SAT

Armin Biere

JOHANNES KEPLER
UNIVERSITÄT LINZ

CPAIOR’20 Master Class

Online

September 21, 2020

Dress Code of a Speaker at a Master Class as SAT Problem

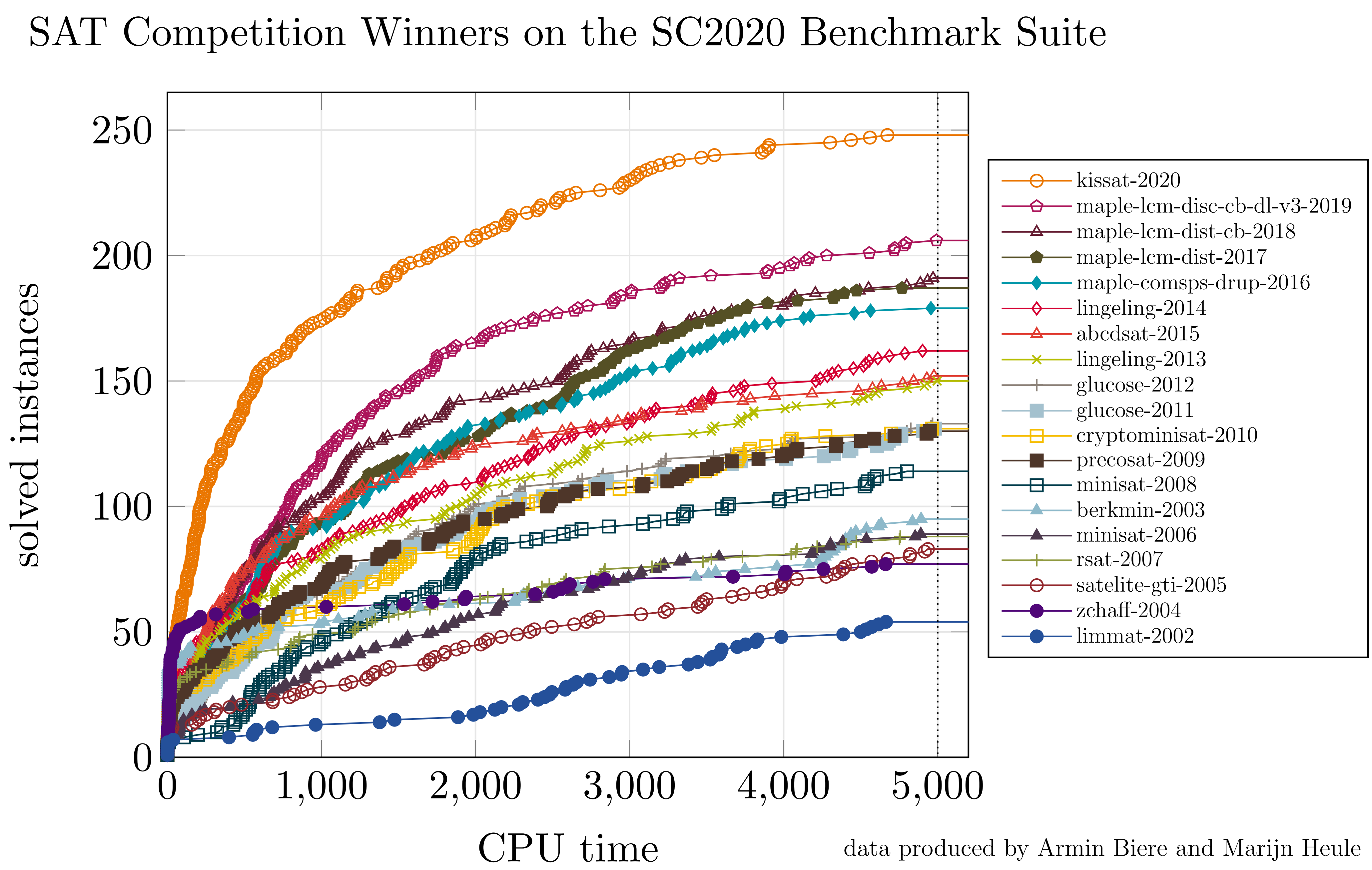
- propositional logic:
  - variables **tie** **shirt**
  - negation $\neg$ (not)
  - disjunction $\vee$ (or)
  - conjunction $\wedge$ (and)
- clauses (conditions / constraints)

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?**

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

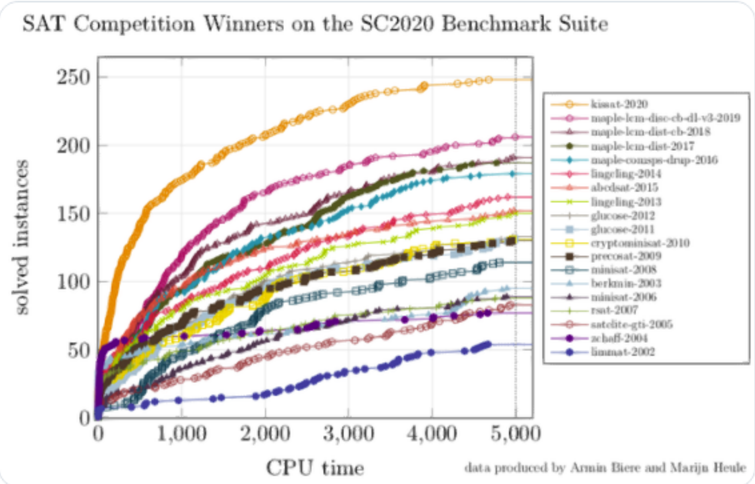

SAT-Race 2010 Award
"Plingeling"
by Armin Biere
is awarded the title of
Best Parallel Solver
Chair of SAT-Race Organizing Committee
SAT-Race 2010 Award
"Lingeling"
by Armin Biere
is awarded the title of
Second Prize Winn
Edinburgh, Scotland, July 14, 2010
Tenth International Conference
On Theory and Applications of Satisfiability Testing
SAT
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
SAT
tition 2009
ld medal
Eighteenth International Conference on Theory and Applications of Satisfiability Testing
SAT-Race 2015
Main Track
Special Prize: Most Innovative Solver
Awarded to:
Lingeling sr15bal
Written by:
Armin Biere
SAT-Race 201
Parallel Track
3rd Prize
Treengeling sr15bag
2nd Prize
Challenge 2014 · Configurable SAT Solver
SAT-Race 2010
Best Parallel Solver 2010
SAT COMPETITION 2013
Competition 2009
Silver medal
International Symposium
Delft
SAT Competition 2002
Winner
Cincinnati


SAT Competition Winners on the SC2020 Benchmark Suite
solved instances
CPU time
0
50
100
150
200
250
0
1,000
2,000
3,000
4,000
5,000
kissat-2020
maple-lcm-disc-cb-dl-v3-2019
maple-lcm-dist-cb-2018
maple-lcm-dist-2017
maple-comsps-drup-2016
lingeling-2014
abcdsat-2015
lingeling-2013
glucose-2012
glucose-2011
cryptominisat-2010
precosat-2009
minisat-2008
berkmin-2003
minisat-2006
rsat-2007
satelite-gti-2005
zchaff-2004
limmat-2002
data produced by Armin Biere and Marijn Heule

some recent **Tweets**

Armin Biere @ArminBiere

SAT solvers get faster and faster: all-time winners of the SAT Competition on 2020 instances, featuring our new solver Kissat (fmv.jku.at/kissat), which won in 2020. The web page also has runtime CDFs for 2011 and 2019.



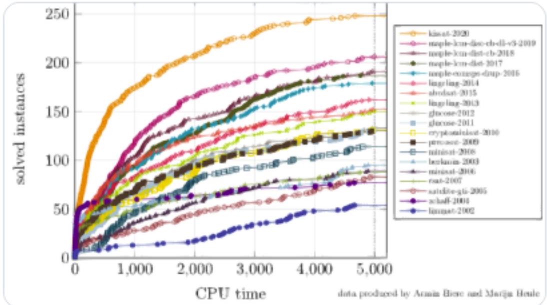
5:20 PM · Jul 28, 2020 · Twitter Web App

View Tweet activity

57 Retweets **7** Quote Tweets **327** Likes

Armin Biere @ArminBiere · Jul 28

SAT solvers get faster and faster: all-time winners of the SAT Competition on 2020 instances, featuring our new solver Kissat (fmv.jku.at/kissat), which won in 2020. The web page also has runtime CDFs for 2011 and 2019.



5 64 327

joao @_joaogui1 · Jul 29

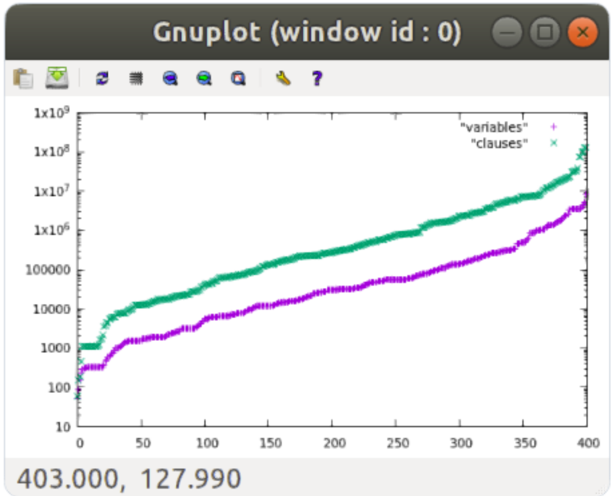
How big are the instances?

1 1

Armin Biere @ArminBiere

Replying to @_joaogui1

The largest ones have millions of variables and clauses. The planning track had even larger ones. See the variable and clause distribution plot for the main track:



Armin Biere @ArminBiere

Eventually I will need to support 64-bit variable indices (Lingeling has 2^27, CaDiCaL indeed 2^31 and Kissat 2^28 as compromise though it could easily do half a billion)

T-Mobile A 89 % 21:12

Hi,
We are trying to verify Deep Neural Networks with our verification machine ESBMC, that uses Boolector. Our experiments are geting the following error:

- internal error in 'lglib.c': more than 134217724 variables.
  Could we increase this variable number? Since we are performing our experiments in a huge RAM memory.

—
You are receiving this because you are subscribed to this thread.
Reply to this email directly, view it on GitHub, or unsubscribe.

Andrew V. Jones 13:40
an Boolector/boolector, S...

Can you try compiling Boolector with a different SAT solver? I believe that CaDiCaL has a much higher limit (maybe INT_MAX vars).

Zitierten Text anzeigen

Aina Niemetz 18:16
an Boolector/boolector, S...

As @andrewvaughanj points out, this is a limitation in the SAT solver that we can not control. Let me add that CaDiCaL typically outperforms Lingeling in combination with Boolector, so it might be a good idea to switch to CaDiCaL anyways.

Satisfiability (SAT) related topics have attracted researchers from various disciplines. Logic, applied areas such as planning, scheduling, operations research and combinatorial optimization, but also theoretical issues on the theme of complexity, and much more, they all are connected through SAT.

My personal interest in SAT stems from actual solving: The increase in power of modern SAT solvers over the past 15 years has been phenomenal. It has become the key enabling technology in automated verification of both computer hardware and software. Bounded Model Checking (BMC) of computer hardware is now probably the most widely used model checking technique. The counterexamples that it finds are just satisfying instances of a Boolean formula obtained by unwinding to some fixed depth a sequential circuit and its specification in linear temporal logic. Extending model checking to software verification is a much more difficult problem on the frontier of current research. One promising approach for languages like C with finite word-length integers is to use the same idea as in BMC but with a decision procedure for the theory of bit-vectors instead of SAT. All decision procedures for bit-vectors that I am familiar with ultimately make use of a fast SAT solver to handle complex formulas.

Decision procedures for more complicated theories, like linear real and integer arithmetic, are also used in program verification. Most of them use powerful SAT solvers in an essential way.

Clearly, efficient SAT solving is a key technology for 21st century computer science. I expect this collection of papers on all theoretical and practical aspects of SAT solving will be extremely useful to both students and researchers and will lead to many further advances in the field.

Edmund Clarke

Edmund M. Clarke, FORE Systems University Professor of Computer Science and Professor of Electrical and Computer Engineering at Carnegie Mellon University, is one of the initiators and main contributors to the field of Model Checking, for which he also received the 2007 ACM Turing Award.

In the late 90s Professor Clarke was one of the first researchers to realize that SAT solving has the potential to become one of the most important technologies in model checking.

ISBN 978-1-58603-929-5
ISSN 0922-6389

185

HANDBOOK

of satisfiability

Editors:
Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

IOS Press

Frontiers in Artificial Intelligence and Applications

HANDBOOK

of satisfiability

Editors:
Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

IOS Press

Part I. Theory and Algorithms

John Franco, John Martin:
A History of Satisfiability. 3-74

Steven David Prestwich:
CNF Encodings. 75-97

Adnan Darwiche, Knot Pipatsrisawat:
Complete Algorithms. 99-130

João P. Marques Silva, Inês Lynce, Sharad Malik:
Conflict-Driven Clause Learning SAT Solvers. 131-153

Marijn Heule, Hans van Maaren:
Look-Ahead Based SAT Solvers. 155-184

Henry A. Kautz, Ashish Sabharwal, Bart Selman:
Incomplete Algorithms. 185-203

Oliver Kullmann:
Fundaments of Branching Heuristics. 205-244

Dimitris Achlioptas:
Random Satisfiability. 245-270

Carla P. Gomes, Ashish Sabharwal:
Exploiting Runtime Variation in Complete Solvers. 271-288

Karem A. Sakallah:
Symmetry and Satisfiability. 289-338

Hans Kleine Büning, Oliver Kullmann:
Minimal Unsatisfiability and Autarkies. 339-401

Evgeny Dantsin, Edward A. Hirsch:
Worst-Case Upper Bounds. 403-424

Marko Samer, Stefan Szeider:
Fixed-Parameter Tractability. 425-454

Part II. Applications and Extensions

Armin Biere:
Bounded Model Checking. 457-481

Jussi Rintanen:
Planning and SAT. 483-504

Daniel Kroening:
Software Verification. 505-532

Hantao Zhang:
Combinatorial Designs by SAT Solvers. 533-568

Fabrizio Altarelli, Rémi Monasson, Guilhem Semerjian, Francesco Zamponi:
Connections to Statistical Physics. 569-611

Chu Min Li, Felip Manyà:
MaxSAT, Hard and Soft Constraints. 613-631

Carla P. Gomes, Ashish Sabharwal, Bart Selman:
Model Counting. 633-654

Rolf Drechsler, Tommi A. Junttila, Ilkka Niemelä:
Non-Clausal SAT and ATPG. 655-693

Olivier Roussel, Vasco M. Manquinho:
Pseudo-Boolean and Cardinality Constraints. 695-733

Hans Kleine Büning, Uwe Bubeck:
Theory of Quantified Boolean Formulas. 735-760

Enrico Giunchiglia, Paolo Marin, Massimo Narizzano:
Reasoning with Quantified Boolean Formulas. 761-780

Roberto Sebastiani, Armando Tacchella:
SAT Techniques for Modal and Description Logics. 781-824

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, Cesare Tinelli:
Satisfiability Modulo Theories. 825-885

Stephen M. Majercik:
Stochastic Boolean Satisfiability. 887-925


P

NEWLY AVAILABLE SECTION OF THE CLASSIC WORK

The Art of Computer Programming

VOLUME 4

Satisfiability

FASCICLE 6

Buchdeckel

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.

* * *

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.)

I've tried to ameliorate the reader's navigation problem by adding subheadings at the top of each right-hand page. Furthermore, as in other sections, the exercises appear in an order that roughly parallels the order in which corresponding topics are taken up in the text. Numerous cross-references are provided

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

SAT Handbook upcoming 2nd Edition

editors Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh

*with many updated chapters and the **following 7 new chapters**:*

Proof Complexity   Jakob Nordström and Sam Buss

Preprocessing   Armin Biere, Matti Järvisalo and Benjamin Kiesl

Tuning and Configuration

Holger Hoos, Frank Hutter and Kevin Leyton-Brown

Proofs of Unsatisfiability   Marijn Heule

Core-Based MaxSAT

Fahiem Bacchus, Matti Järvisalo and Ruben Martins

Proof Systems for Quantified Boolean Formulas

Olaf Beyersdorff, Mikoláš Janota, Florian Lonsing and Martina Seidl

Approximate Model Counting   Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi

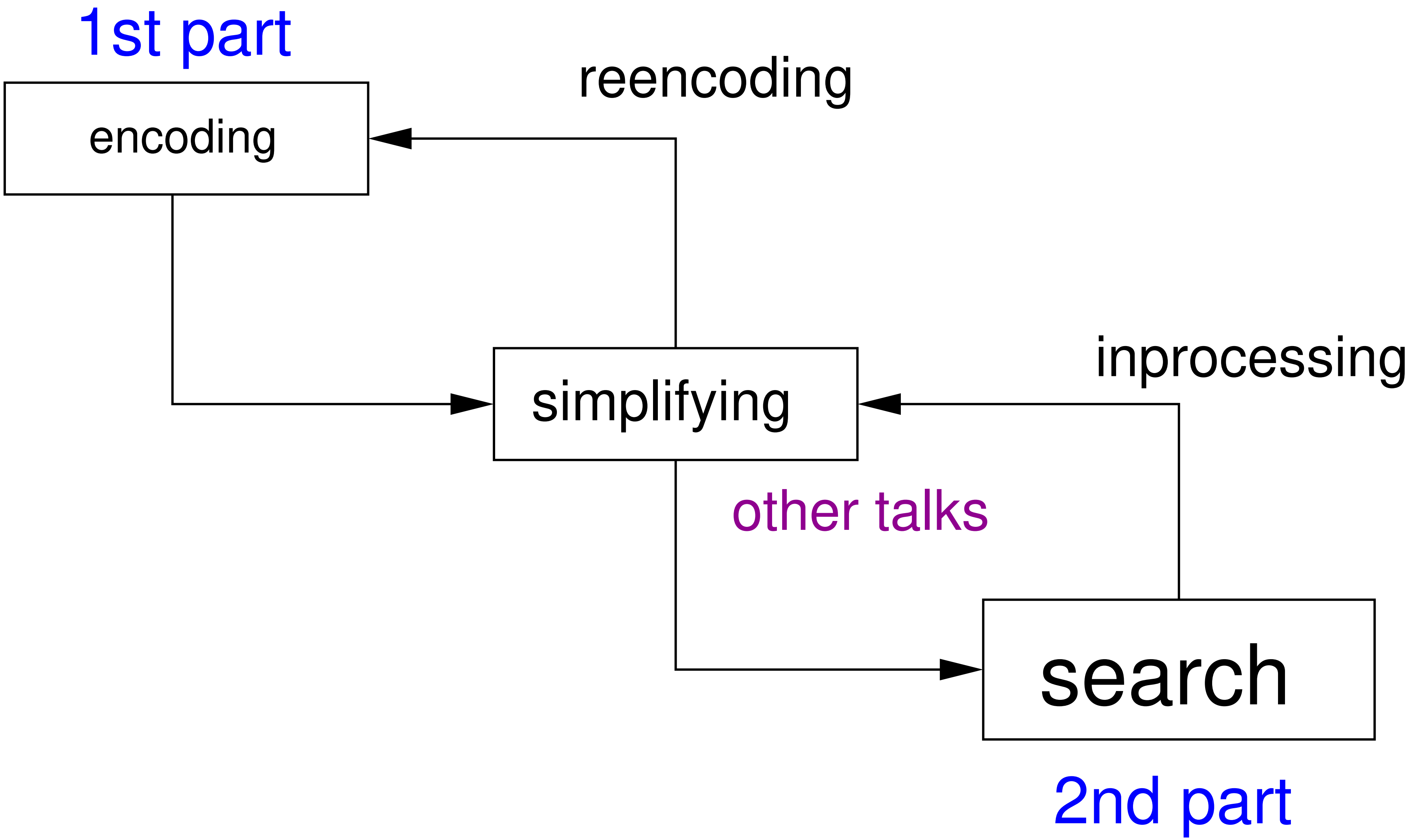
The SAT problem is evidently a killer app, because it is key to the solution of so many other problems. SAT-solving techniques are among computer science's best success stories so far, and these volumes tell that fascinating tale in the words of the leading SAT experts.

Donald Knuth

. . . Clearly, efficient SAT solving is a key technology for 21st century computer science. I expect this collection of papers on all theoretical and practical aspects of SAT solving will be extremely useful to both students and researchers and will lead to many further advances in the field.

Edmund Clarke

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

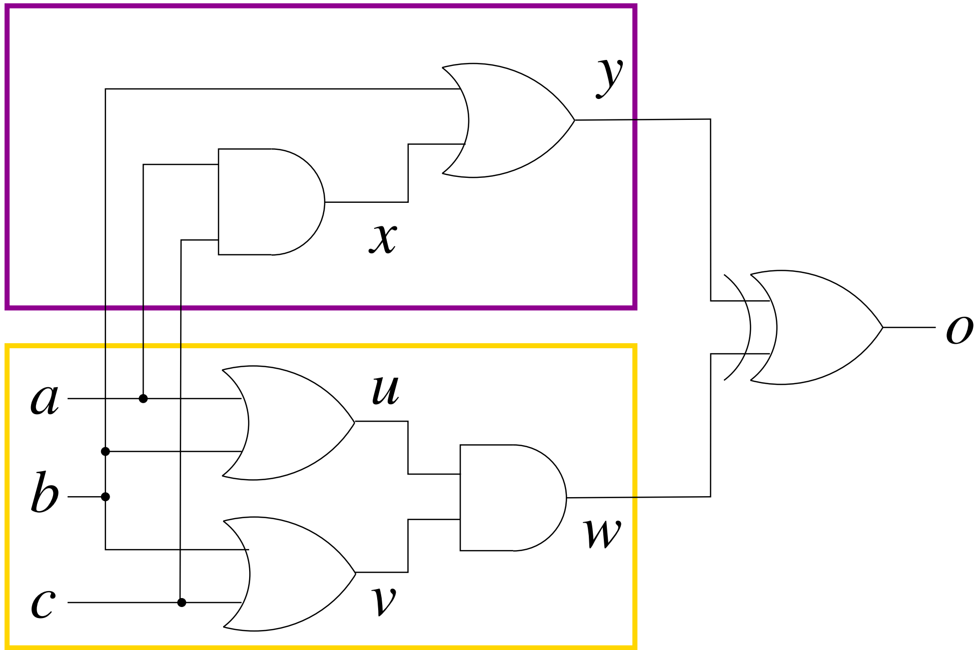
$$
\begin{aligned}
\mathit{original} &\equiv \textbf{if } \neg a \wedge \neg b \textbf{ then } h \textbf{ else } \textbf{ if } \neg a \textbf{ then } g \textbf{ else } f \\
&\equiv (\neg a \wedge \neg b) \wedge h \;\vee\; \neg(\neg a \wedge \neg b) \wedge \textbf{ if } \neg a \textbf{ then } g \textbf{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

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

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

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

$$
\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: $\qquad s = x + y$

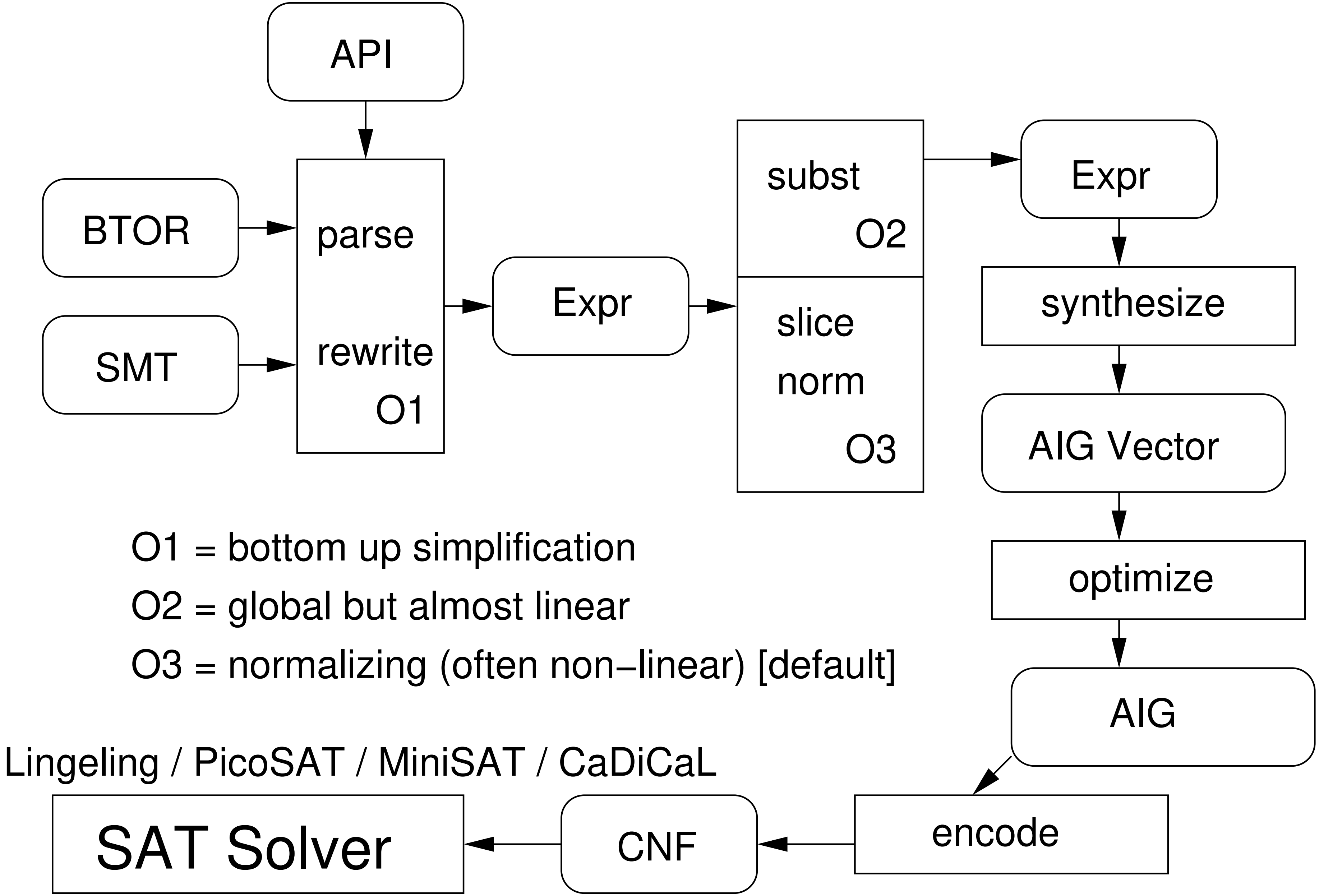
$$[s_3, s_2, s_1, s_0]_4 \;=\; [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

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

where

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

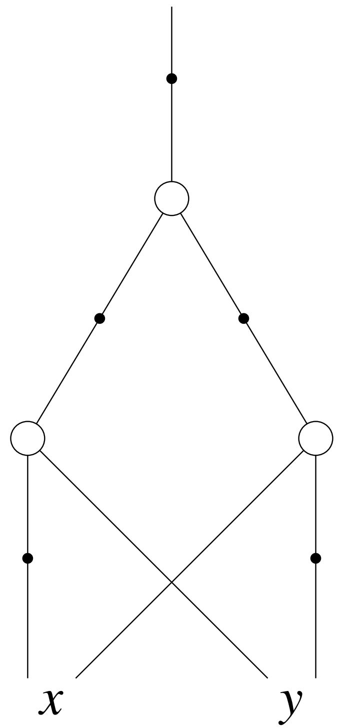
Boolector Architecture



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

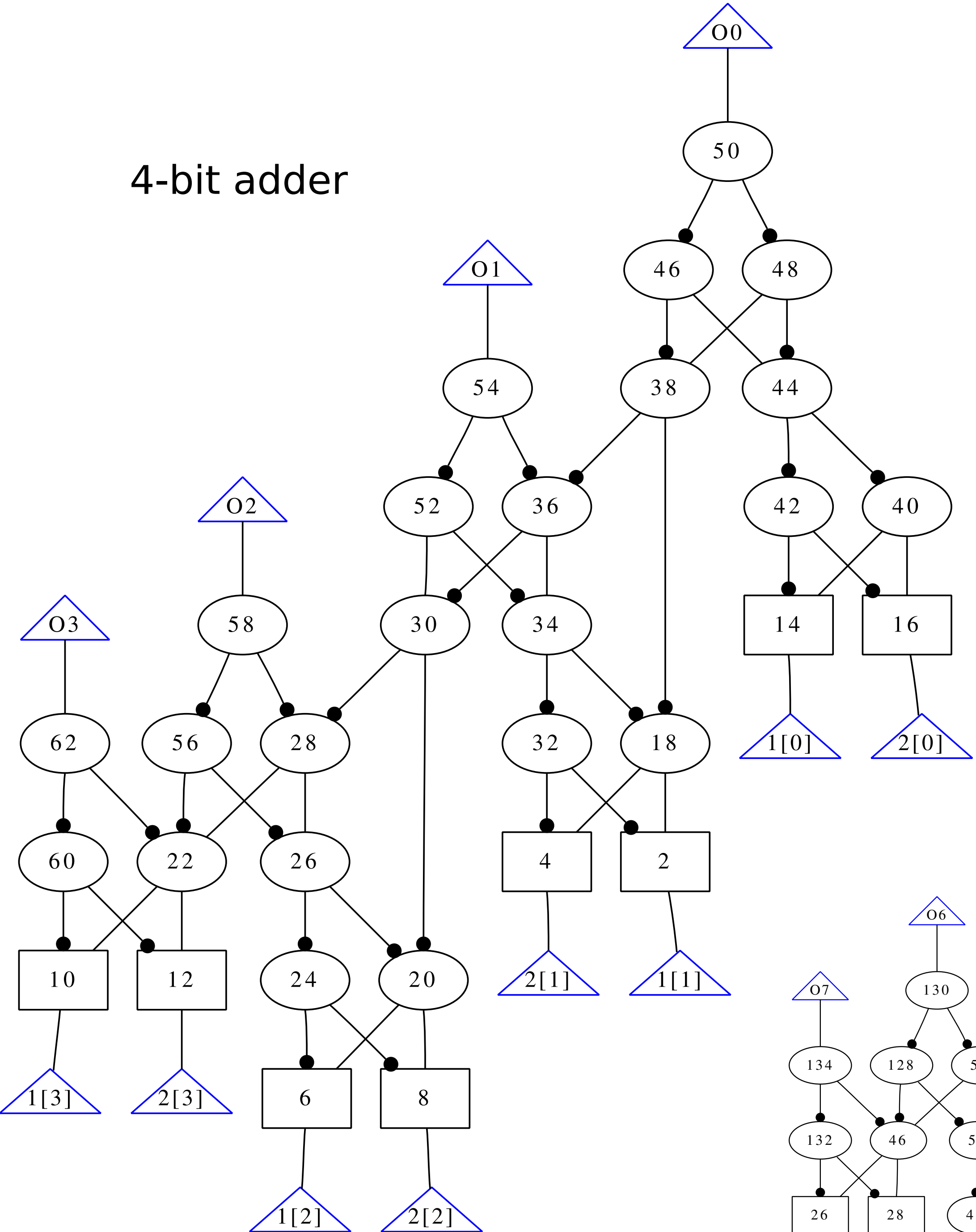


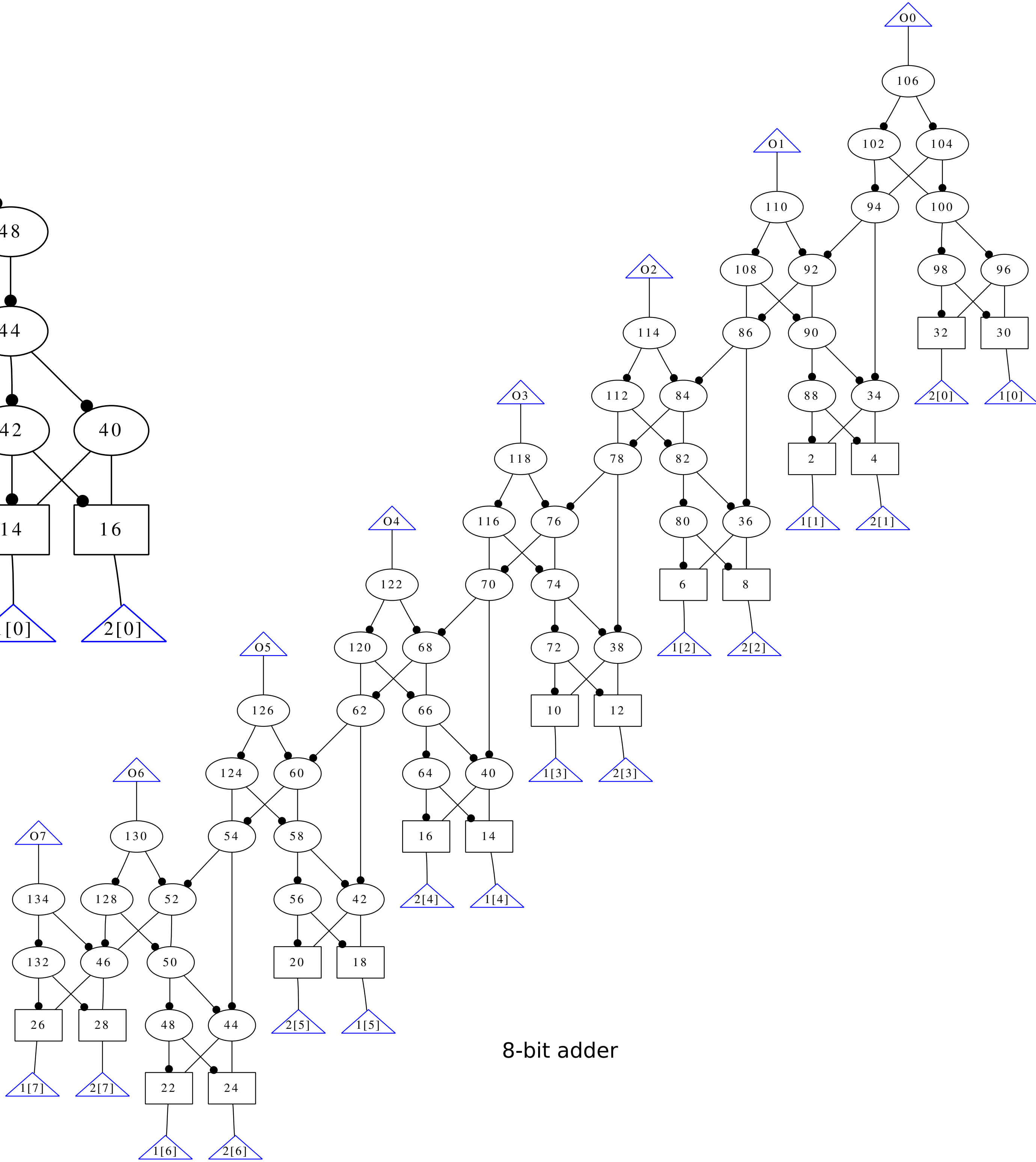
negation/sign are edge attributes

not part of node

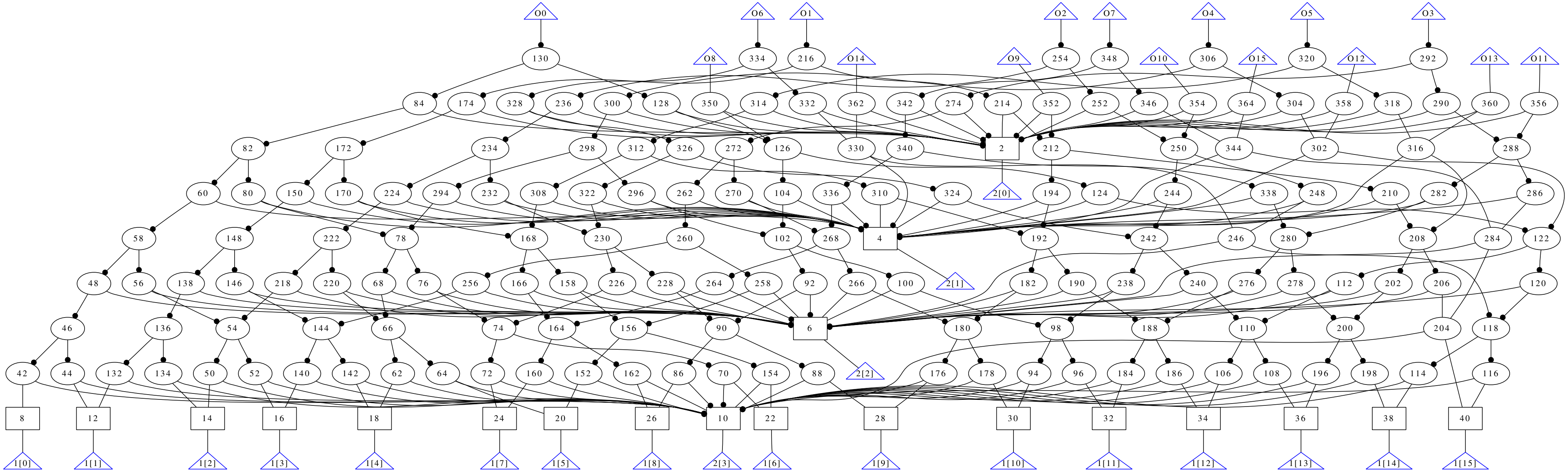
$$x \text{ xor } y \;\equiv\; (\overline{x} \wedge y) \vee (x \wedge \overline{y}) \;\equiv\; \overline{\overline{(\overline{x} \wedge y)} \wedge \overline{(x \wedge \overline{y})}}$$

4-bit adder

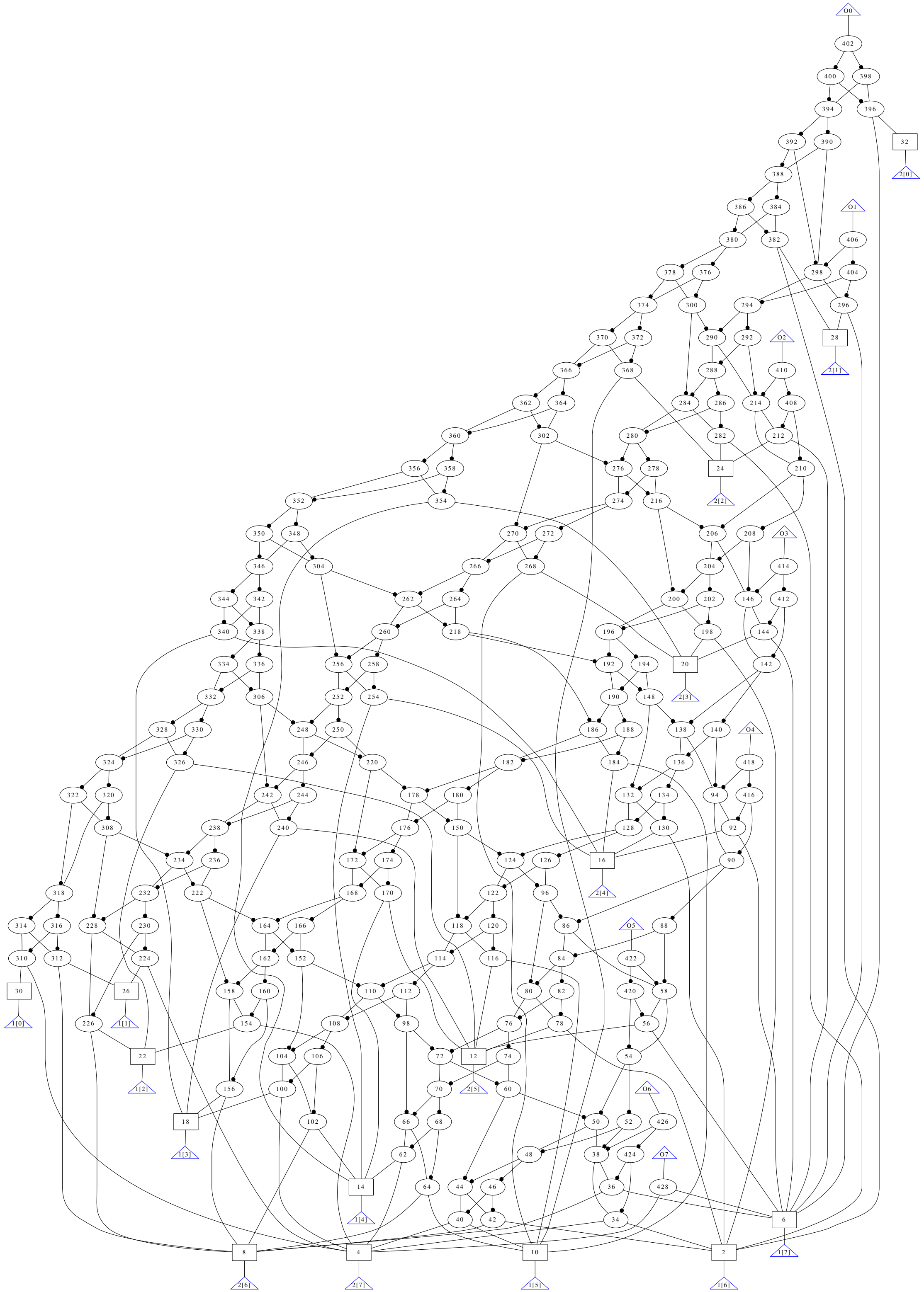




8-bit adder



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

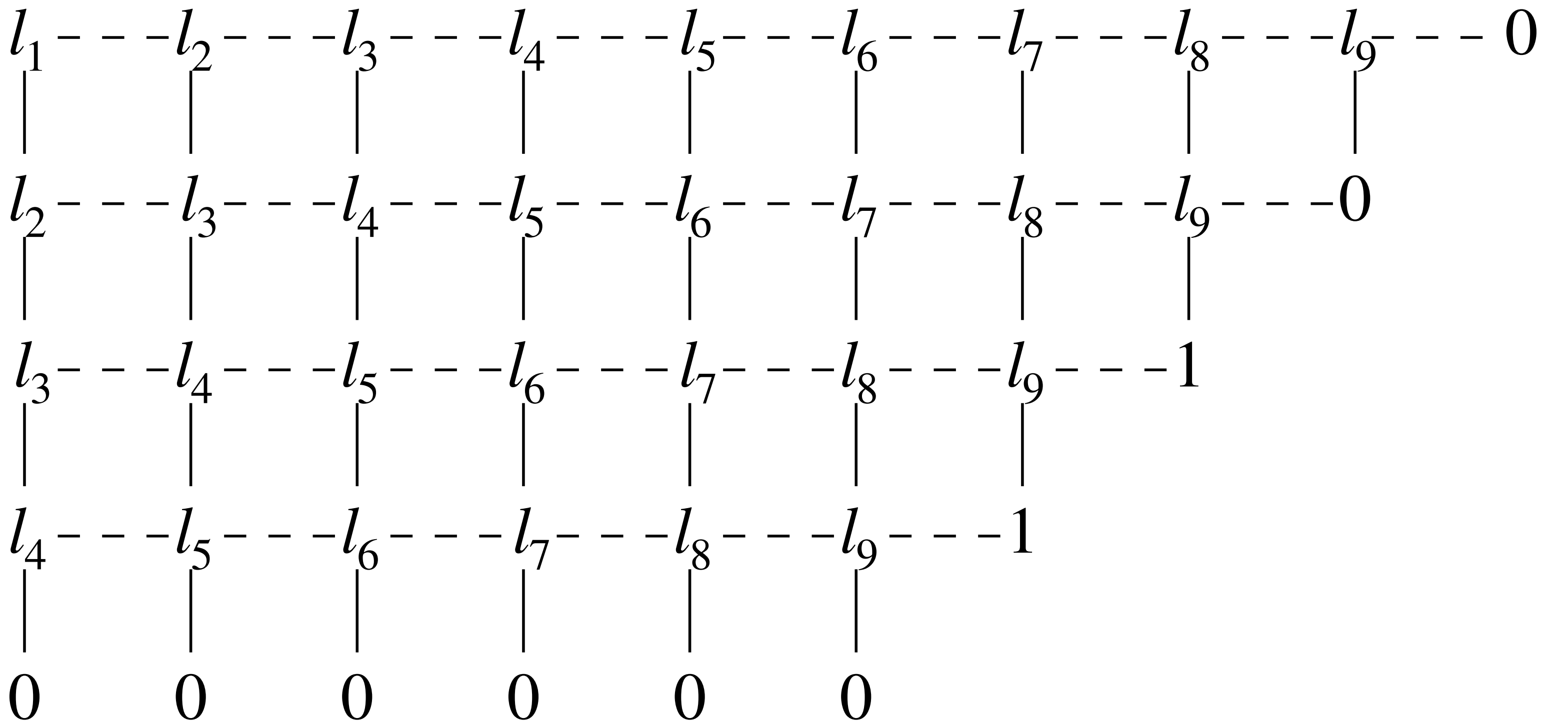
- given a set of literals $\{l_1, \dots l_n\}$
  - constraint the number of literals assigned to *true*
  - $l_1 + \cdots + l_n \geq k$ or $l_1 + \cdots + l_n \leq k$ or $l_1 + \cdots + l_n = k$
  - combined make up exactly all fully symmetric boolean functions
- multiple encodings of cardinality constraints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

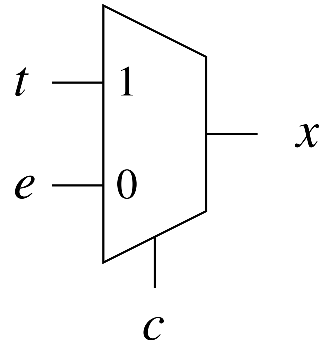
BDD-Based Encoding of Cardinality Constraints

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



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

Tseitin Encoding of If-Then-Else Gate



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

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

minimal but not arc consistent:

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

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

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

DIMACS Format

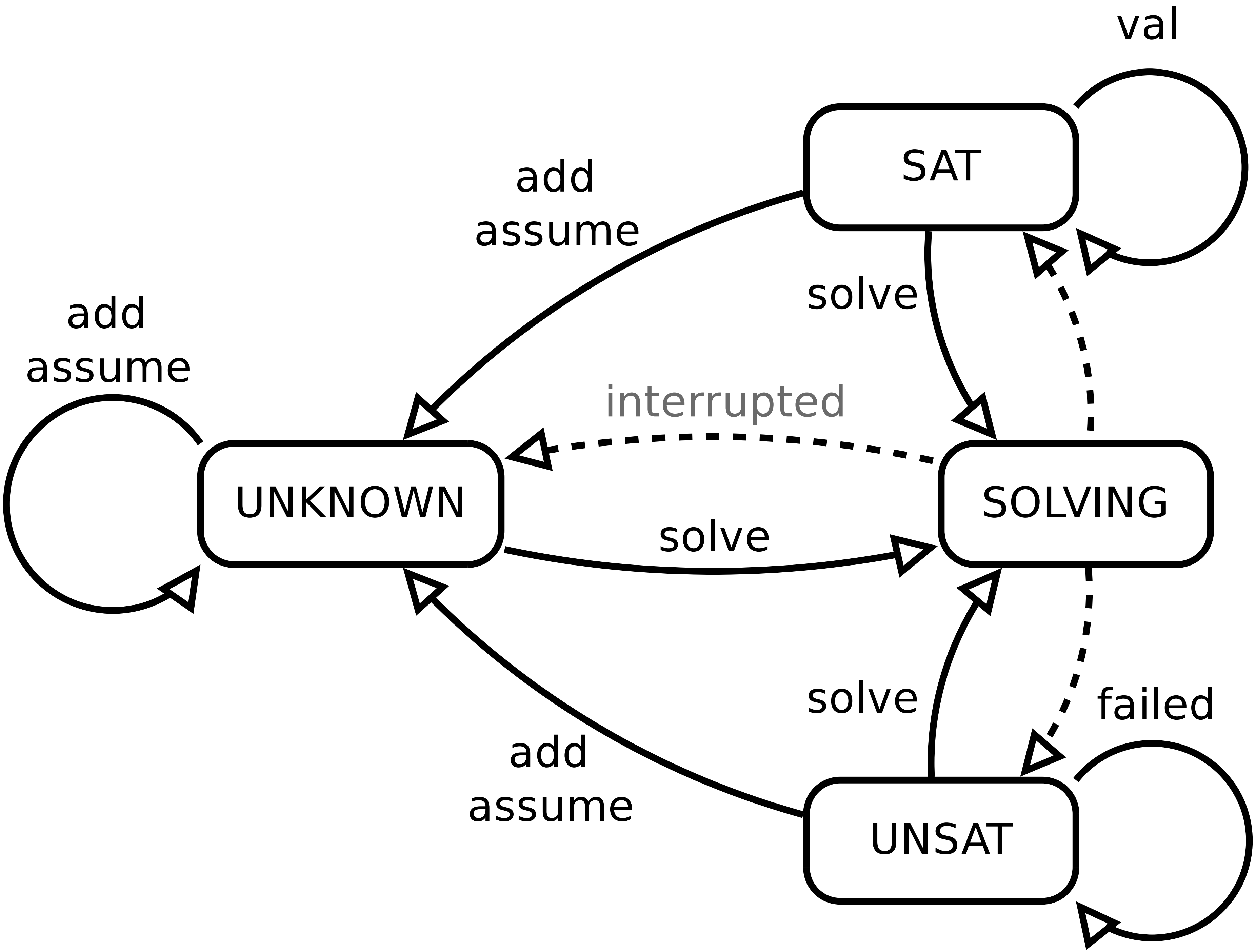
```
$ cat example.cnf
c comments start with 'c' and extend until the end of the line
c
c variables are encoded as integers:
c
c   'tie'   becomes '1'
c   'shirt' becomes '2'
c
c header 'p cnf <variables> <clauses>'
c
p cnf 2 3
-1  2 0           c  !tie  or  shirt
 1  2 0           c   tie  or  shirt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

IPASIR Functions

```c
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));
```

```cpp
#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
```

DP / DPLL

- dates back to the 50'ies:

  1st version DP is resolution based $\Rightarrow$ 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

DP Procedure

forever

if $F = \top$ **return** <u>satisfiable</u>

if $\bot \in F$ **return** <u>unsatisfiable</u>

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

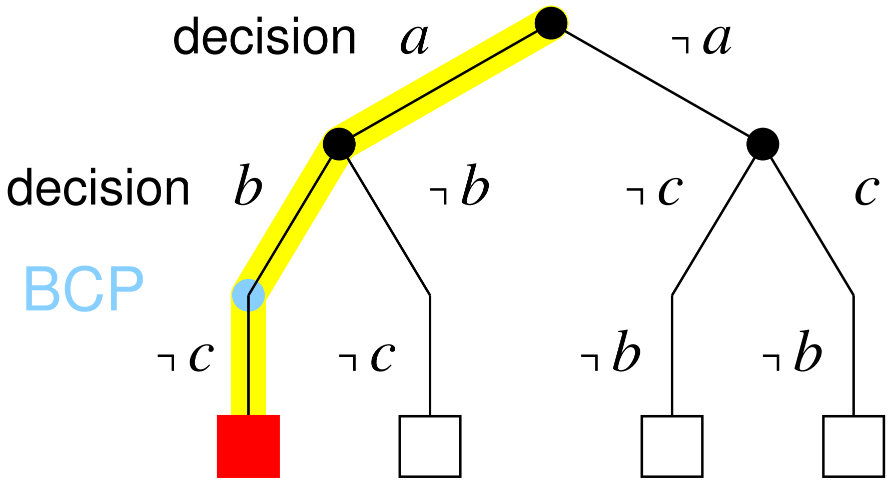
$\Rightarrow$ $\boxed{\text{CDCL}}$

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]

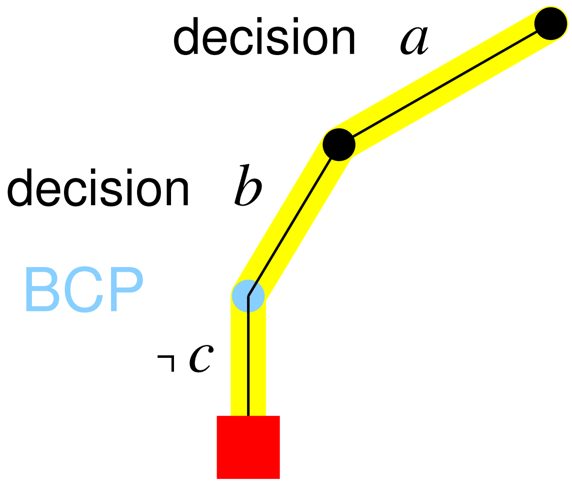
- first implemented in the context of GRASP SAT solver
  - name given later to distinguish it from DPLL
  - not recursive anymore
- essential for SMT
- learning clauses as no-goods
- notion of implication graph
- (first) unique implication points

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 $\neg a \vee \neg b$

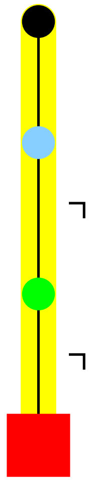
Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$

decision a



$\neg b$ BCP

$\neg c$ BCP

clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

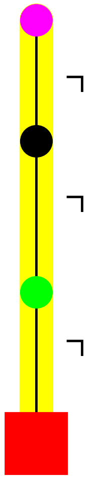
learn $\neg a$

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

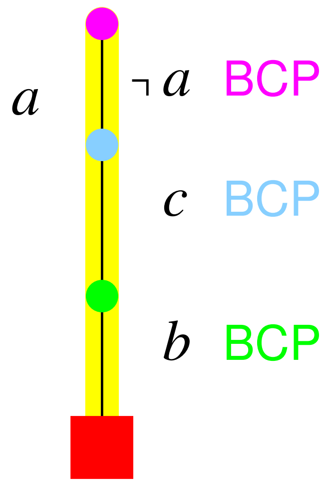
learn c

Conflict Driven Clause Learning (CDCL)

$a = 1$

$b = 0$

$c = 0$



clauses

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

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

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

$\neg a \vee b \vee c$

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

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

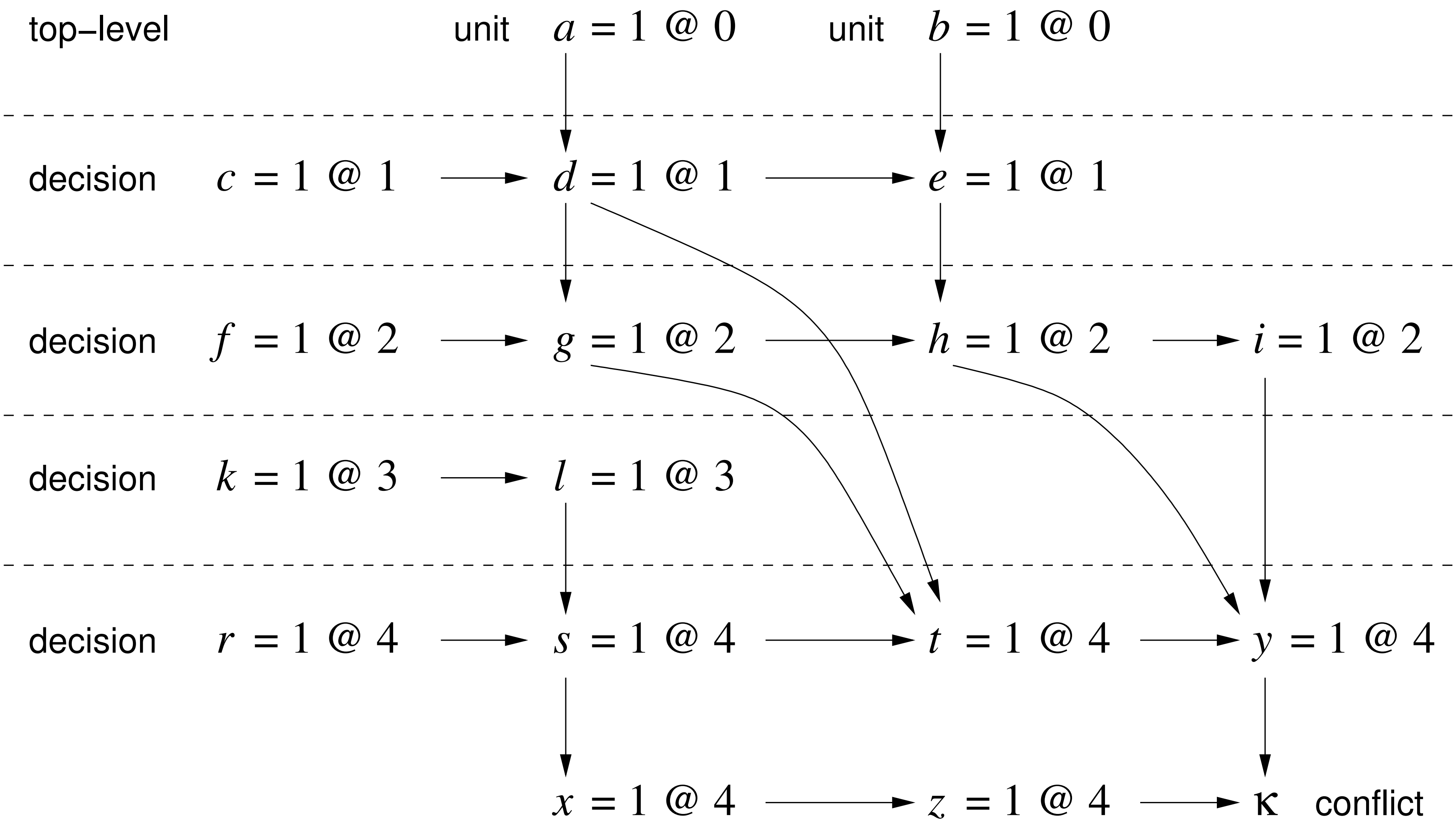
$\neg a$

c

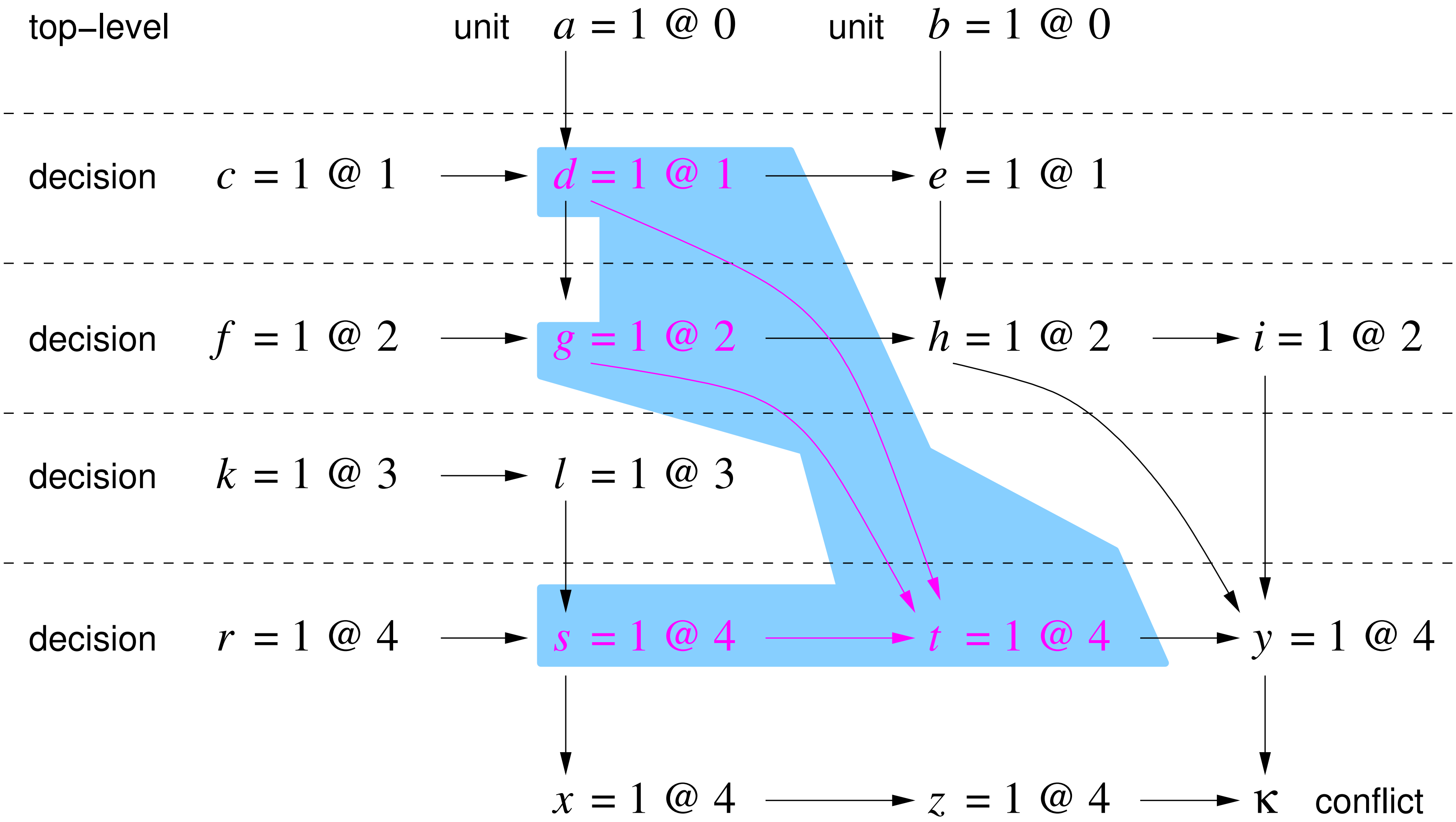
learn $\perp$

empty clause

Implication Graph

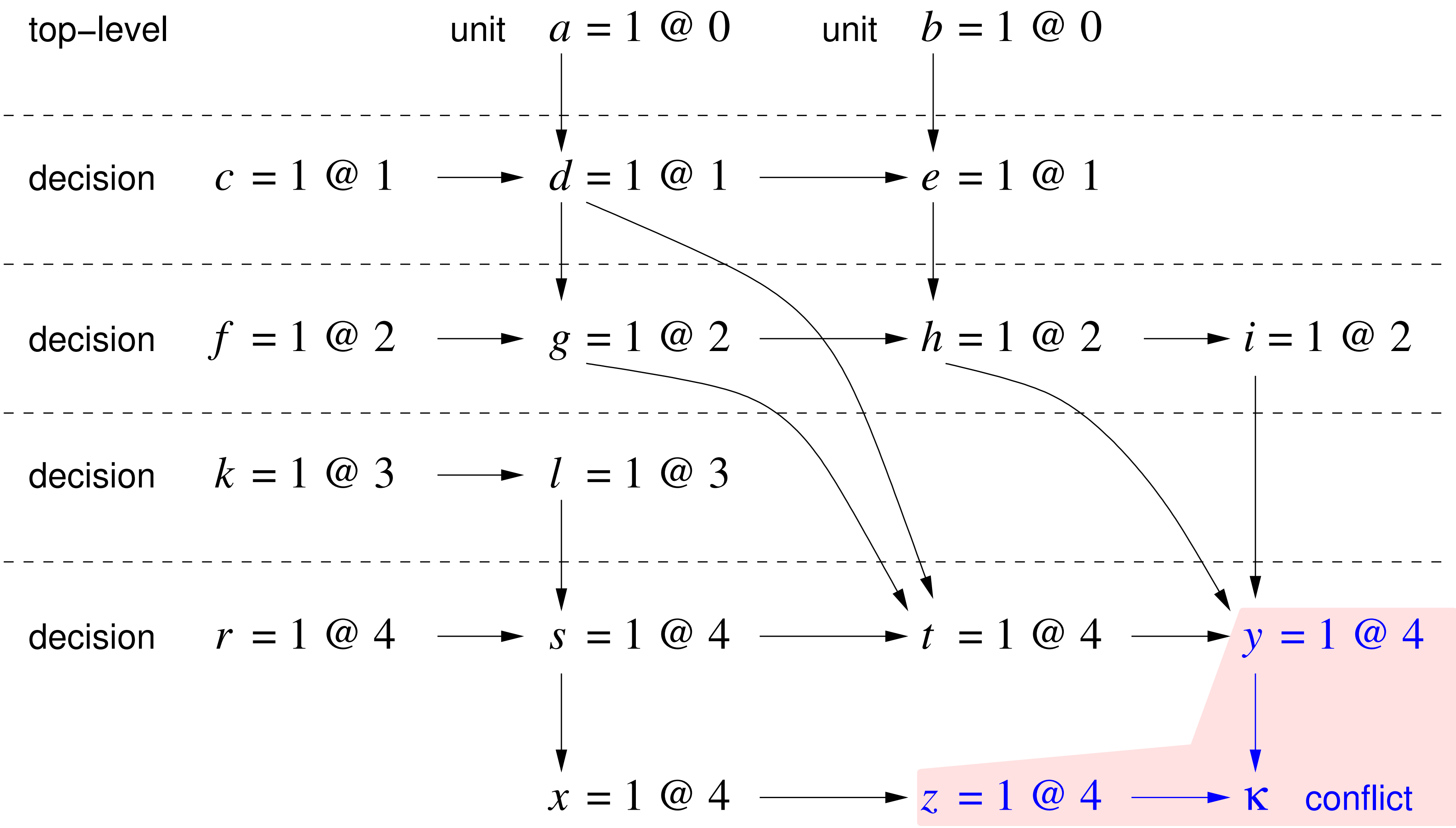


Antecedents / Reasons



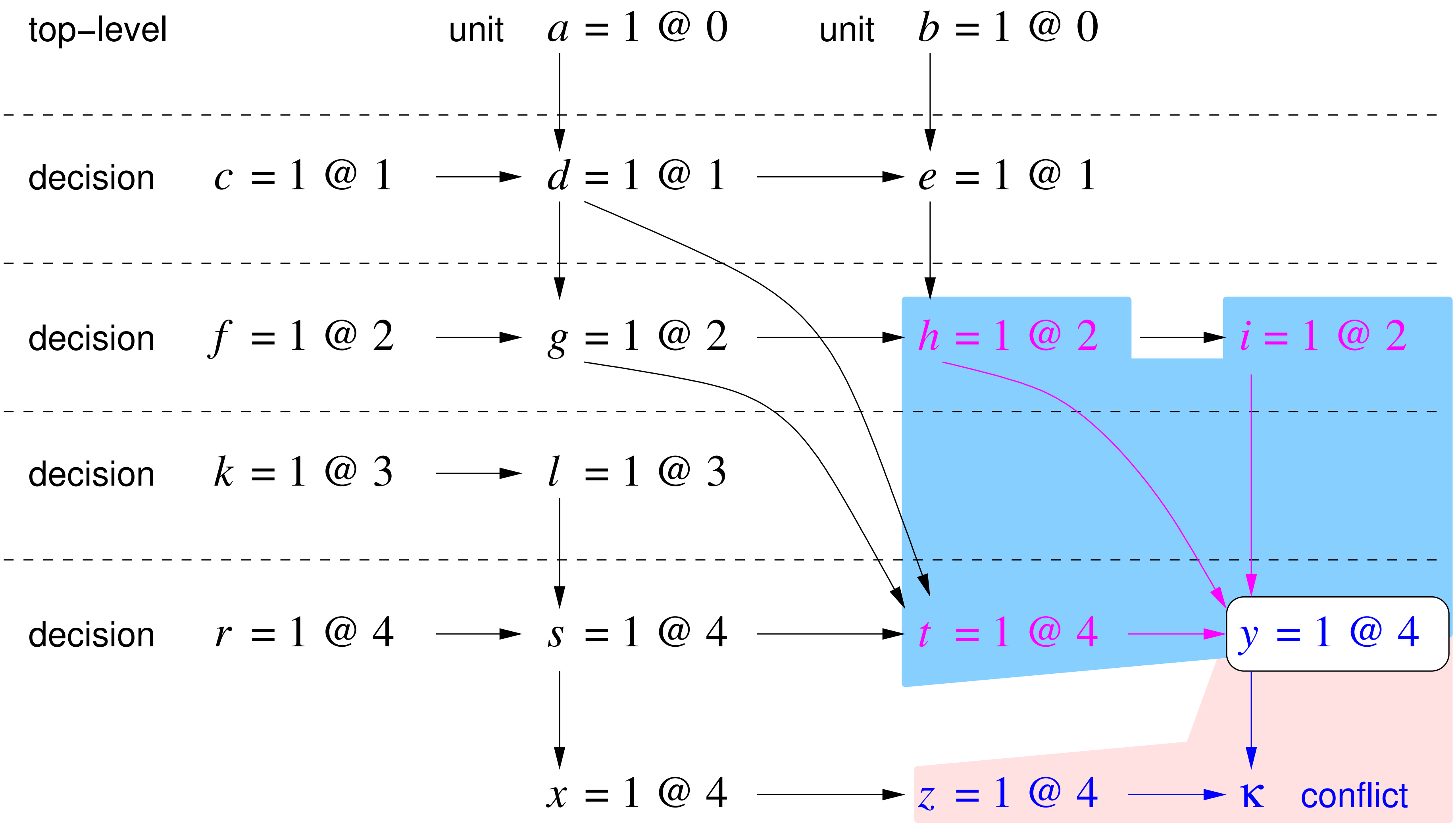
$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee t)$$

Conflicting Clauses



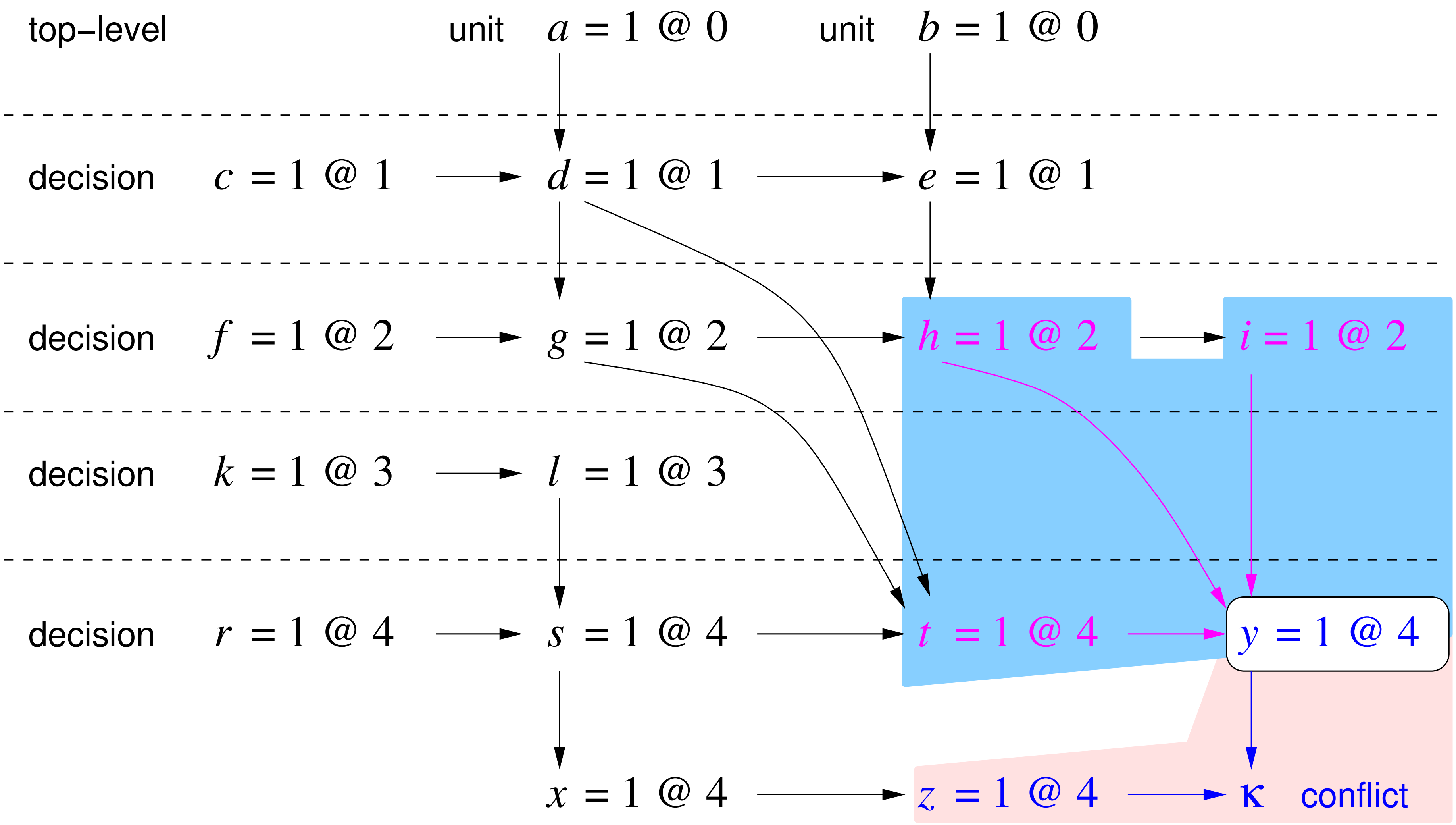
$$\neg(y \wedge z) \quad \equiv \quad (\bar{y} \vee \bar{z})$$

Resolving Antecedents 1st Time



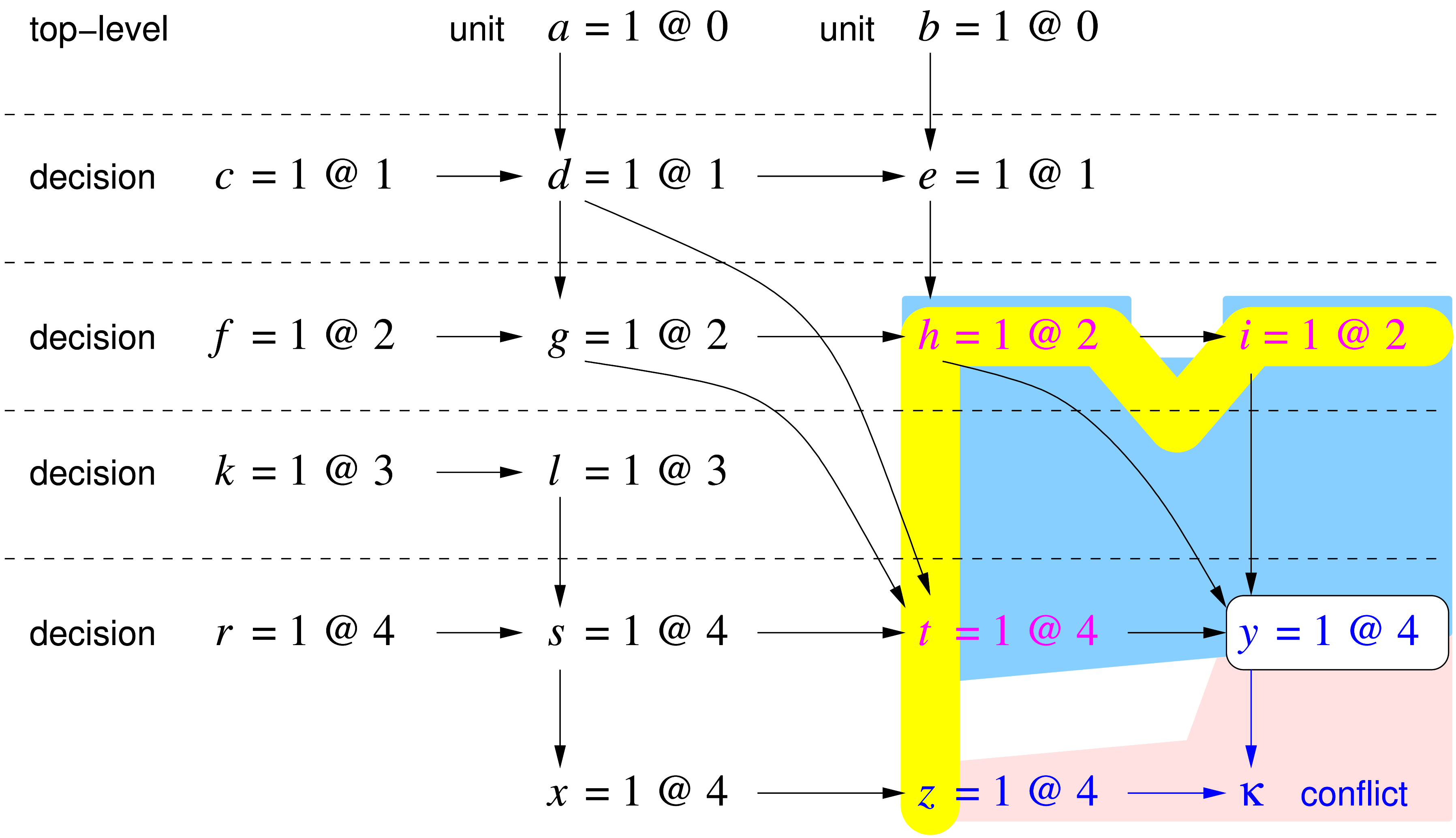
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

Resolving Antecedents 1st Time



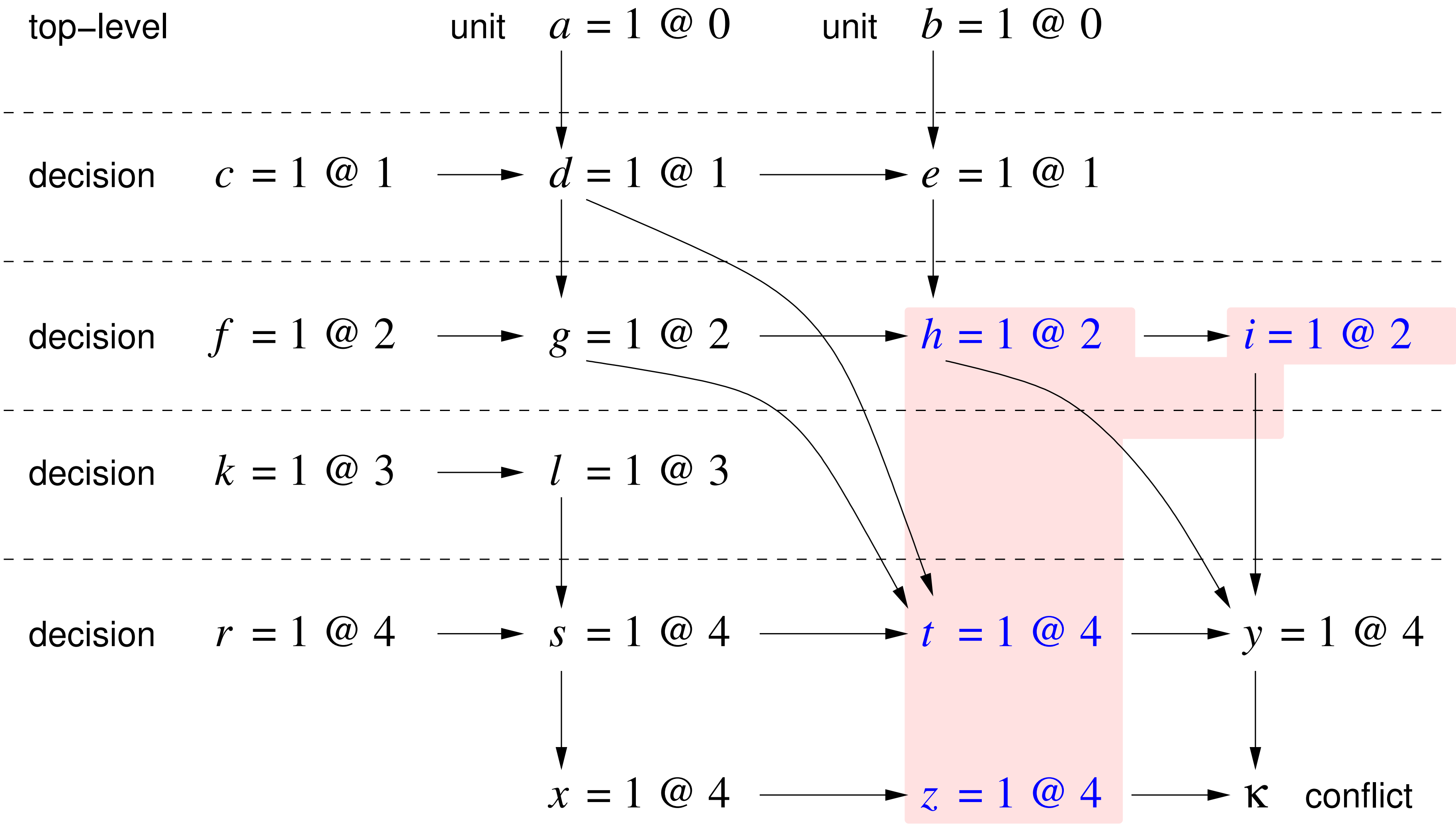
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{\boxed{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}}$$

Resolvents = Cuts = Potential Learned Clauses



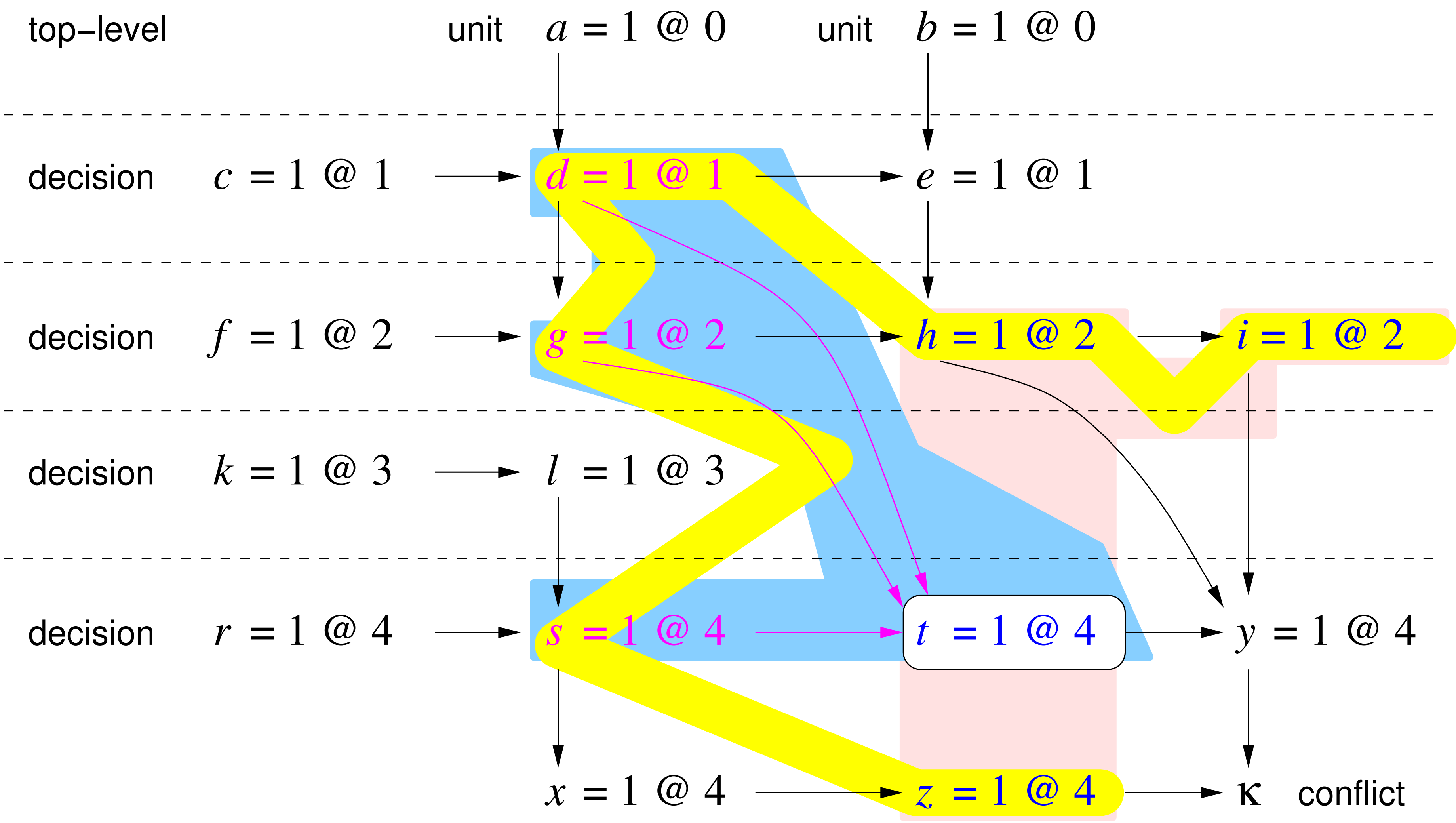
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}$$

Potential Learned Clause After 1 Resolution



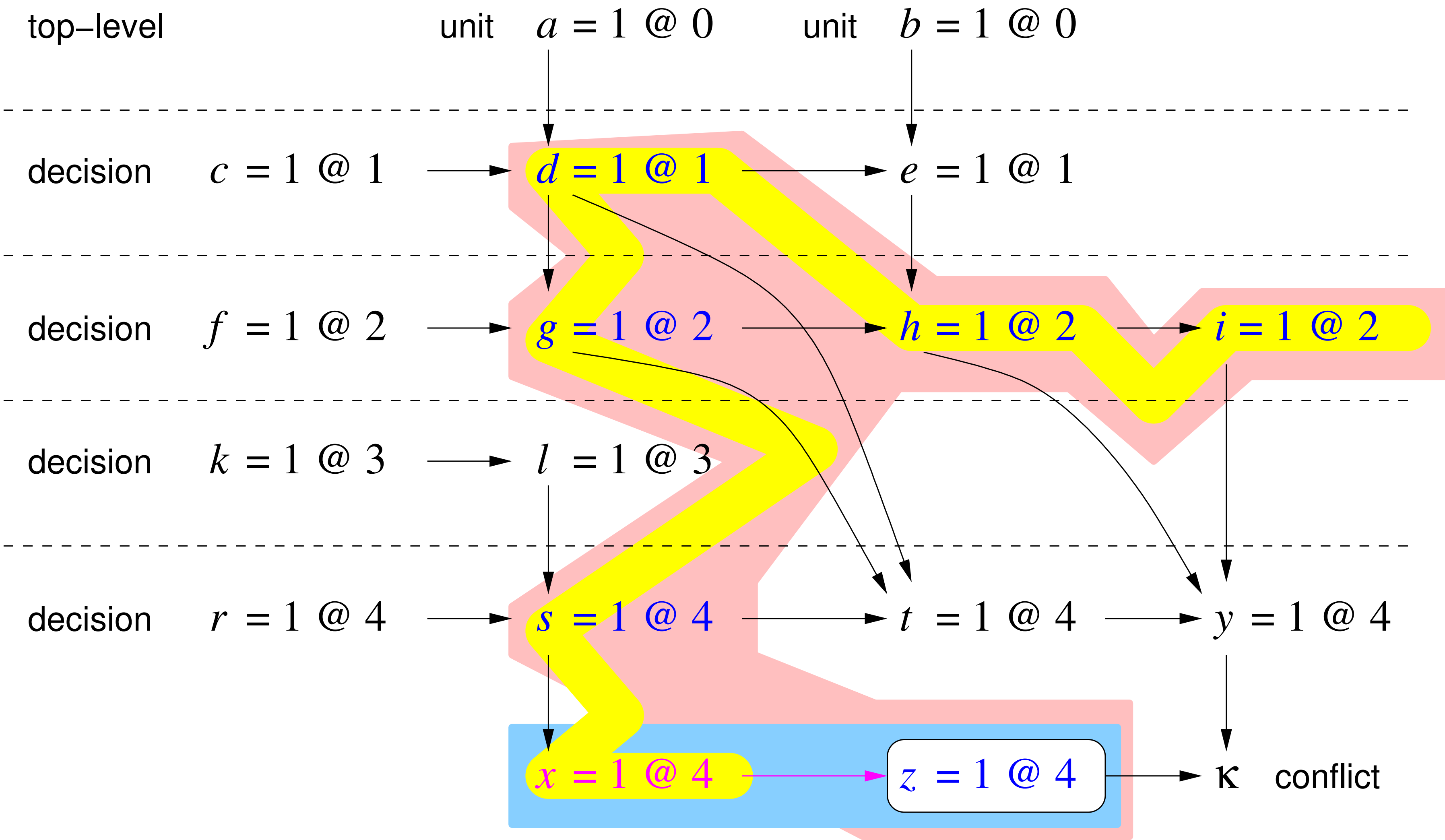
$$(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})$$

Resolving Antecedents 2nd Time



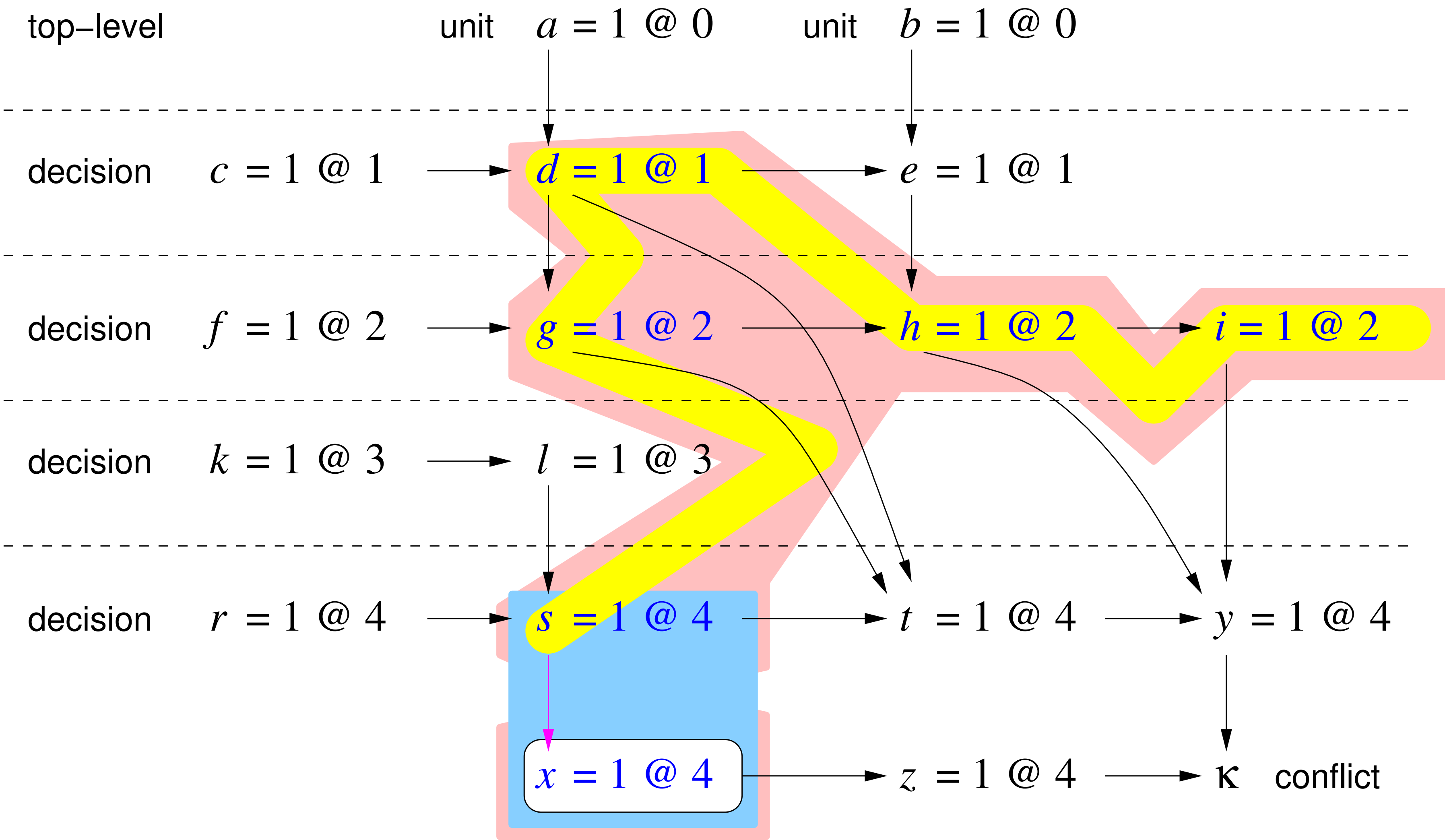
$$\frac{(\bar{d} \vee \bar{g} \vee \bar{s} \vee t) \qquad (\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}$$

Resolving Antecedents 3rd Time



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

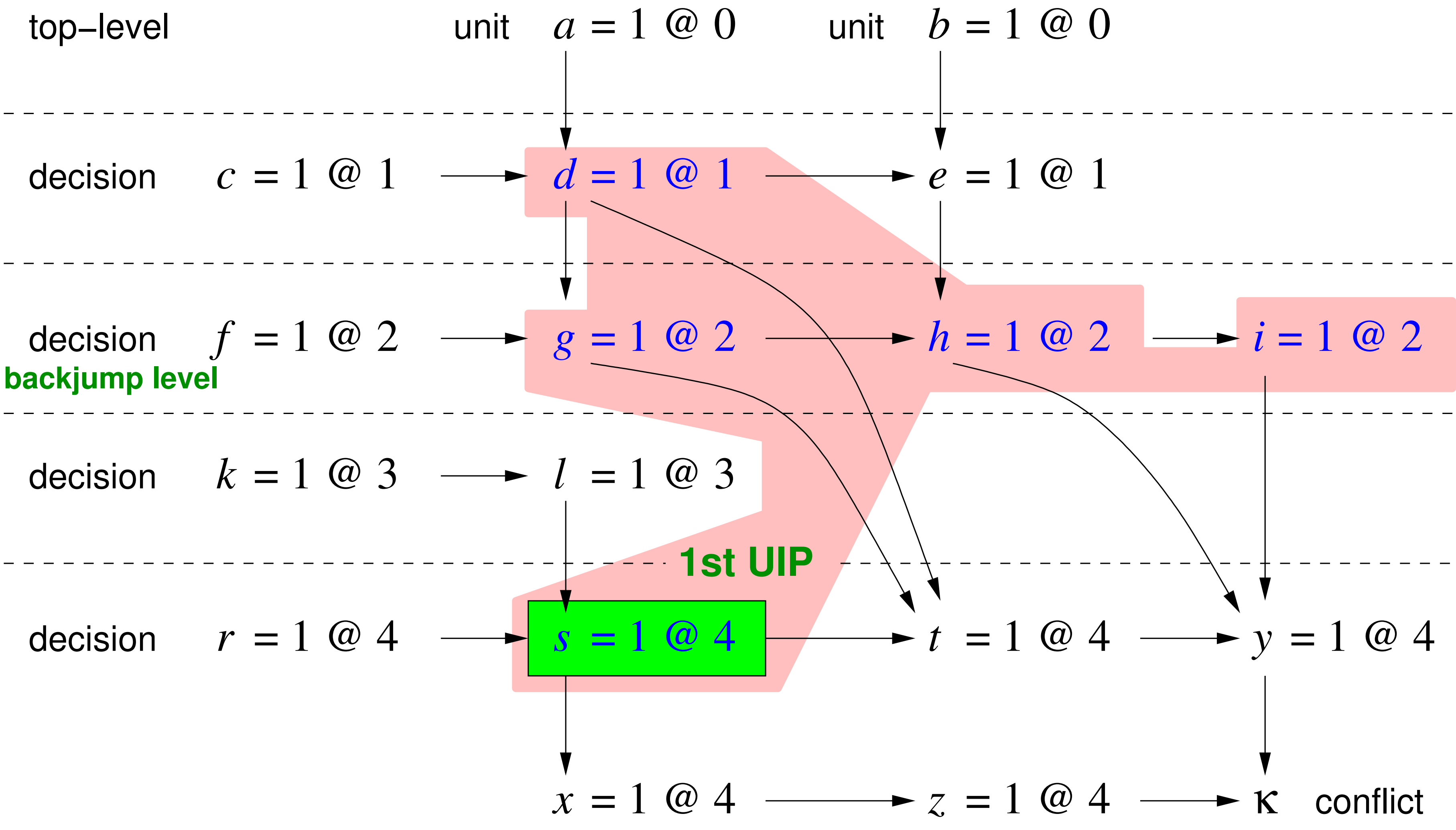
Resolving Antecedents 4$^{\text{th}}$ Time



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

self subsuming resolution

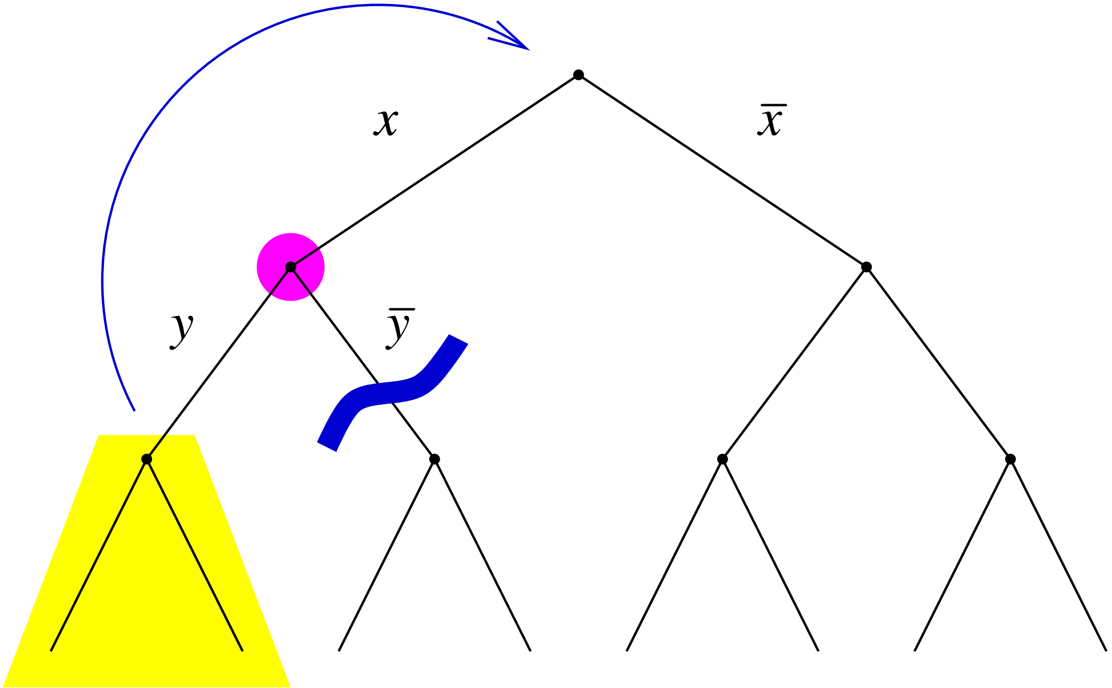
1^{st} UIP Clause after 4 Resolutions



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

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

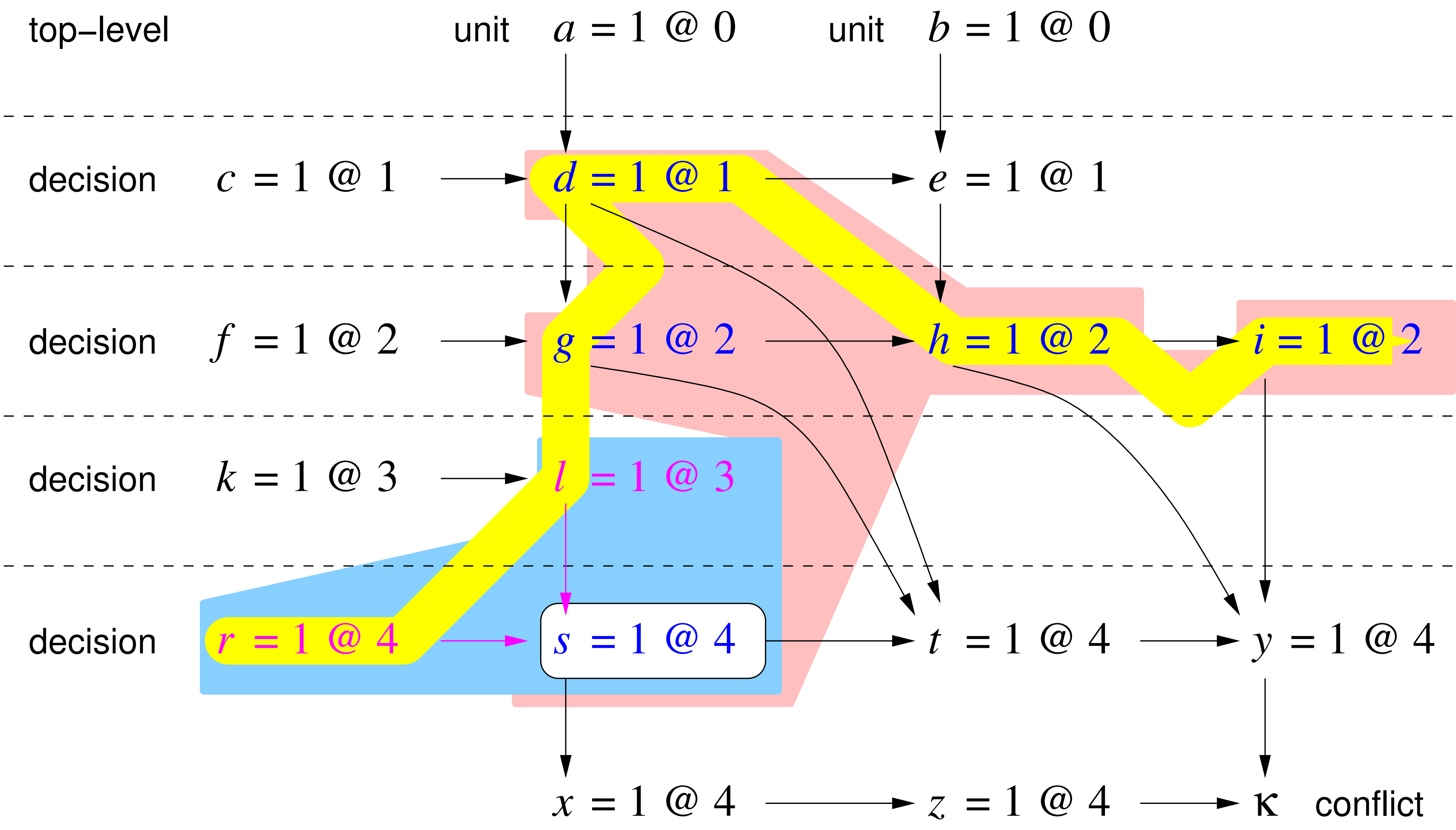
Backjumping



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

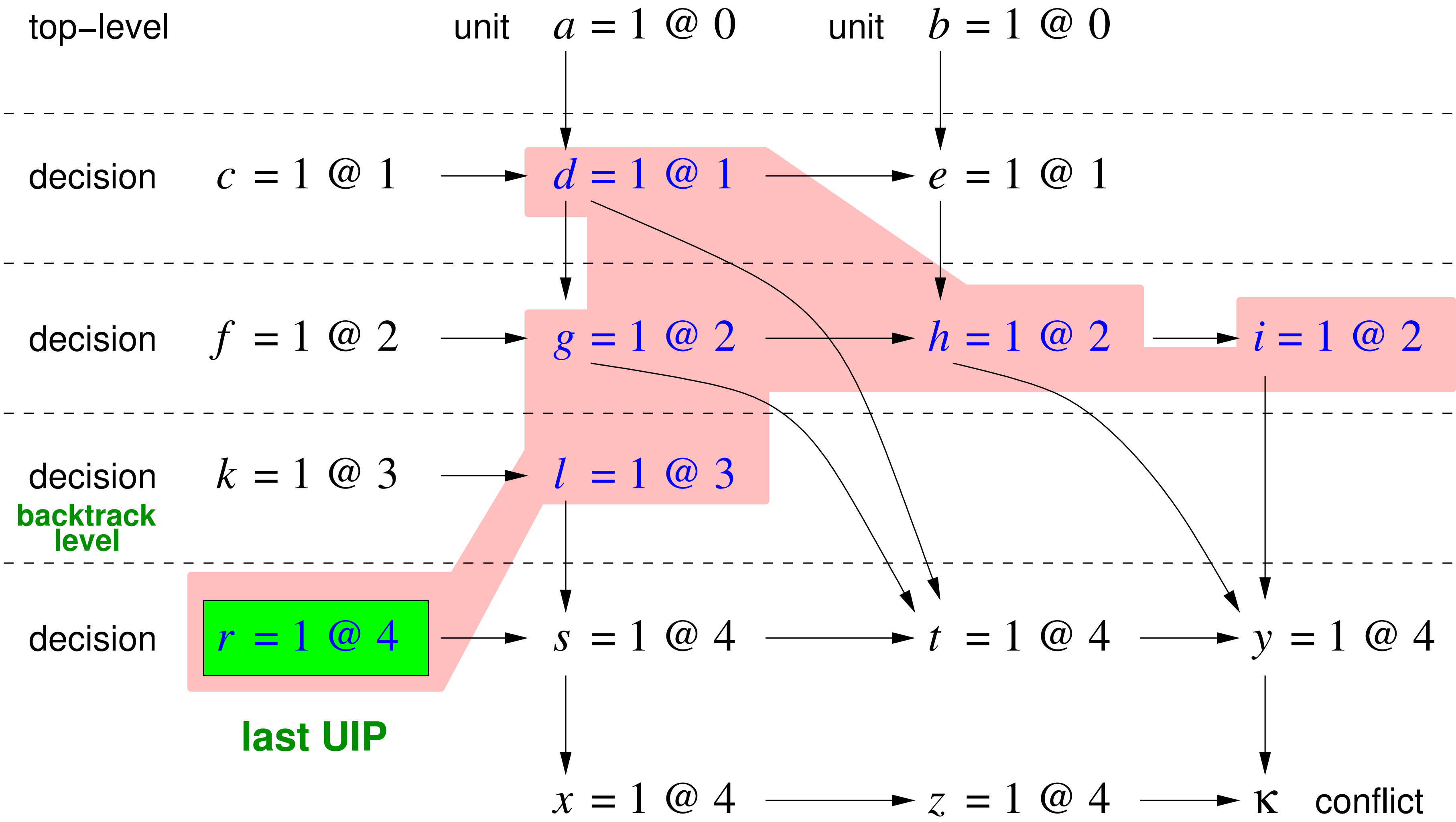
Immediately jump back to the $\overline{x}$ case – assuming x was used.

Resolving Antecedents 5^{th} Time



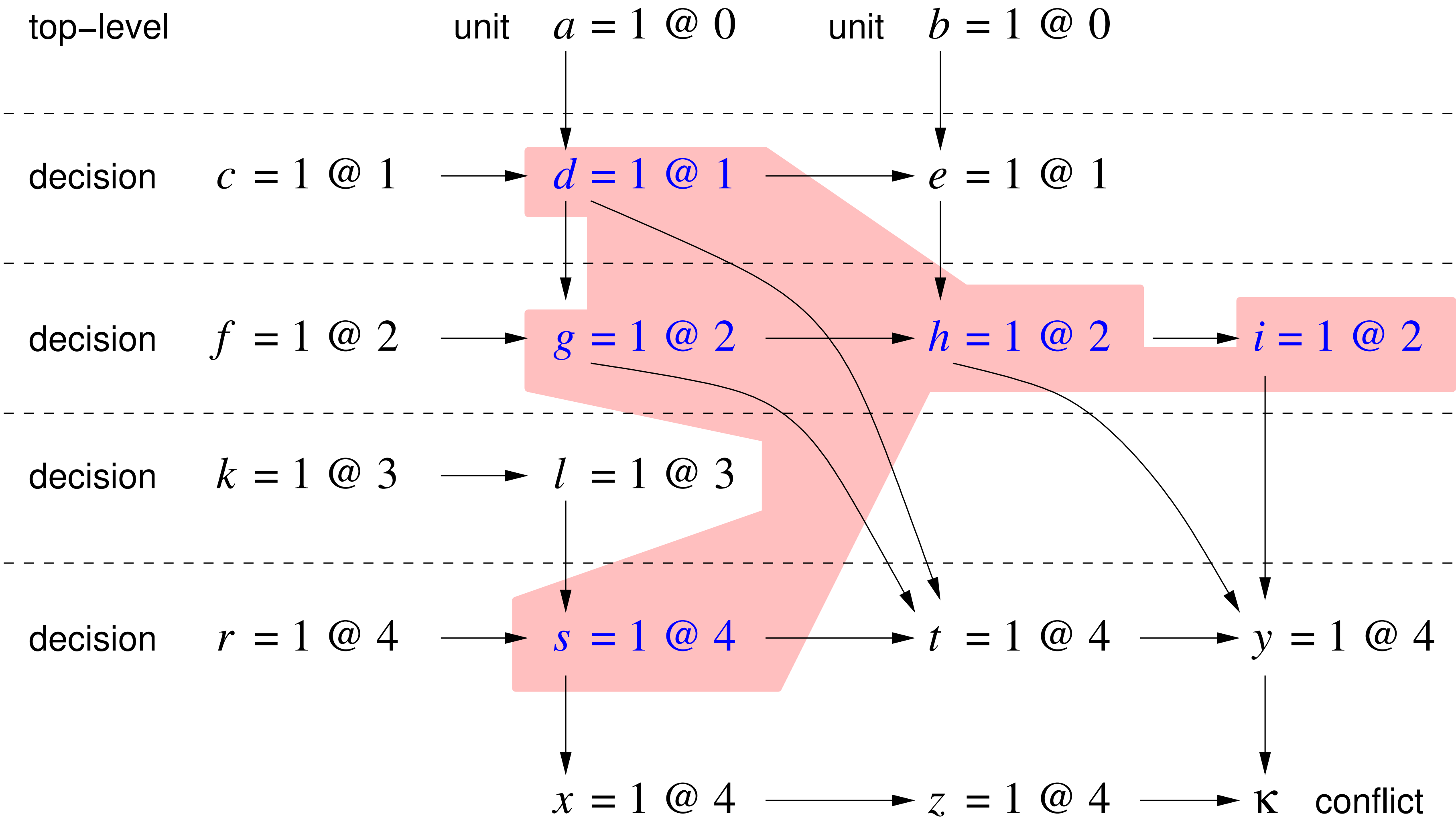
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



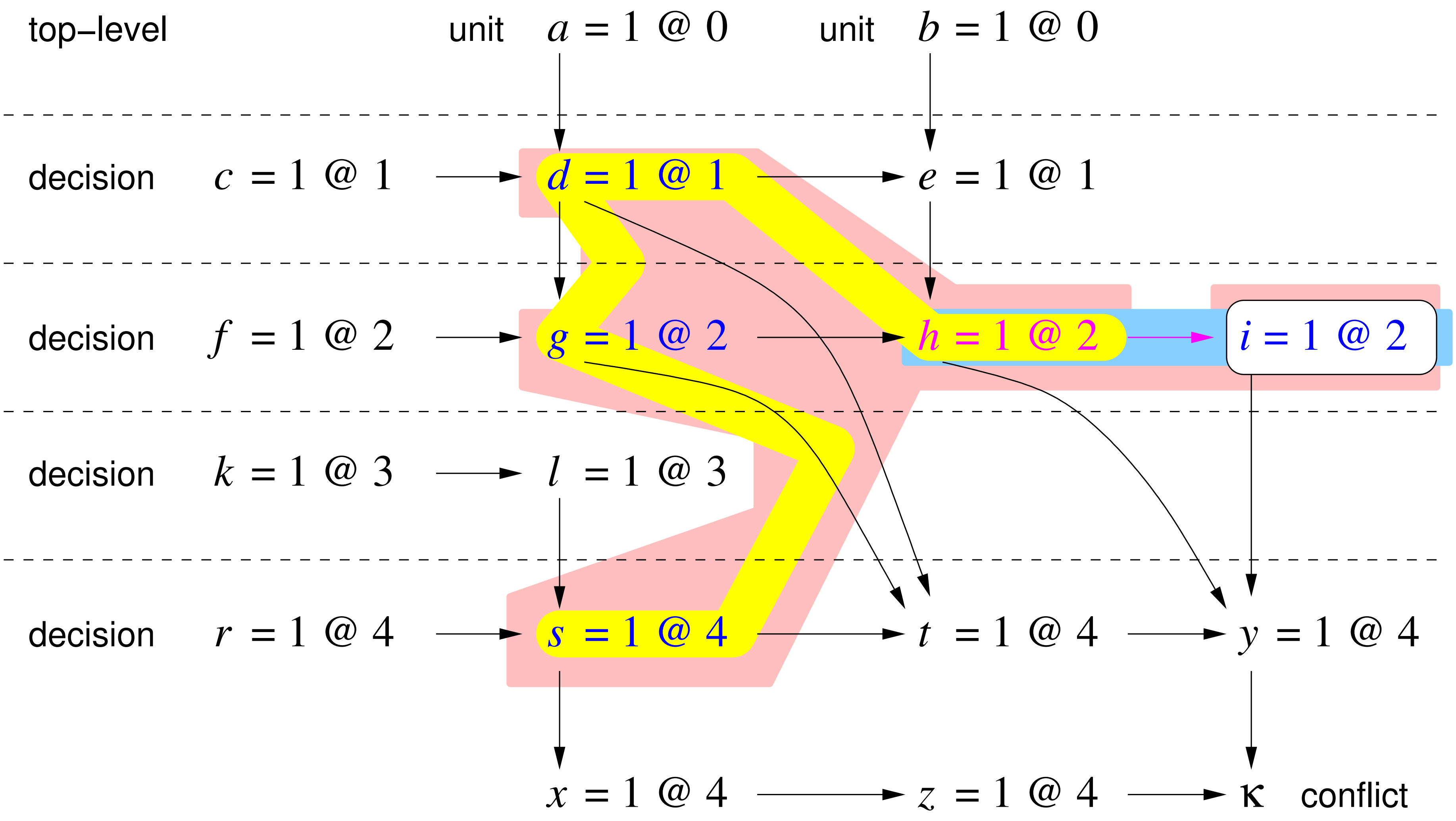
$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

1st UIP Clause after 4 Resolutions



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

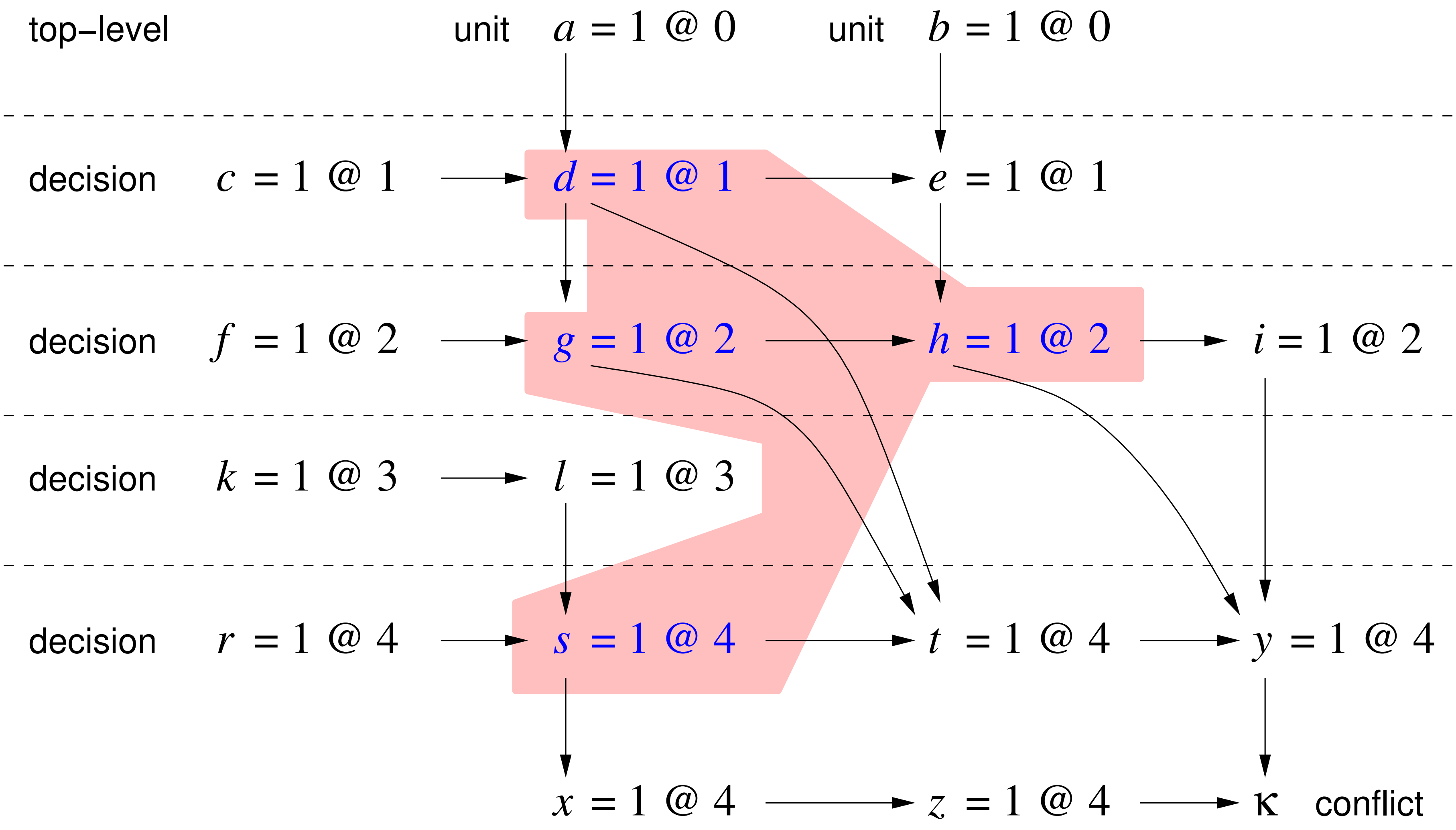
Locally Minimizing 1^{st} UIP Clause



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

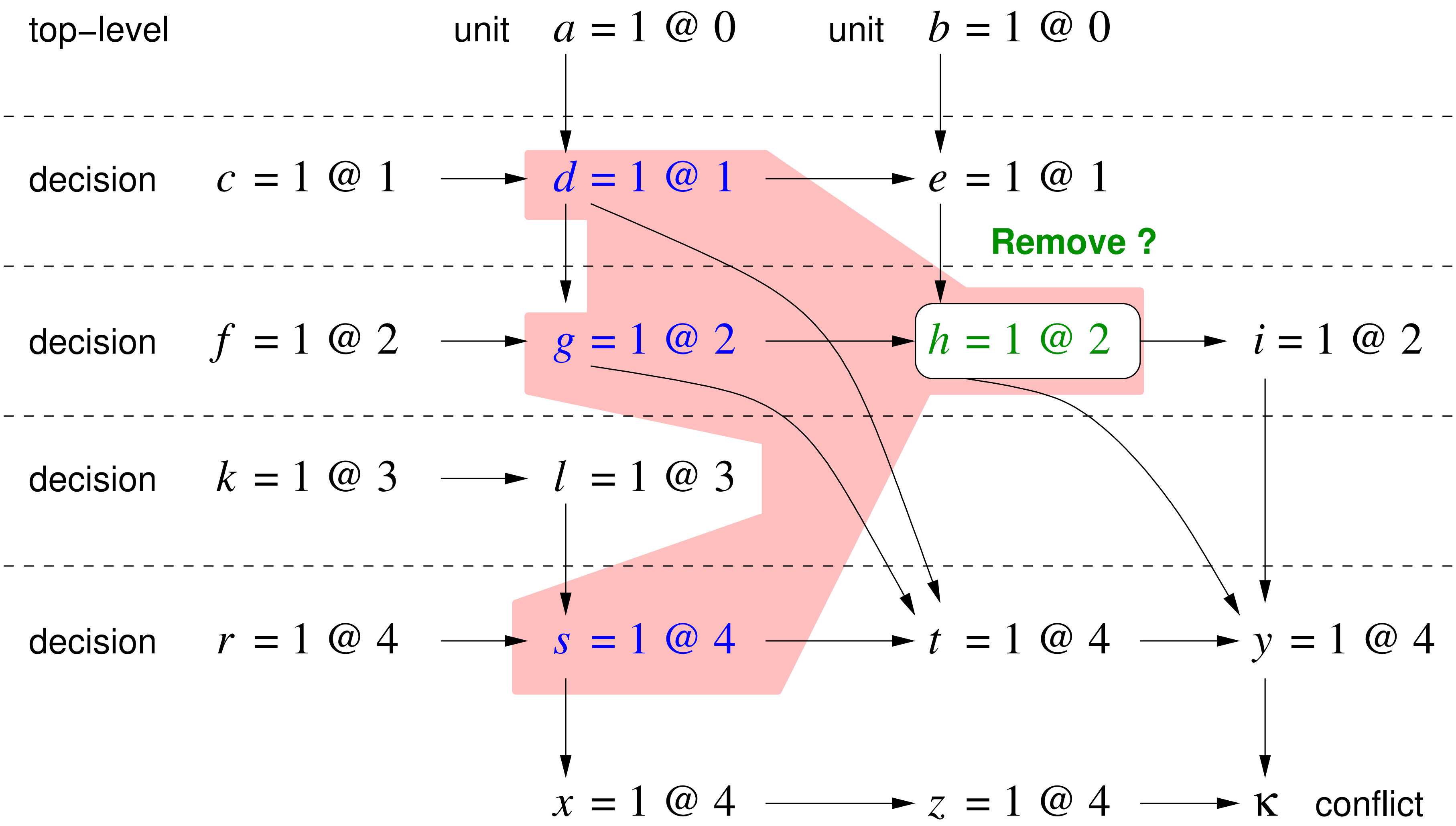
self subsuming resolution

Locally Minimized Learned Clause



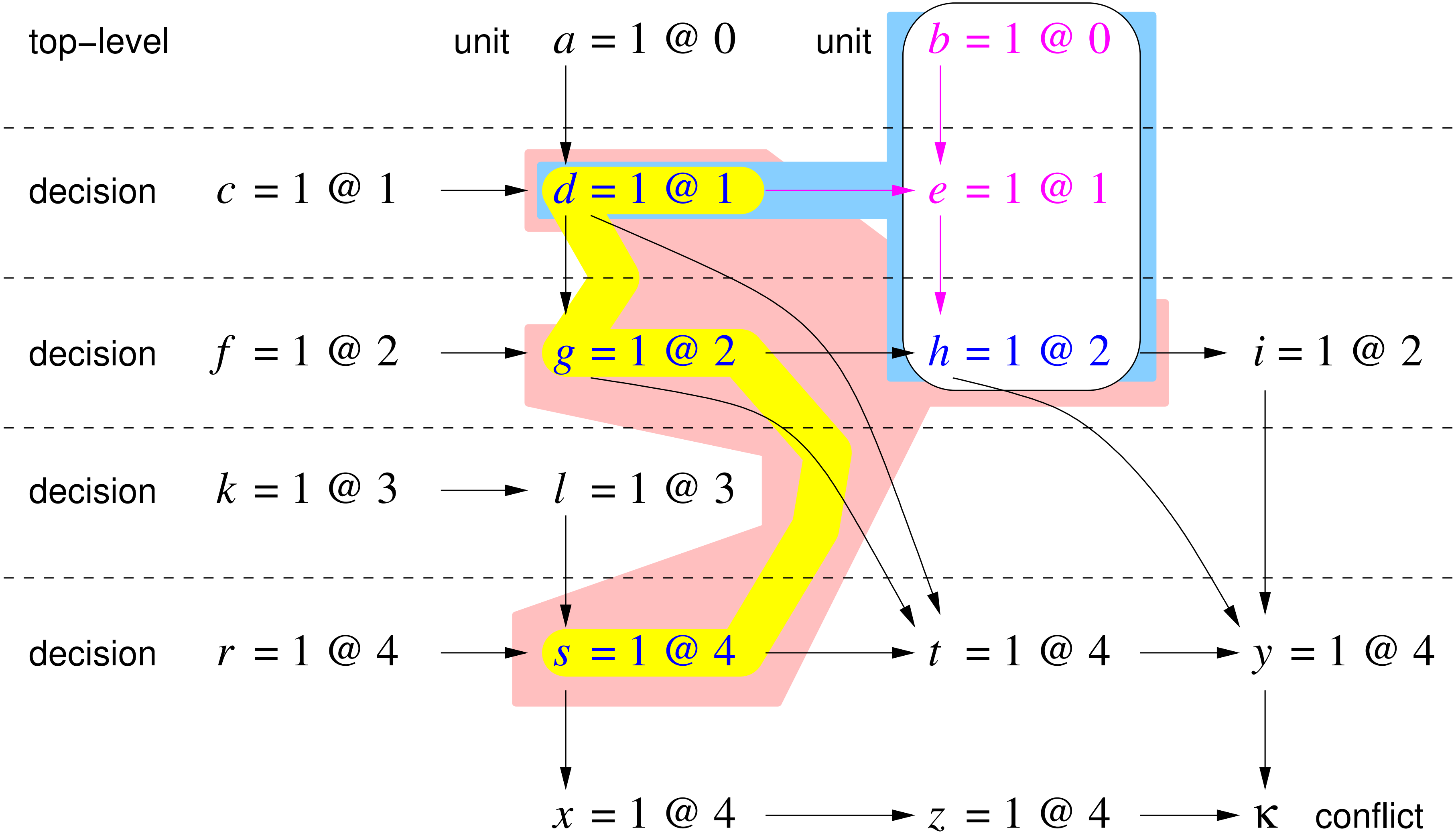
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$$

Minimizing Locally Minimized Learned Clause Further?



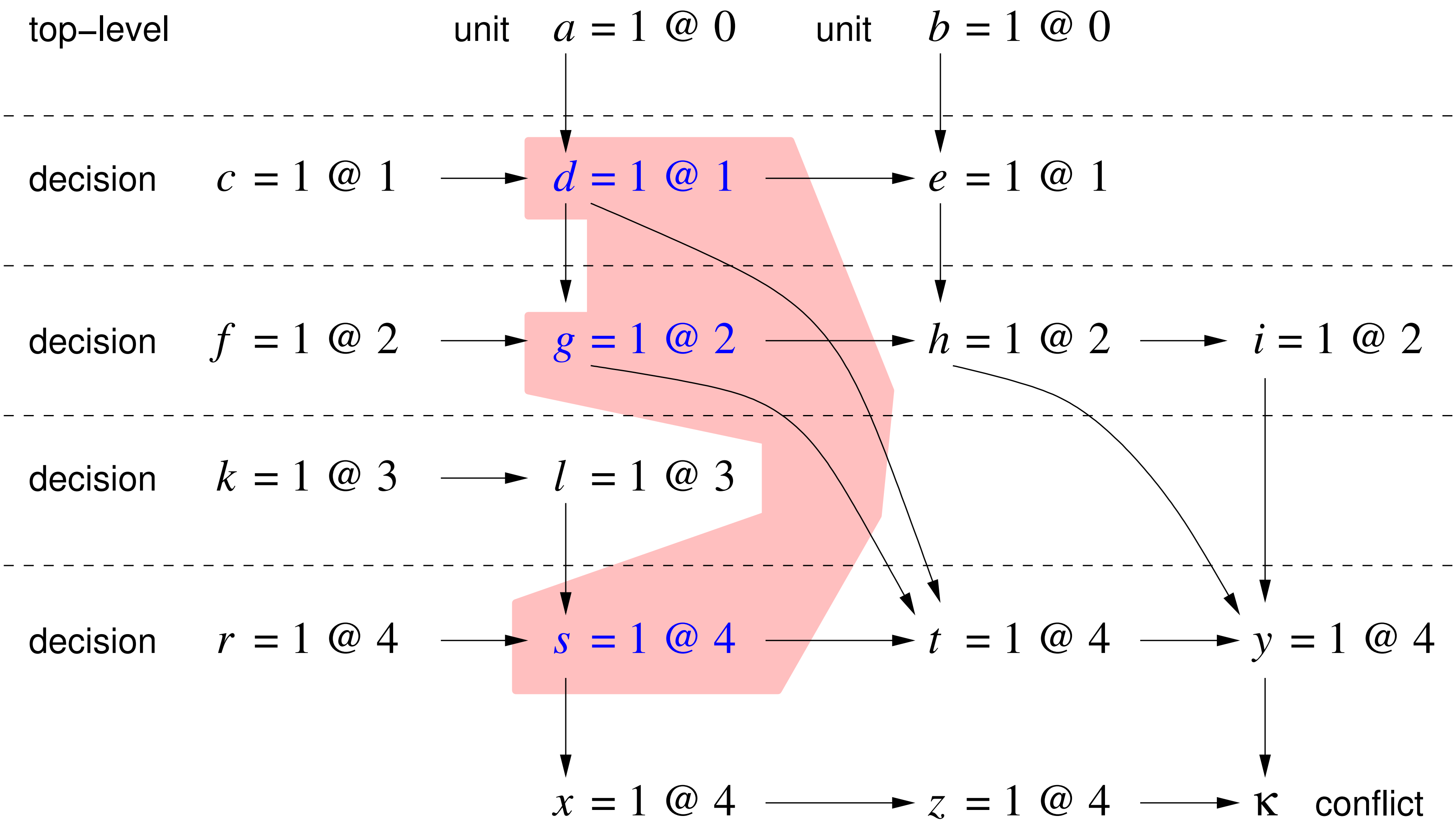
$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

Recursively Minimizing Learned Clause



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

Recursively Minimized Learned Clause



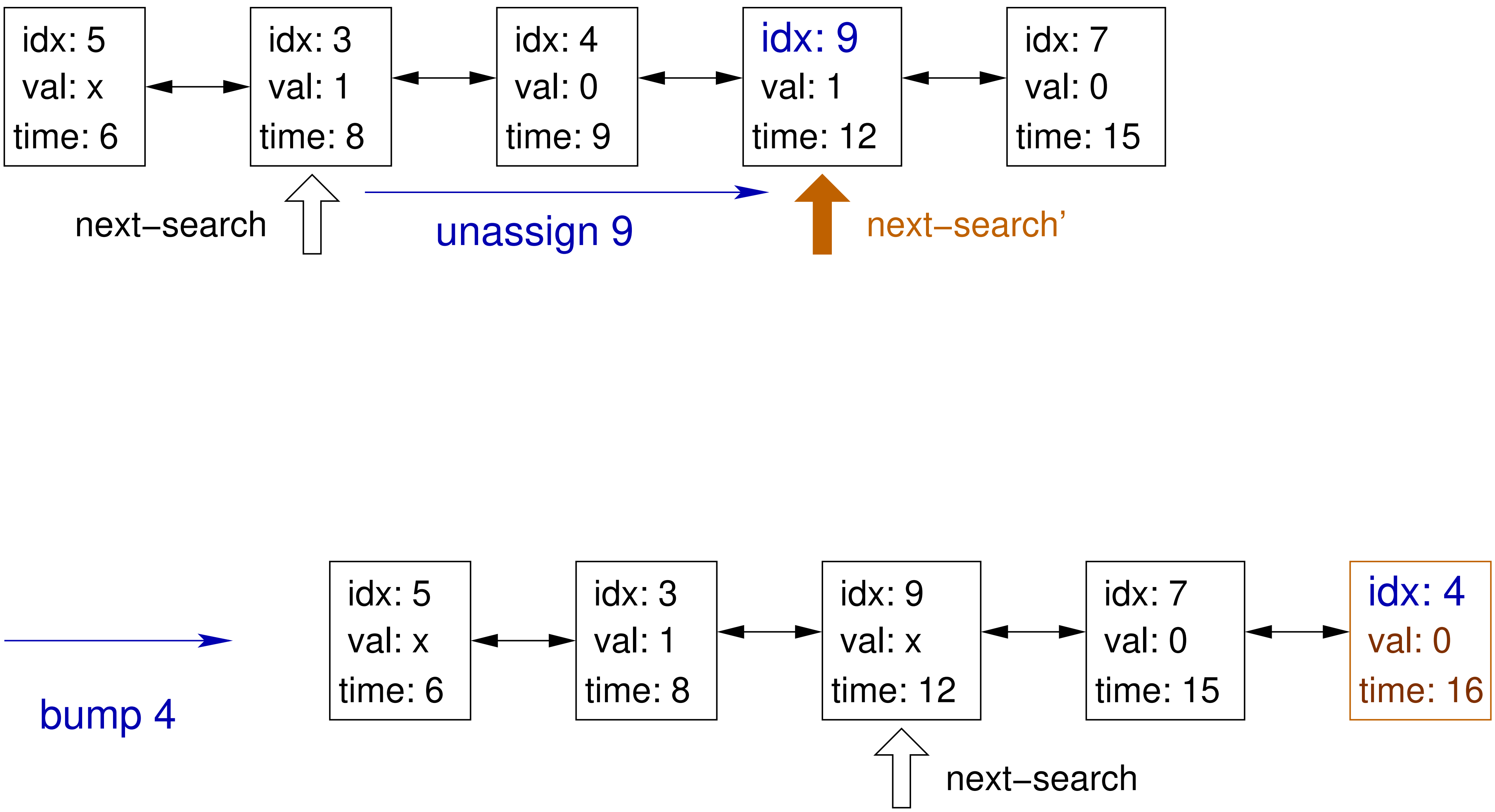
$(\overline{d} \vee \overline{g} \vee \overline{s})$

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 $\quad s'$ 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

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

i conflict index $\qquad b$ 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]
- eagerly reduce hyper-binary resolvents derived in inprocessing

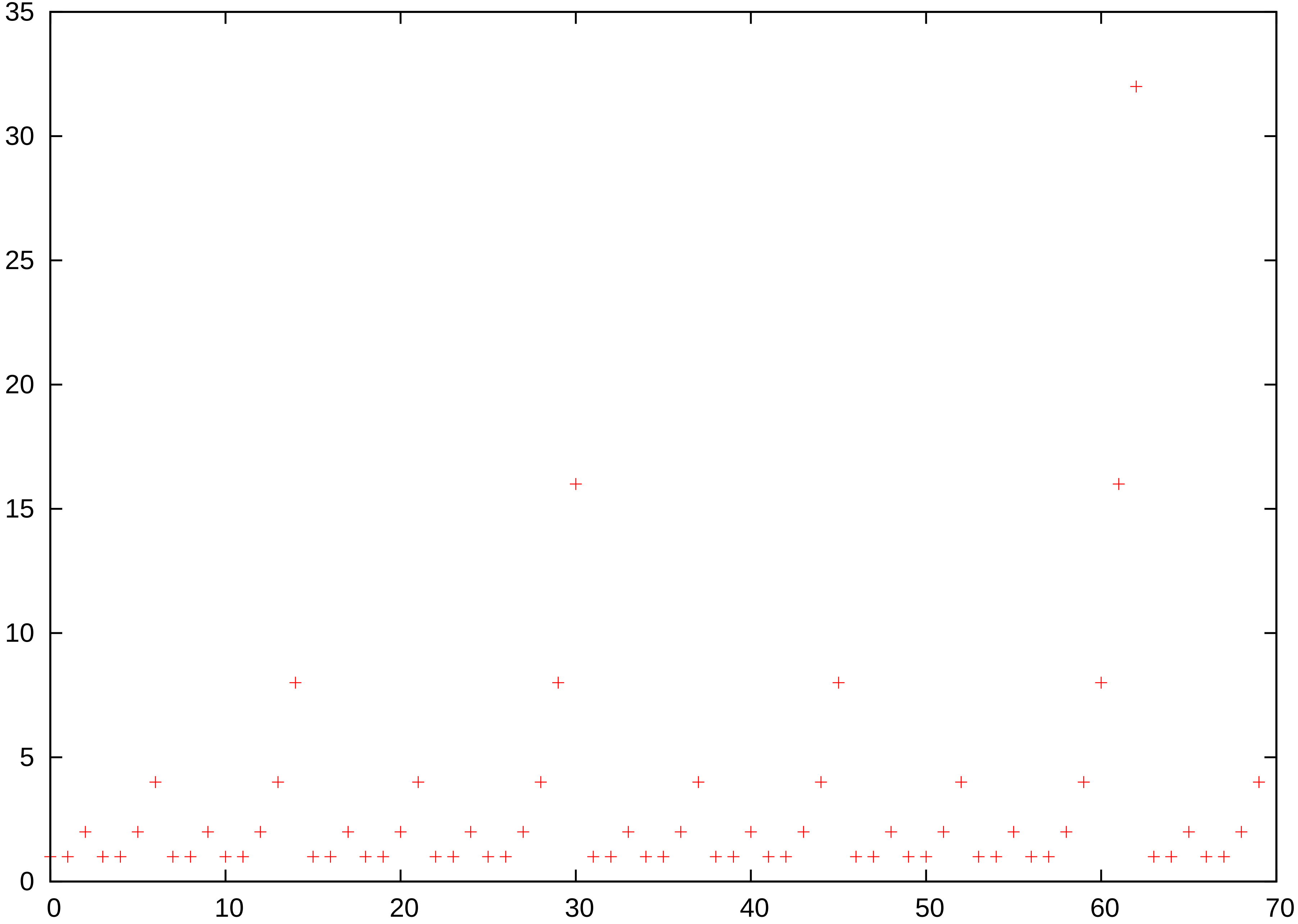
Restarts

- 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]

  call it now "stabilizing mode" and "focused mode"

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

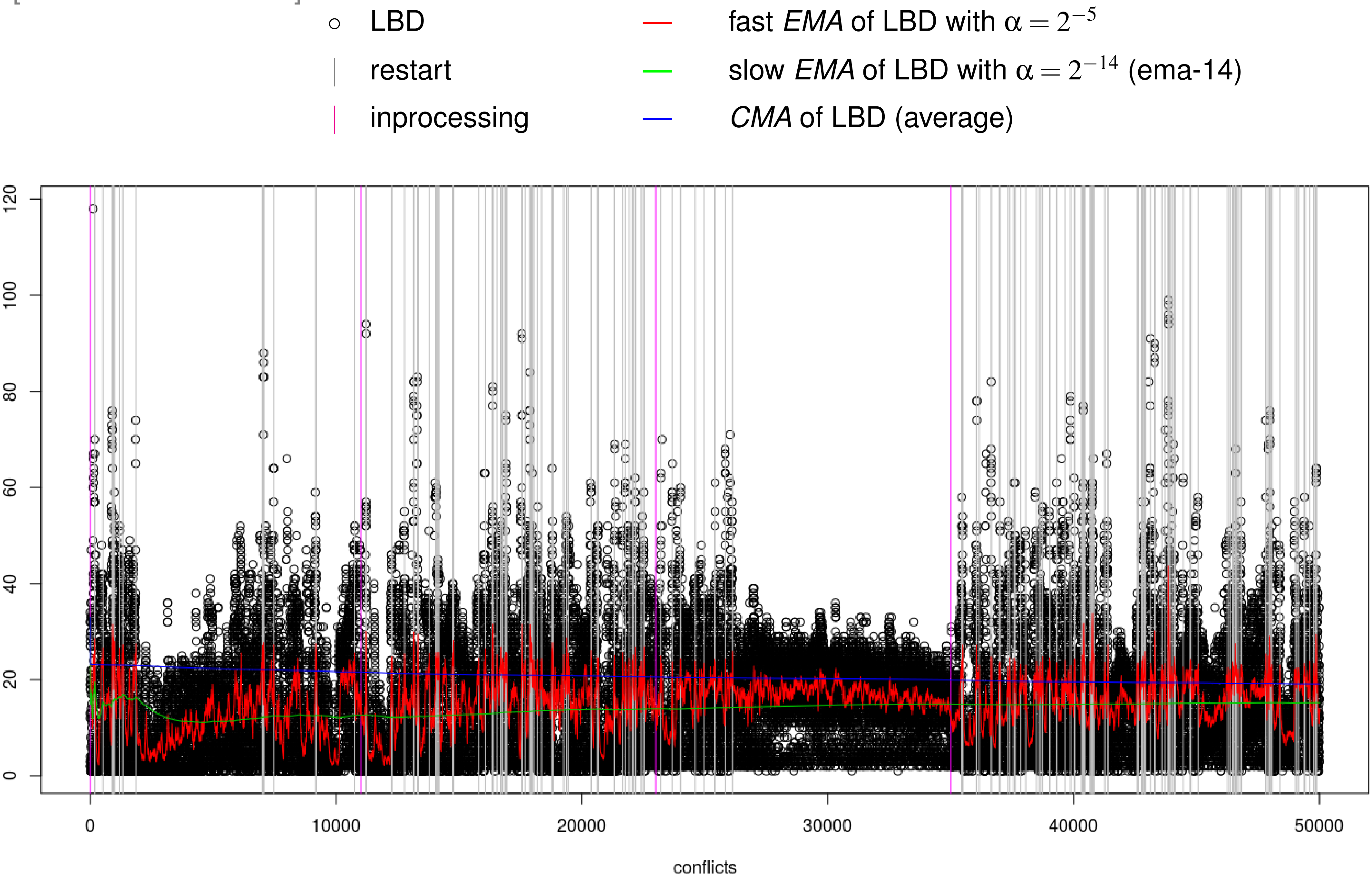
[Knuth'12]

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

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

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] [BiereFleury-POS-2020]

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

newest Version 1.3.1 from June 18

```
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 (search_limits_hit ()) break;    // decision or conflict limit
  else if (terminated_asynchronously ())   // externally terminated
    break;
  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
}
```

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]

Things we did not discuss ...

- advanced preprocessing and inprocessing
  IJCAI-JAIR 2019 award for [HeuleJärvisaloLonsingSeidlBiere-JAIR-2015]
  (many) best papers with Marijn Heule and Benjamin Kiesl
  [PhD thesis of Bejamin Kiesl 2019]
- proofs (Marijn Heule), certificates for UNSAT, interpolation
- relation to proof complexity   Banff, Fields, Dagstuhl seminars
- extensions formalisms: QBF, Pseudo-Boolean, #SAT, ...
- local search   this year's best solvers have all local search in it
- challenges: arithmetic reasoning (and proofs)
  best paper [KaufmannBiereKauers-FMCAD'17]   [PhD thesis Daniela Kaufmann 2020]
- chronological backtracking   [RyvchinNadel-SAT'18] [MöhleBiere-SAT'19]
- incremental SAT solving
  best student paper [FazekasBiereScholl-SAT'19]   [PhD thesis of Katalin Fazekas in 2020]
- parallel and distributed SAT solving   Handbook of Parallel Constraint Reasoning, ...

Personal SAT Solver History