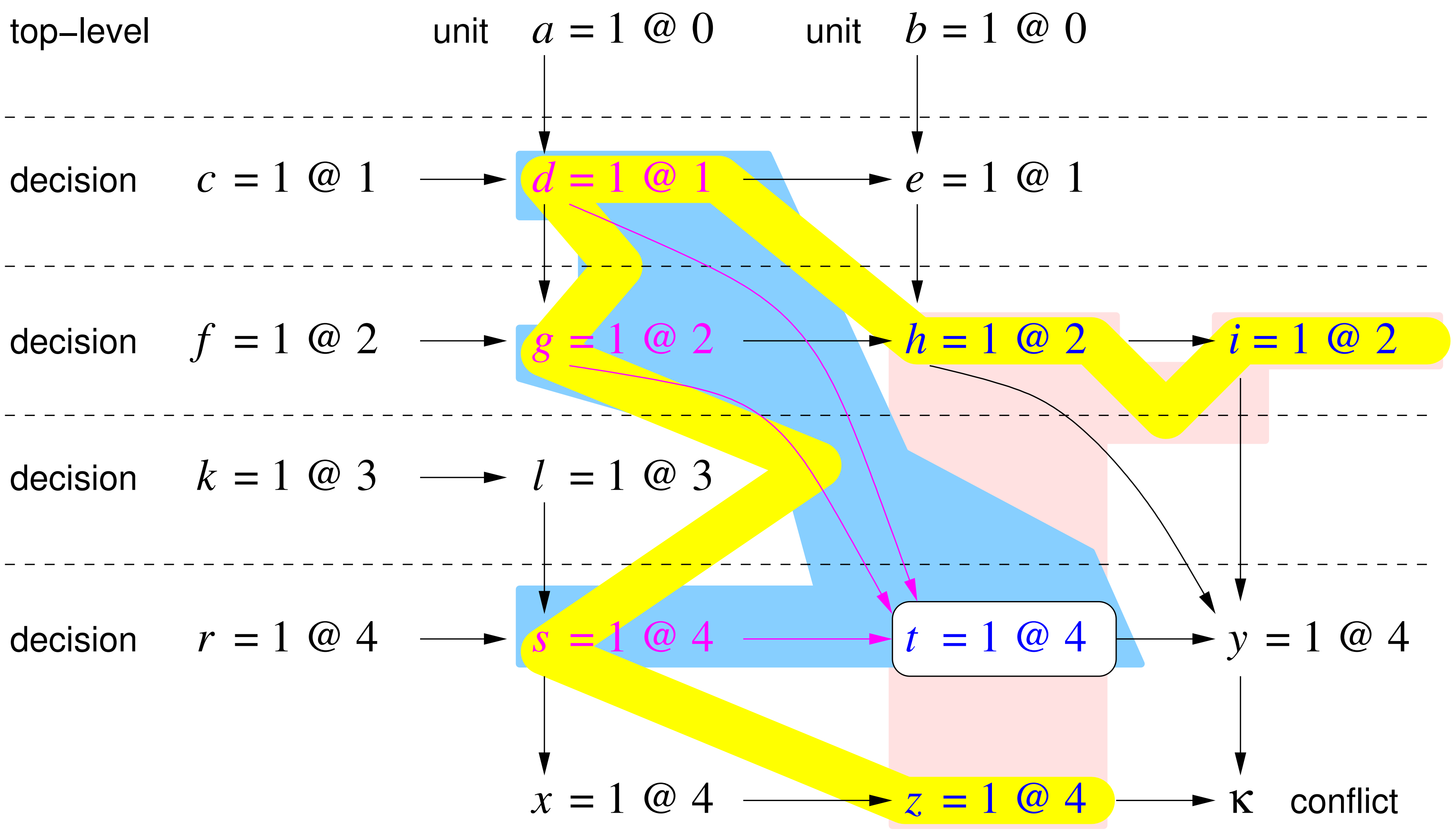
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Potential Learned Clause After 1 Resolution



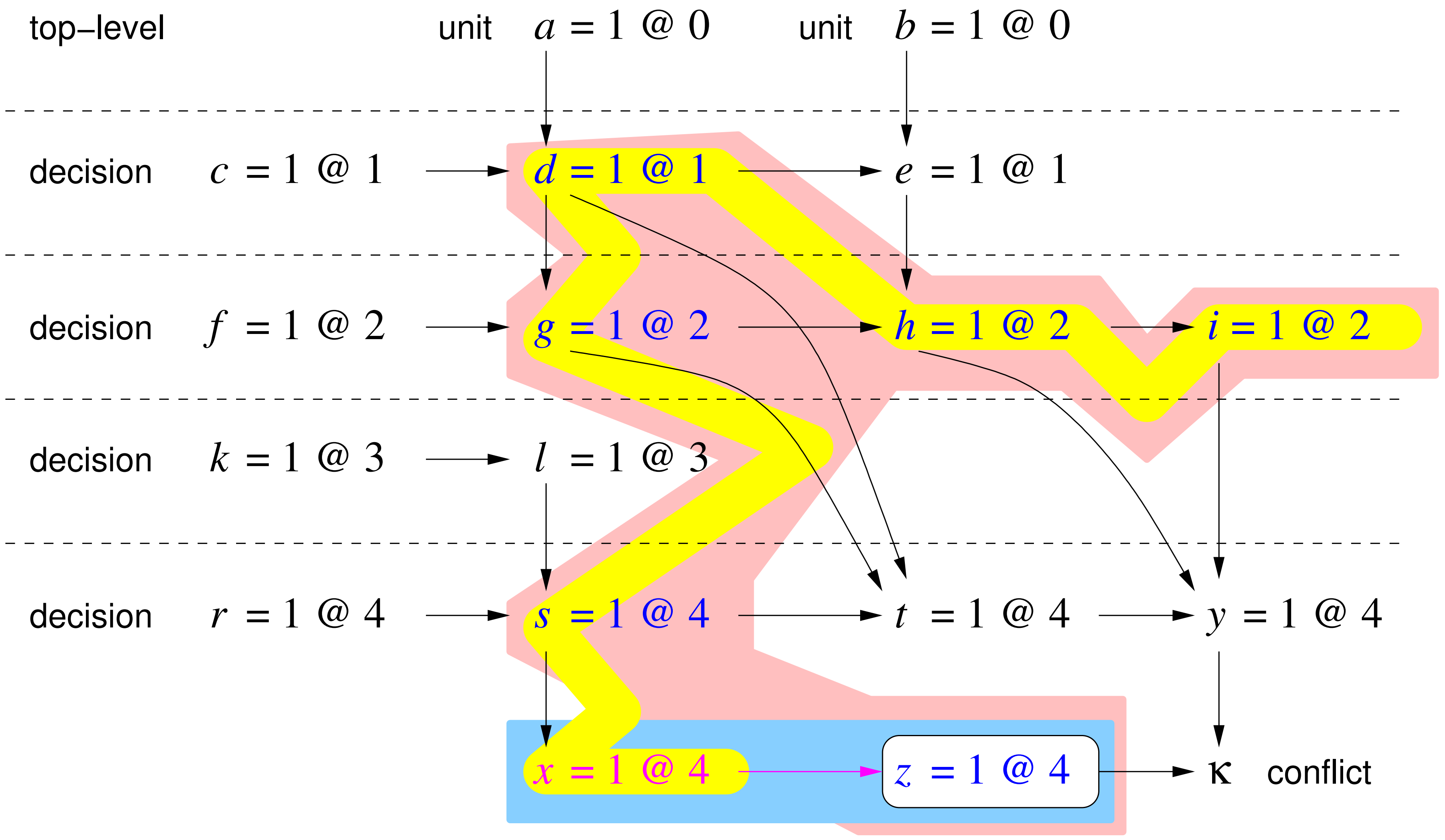
$(\overline{h} \vee \overline{i} \vee \overline{t} \vee \overline{z})$

Resolving Antecedents 2nd Time



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

Resolving Antecedents 3rd Time



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

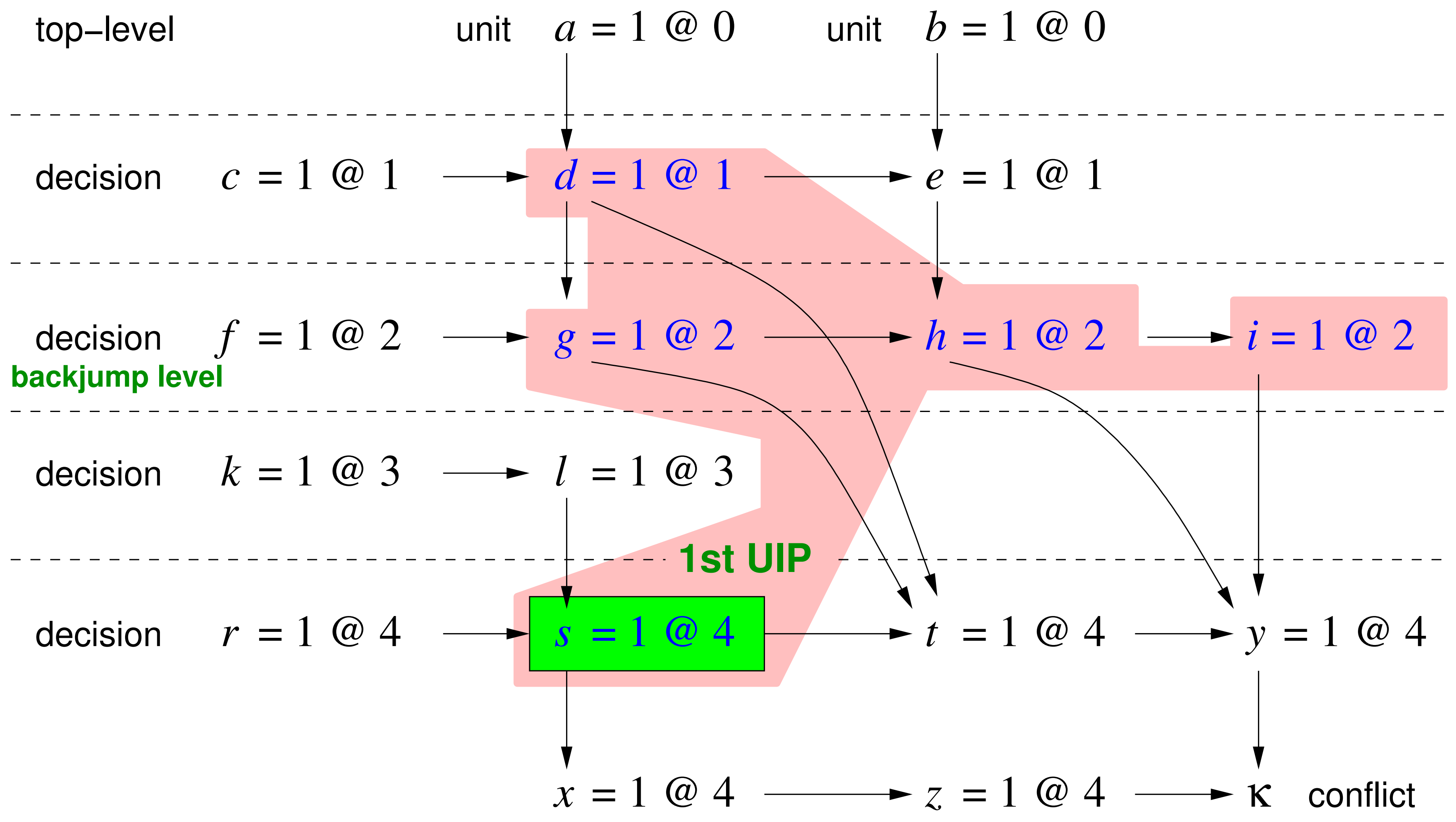
Resolving Antecedents 4th Time

$$\begin{tikzcd}
\text{top-level} & & \text{unit } a = 1\ @\ 0 \arrow[d] & \text{unit } b = 1\ @\ 0 \arrow[d] & \\
\text{decision} & c = 1\ @\ 1 \arrow[r] & d = 1\ @\ 1 \arrow[r] \arrow[d] \arrow[rddd, bend left] & e = 1\ @\ 1 \arrow[d] & \\
\text{decision} & f = 1\ @\ 2 \arrow[r] & g = 1\ @\ 2 \arrow[r] \arrow[rdd, bend left] & h = 1\ @\ 2 \arrow[r] \arrow[rdd, bend left] & i = 1\ @\ 2 \arrow[dd] \\
\text{decision} & k = 1\ @\ 3 \arrow[r] & l = 1\ @\ 3 \arrow[d] & & \\
\text{decision} & r = 1\ @\ 4 \arrow[r] & s = 1\ @\ 4 \arrow[r] \arrow[d] & t = 1\ @\ 4 \arrow[r] & y = 1\ @\ 4 \arrow[d] \\
& & \boxed{x = 1\ @\ 4} \arrow[r] & z = 1\ @\ 4 \arrow[r] & \kappa \quad \text{conflict}
\end{tikzcd}$$

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

self subsuming resolution

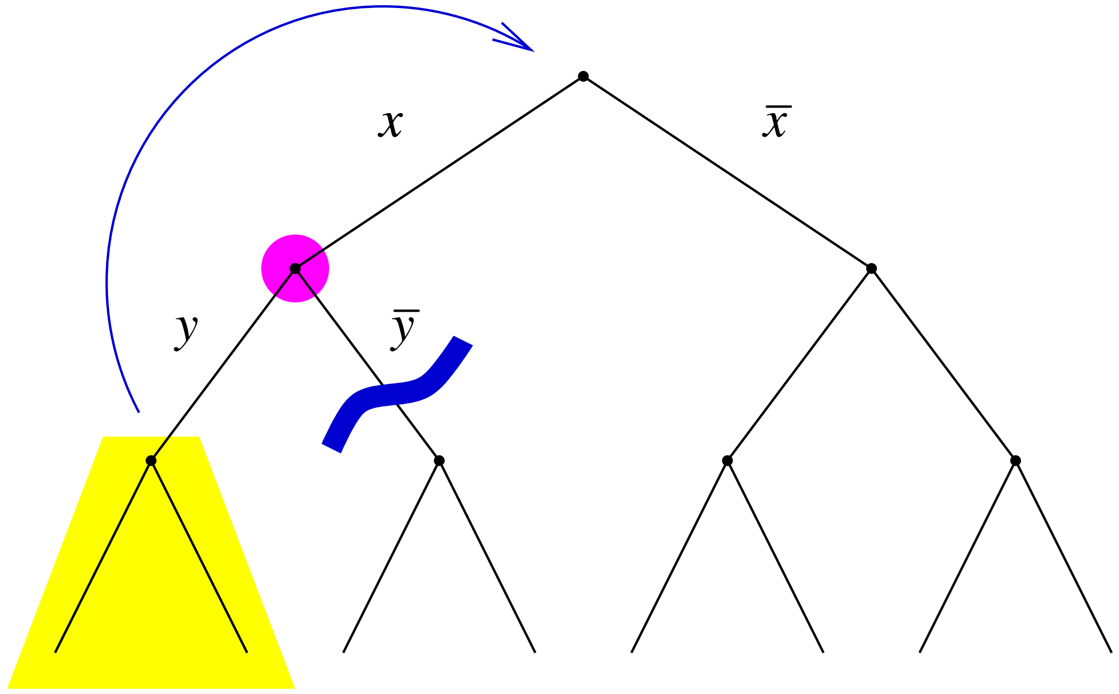
1st UIP Clause after 4 Resolutions



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

UIP = unique implication point   dominates conflict on the last level

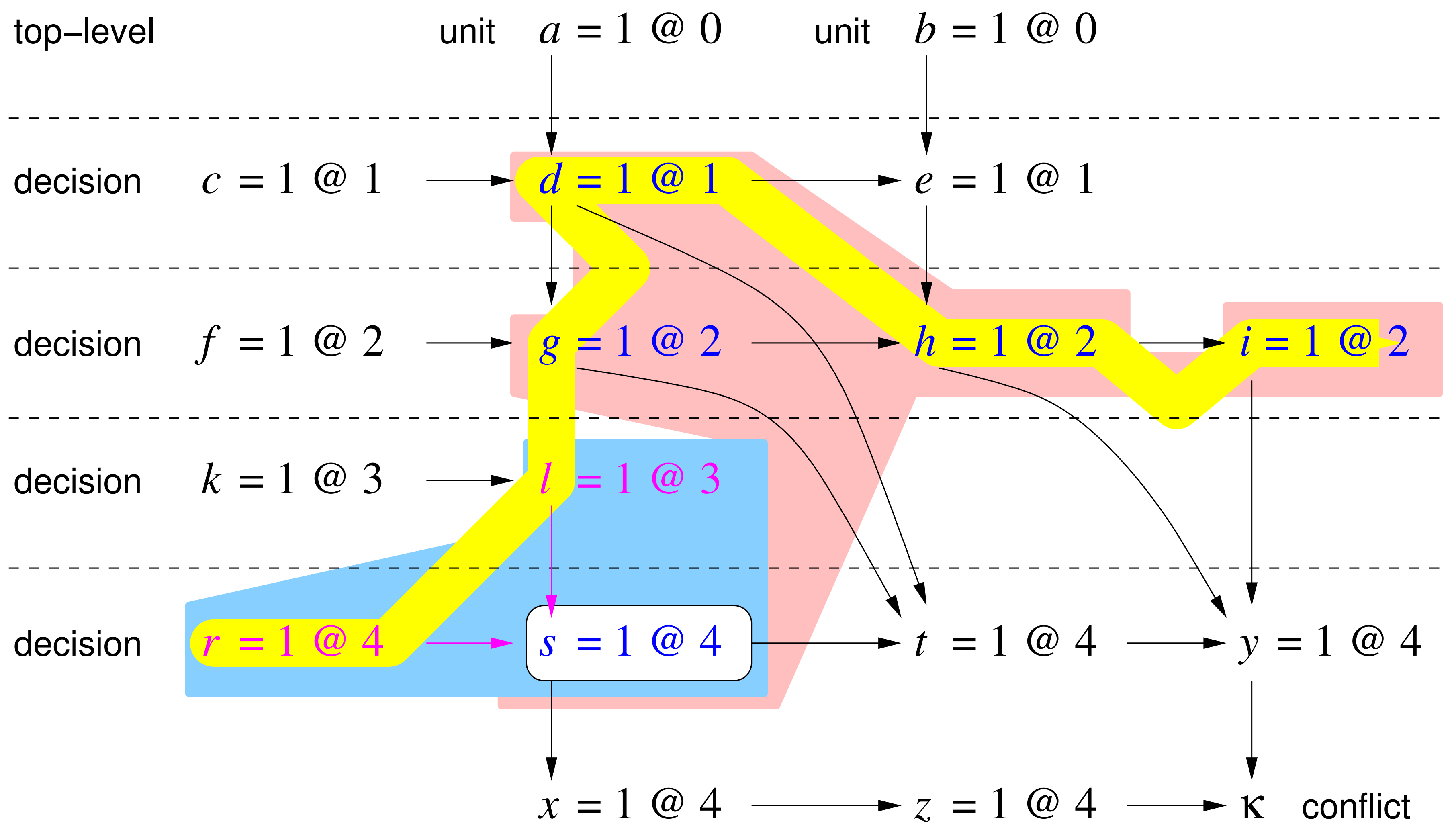
Backjumping



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

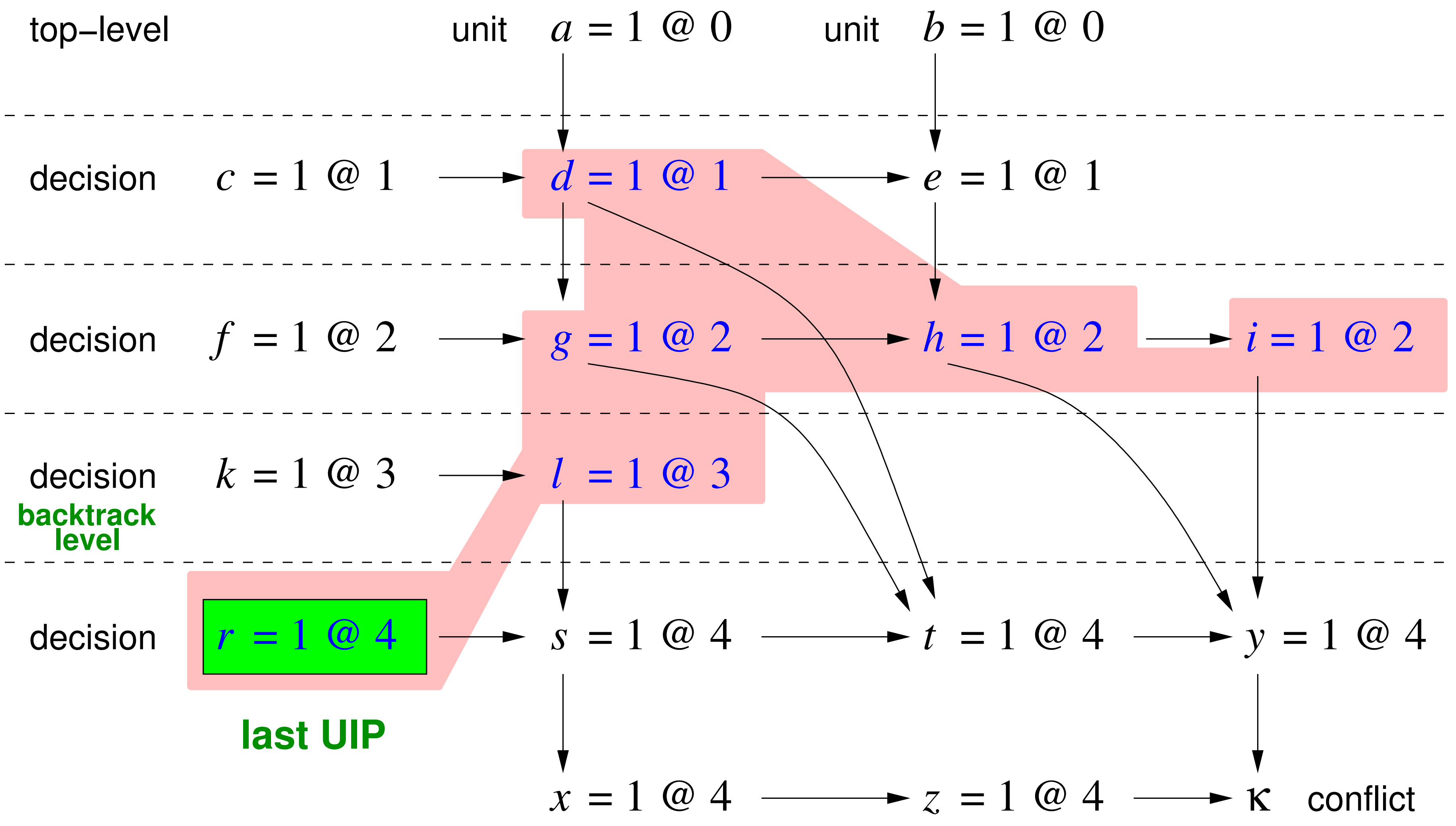
Immediately jump back to the $\overline{x}$ case – assuming x was used.

Resolving Antecedents 5th Time



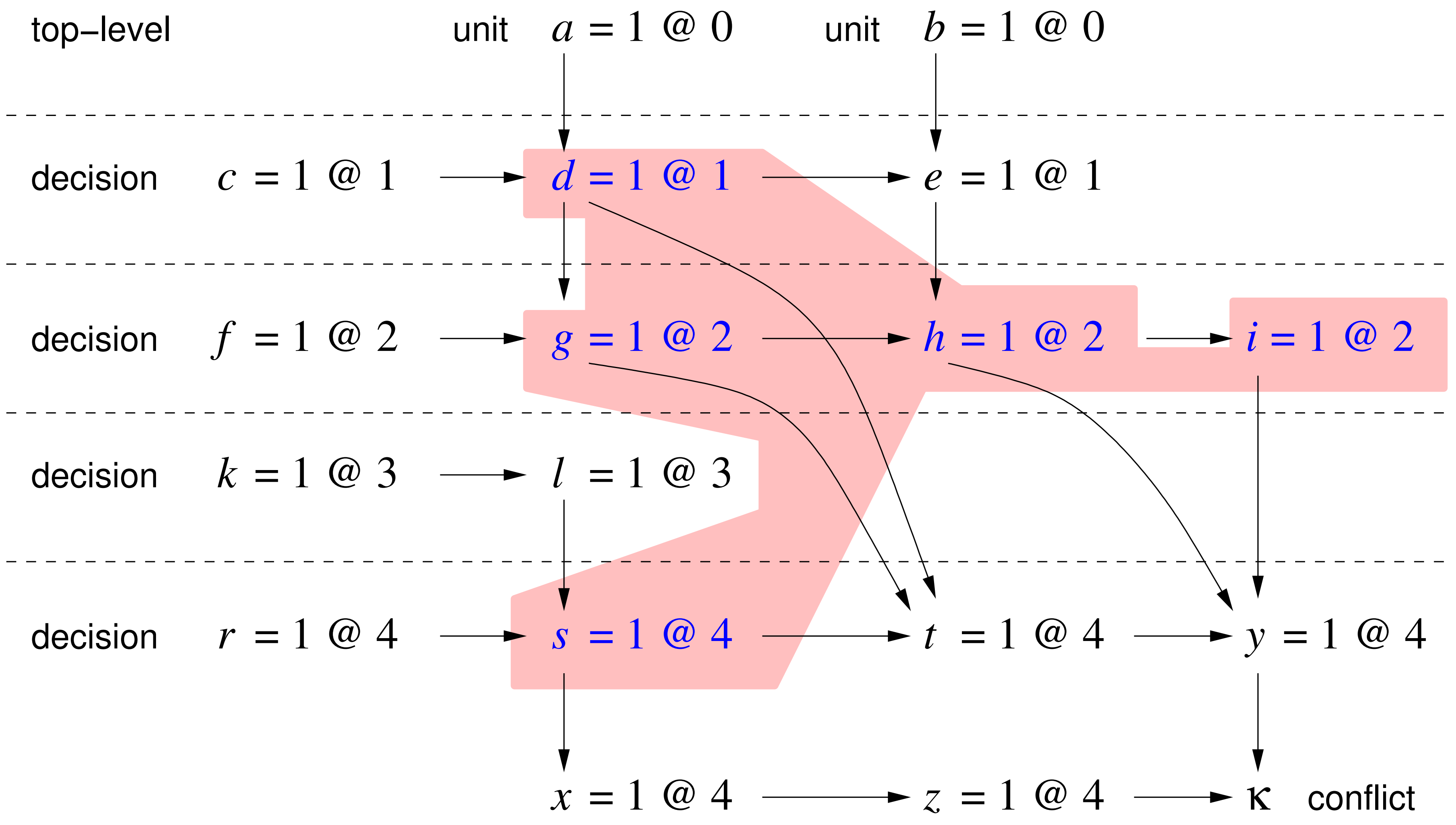
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



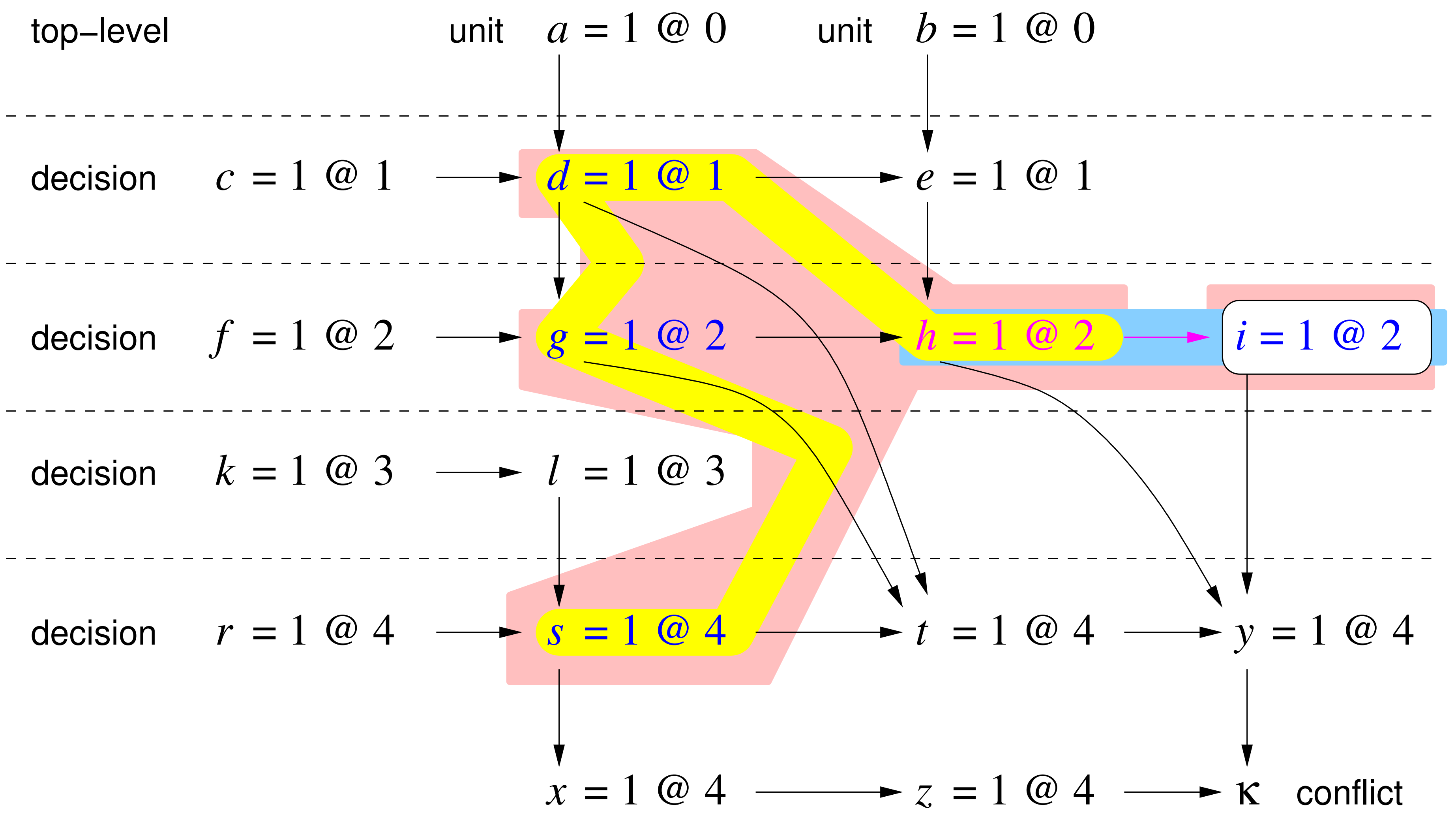
$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

1st UIP Clause after 4 Resolutions



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

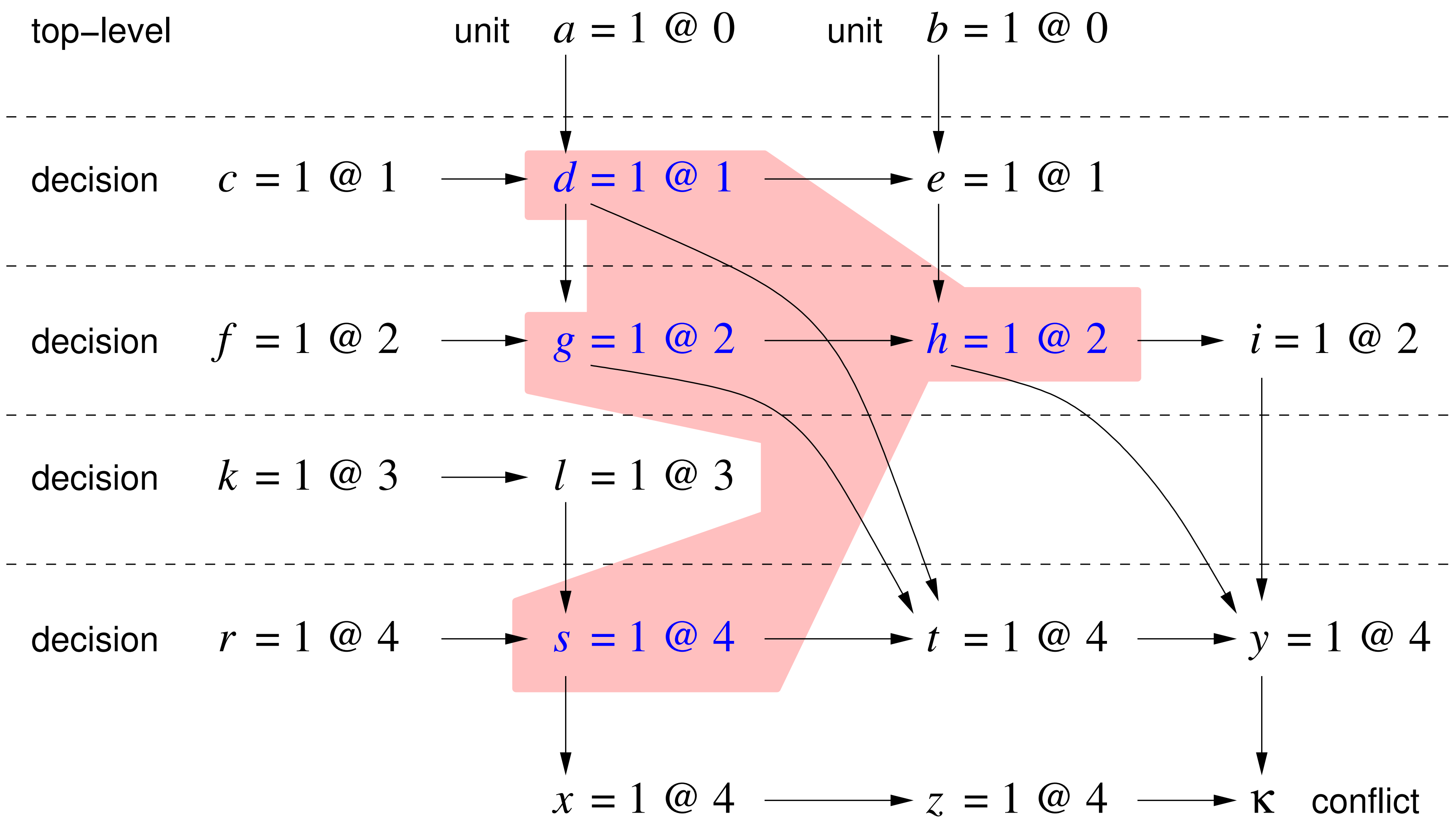
Locally Minimizing 1st UIP Clause



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

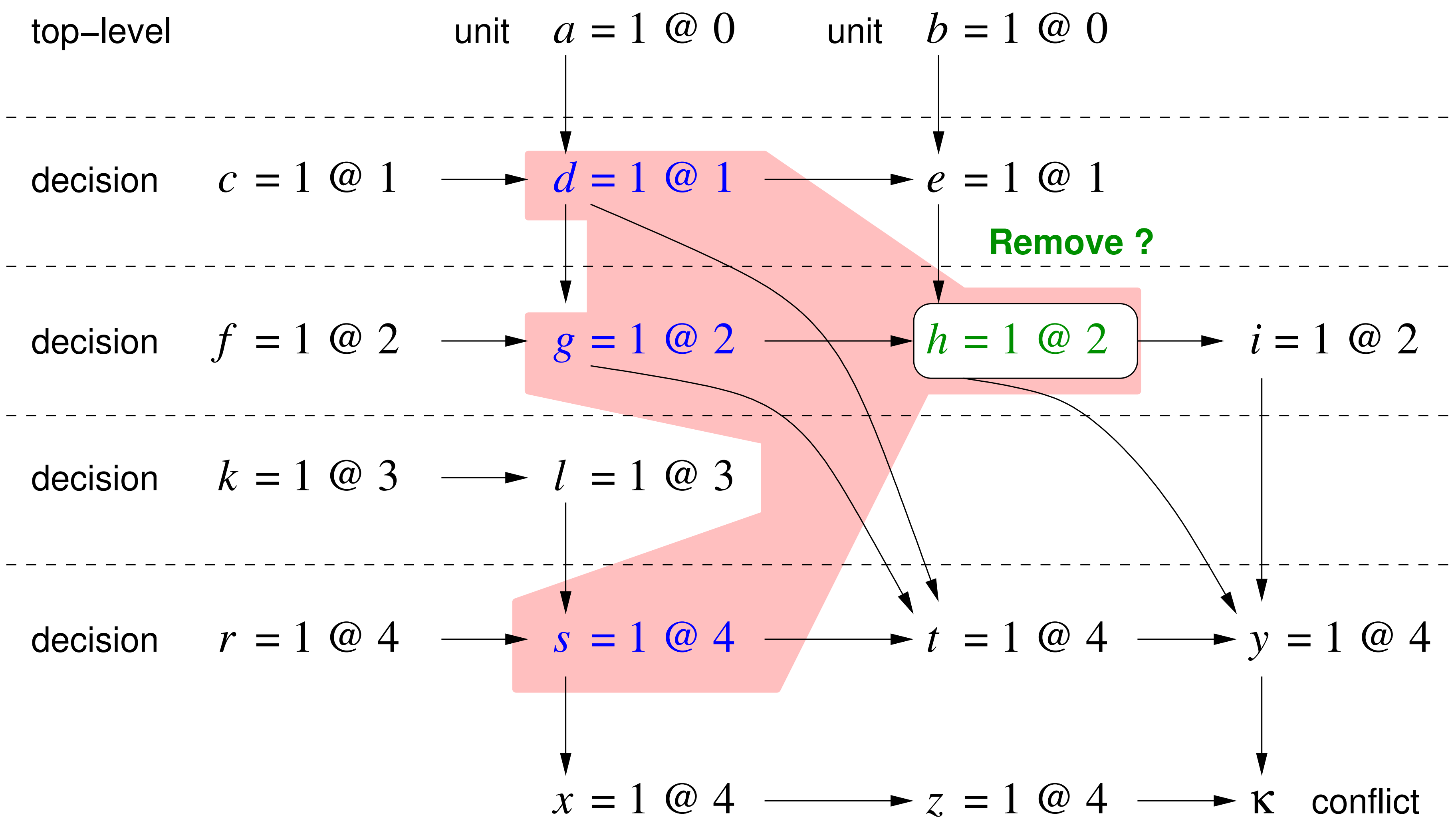
self subsuming resolution

Locally Minimized Learned Clause



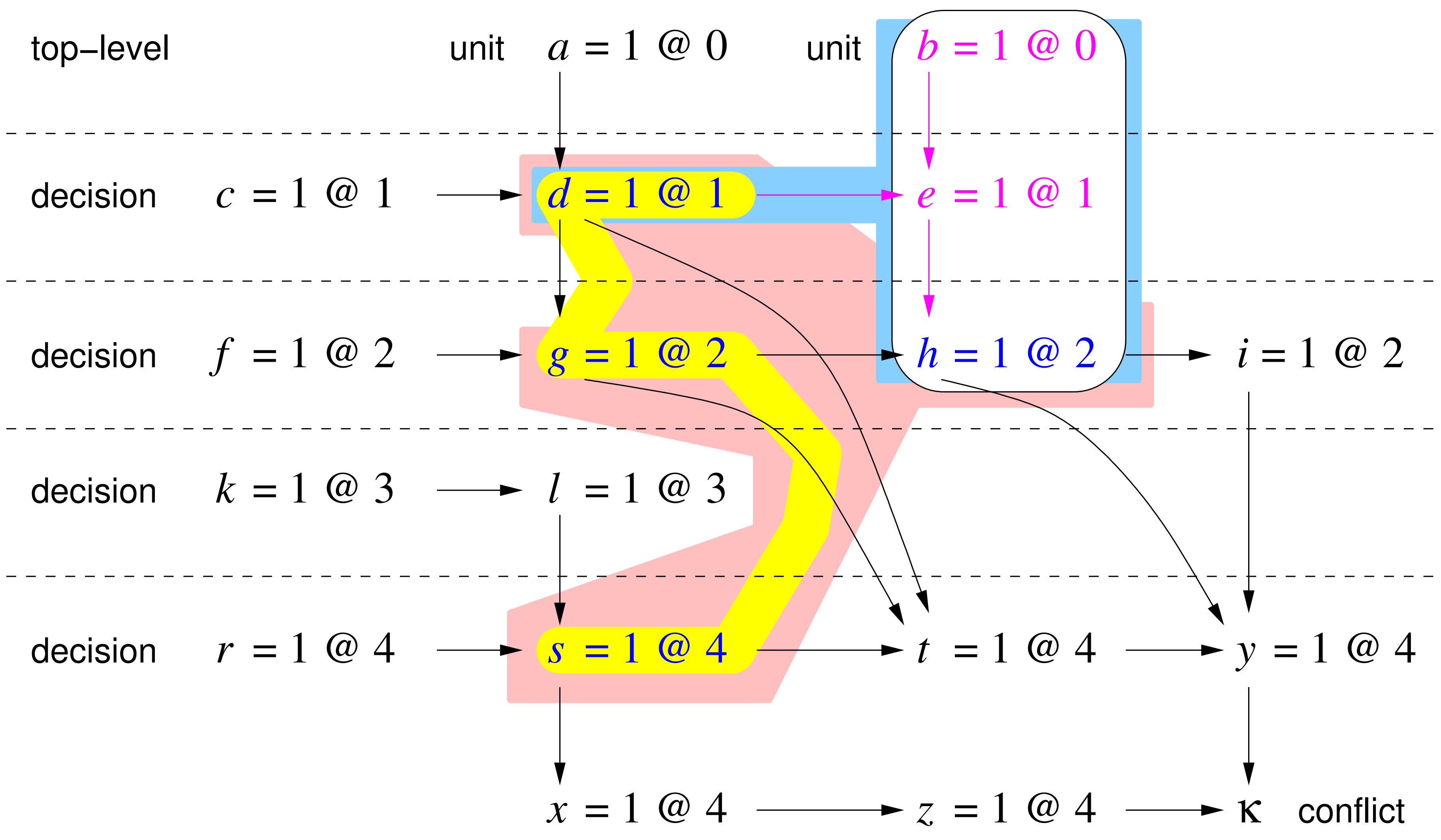
$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$

Minimizing Locally Minimized Learned Clause Further?



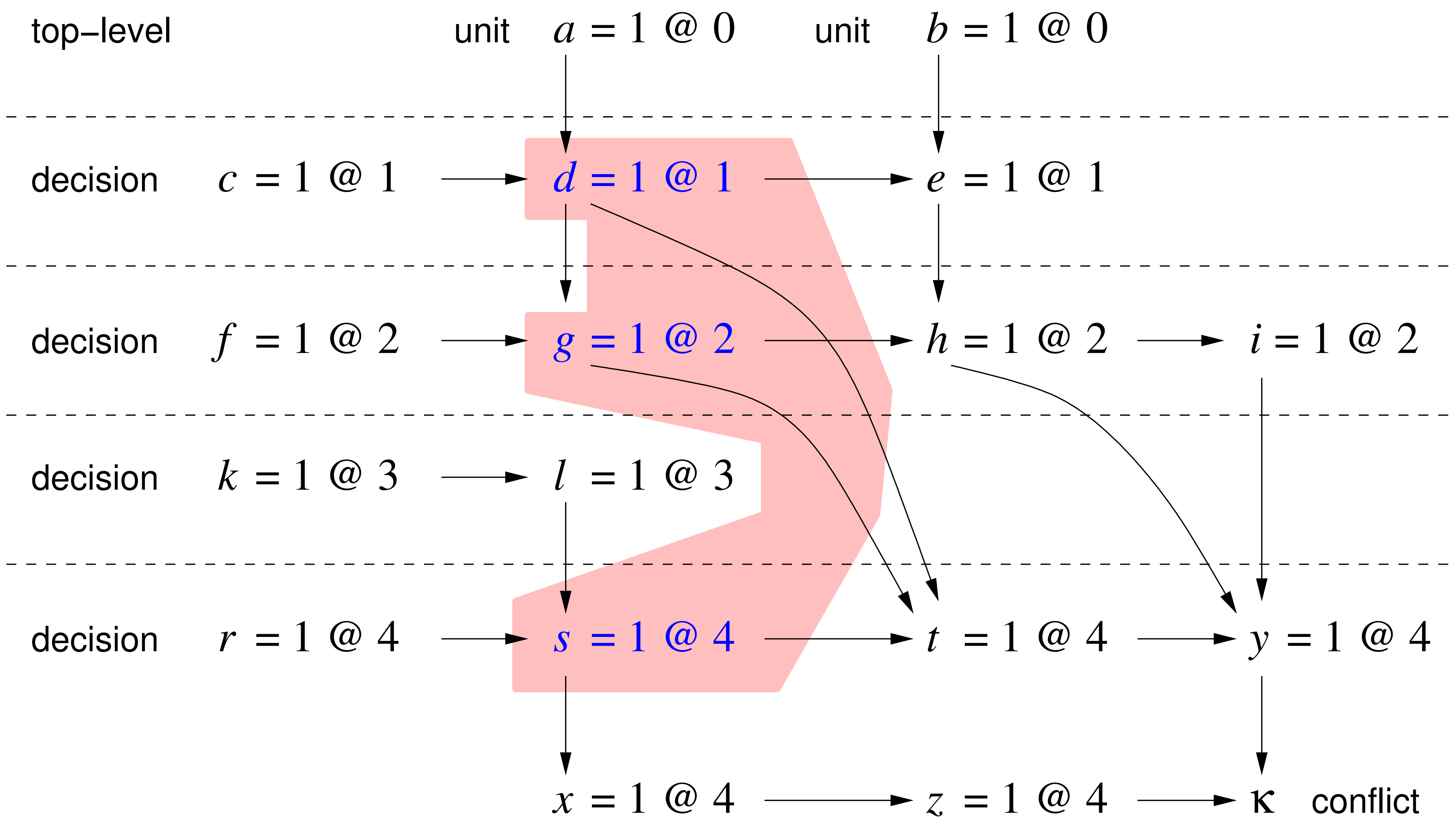
$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$

Recursively Minimizing Learned Clause



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

Recursively Minimized Learned Clause



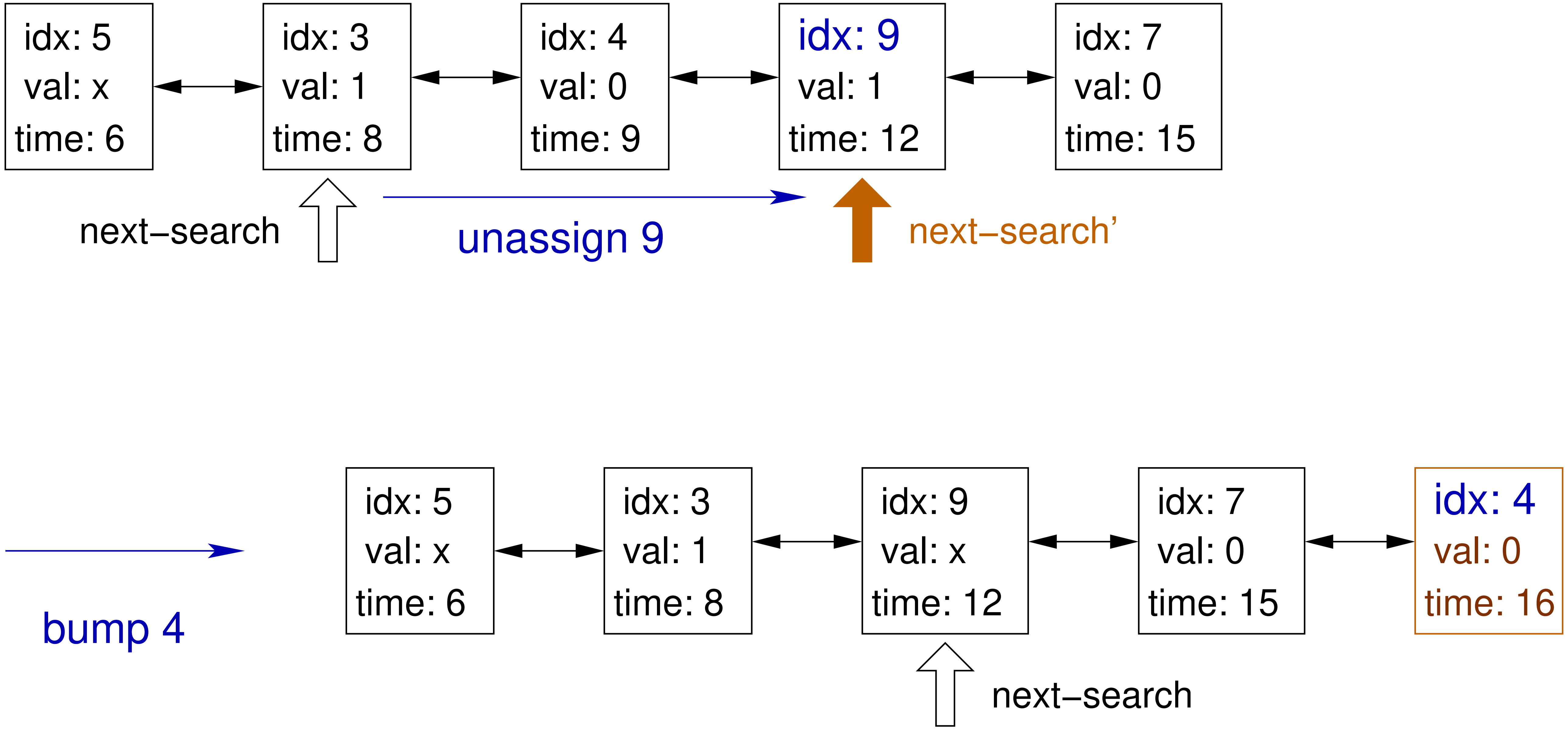
$(\overline{d} \vee \overline{g} \vee \overline{s})$

Decision Heuristics

- number of variable occurrences in (remaining unsatisfied) clauses (LIS)
  - eagerly satisfy many clauses with many variations 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]
- EVSIDS during stabilization VMTF otherwise [Biere-SAT-Race-2019]

Fast VMTF Implementation

- Siege SAT solver [Ryan Thesis 2004] used variable move to front (VMTF)
  - bumped variables moved to head of doubly linked list
  - search for unassigned variable starts at head
  - variable selection is an online sorting algorithm of scores
  - classic “move-to-front” strategy achieves good amortized complexity
- fast simple implementation for caching searches in VMTF [BiereFröhlich’SAT15]
  - doubly linked list does not have positions as an ordered array
  - bump $=$ move-to-front $=$ dequeue then insertion at the head
- time-stamp list entries with “insertion-time”
  - maintained invariant: all variables right of next-search are assigned
  - requires (constant time) update to next-search while unassigning variables
  - occassionally (32-bit) time-stamps will overflow: update all time stamps


idx: 5
val: x
time: 6
idx: 3
val: 1
time: 8
idx: 4
val: 0
time: 9
idx: 9
val: 1
time: 12
idx: 7
val: 0
time: 15
next–search
unassign 9
next–search'
bump 4
idx: 5
val: x
time: 6
idx: 3
val: 1
time: 8
idx: 9
val: x
time: 12
idx: 7
val: 0
time: 15
idx: 4
val: 0
time: 16
next–search

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$_1$	i	s	variable move-to-front
VMTF$_2$	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

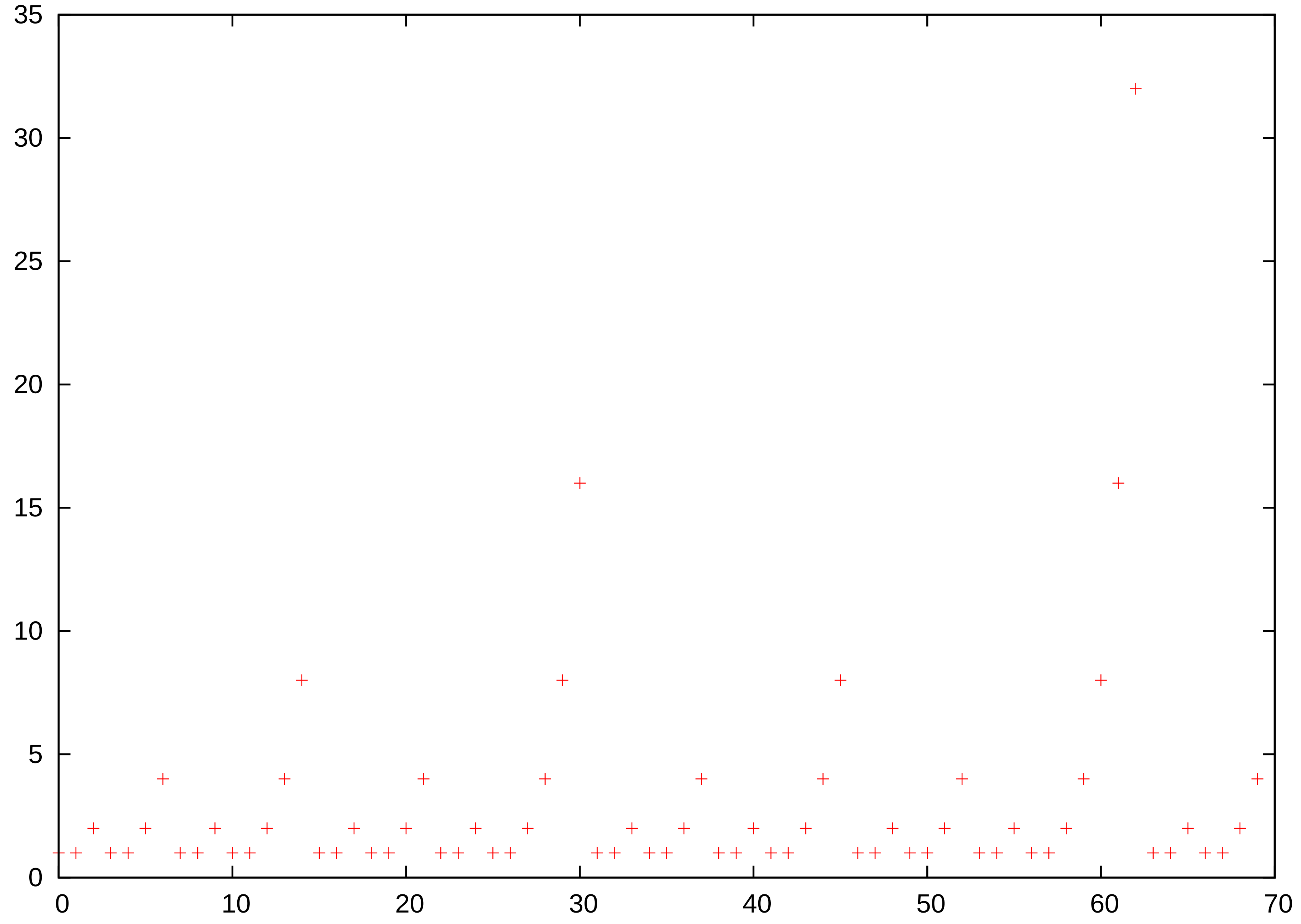
- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only "short" clauses
- better periodically delete "useless" learned clauses
  - keep a certain number of learned clauses — "search cache"
  - if this number is reached MiniSAT reduces (deletes) half 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 ($\leq 2 \ldots 5$)
  - three Tier system by [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
- interleave “stabilizing” (no restarts) and “non-stabilizing” phases [Chanseok Oh]

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

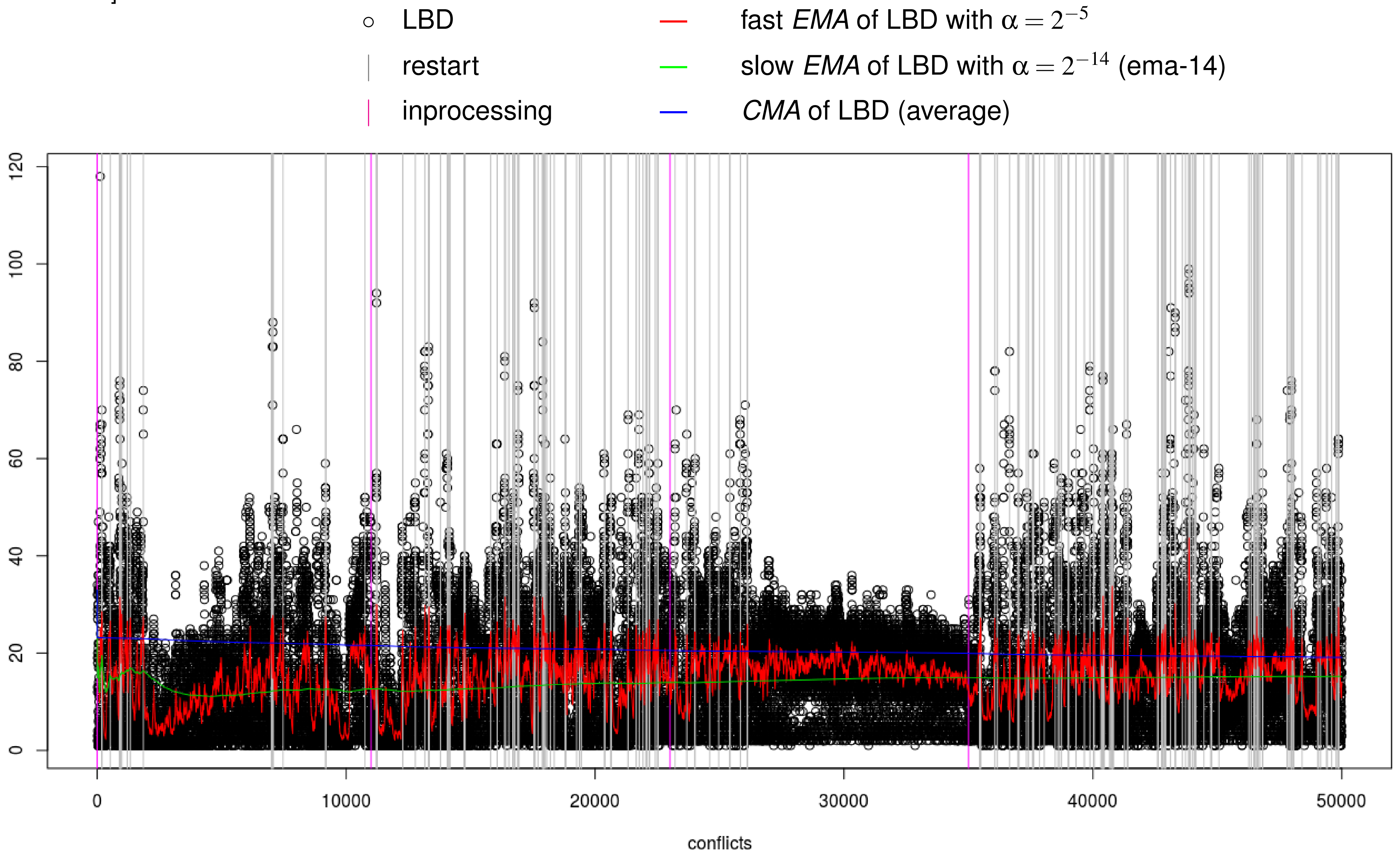
[Knuth'12]

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

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

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” [Hans van Maaren]
- “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**
- 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]
- target phases of largest conflict free trail / assignment [Biere-SAT-Race-2019]

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

new Version 1.2.1 from Oct. 9

```
int Internal::cdcl_loop_with_inprocessing () {

  int res = 0;

  while (!res) {
         if (unsat) res = 20;
    else if (!propagate ()) analyze ();      // propagate and analyze
    else if (iterating) iterate ();          // report learned unit
    else if (satisfied ()) res = 10;         // found model
    else if (terminating ()) break;          // limit hit or async abort
    else if (restarting ()) restart ();      // restart by backtracking
    else if (rephasing ()) rephase ();       // reset variable phases
    else if (reducing ()) reduce ();         // collect useless clauses
    else if (probing ()) probe ();           // failed literal probing
    else if (subsuming ()) subsume ();       // subsumption algorithm
    else if (eliminating ()) elim ();        // variable elimination
    else if (compacting ()) compact ();      // collect variables
    else if (conditioning ()) condition ();  // globally blocked clauses
    else res = decide ();                    // next decision
  }

  return res;
}
```

https://github.com/arminbiere/cadical

https://fmv.jku.at/cadical

Two-Watched Literal Schemes

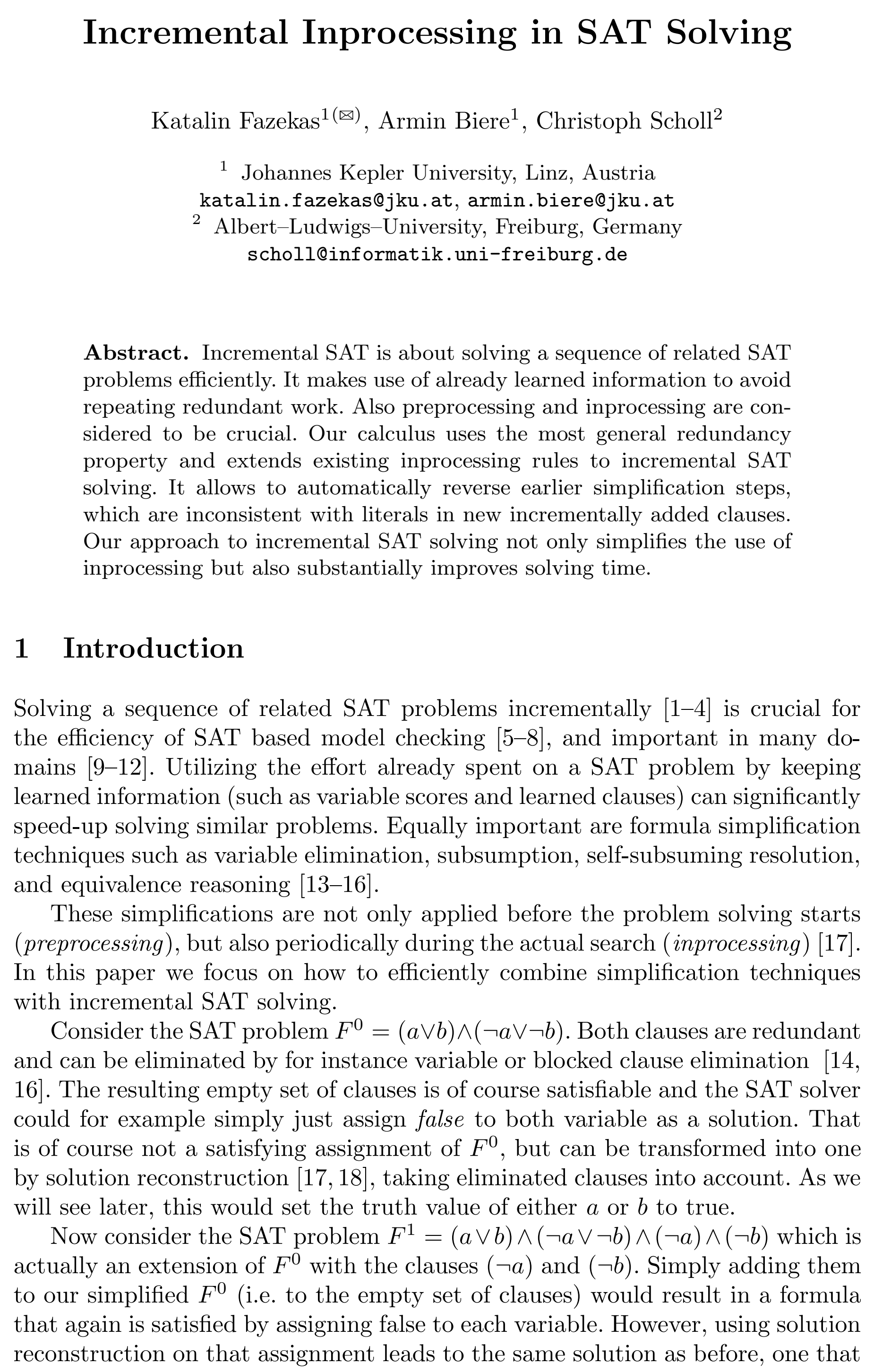
- original idea from SATO [ZhangStickel'00]
  - invariant: $\boxed{\text{always watch two non-false literals}}$
  - if a watched literal becomes <u>false</u> replace it
  - if no replacement can be found clause is either unit or empty
  - original version used <u>head</u> and <u>tail</u> pointers on Tries
- improved variant from Chaff [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily — SATO: <u>head</u> forward, <u>tail</u> backward
  - no update needed during backtracking
- <u>one</u> watch is enough to ensure correctness — but looses <u>arc consistency</u>
- reduces <u>visiting</u> 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]

two papers at

SAT'19

7 - 12 July

Lisbon, Portugal



Incremental Inprocessing in SAT Solving

Katalin Fazekas[1(✉)], Armin Biere[1], Christoph Scholl[2]

[1] Johannes Kepler University, Linz, Austria
katalin.fazekas@jku.at, armin.biere@jku.at
[2] Albert–Ludwigs–University, Freiburg, Germany
scholl@informatik.uni-freiburg.de

Abstract. Incremental SAT is about solving a sequence of related SAT problems efficiently. It makes use of already learned information to avoid repeating redundant work. Also preprocessing and inprocessing are considered to be crucial. Our calculus uses the most general redundancy property and extends existing inprocessing rules to incremental SAT solving. It allows to automatically reverse earlier simplification steps, which are inconsistent with literals in new incrementally added clauses. Our approach to incremental SAT solving not only simplifies the use of inprocessing but also substantially improves solving time.

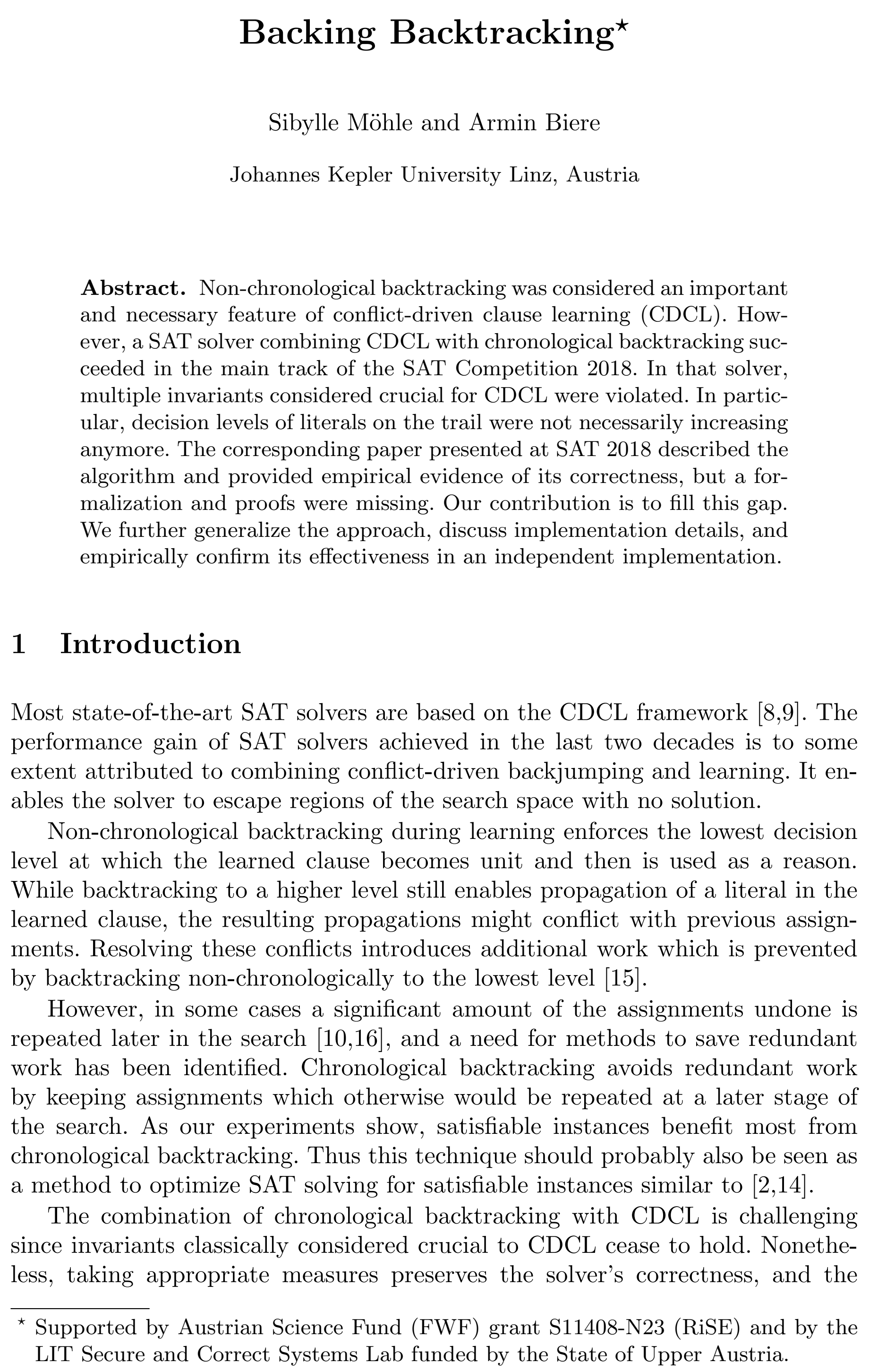
1 Introduction

Solving a sequence of related SAT problems incrementally [1–4] is crucial for the efficiency of SAT based model checking [5–8], and important in many domains [9–12]. Utilizing the effort already spent on a SAT problem by keeping learned information (such as variable scores and learned clauses) can significantly speed-up solving similar problems. Equally important are formula simplification techniques such as variable elimination, subsumption, self-subsuming resolution, and equivalence reasoning [13–16].

These simplifications are not only applied before the problem solving starts (*preprocessing*), but also periodically during the actual search (*inprocessing*) [17]. In this paper we focus on how to efficiently combine simplification techniques with incremental SAT solving.

Consider the SAT problem $F^0 = (a \vee b) \wedge (\neg a \vee \neg b)$. Both clauses are redundant and can be eliminated by for instance variable or blocked clause elimination [14, 16]. The resulting empty set of clauses is of course satisfiable and the SAT solver could for example simply just assign *false* to both variable as a solution. That is of course not a satisfying assignment of F^0, but can be transformed into one by solution reconstruction [17, 18], taking eliminated clauses into account. As we will see later, this would set the truth value of either a or b to true.

Now consider the SAT problem $F^1 = (a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg a) \wedge (\neg b)$ which is actually an extension of F^0 with the clauses $(\neg a)$ and $(\neg b)$. Simply adding them to our simplified F^0 (i.e. to the empty set of clauses) would result in a formula that again is satisfied by assigning false to each variable. However, using solution reconstruction on that assignment leads to the same solution as before, one that



Backing Backtracking*

Sibylle Möhle and Armin Biere

Johannes Kepler University Linz, Austria

Abstract. Non-chronological backtracking was considered an important and necessary feature of conflict-driven clause learning (CDCL). However, a SAT solver combining CDCL with chronological backtracking succeeded in the main track of the SAT Competition 2018. In that solver, multiple invariants considered crucial for CDCL were violated. In particular, decision levels of literals on the trail were not necessarily increasing anymore. The corresponding paper presented at SAT 2018 described the algorithm and provided empirical evidence of its correctness, but a formalization and proofs were missing. Our contribution is to fill this gap. We further generalize the approach, discuss implementation details, and empirically confirm its effectiveness in an independent implementation.

1 Introduction

Most state-of-the-art SAT solvers are based on the CDCL framework [8,9]. The performance gain of SAT solvers achieved in the last two decades is to some extent attributed to combining conflict-driven backjumping and learning. It enables the solver to escape regions of the search space with no solution.

Non-chronological backtracking during learning enforces the lowest decision level at which the learned clause becomes unit and then is used as a reason. While backtracking to a higher level still enables propagation of a literal in the learned clause, the resulting propagations might conflict with previous assignments. Resolving these conflicts introduces additional work which is prevented by backtracking non-chronologically to the lowest level [15].

However, in some cases a significant amount of the assignments undone is repeated later in the search [10,16], and a need for methods to save redundant work has been identified. Chronological backtracking avoids redundant work by keeping assignments which otherwise would be repeated at a later stage of the search. As our experiments show, satisfiable instances benefit most from chronological backtracking. Thus this technique should probably also be seen as a method to optimize SAT solving for satisfiable instances similar to [2,14].

The combination of chronological backtracking with CDCL is challenging since invariants classically considered crucial to CDCL cease to hold. Nonetheless, taking appropriate measures preserves the solver's correctness, and the

* Supported by Austrian Science Fund (FWF) grant S11408-N23 (RiSE) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.