Modern SAT Solvers

Part B

Vienna Winter School on Verification

9 February 2012

TU Vienna, Austria

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

http://fmv.jku.at

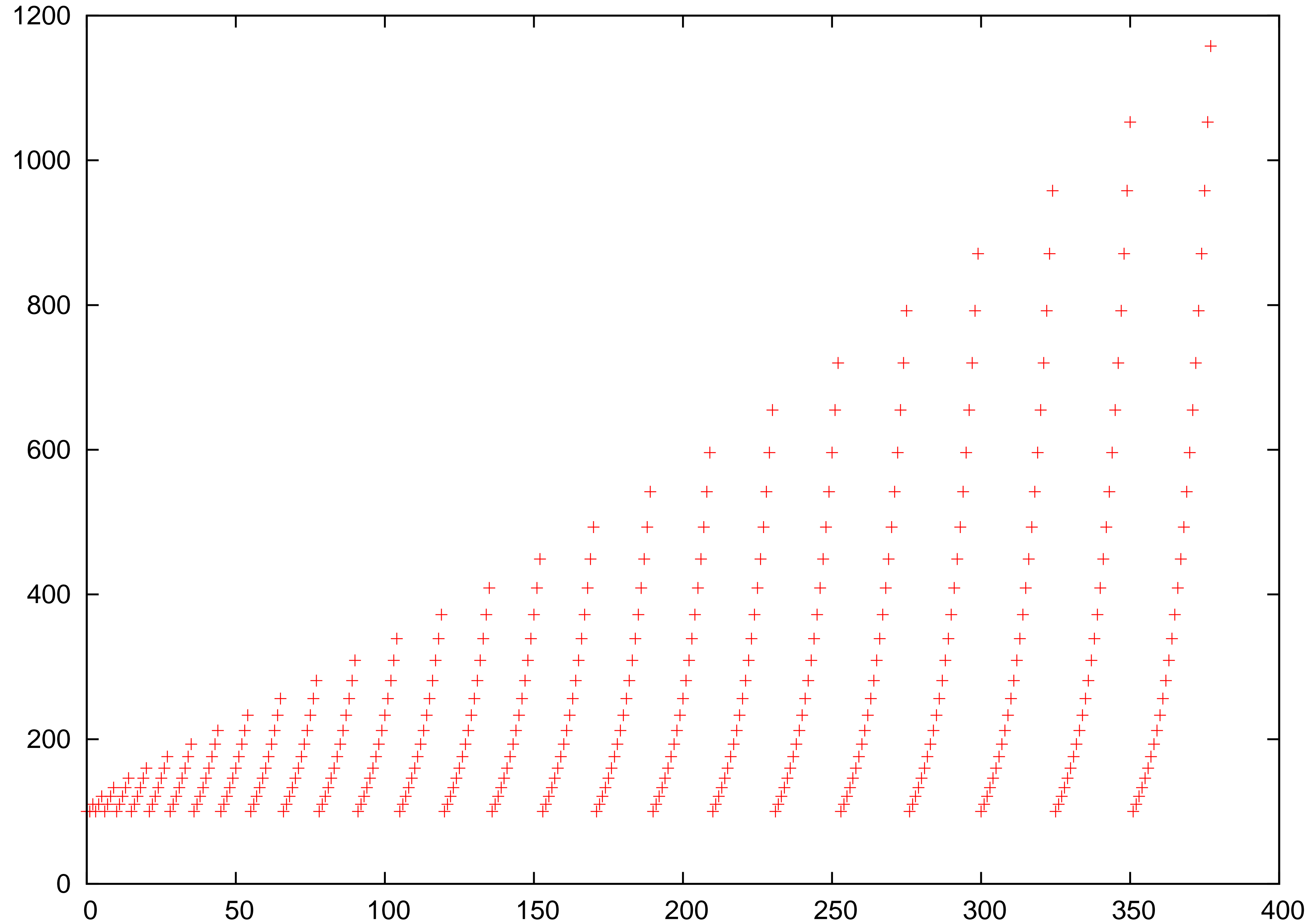
Restarts

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

RiSE

Inner/Outer Restart Intervals

378 restarts in 104408 conflicts

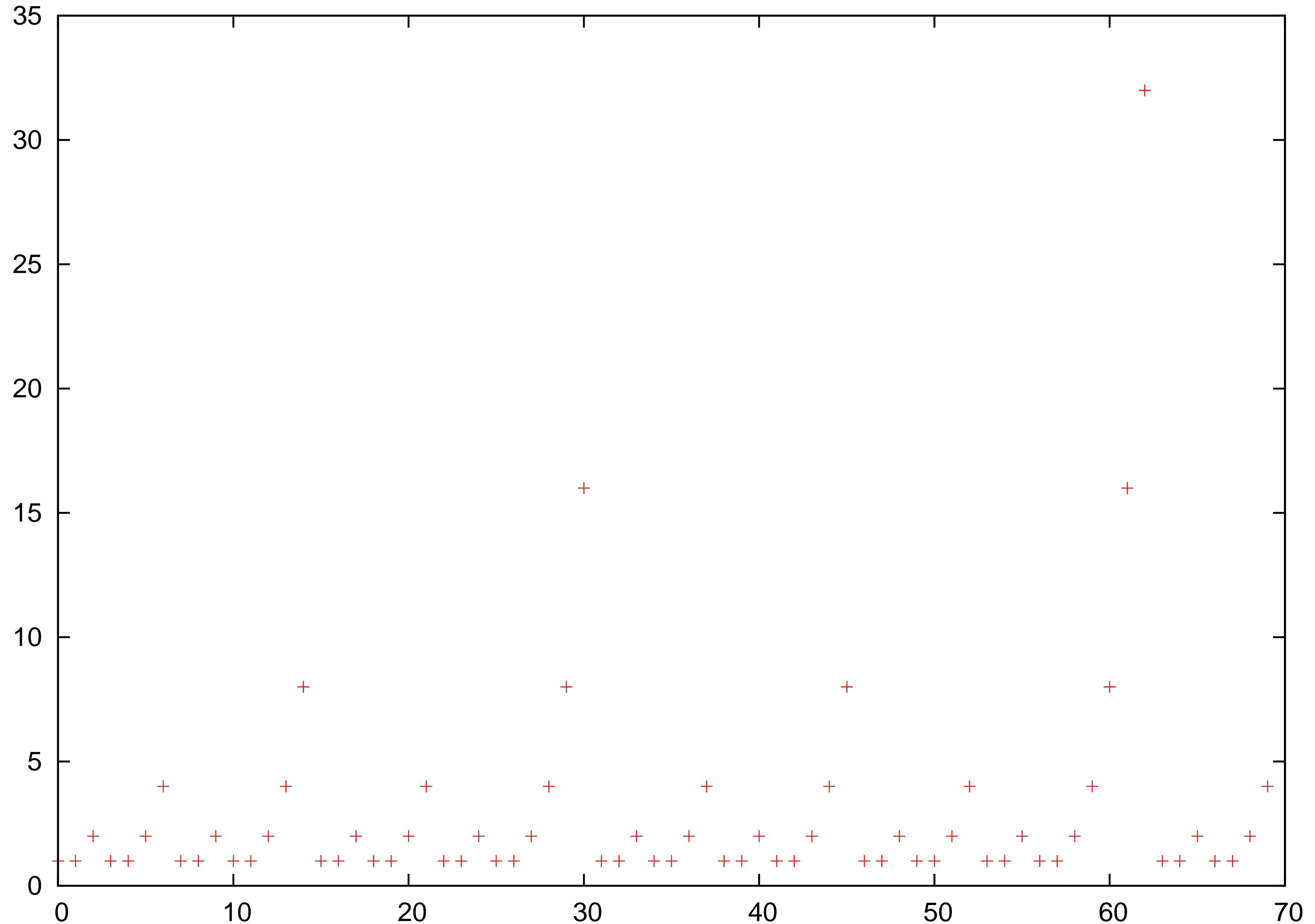


Inner/Outer Restart Scheduling


```
int inner = 100, outer = 100;
int restarts = 0, conflicts = 0;

for (;;)
  {
    ... // run SAT core loop for 'inner' conflicts

    restarts++;
    conflicts += inner;

    if (inner >= outer)
      {
        outer *= 1.1;
        inner = 100;
      }
    else
      inner *= 1.1;
  }
```

Luby's Restart Intervals

70 restarts in 104448 conflicts

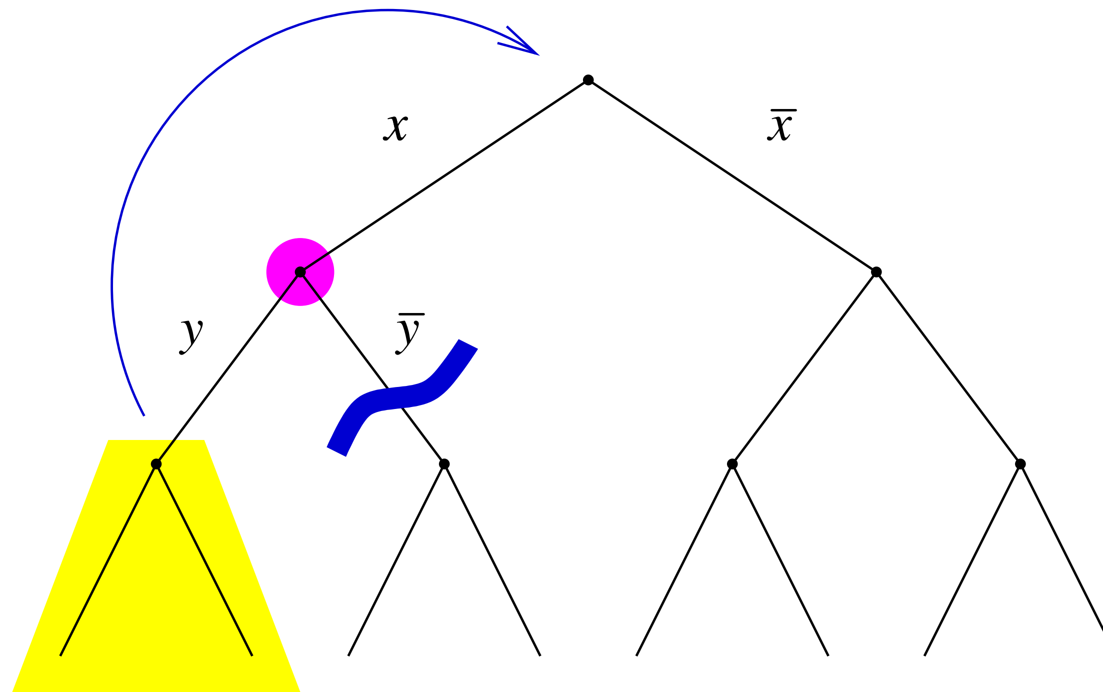


RiSE

Luby Restart Scheduling

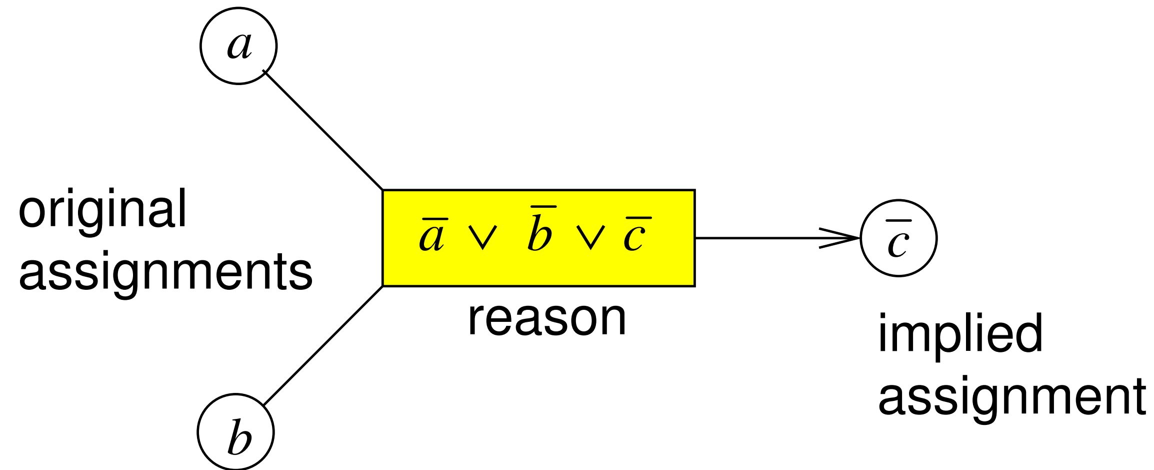
```
unsigned
luby (unsigned i)
{
  unsigned k;

  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
```

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

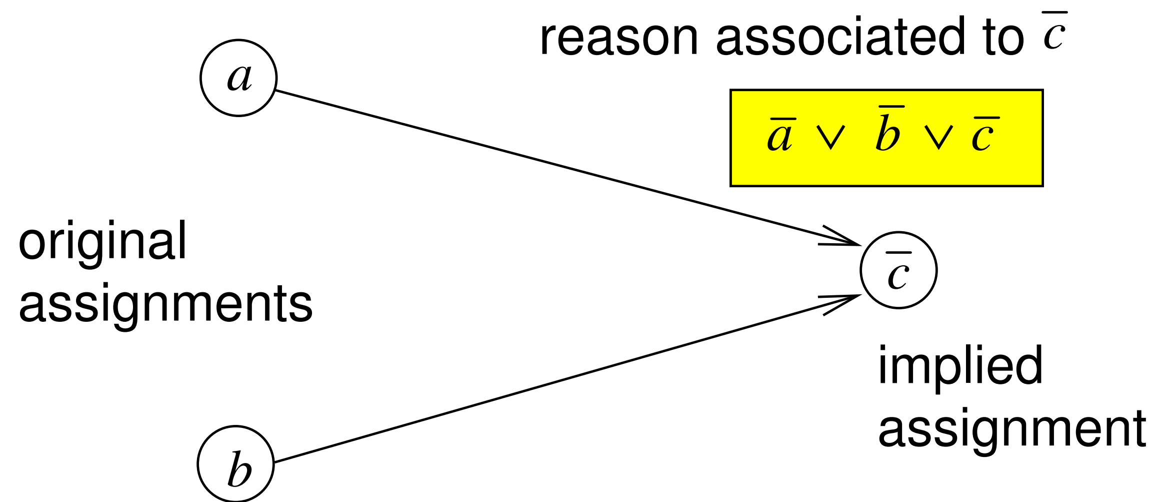


If y has never been used to derive a conflict, then skip $\overline{y}$ case.

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


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

Optimized Implication Graph for Unit Resolution in DP



RiSE

CNF for following Examples


```
-3 1 2 0
3 -1 0
3 -2 0
-4 -1 0
-4 -2 0
-3 4 0
3 -4 0
-3 5 6 0
3 -5 0
3 -6 0
4 5 6 0
```

We use a version of the DIMACS format.

Variables are represented as positive integers.

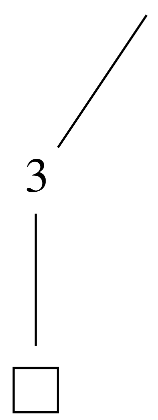
Integers represent literals.

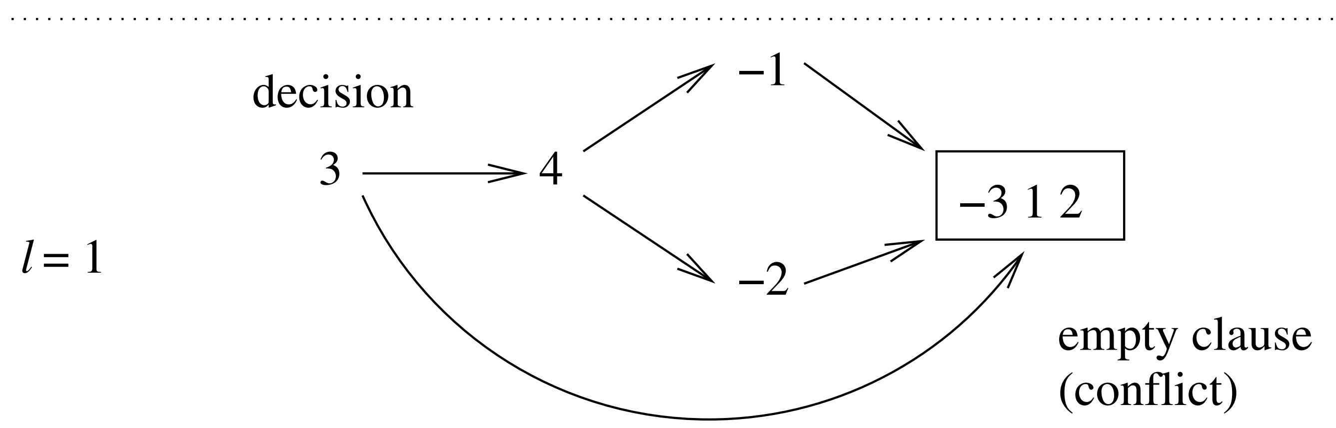
Subtraction means negation.

A clause is a zero terminated list of integers.

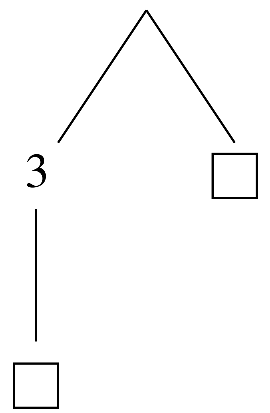
CNF has a good cut made of variables 3 and 4
(but we are going to apply DP with learning to it)

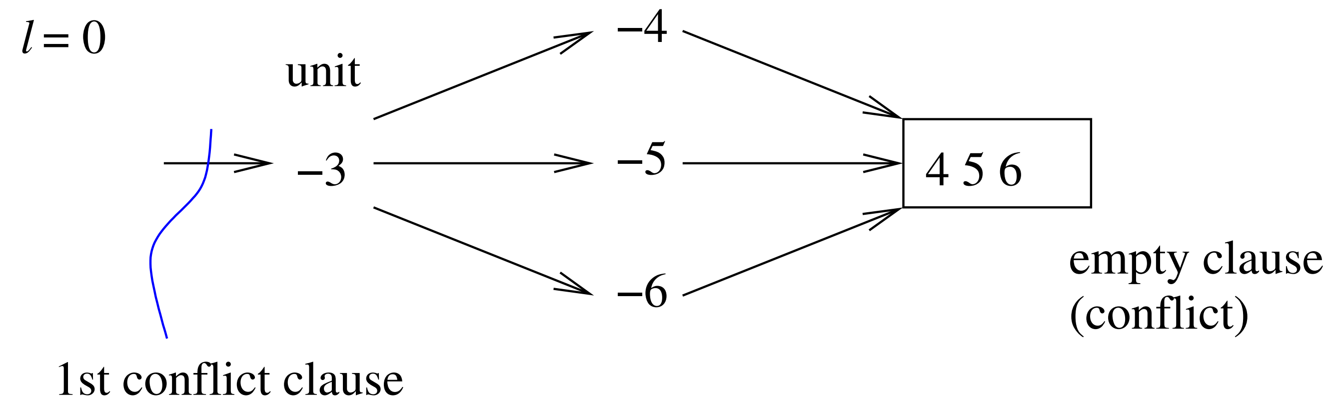
DP with Learning Run 1 (3 as 1st decision)





unit clause −3 is generated as learned clause and we backtrackt to $l = 0$

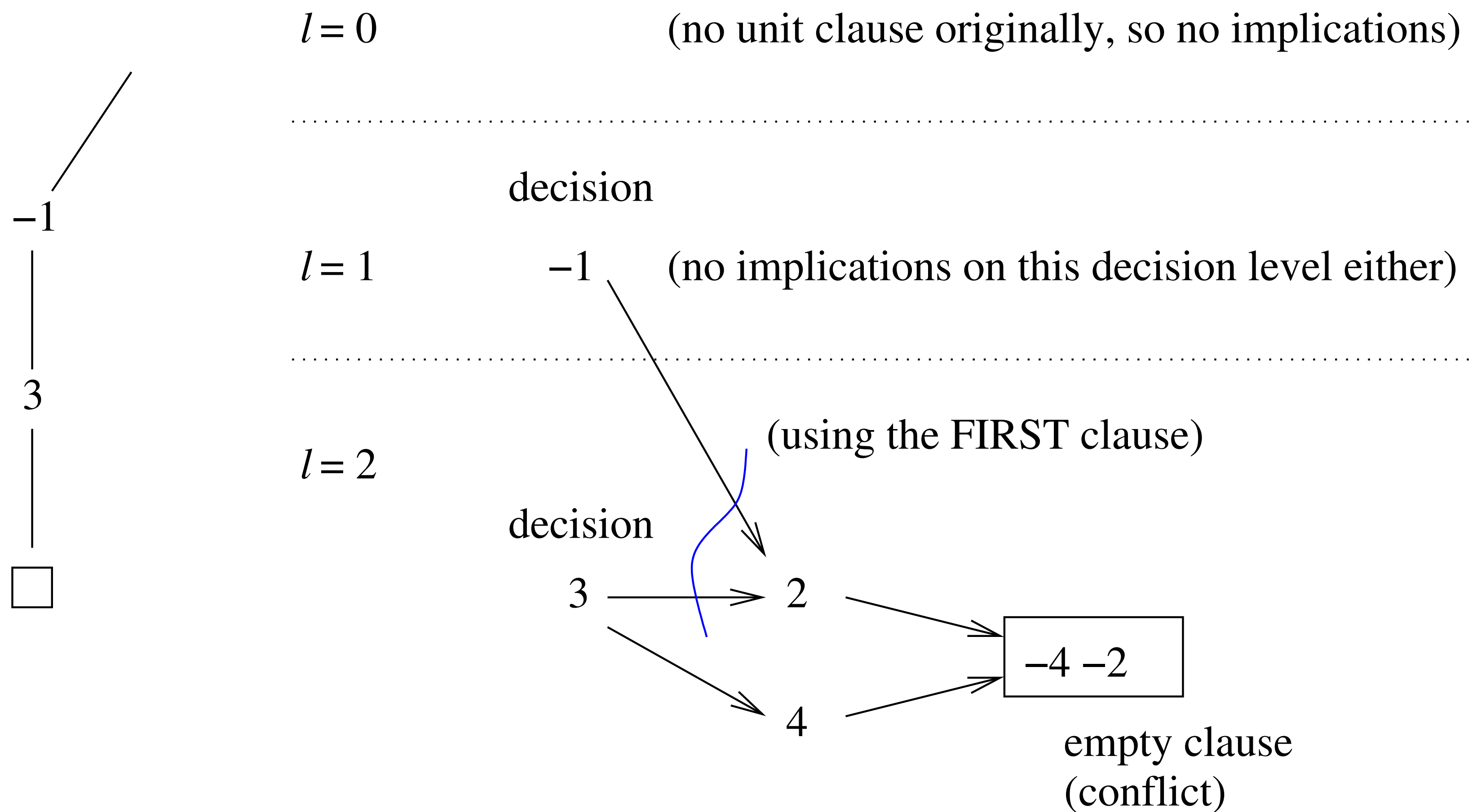




since −3 has a real unit clause as reason, an empty conflict clause is learned

RiSE

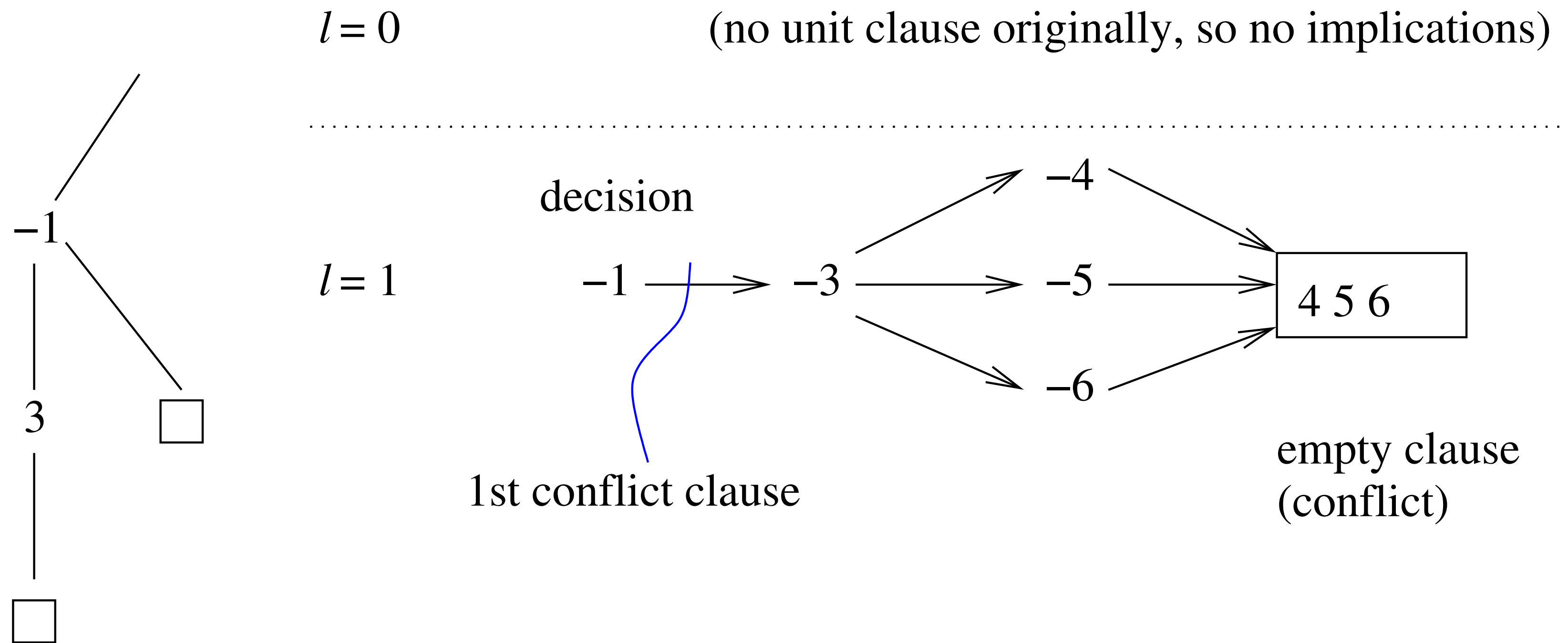
DP with Learning Run 2 Fig. 1 (-1, 3 as decision order)



since FIRST clause was used to derive 2, conflict clause is (1 −3)

backtrack to $l = 1$ (smallest level for which conflict clause is a unit clause)

RiSE

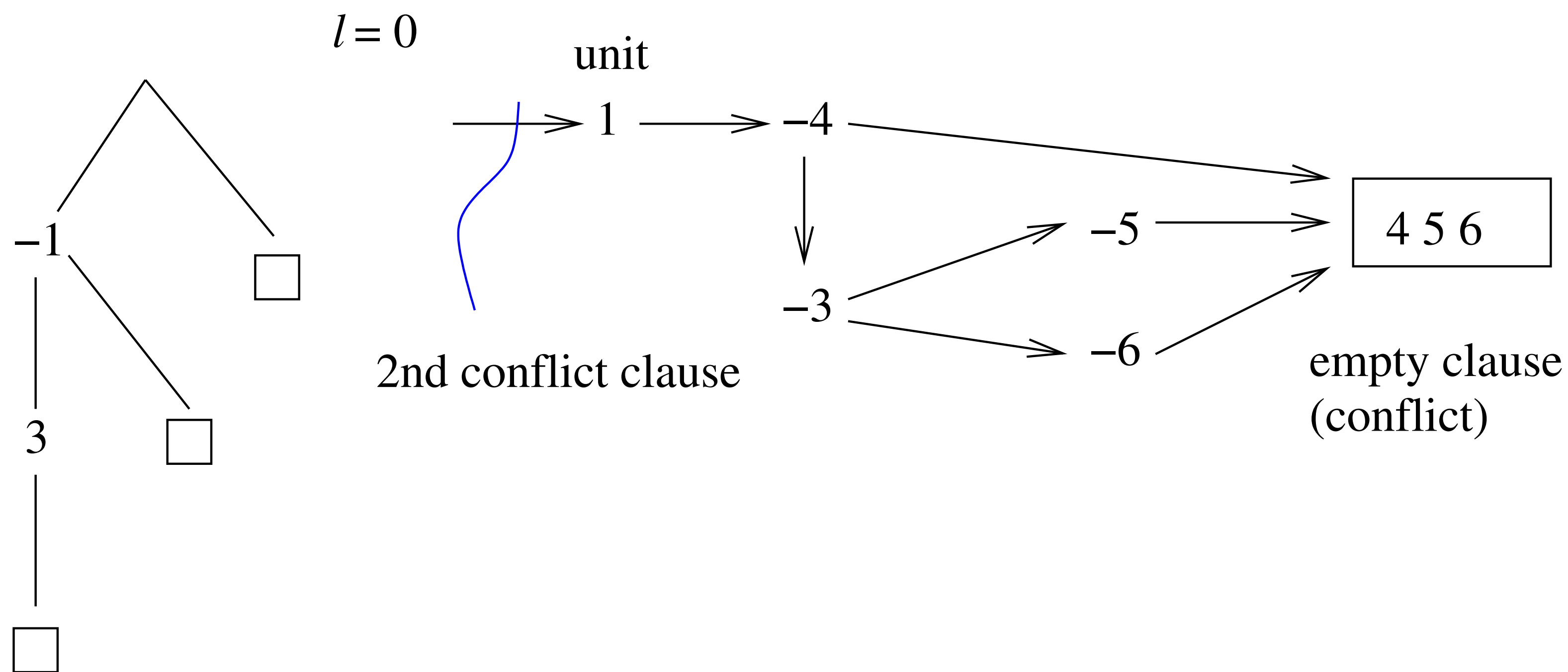


learned conflict clause is the unit clause 1

backtrack to decision level $l = 0$

RiSE

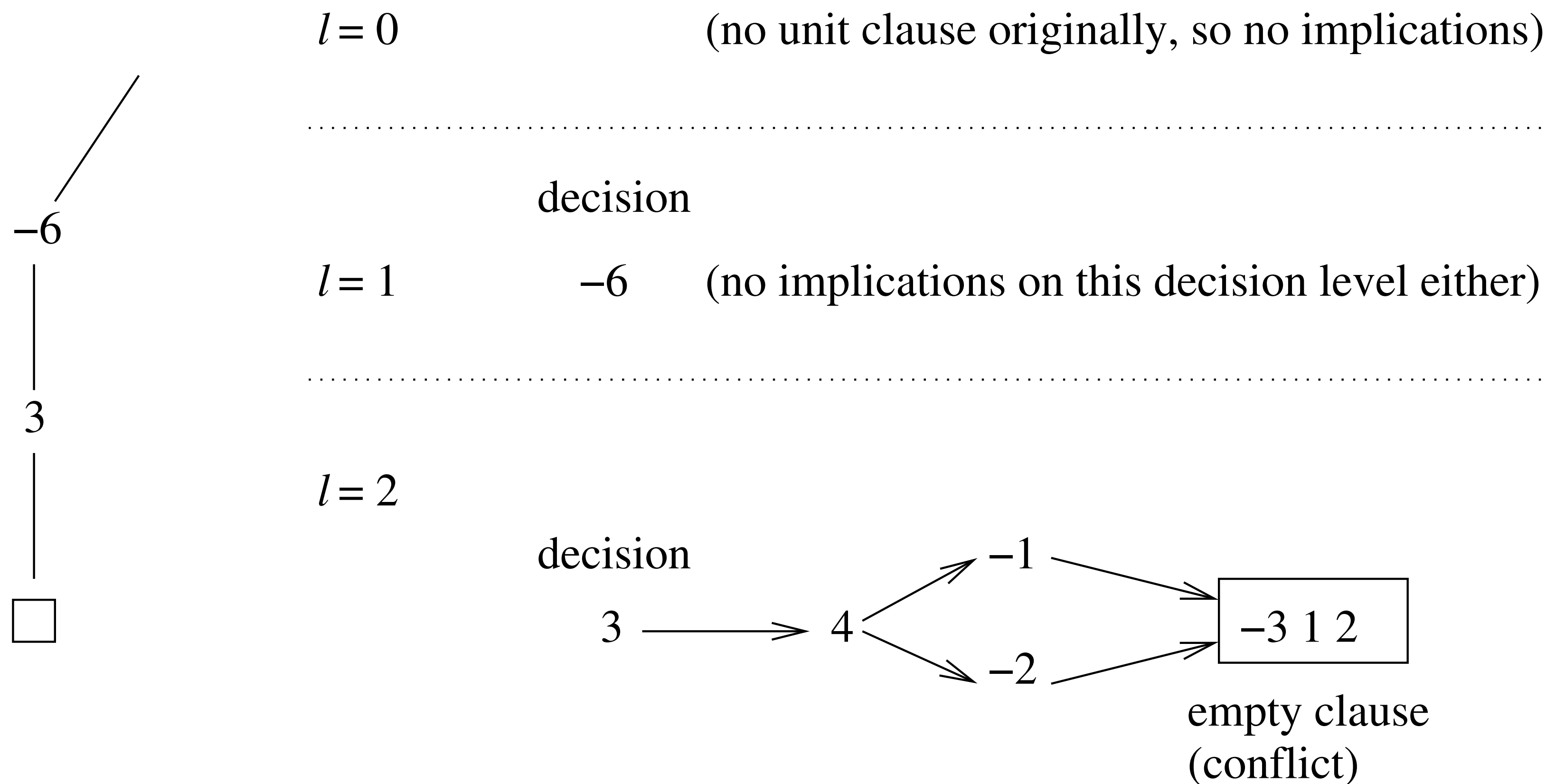
DP with Learning Run 2 Fig. 3 (-1, 3 as decision order)



since the learned clause is the empty clause, conclude unsatisfiability

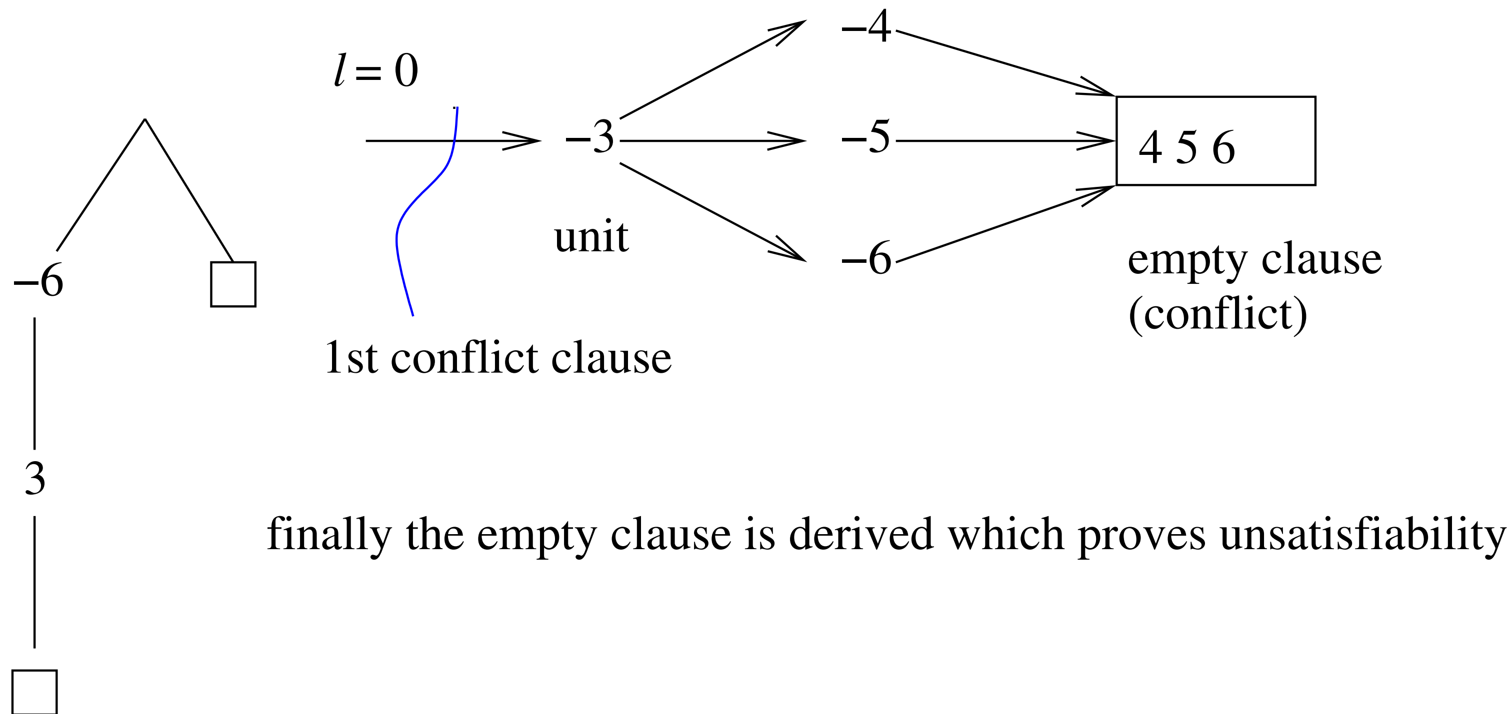
RiSE

DP with Learning Run 3 Fig. 1 (-6, 3 as decision order)



learn the unit clause −3 and BACKJUMP to decision level $l = 0$

RiSE


$l = 0$
−4
−3
−5
−6
4 5 6
unit
empty clause
(conflict)
1st conflict clause
−6
3
finally the empty clause is derived which proves unsatisfiability

Toplevel Loop in DP with Learning

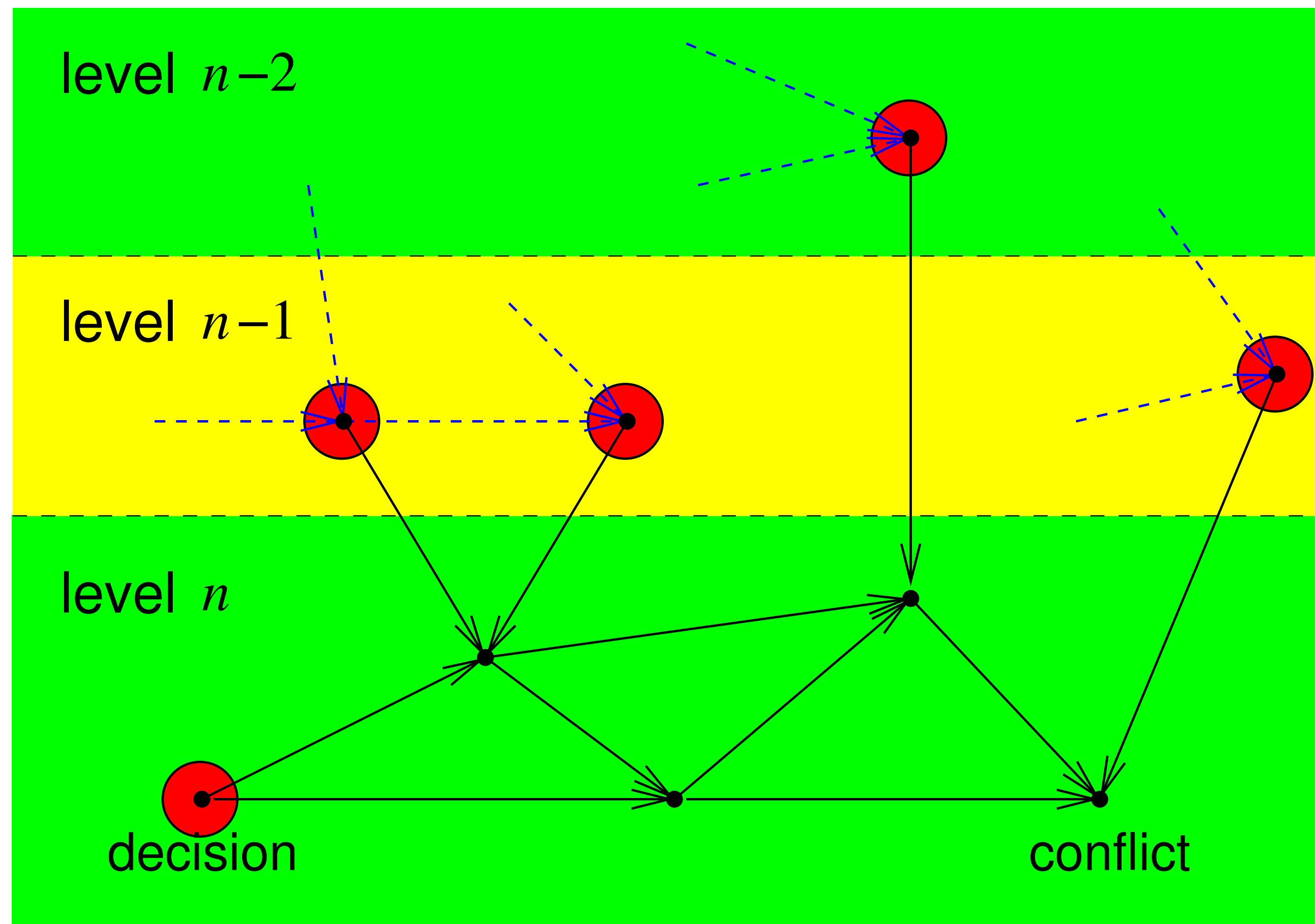

```
int
sat (Solver solver)
{
  Clause conflict;

  for (;;)
    {
      if (bcp_queue_is_empty (solver) && time_to_simplify (solver))
        simplify (solver);
      if (bcp_queue_is_empty (solver)) {
        if (all_variables_assigned (solver)) return SATISFIABLE;
        if (should_restart (solver)) restart ();
        decide (solver);
      }
      conflict = bcp (solver);
      if (conflict && !backtrack (solver, conflict)) return UNSATISFIABLE;
    }
}
```

Backtracking in DP with Learning


```
int
backtrack (Solver solver, Clause conflict)
{
  Clause learned_clause; Assignment assignment; int new_level;

  if (decision_level(solver) == 0)
    return 0;

  analyze (solver, conflict);
  learned_clause = add (solver);

  assignment = drive (solver, learned_clause);
  enqueue_bcp_queue (solver, assignment);

  new_level = jump (solver, learned_clause);
  undo (solver, new_level);

  return 1;
}
```

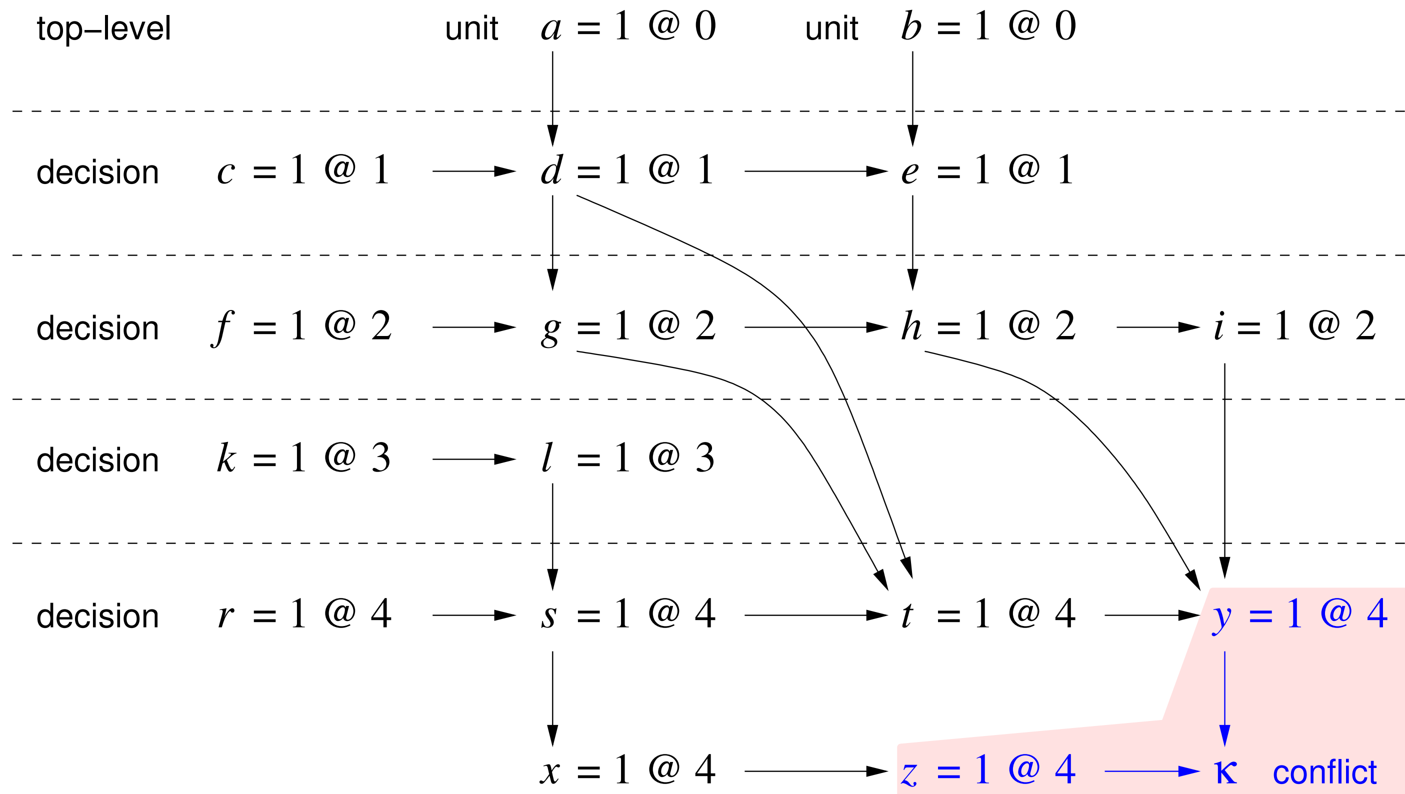

RiSE

Conflict Clauses as Cuts in the Implication Graph



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

Unique Implication Points (UIP)



UIP = *articulation point* in graph decomposition into biconnected components

(simply a node which, if removed, would disconnect two parts of the graph)

Detection of UIPs

- can be found by graph traversal in the order of made assignments

- *trail* respects this order

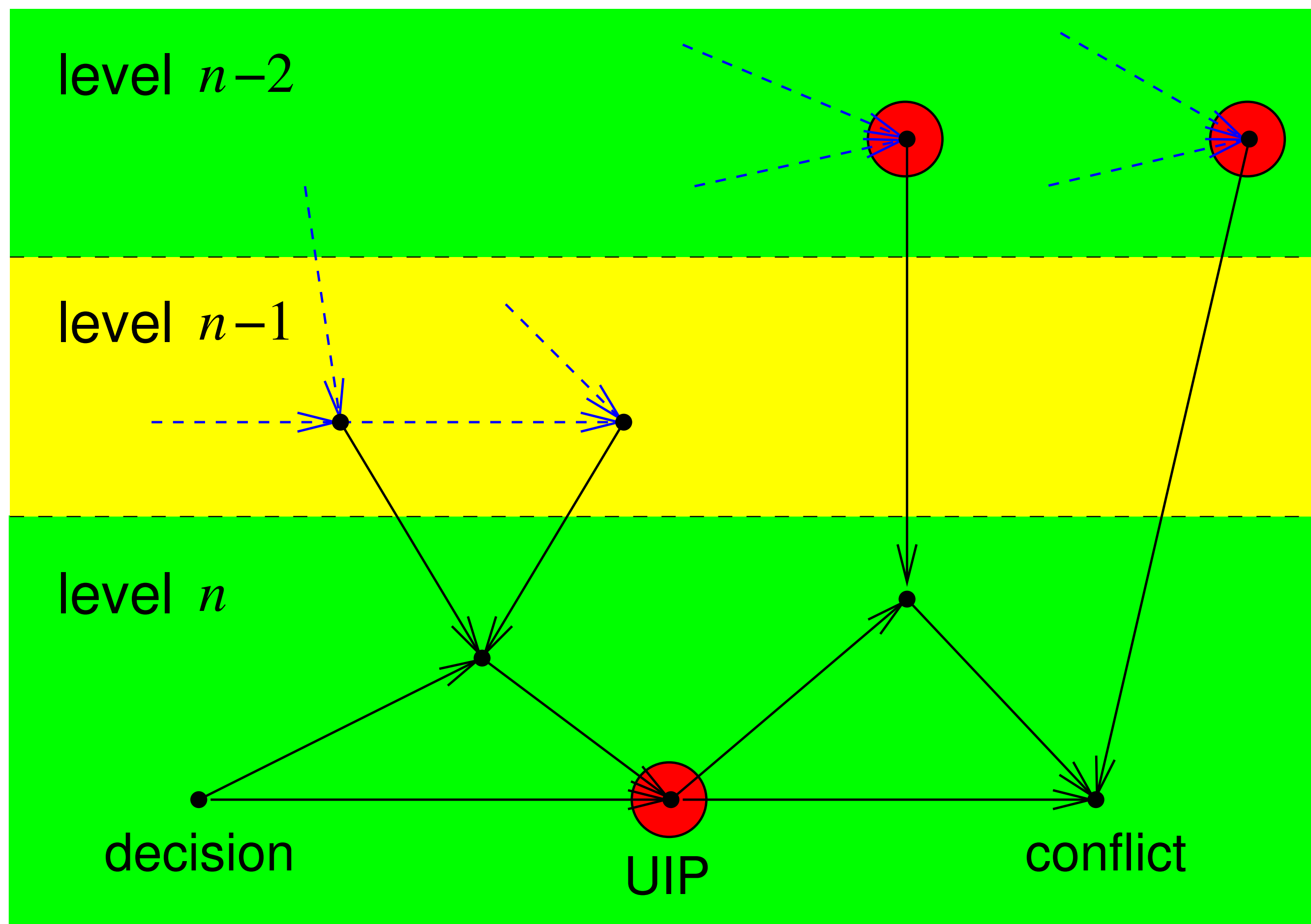
- traverse reasons of variables on trail starting with conflict

- count "open paths"
  (initially size of clause with only false literals)

- if all paths converged at one node, then UIP is found

- decision of current decision level is a UIP and thus a *sentinel*

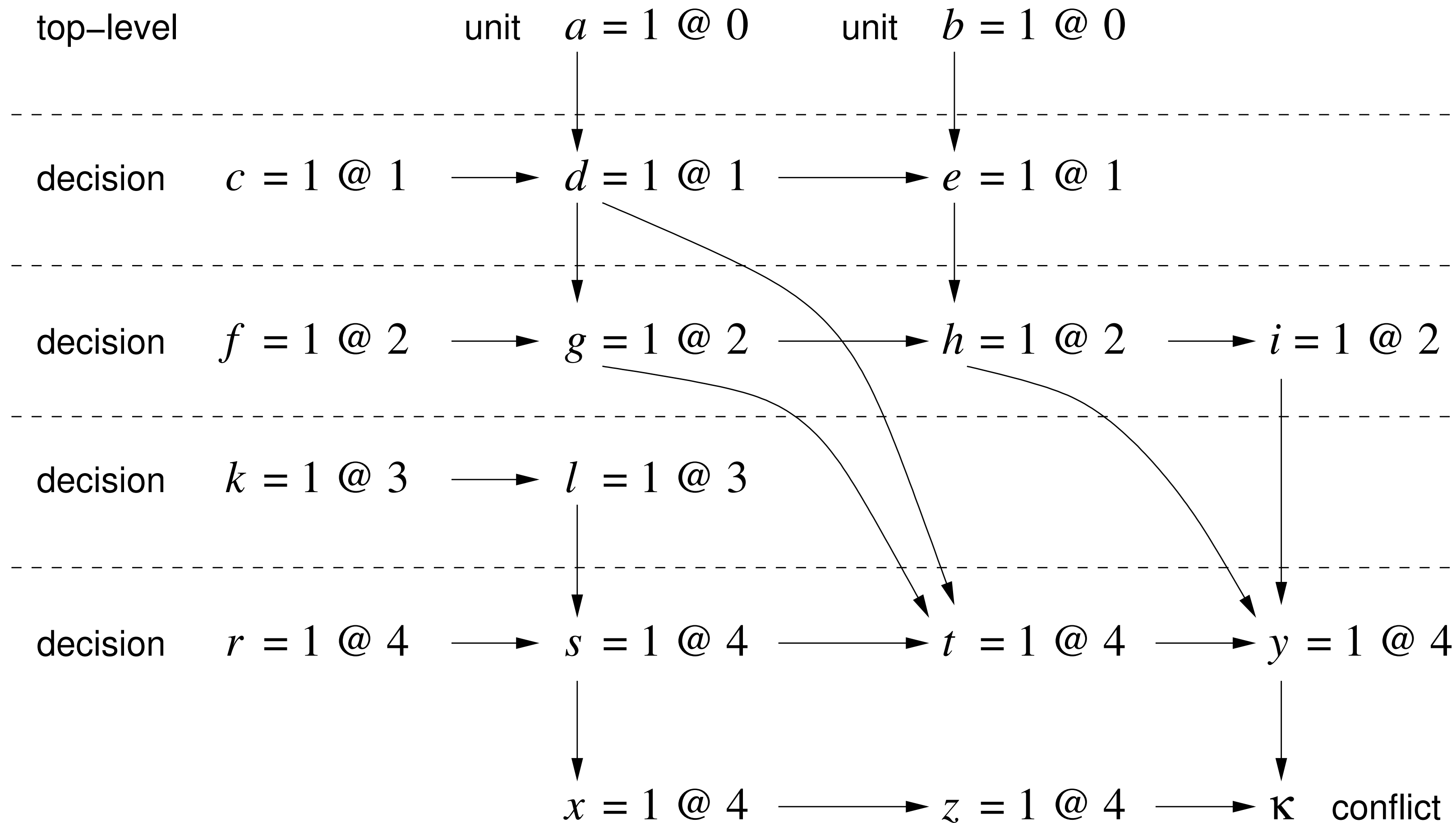
RiSE

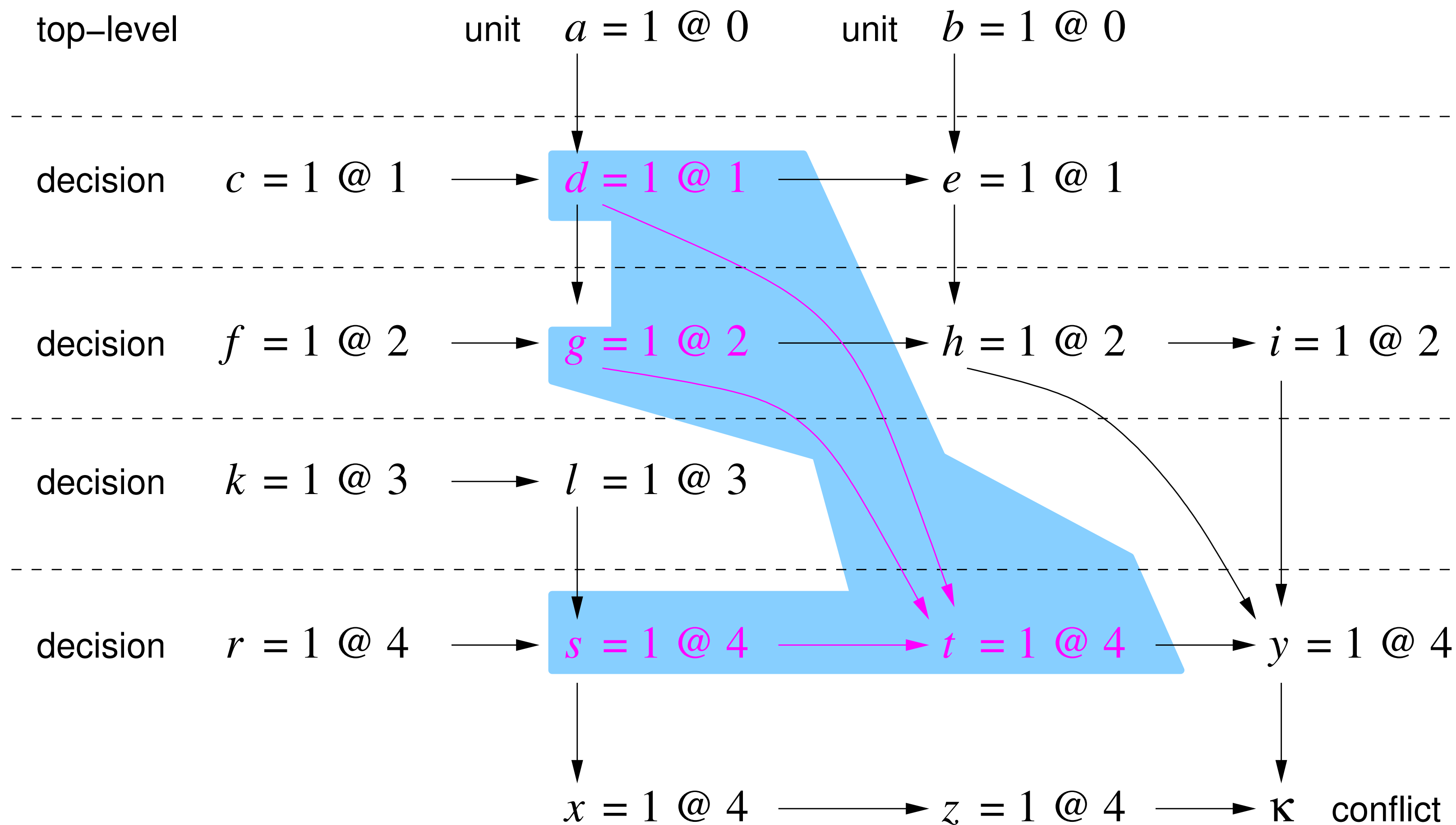


1st UIP learned clause increases chance of backjumping
("pulls in" as few decision levels as possible)

- intuitively is is important to localize the search (cf cutwidth heuristics)
- cuts for learned clauses may only include UIPs of current decision level
- on lower decision levels an arbitrary cut can be chosen
- multiple alternatives
  - include all the roots contributing to the conflict
  - find minimal cut (heuristically)
  - **cut off at first literal of lower decision level** (works best)

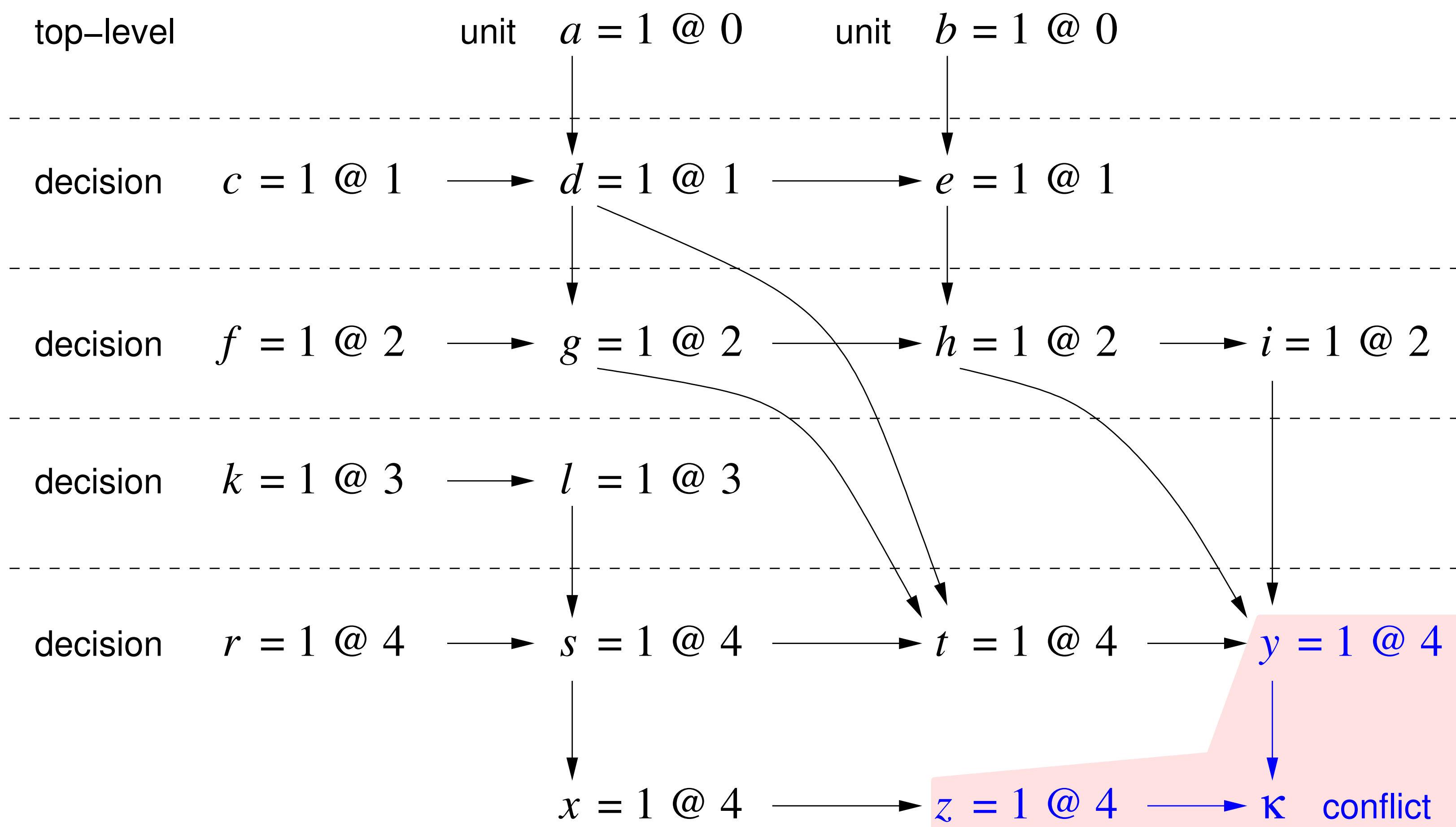
Implication Graph





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

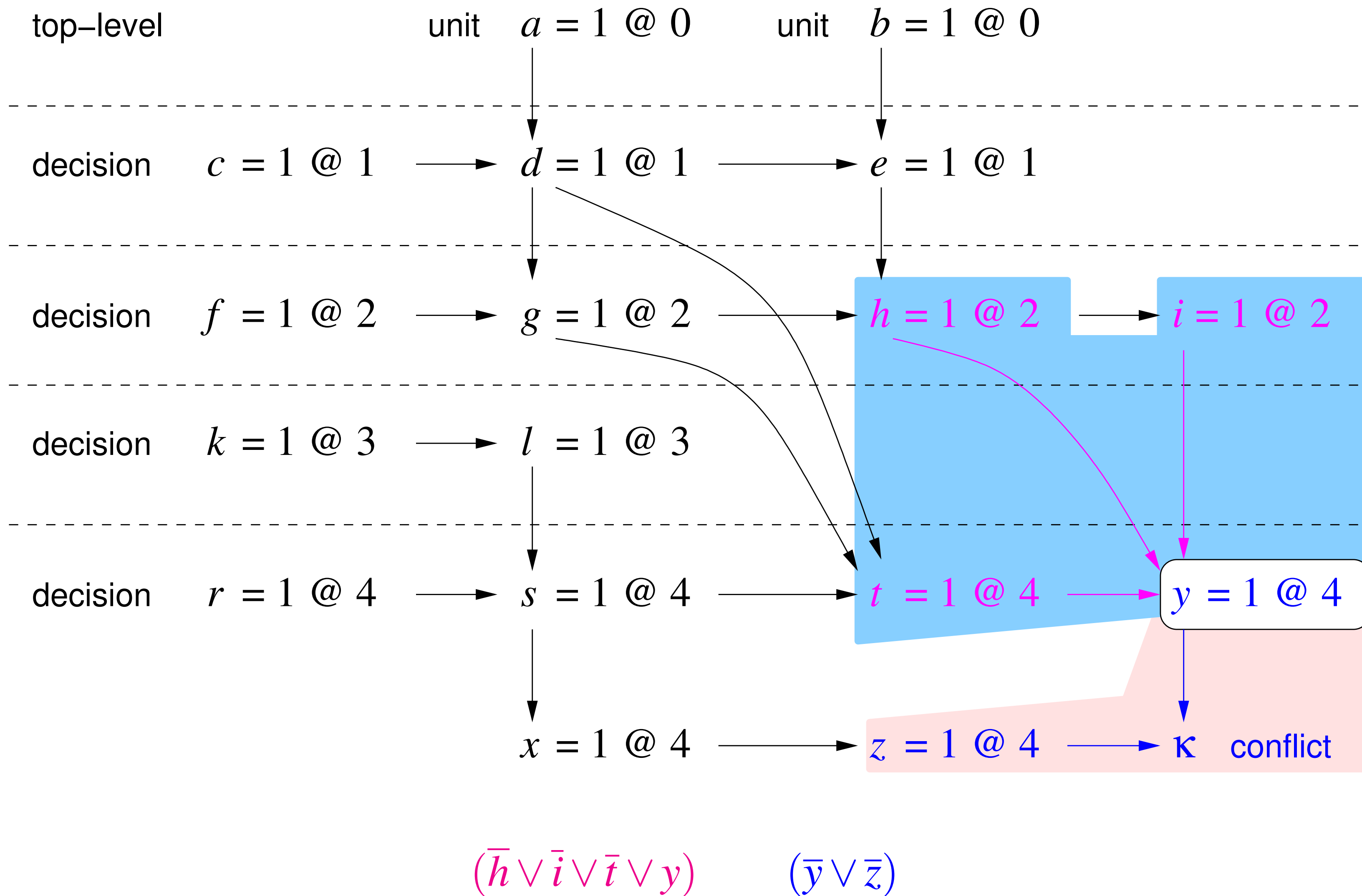
RiSE



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

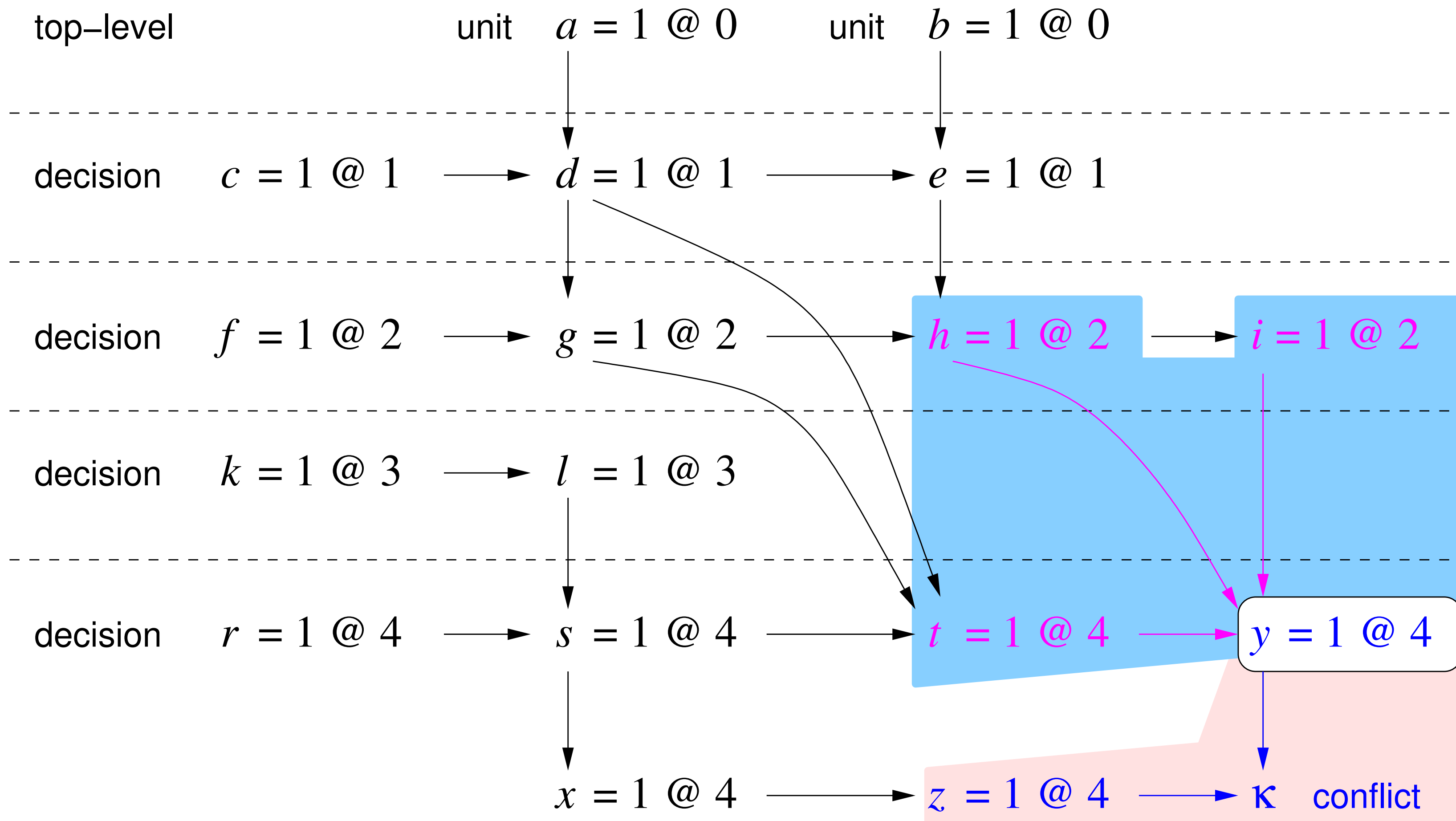
RiSE

Resolving Antecedents 1st Time



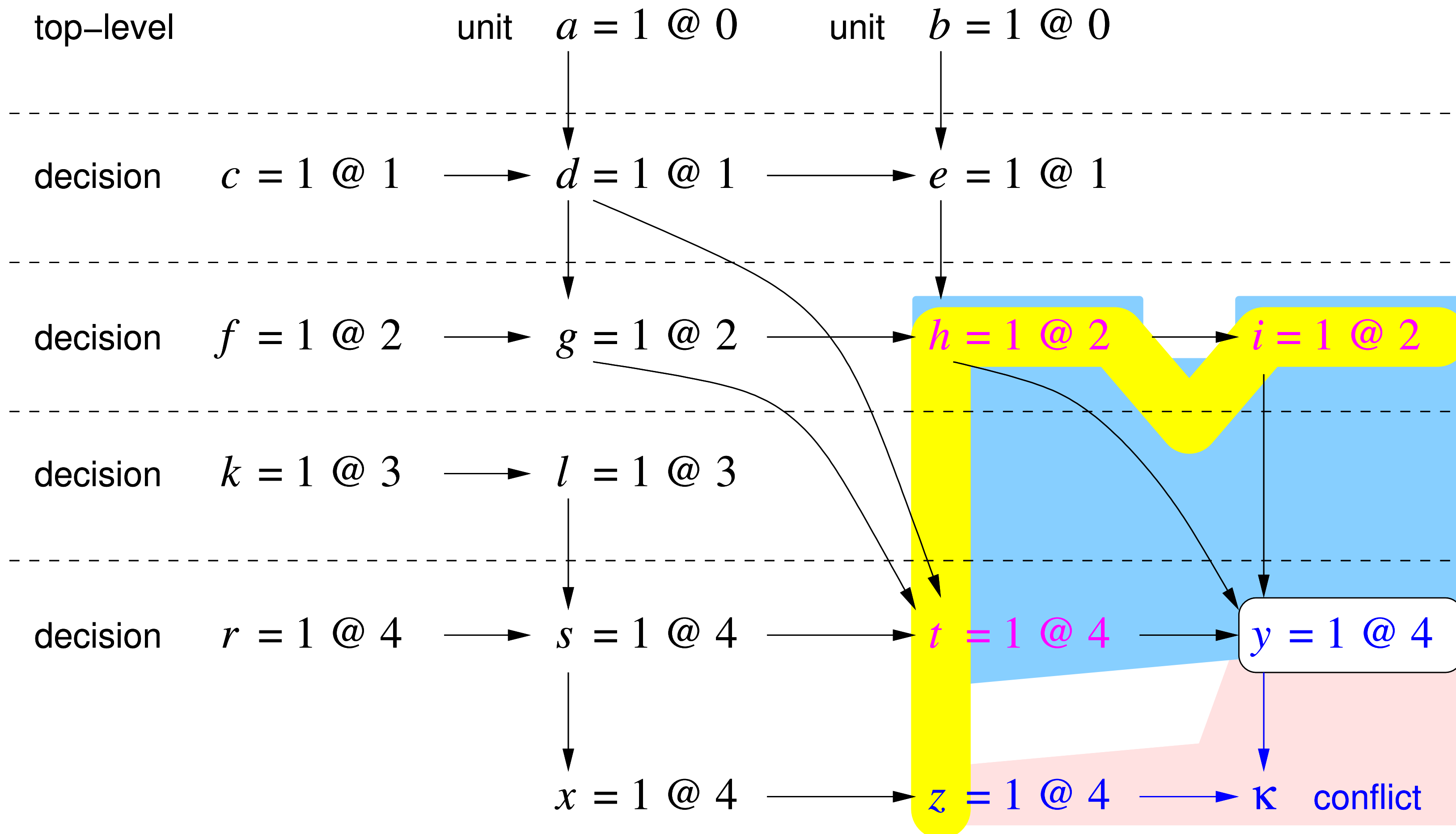
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

RiSE



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

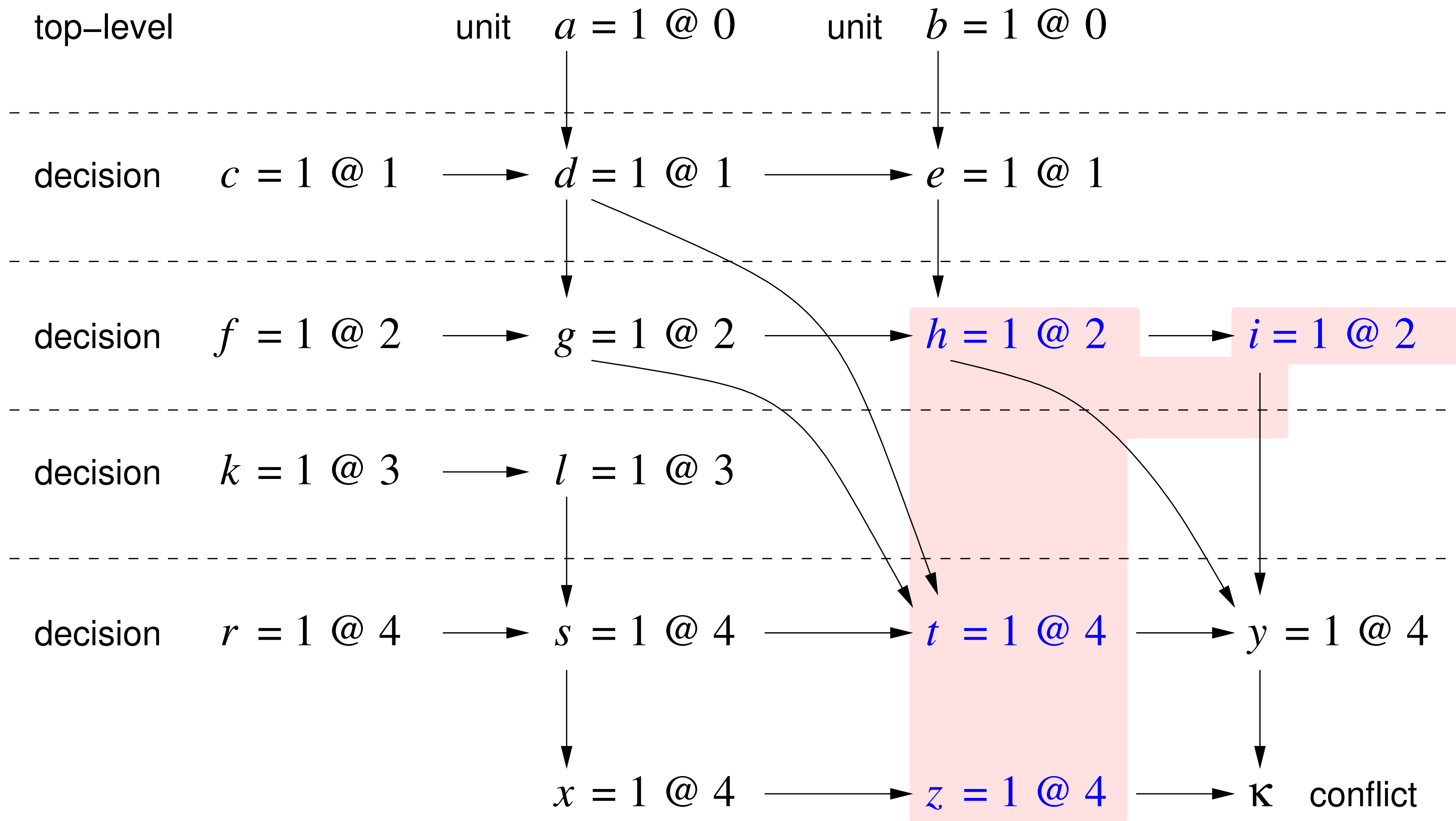
Resolvents = Cuts = Potential Learned Clauses



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

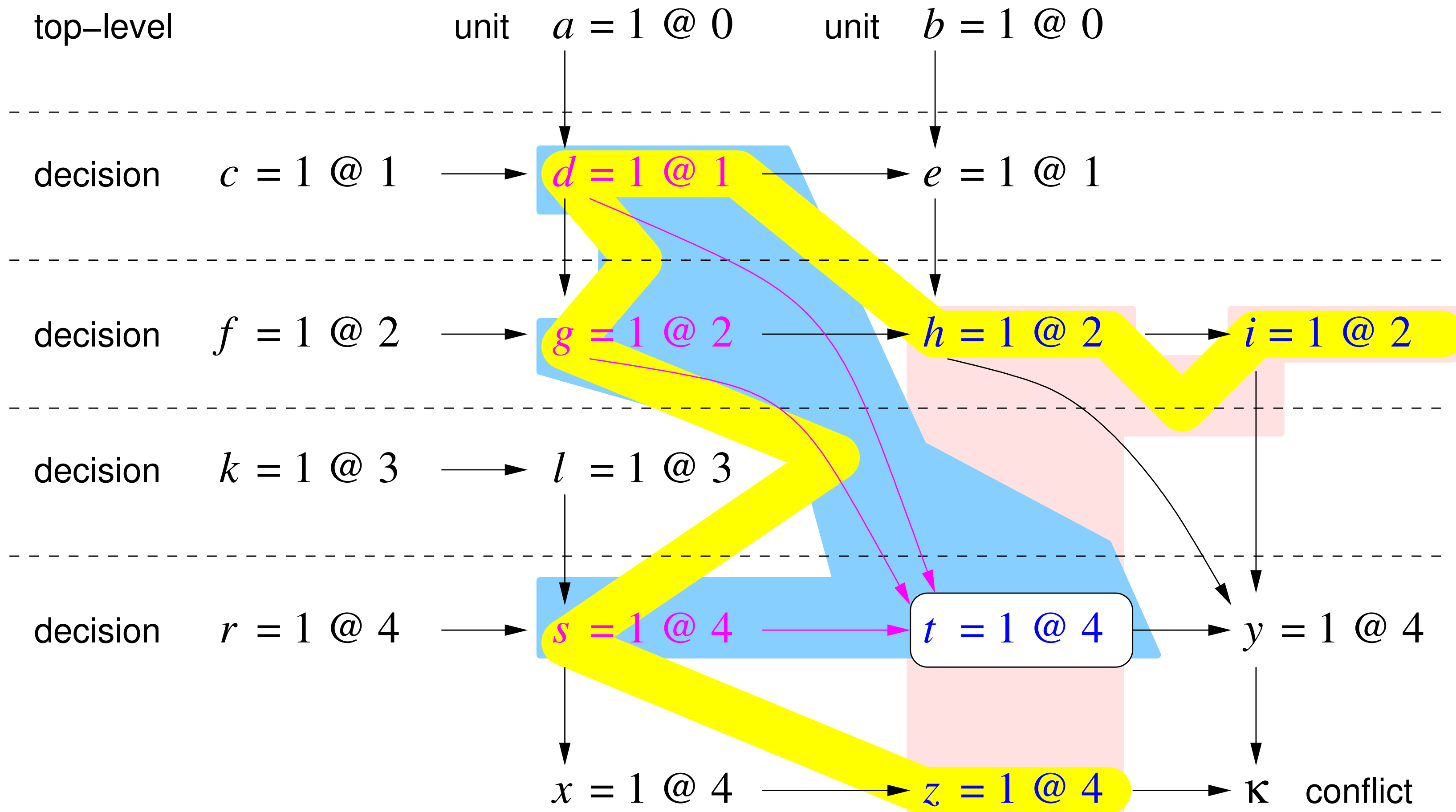
RiSE

Potential Learned Clause After 1 Resolution



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

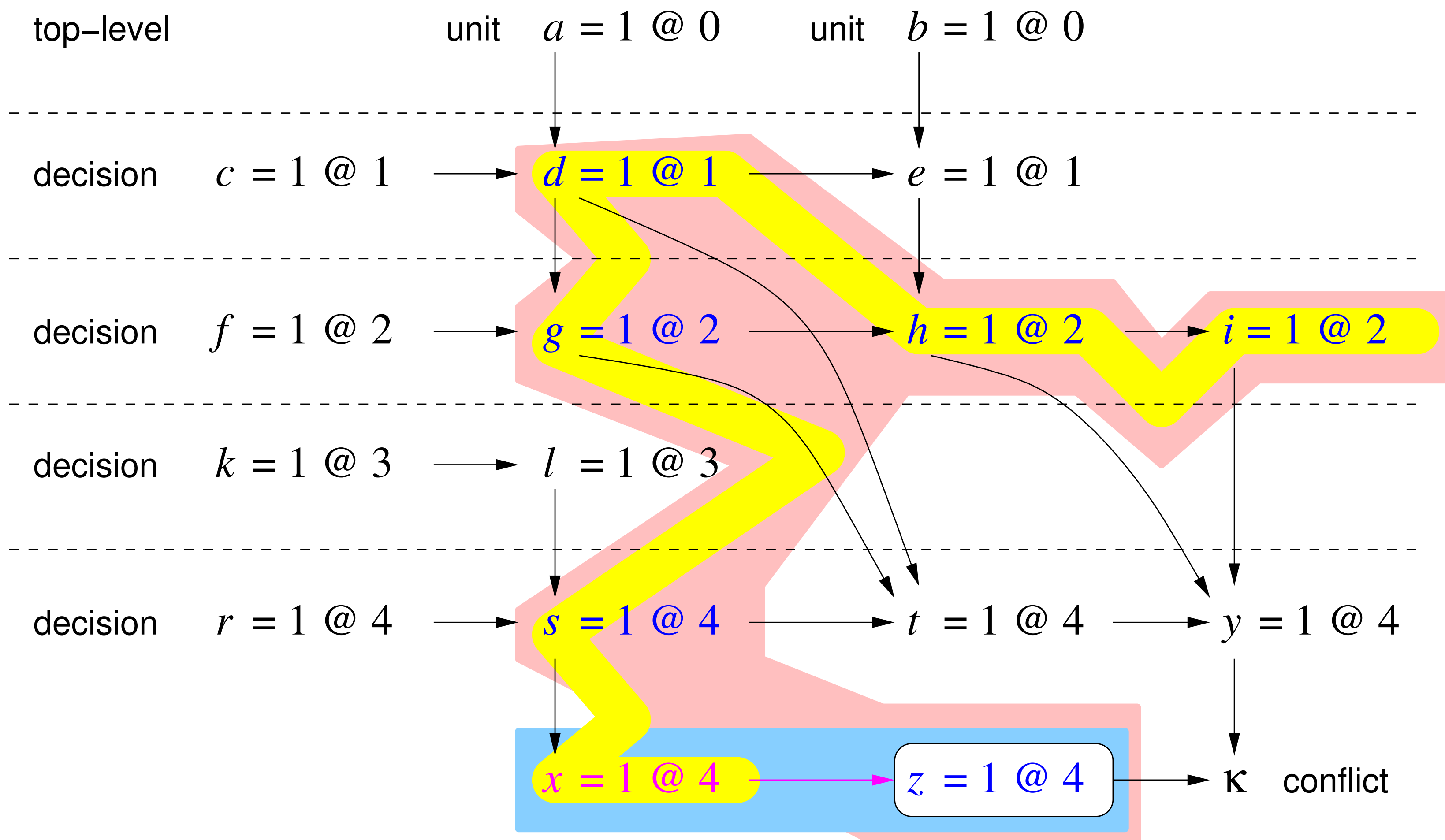
RiSE



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

RiSE

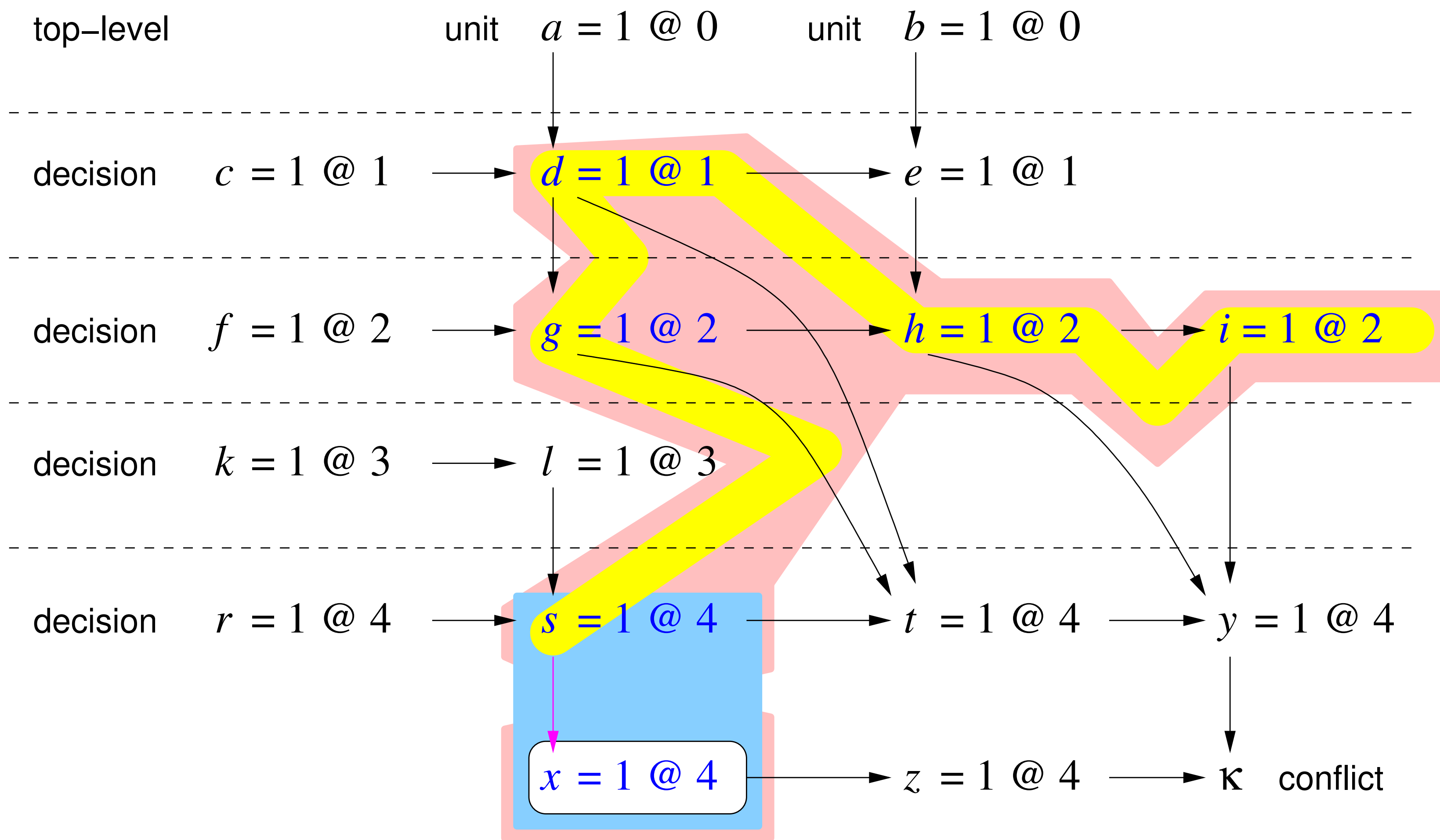
Resolving Antecedents 3rd Time



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

RiSE

Resolving Antecedents 4th Time

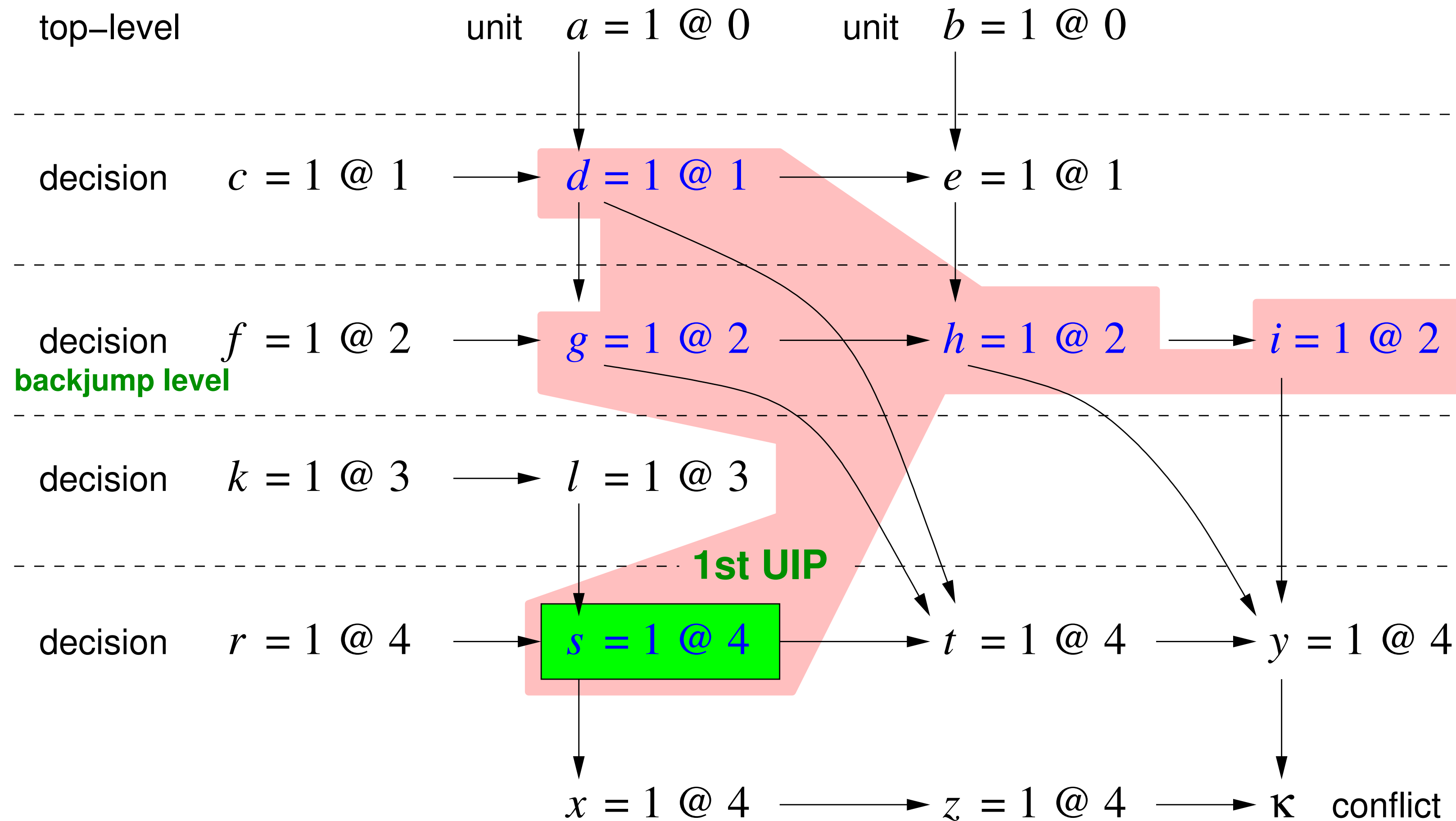


$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

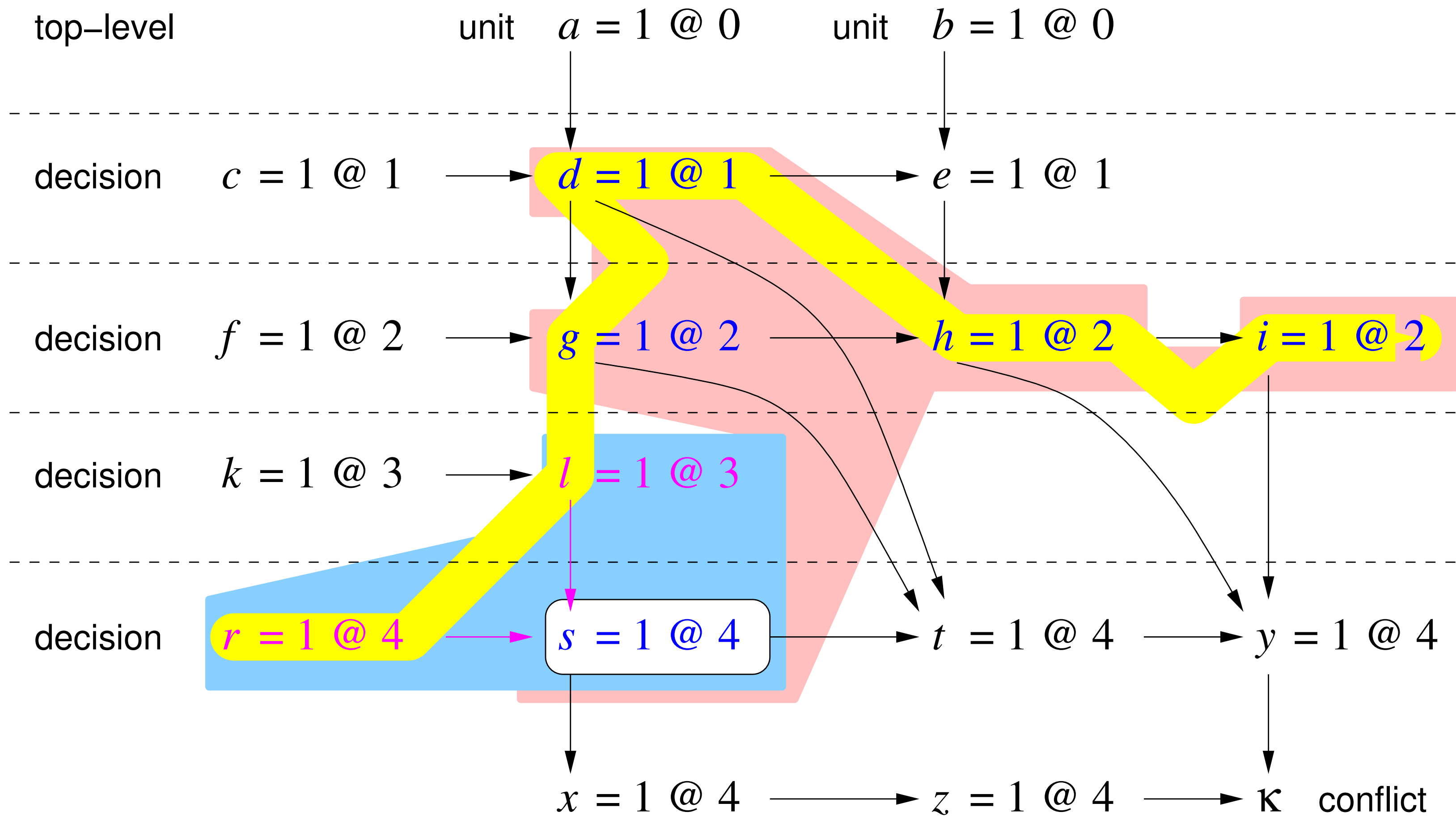
RiSE

1st UIP Clause after 4 Resolutions



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

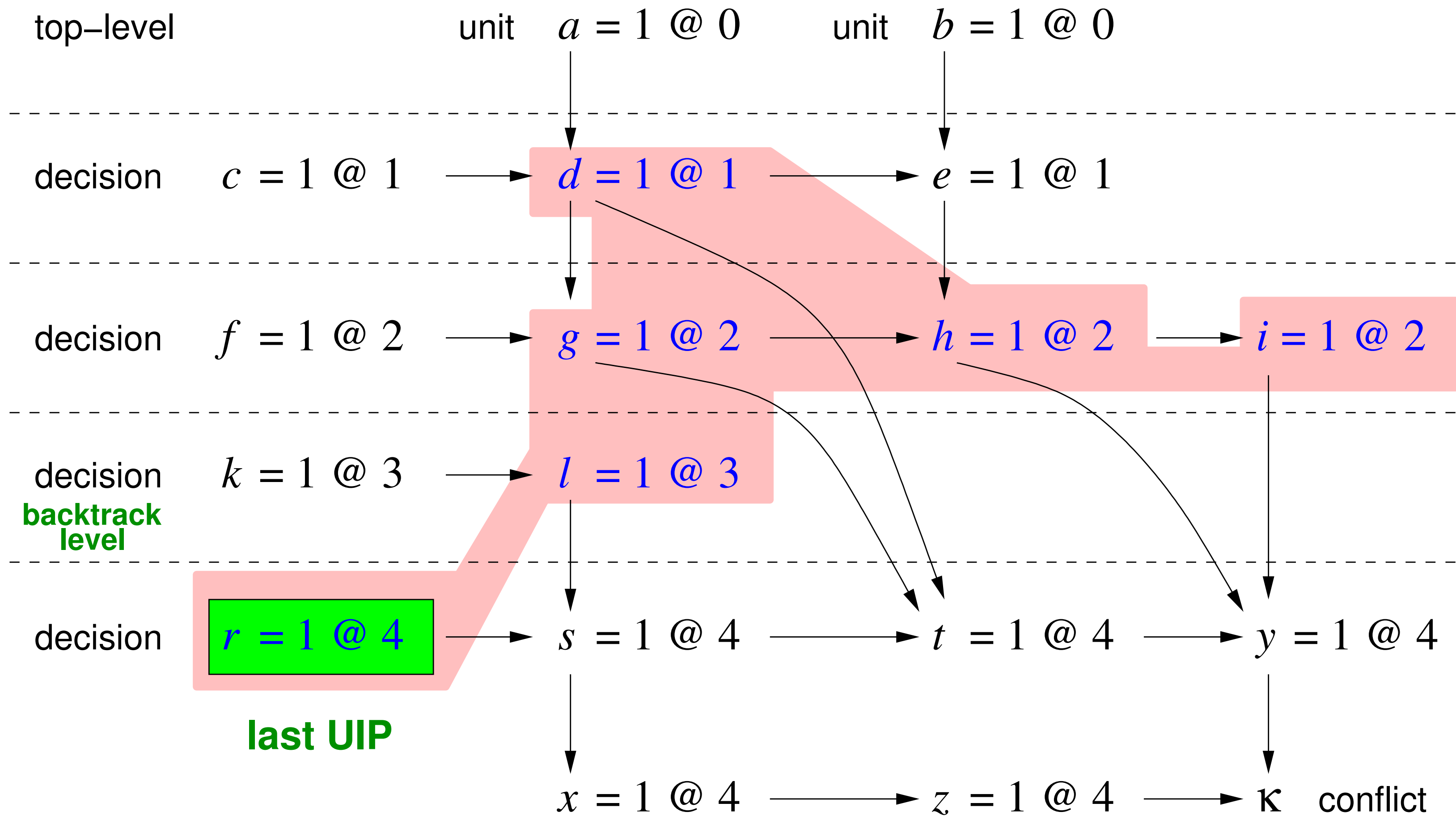
RiSE



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

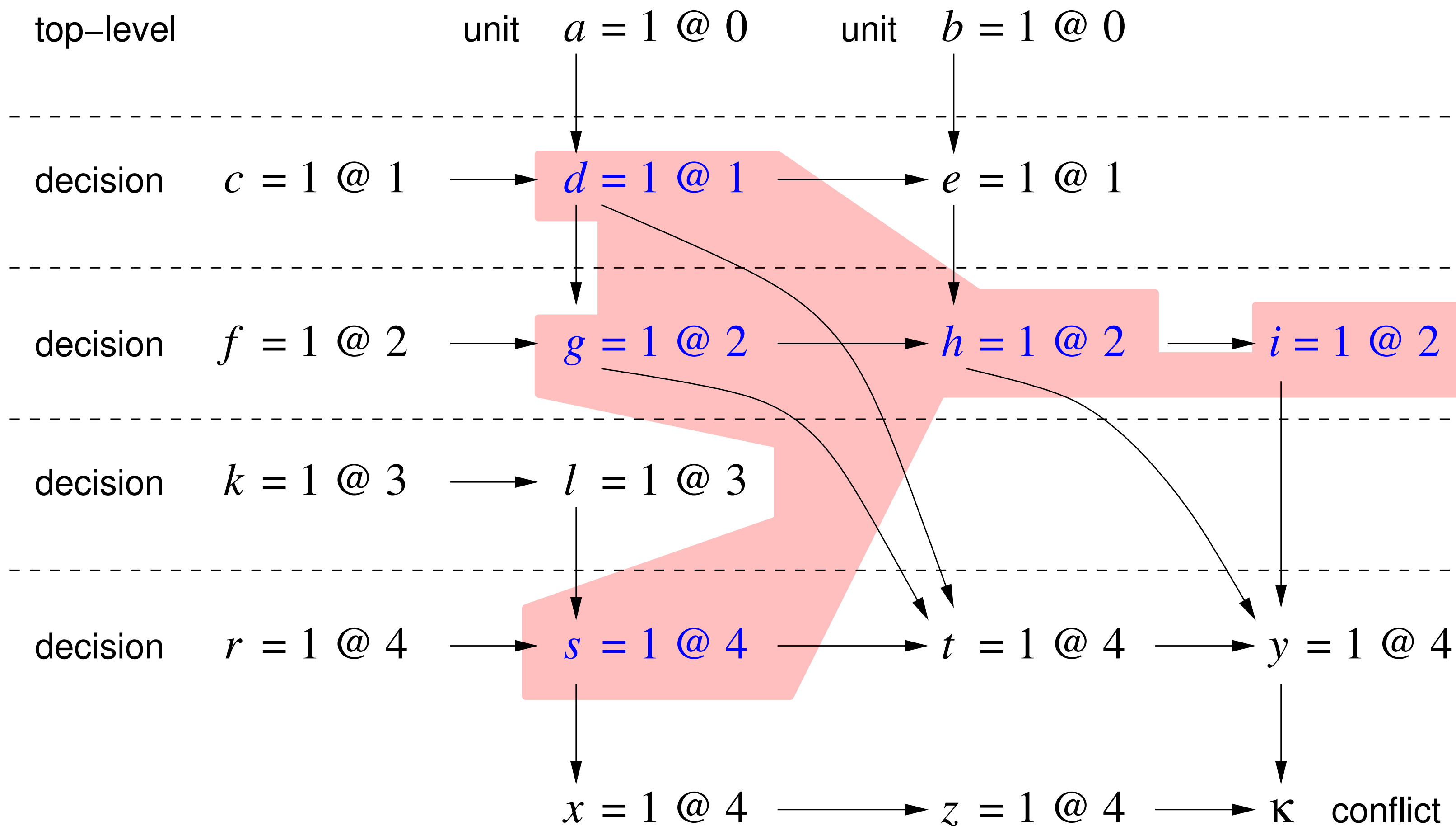
RiSE

Decision Learned Clause



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

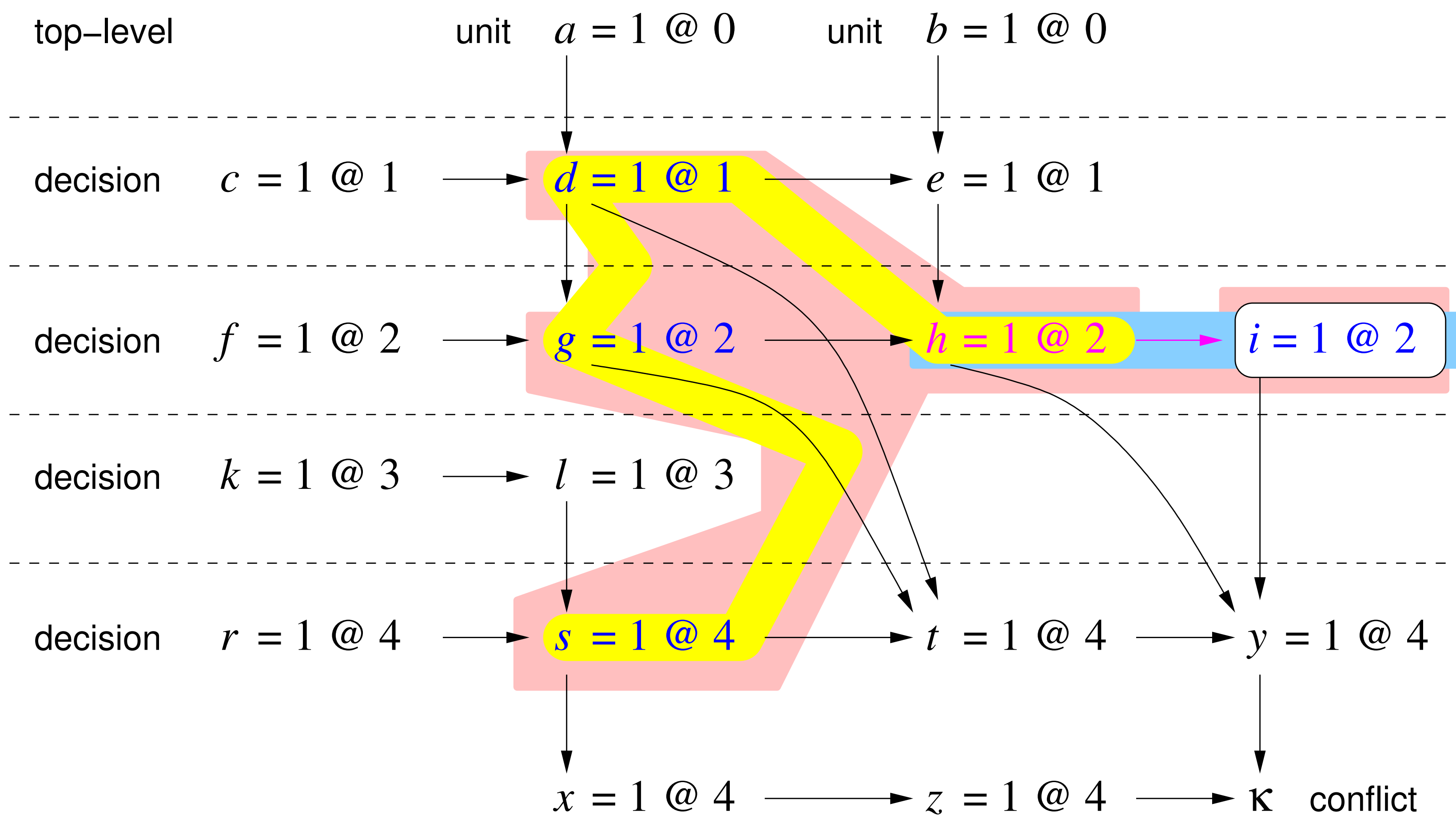
1st UIP Clause after 4 Resolutions



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

RiSE

Locally Minimizing 1st UIP Clause

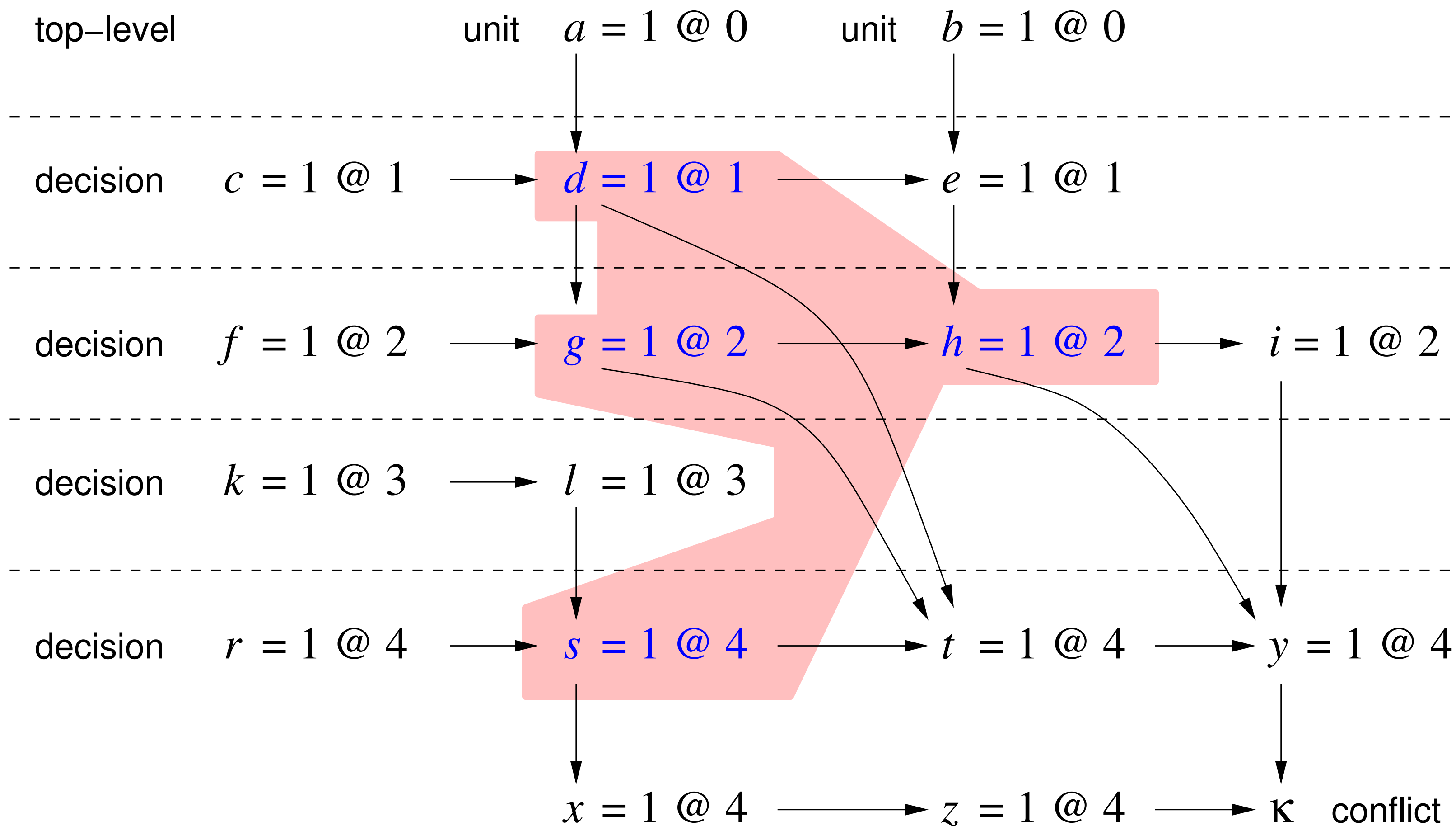


$$\frac{(\overline{h} \vee i) \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})}{(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})}$$

self subsuming resolution

RiSE

Locally Minimized Learned Clause



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

Two step algorithm:

1. mark all variables in 1st UIP clause

2. remove literals with all antecedent literals also marked

Correctness:

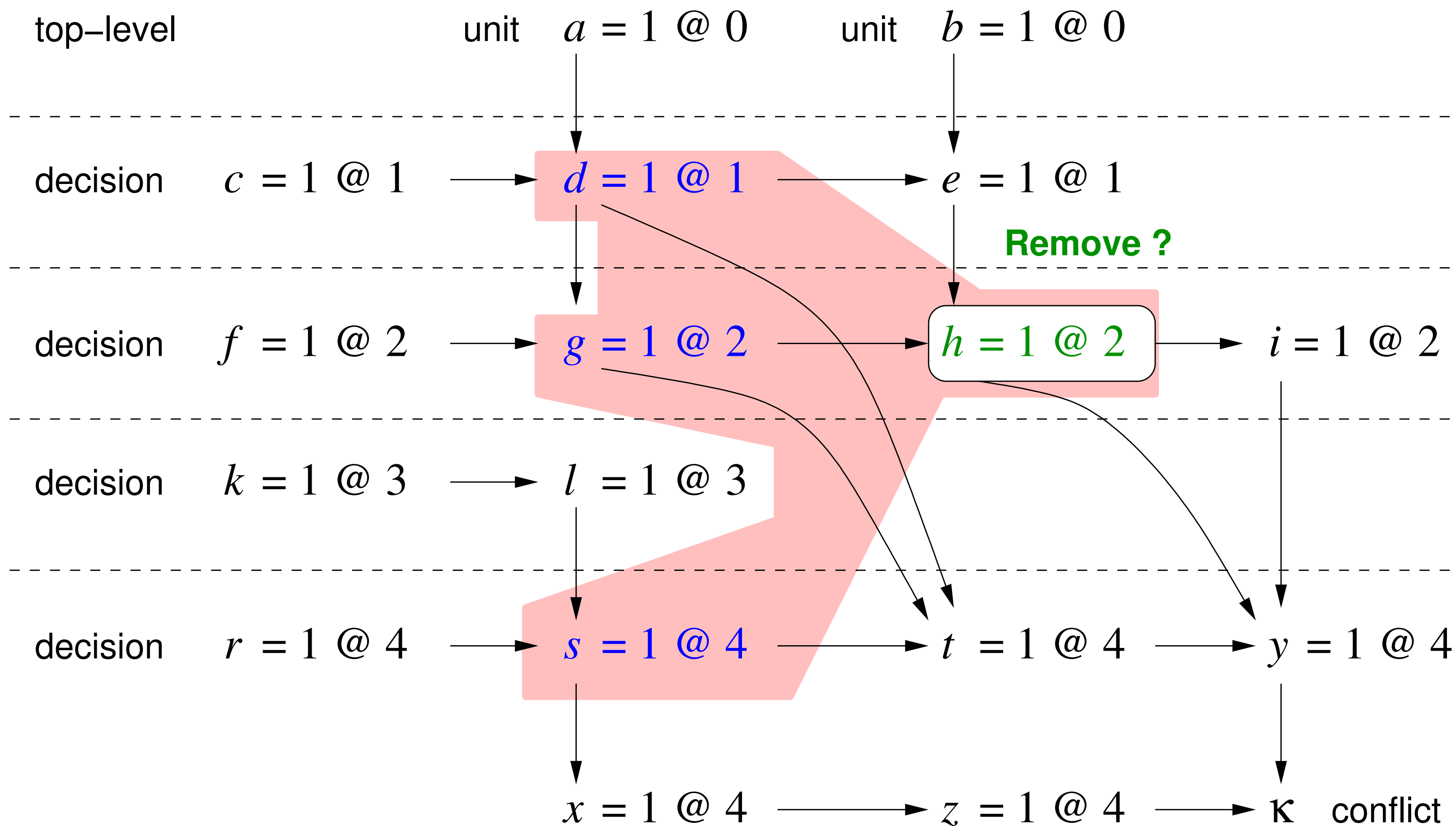
- removal of literals in step 2 are self subsuming resolution steps.

- implication graph is acyclic.

Confluence: produces a unique result.

RiSE

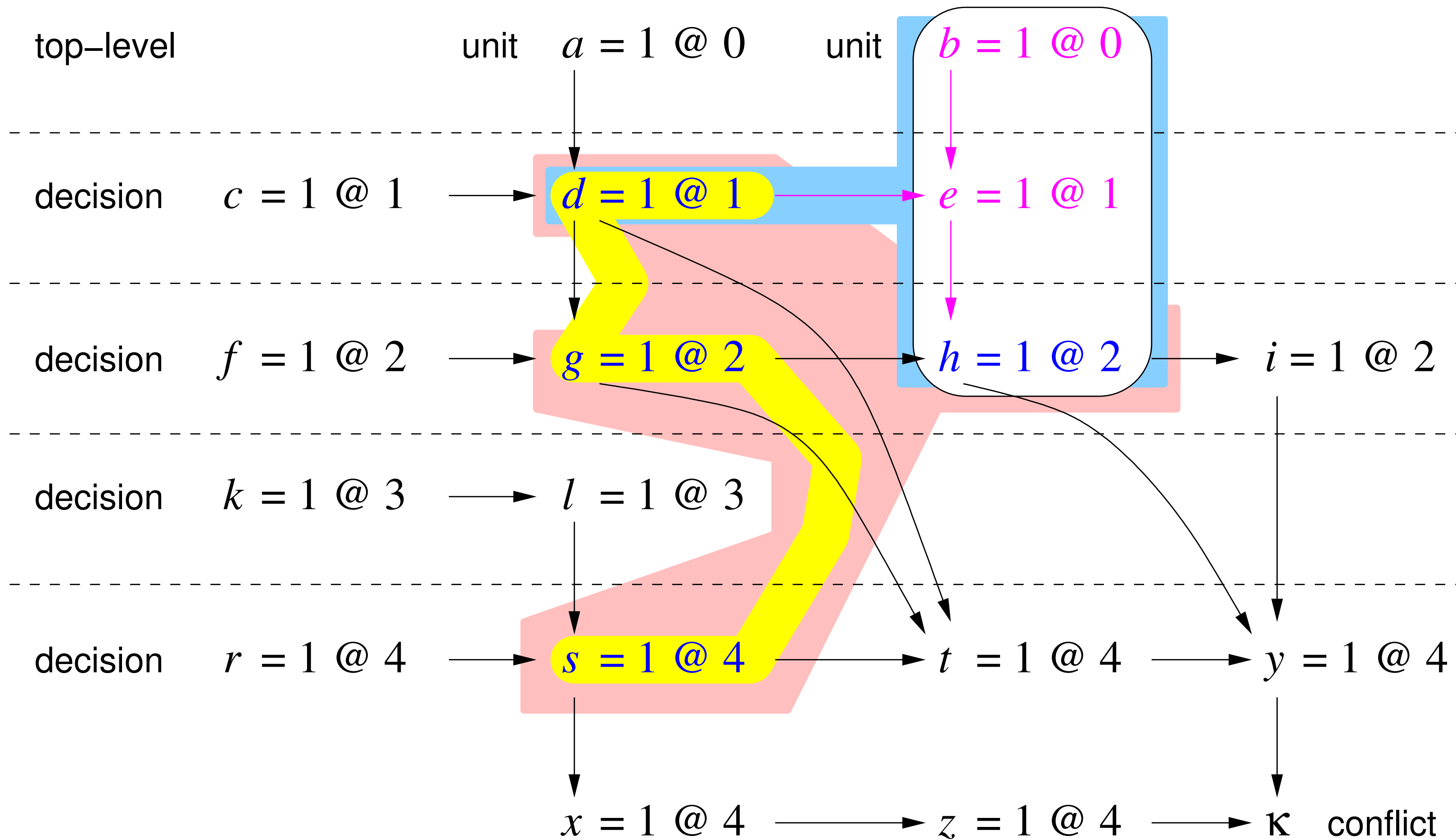
Minimizing Locally Minimized Learned Clause Further?



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

RiSE

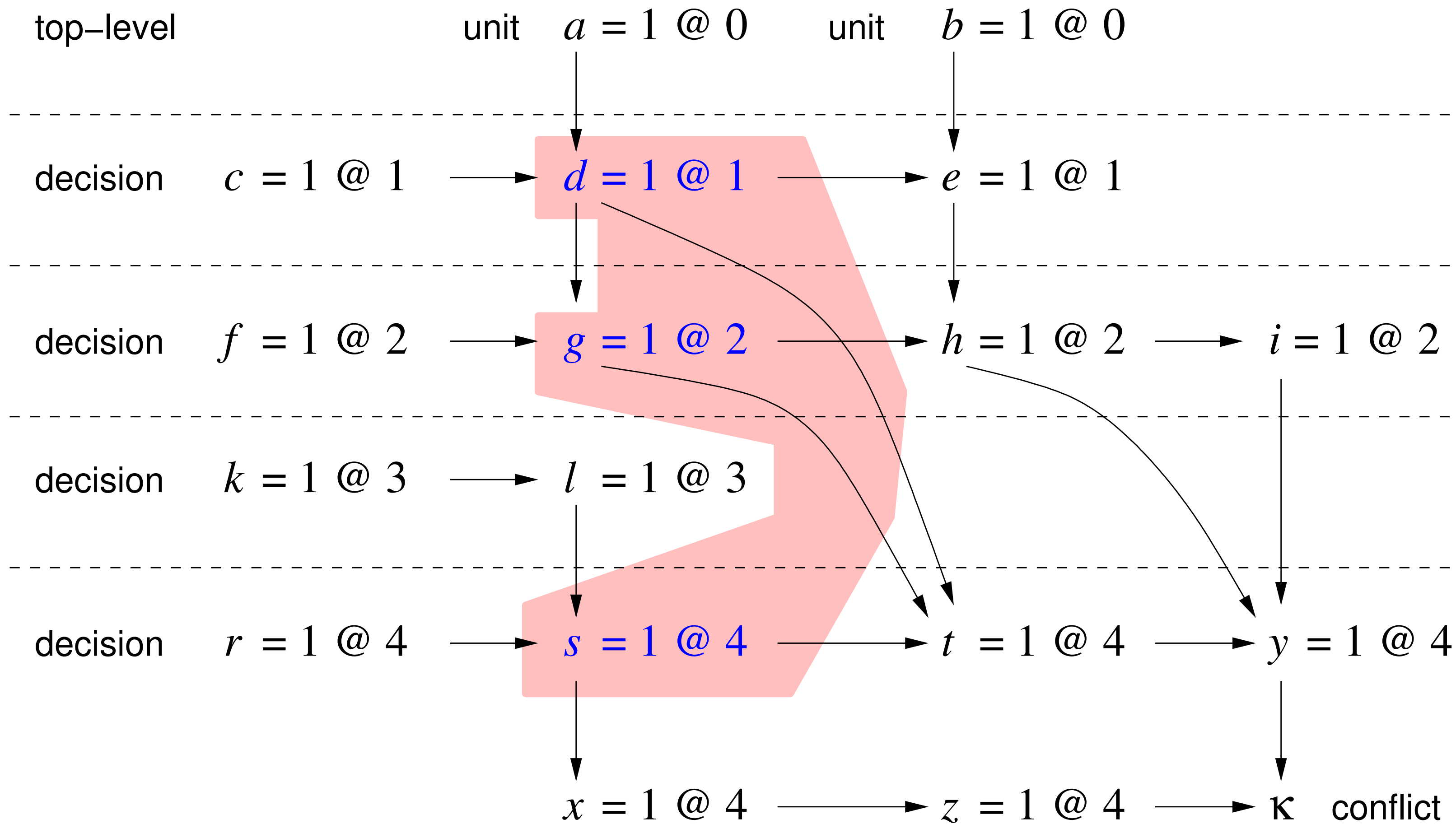
Recursively Minimizing Learned Clause



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

RiSE

Recursively Minimized Learned Clause



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

RiSE

Recursive Minimization Algorithm

[MiniSAT 1.13]

Four step algorithm:

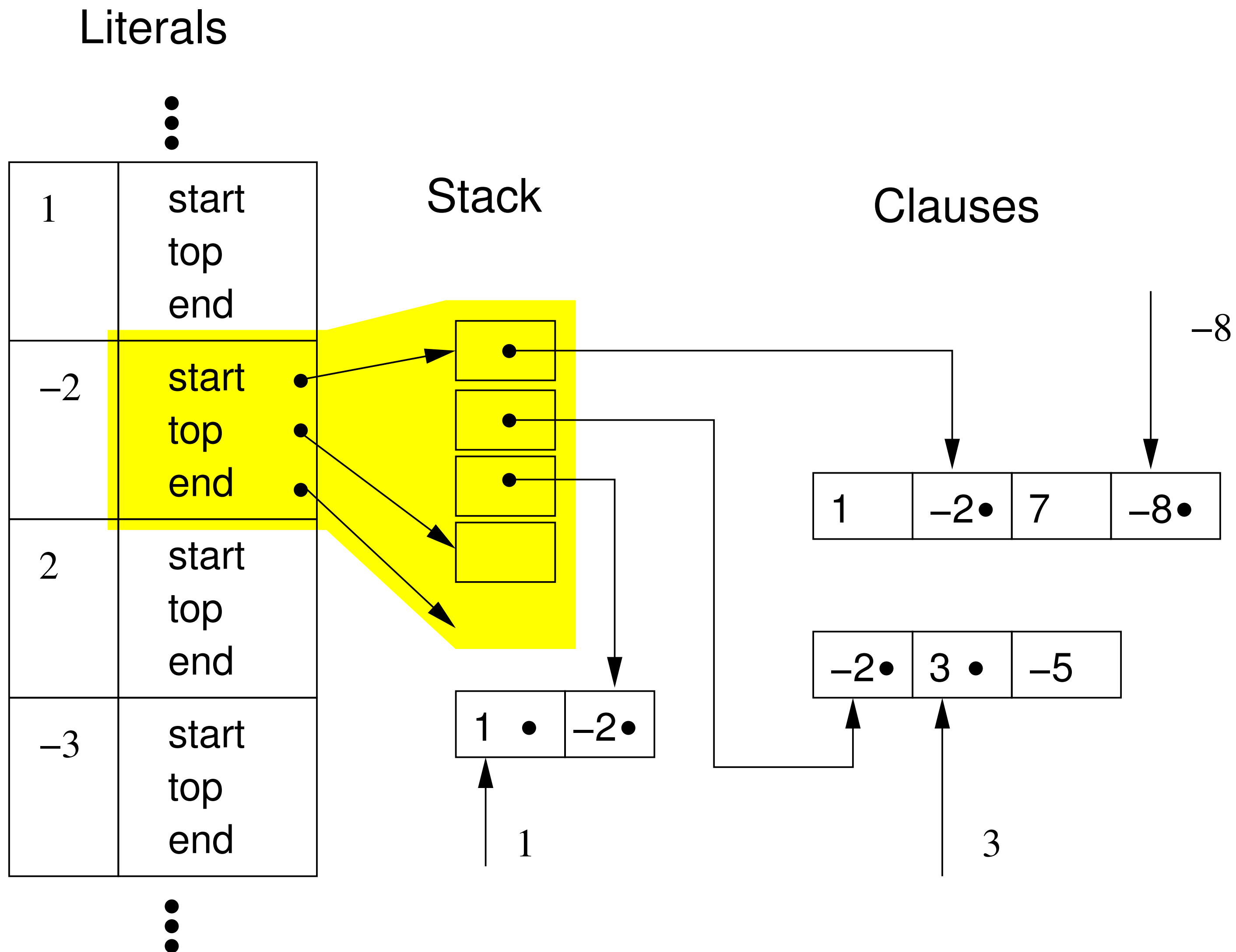
1. mark all variables in 1st UIP clause
2. for each candidate literal: search implication graph
3. start at antecedents of candidate literals
4. if search always terminates at marked literals remove candidate

Correctness and Confluence as in local version!!!

Optimization: terminate early with failure if new decision level is “pulled in”

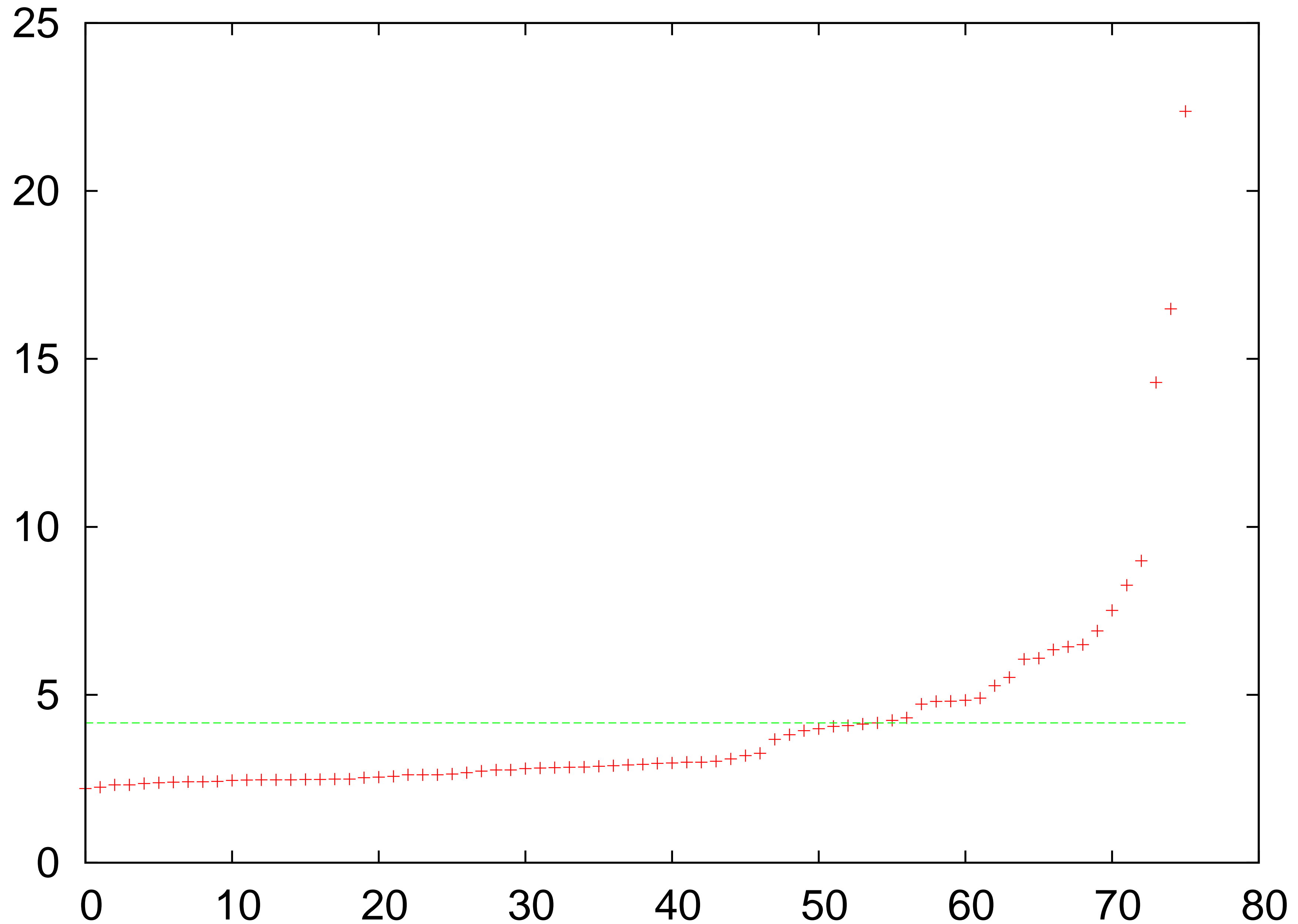
RiSE

ZChaff Occurrence Stacks



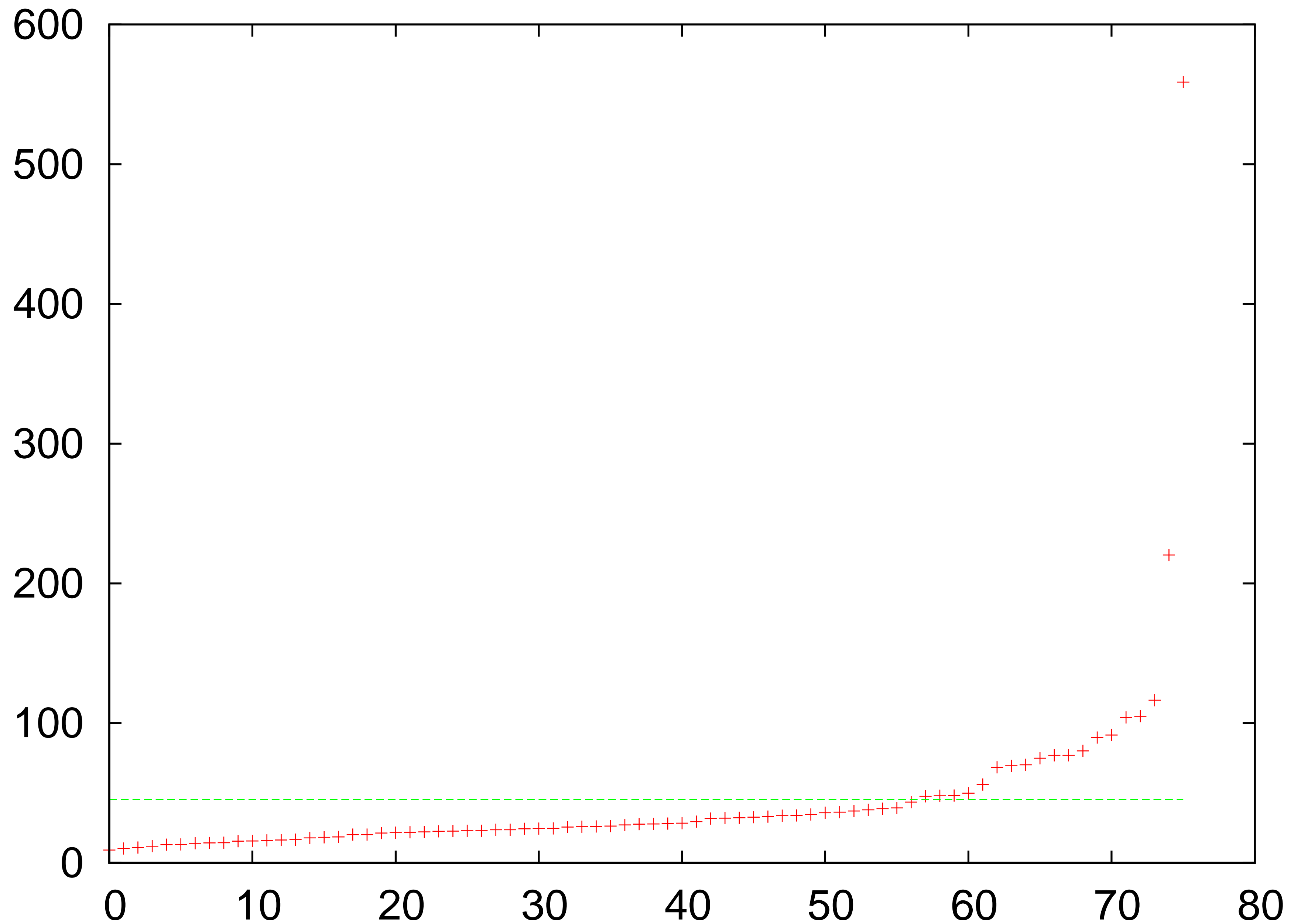
RiSE

Average Number Clauses Visited Per Propagation

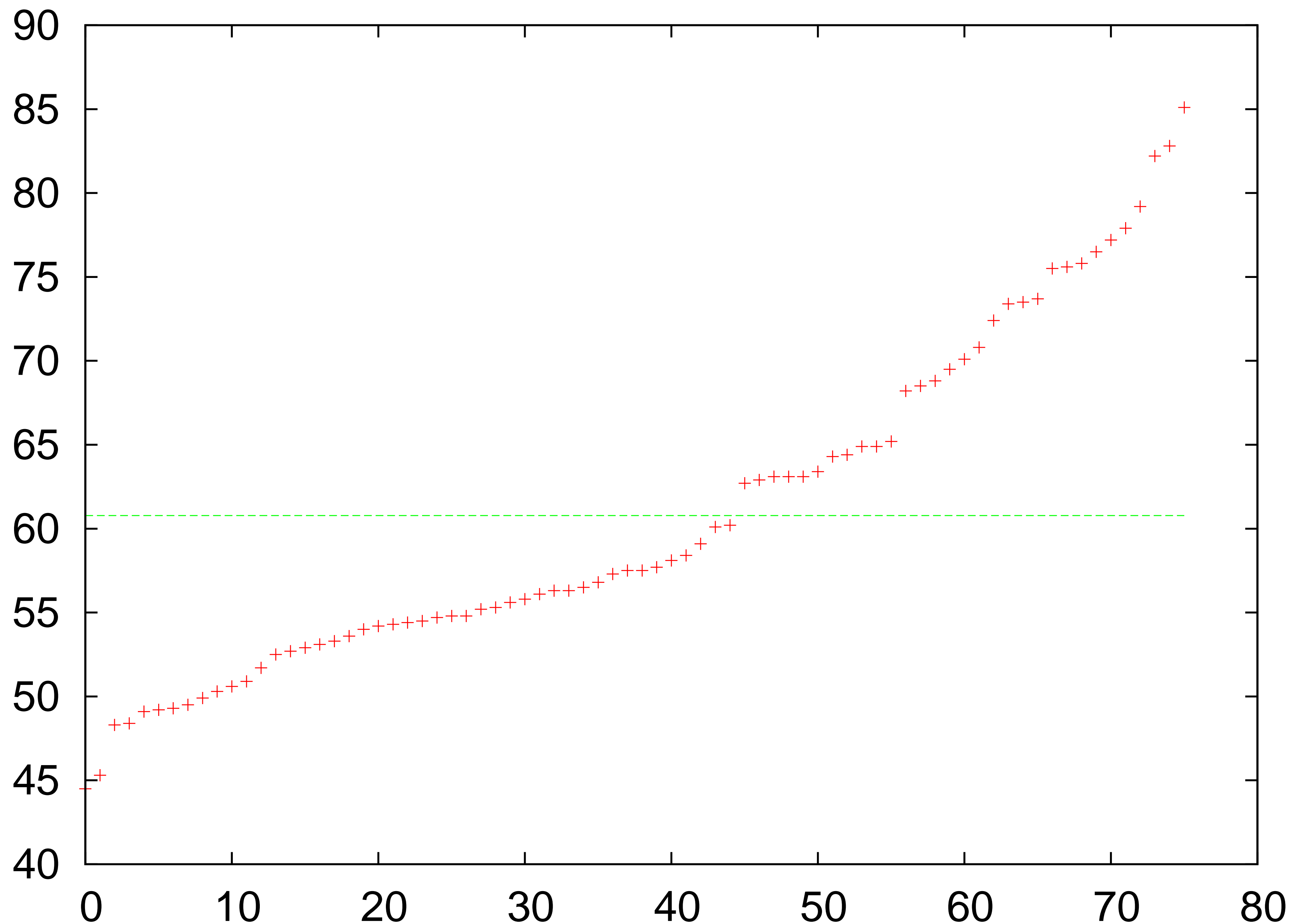


RiSE

Average Learned Clause Length

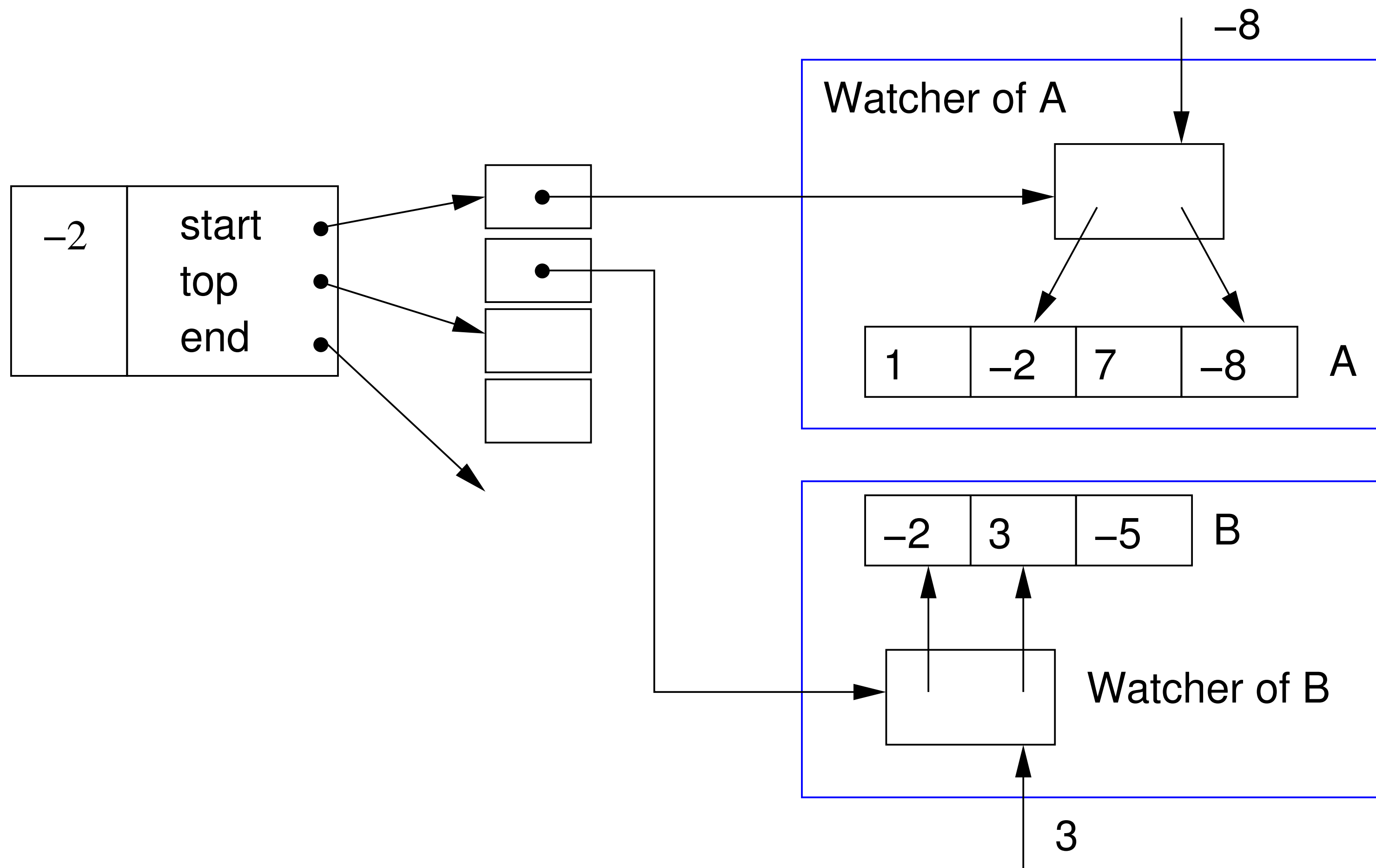


Percentage Visited Clauses With Other Watched Literal True



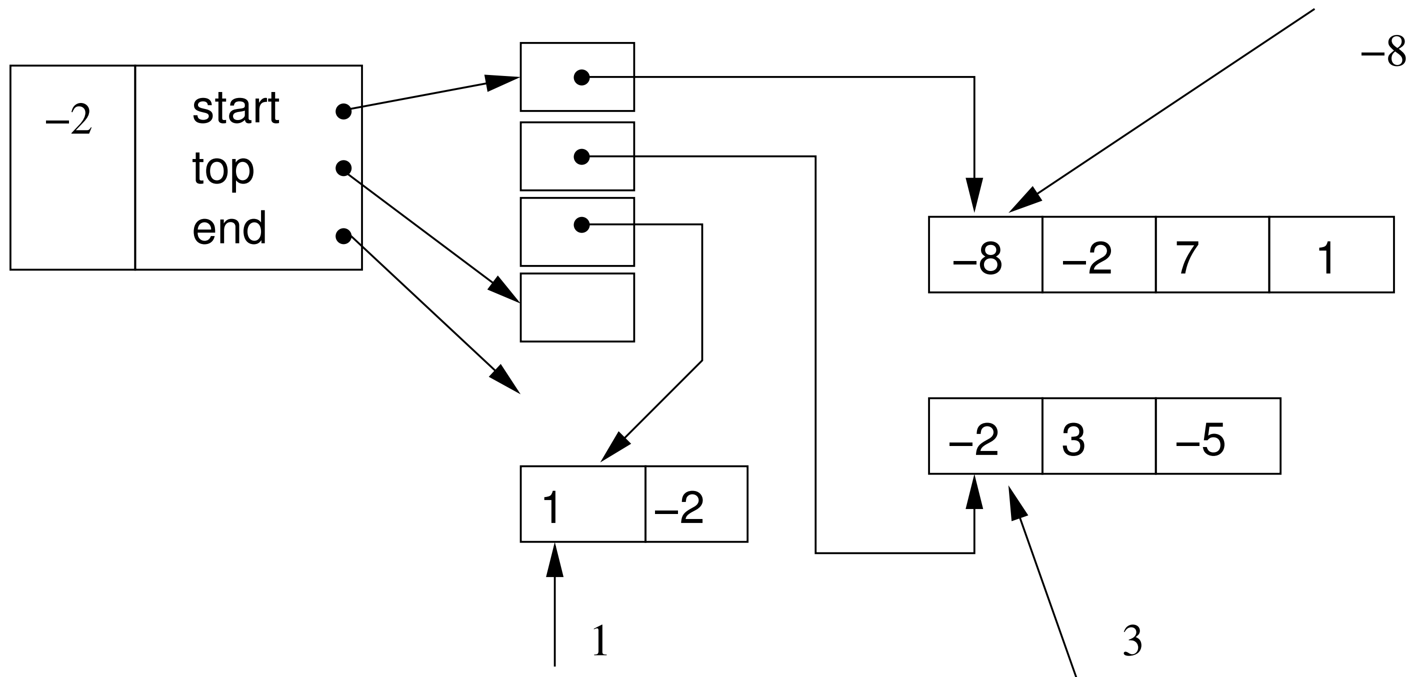
RiSE

Limmat / FunEx Occurrence Stacks



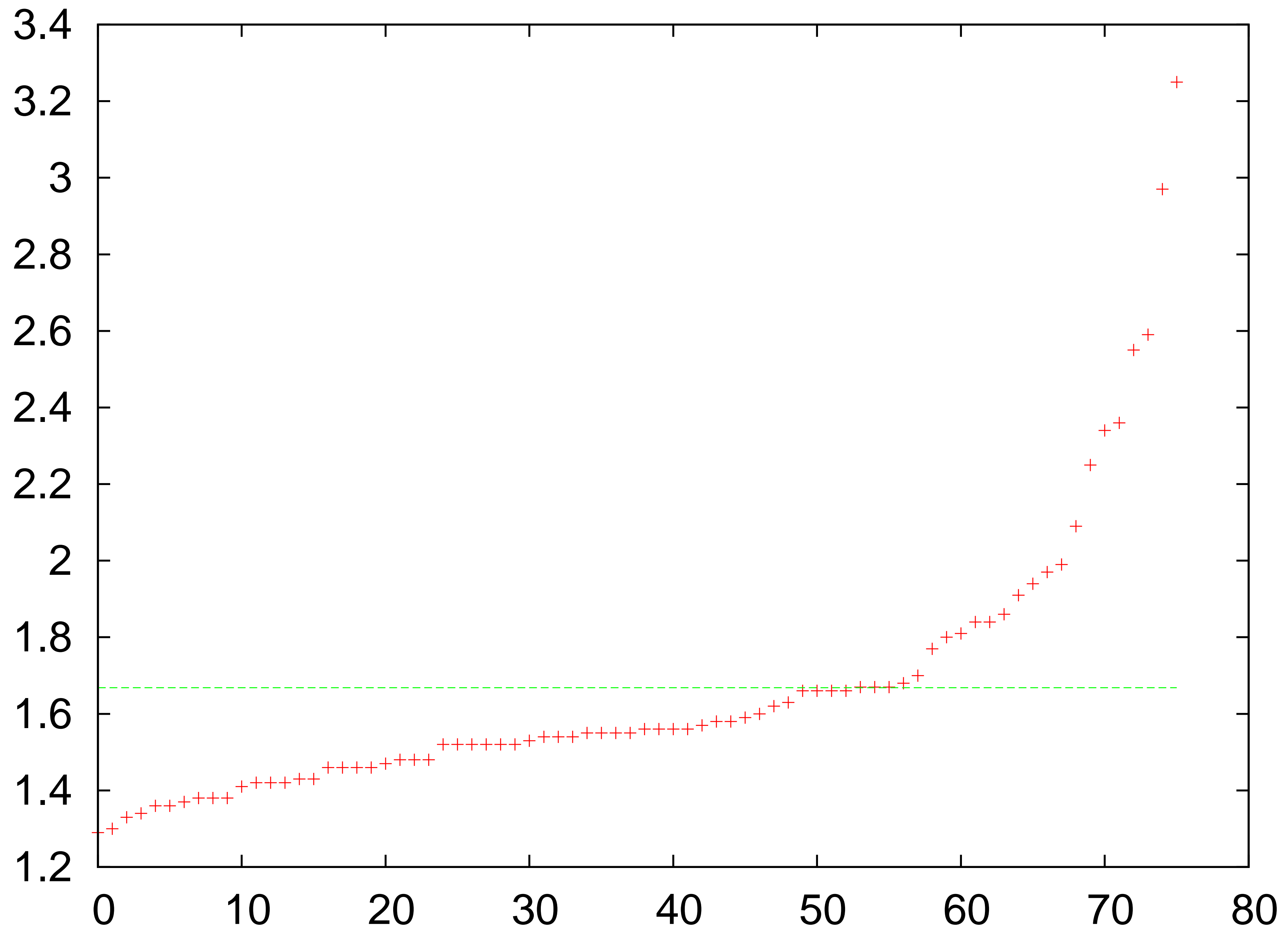
RiSE

CompSAT / MiniSAT Occurrence Stacks



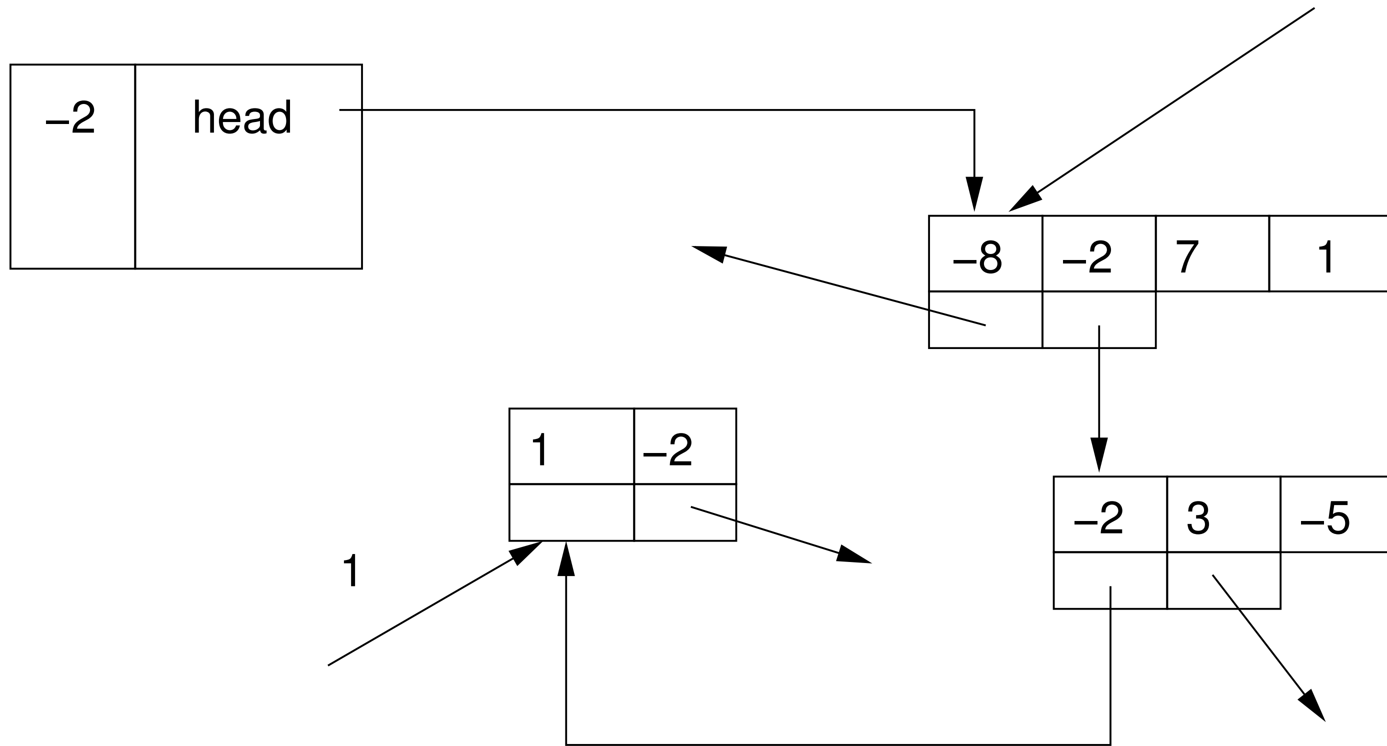
invariant: first two literals are watched

Average Number Literals Traversed Per Visited Clause



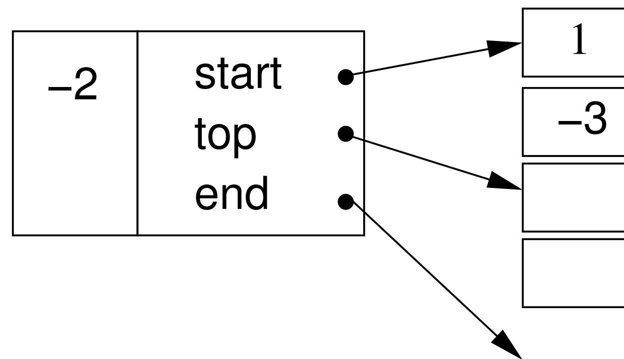
RiSE

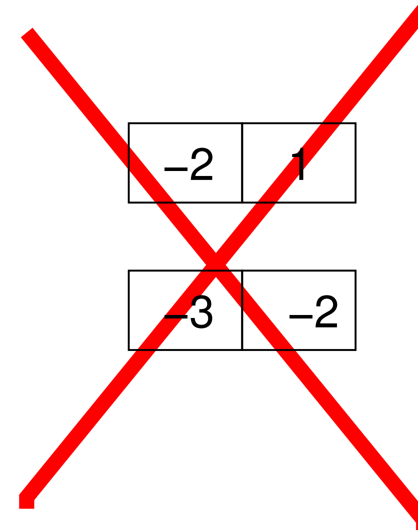
MChaff / PicoSAT Occurrence Lists



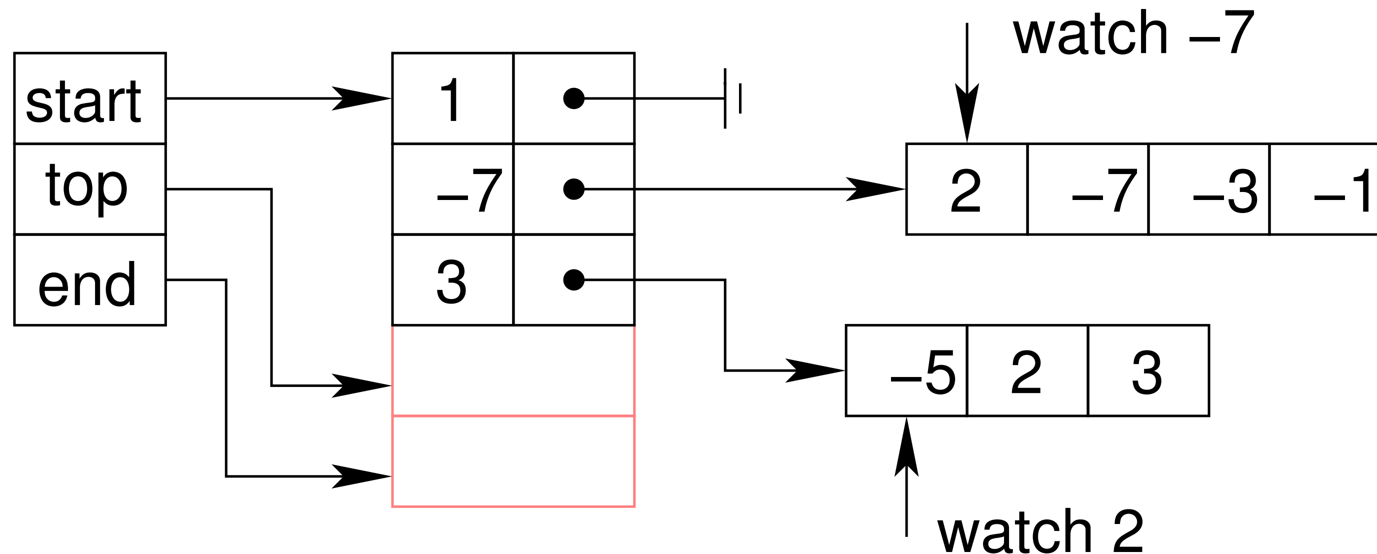
invariant: first two literals are watched

Additional Binary Clause Watcher Stack





Caching Potential Satisfied Literals (Blocked Literals)



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

so cache this literals in watch list to avoid pointer dereference

for binary clause no need to store clause at all

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

LINGELING uses more compact pointer-less variant

Failed Literal Probing

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

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

Extensions

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

RiSE

Other Types of Learning

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

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

1. BCP over (in)equalities: $\dfrac{x = y \quad z = (x \oplus y)}{z = 0}$ $\dfrac{x = 0 \quad z = (x \vee y)}{z = y}$ etc.

2. structural rules: $\dfrac{x = (a \vee b) \quad y = (a \vee b)}{x = y}$ etc.

3. test rule:

$$\dfrac{\begin{matrix}\{x = 0\} \cup E & & \{x = 1\} \cup E \\ \Downarrow & & \Downarrow \\ E_0 \cup E & & E_1 \cup E\end{matrix}}{(E_0 \cap E_1) \cup E}$$

Assume $x = 0$, BCP and derive (in)equalities E_0.

Then assume $x = 1$, BCP and derive (in)equalities E_1.

The intersection of E_0 and E_1 contains the (in)equalities valid in *any* case.

Stålmarck's Method Recursively

$$
\begin{array}{cccc}
\multicolumn{2}{c}{x = 0} & \multicolumn{2}{c}{x = 1} \\
\multicolumn{2}{c}{\Downarrow} & \multicolumn{2}{c}{\Downarrow} \\
y = 0 & y = 1 & y = 0 & y = 1 \\
\Downarrow & \Downarrow & \Downarrow & \Downarrow \\
E_{00} & E_{01} & E_{10} & E_{11}
\end{array}
$$

$$
\frac{\dfrac{E_{00} \quad E_{01}}{E_0} \qquad \dfrac{E_{10} \quad E_{11}}{E_1}}{E}
$$

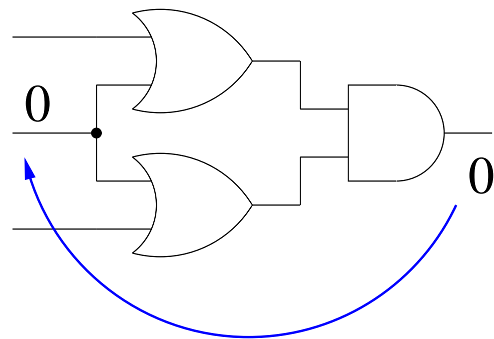
(we do not show the (in)equalities that do not change)

RiSE

Stålmarck's Method Summary

- recursive application
  - depth of recursion bounded by number of variables
  - complete procedures (determines satisfiability or unsatisfiability)
  - for a fixed (constant) recursion depth k polynomial!
- k-saturation:
  - apply split rule on recursively up to depth k on all variables
  - 0-saturation: apply all rules accept test rule (just BCP: linear)
  - 1-saturation: apply test rule (not recursively) for all variables
    (until no new (in)equalities can be derived)

- circuits



output 0 implies middle input 0 *indirectly*

- CNF
  - for each clause c in the CNF
    - $*$ for each literal l in the clause c
      - · assume l and propagate
      - · collect set of all implied literals (direct/indirect “implications” of l)
    - $*$ intersect these sets of implied literals over all l in c
    - $*$ literals in the intersection are implied without any assumption

RiSE

(Bounded) Variable Elimination (VE)

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

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

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

  – do not count tautological resolvents

  – detect units on-the-fly

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

  – variables ordered by the number of occurrences

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

RiSE

Fast (Self) Subsumption

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

Variable Instantiation

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

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

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

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

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

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

Autarkies

Generalization of pure literals.

Given a partial assignment σ.

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

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

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

All clauses touched by an autarky can be removed.

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

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

RiSE

Blocked Clause Elimination (BCE)

one clause $C \in F$ with l *all* clauses in F with $\bar{l}$

fix a CNF F

$a \vee b \vee l$

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

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

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

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

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

Blocked Clauses [Kullmann'99]

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

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

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

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

only first clause is not blocked.

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

literal c in the last clause is blocked.

after removing either $(a \vee \bar{b} \vee \bar{c})$ or $(\bar{a} \vee c)$, the clause $(a \vee b)$ becomes blocked

actually all clauses can be removed

RiSE

[JärvisaloBiereHeule-TACAS'10]

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

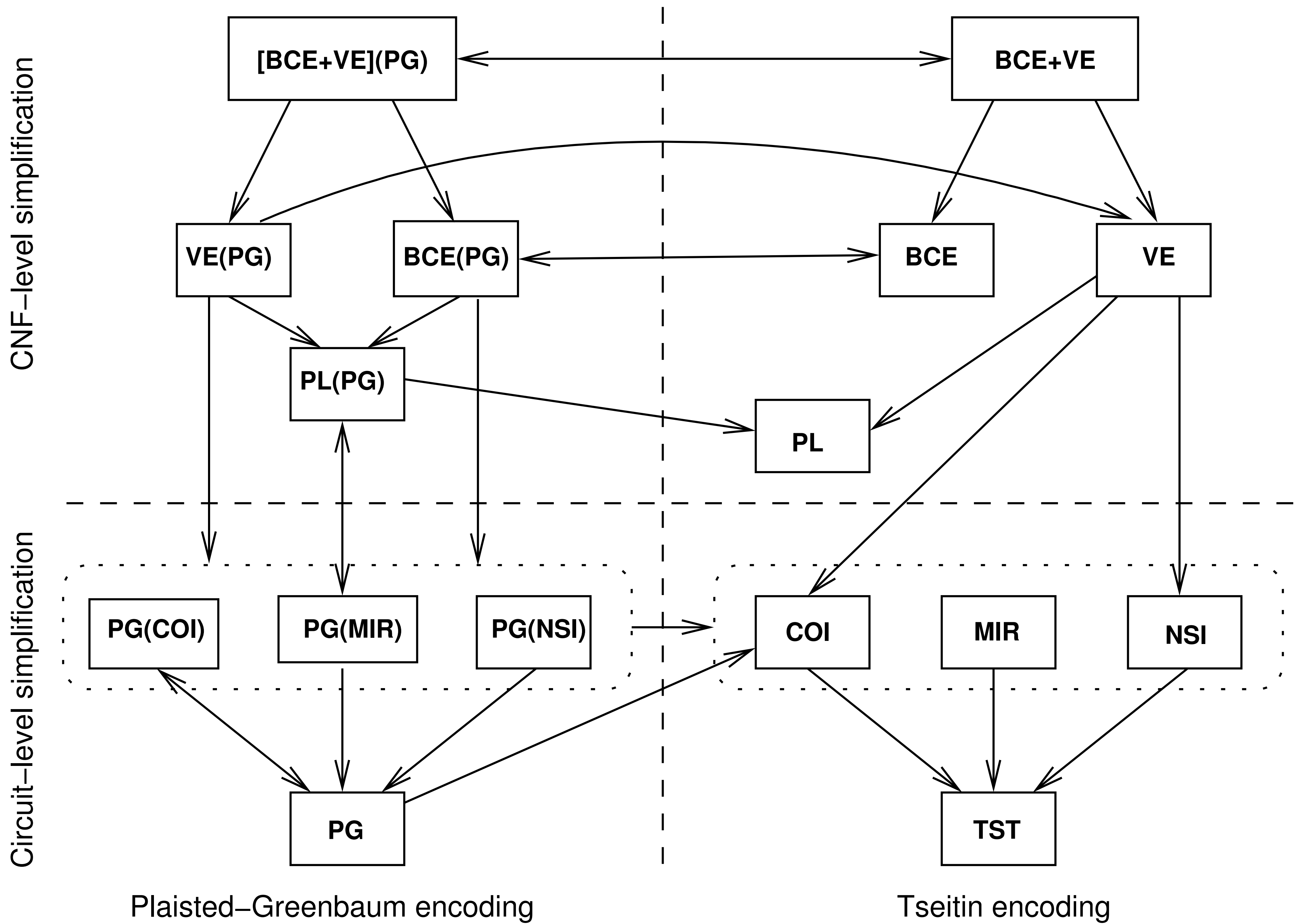
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination



CNF–level simplification
Circuit–level simplification
[BCE+VE](PG)
BCE+VE
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
Plaisted–Greenbaum encoding
Tseitin encoding

Inprocessing: Interleaving Preprocessing and Search

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

- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple idea to benefit from full preprocessing without penalty
  - "preempt" preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time

Benefits of Inprocessing

- allows to use *costly* preprocessors
  - without increasing run-time “much” in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and distillation — even VE can be costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
  - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding “bad” implementation though …
- … and hard(er) to debug

RiSE

Other Inprocessing / Preprocessing Techniques

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

boolean ring reasoning extract XORs, then Gaussian elimination etc.

hyper-binary resolution focus on producing binary resolvents

hidden/asymmetric tautology elimination discover redundant clauses through probing

covered clause elimination use covered literals in probing for redundant clauses

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

RiSE