Karlsruhe Institute of Technology

Entscheidungsverfahren mit Anwendungen in der Softwareverifikation

III: Aussagenlogische Erfüllbarkeit

Carsten Sinz
Institut für Theoretische Informatik

05.11.2019

Definition SAT-Problem

- Gegeben: Aussagenlogische Formel F in konjunktiver Normalform (CNF).
  - Literal: Variable oder negierte Variable; Klausel: Disjunktion von Literalen
- Frage: Ist F erfüllbar, d.h. gibt es ein Modell α von F?
  - Modell: Abbildung $\alpha: Var(F) \to \{0,1\}$, so dass in jeder Klausel von F mindestens ein Literal mit wahr belegt ist
- Beispiel: $F = \{\{x,\neg y\}, \{\neg x,\neg z\}, \{y\}\}$
  - Für $x=y=1$ und $z=0$ evaluiert F zu 1 (also ist F erfüllbar)

- Anmerkung:
  - Ja/Nein-Antwort oft nicht ausreichend
  - Begründung in vielen Anwendungen erforderlich

CNF-Darstellung: DIMACS-Format

- Format zur Darstellung von Formeln in CNF auf dem Computer.
- Aufbau einer DIMACS-Datei:
  1. Optionale Kommentarzeilen: `c Kommentar`
  2. Präambel: `p cnf n m` (n: Anzahl Variablen, m: Anzahl Klauseln)
  3. Klauseln:
     - Liste der Literale, durch Leerzeichen getrennt, 0 als Abschluss
     - Variablen repräsentiert durch Ganzzahlen (>0), „-“ als Negationssymbol

DIMACS-Format: Beispiel

$F = \{\{x_1, x_2, \neg x_3\}, \{x_3, \neg x_4\}, \{\neg x_1, \neg x_3, x_4\}, \{\neg x_2, x_3, x_5\}\}$
im DIMACS-Format:

```
c Dies ist eine Formel in CNF
c mit 5 Variablen und 4 Klauseln.
p cnf 5 4
1 2 -3 0
3 -4 0
-1 -3 4 0
-2 3 5 0
```

(letzte 0 optional)

Vereinfachung durch Unit-Klauseln

- Unit-Klausel: enthält nur ein Literal, d.h. $C = \{m\}$.
- Beobachtung: In jedem Modell α muss *m* mit wahr belegt sein.
- Dadurch Vereinfachung möglich:
  - ¬m kann aus allen anderen Klauseln in F gestrichen werden (da ¬m immer unter α mit falsch belegt ist).
  - Wenn dadurch die leere Klausel (□, entspricht 0) entsteht, ist das Problem unerfüllbar.
- Bezeichnung: Unit-Resolution

DPLL-Algorithmus: Übersicht

- DPLL: Davis-Putnam-Logemann-Loveland
- Grundlegender Algorithmus ~1965
- Grundidee: Fallunterscheidungen und Vereinfachung
- Vereinfachungen:
  - Unit-Resolution
  - Unit-Subsumption

  } Unit-Propagation
  - Löschen purer Literale (pure literal deletion)

Grundlegender DPLL-Algorithmus

boolean DPLL(**ClauseSet** S)
{
  while (S contains a unit clause $\{L\}$) {
    delete from S clauses containing L; // unit-subsumption
    delete $\neg L$ from all clauses in S; // unit-resolution
  }
  if ($\square \in S$) **return** false; // empty clause?
  while (S contains a pure literal L)
    delete from S all clauses containing L;
  if ($S = \emptyset$) **return** true; // no clauses?
  choose a literal L occurring in S; // case-splitting
  if (DPLL($S \cup \{\{L\}\}$) **return** true; // first branch
  else if (DPLL($S \cup \{\{\neg L\}\}$) **return** true; // second branch
  else return false;
}

1(

Beispiel Unit-Propagation

S = {{¬x,y,¬z}, {¬x,z}, {¬y,x}, {y}}

1. Unit-Klausel vorhanden? Ja: {y}
2. Unit-Subsumption: lösche Klauseln, in denen y vorkommt:
   S_1 = {{¬x,z}, {¬y,x}}
3. Unit-Resolution: lösche ¬y aus allen Klauseln:
   S_2 = {{¬x,z}, {x}}
4. Unit-Klausel vorhanden? Ja: {x}
5. Unit-Subsumption: S_3 = {{¬x,z}}
6. Unit-Resolution: S_4 = {{z}}
7. Unit-Klausel vorhanden? Ja: {z}
8. Unit-Subsumption: S_5 = {}
9. Unit-Resolution: keine Klauseln mehr vorhanden

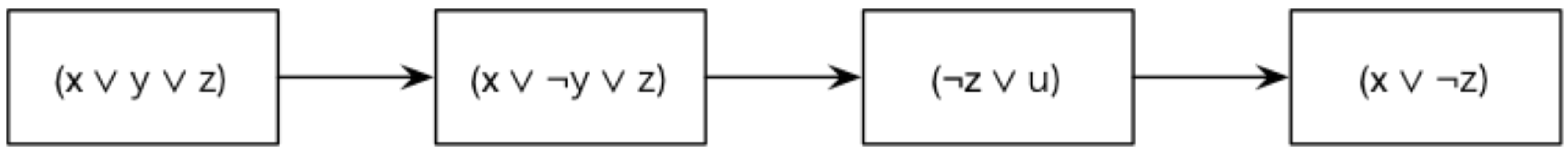
Ergebnis: S erfüllbar!

```
boolean DPLL(ClauseSet S)
{
  while ( S contains a unit clause {L} ) {
    delete from S clauses containing L;  // unit-subsumption
    delete ¬L from all clauses in S;     // unit-resolution
  }
  if ( □ ∈ S ) return false;             // empty clause?
  while ( S contains a pure literal L )
    delete from S all clauses containing L;
  if ( S = ∅ ) return true;              // no clauses?
  choose a literal L occurring in S;     // case-splitting
  if ( DPLL(S ∪ {{L}} ) return true;     // first branch
  else if ( DPLL(S ∪ {{¬L}} ) return true;  // second branch
  else return false;
}
```

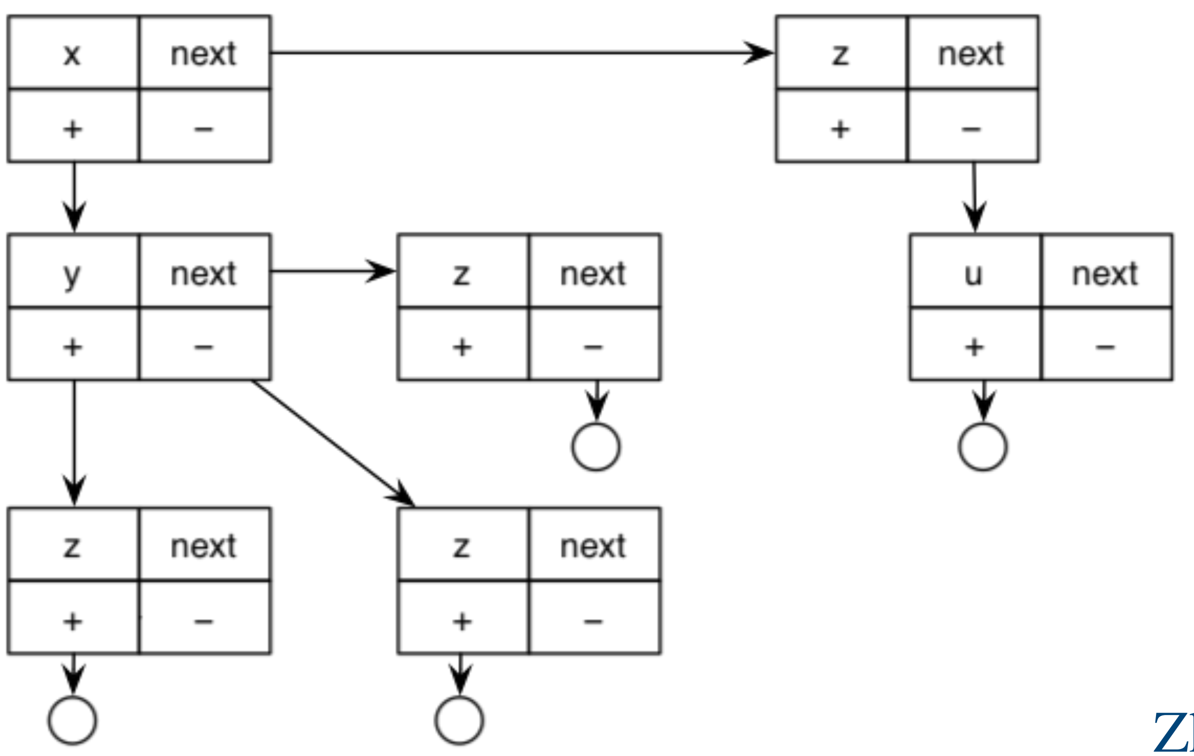
Datenstrukturen zur Repräsentation von Formeln in CNF

Wie können Klauselmengen dargestellt werden?

A) Als Liste von Klauseln



B) Als Trie (= prefix tree) Datenstruktur



Zhang, Stickel (1994)

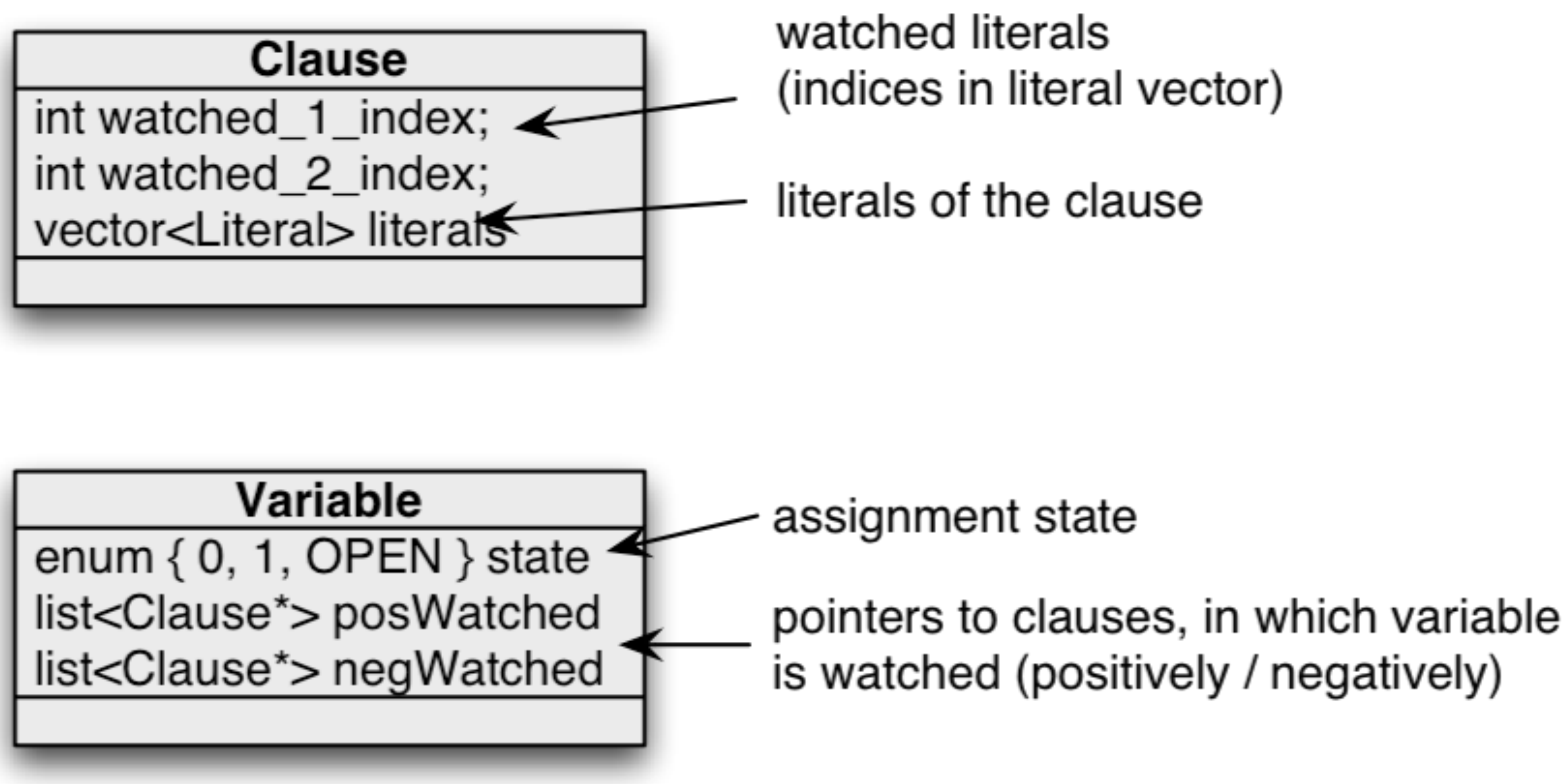
Datenstrukturen für DPLL: Anforderungen

- Sollen schnelle Unit-Propagation ermöglichen
  - Erkennen neuer Units
  - Propagierung von Units
- Unterstützung von Back-Tracking (Wiederherstellung von Klauselmengen)

  Implementationsalternativen:
  - Speichere Kopie der Klauselmenge bei jedem rekursiven Aufruf
  - Speichere nur Änderungen an der Datenstruktur (undo-stack)

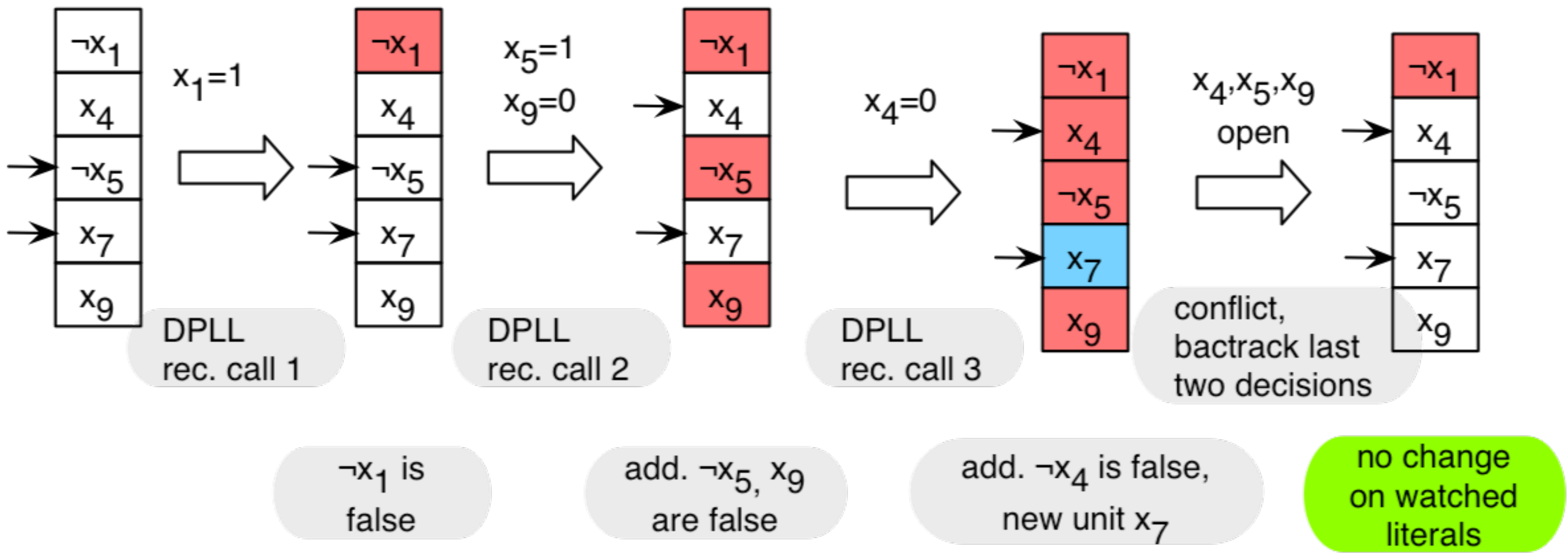
  Ziel: Minimiere Aufwand zur Wiederherstellung
- Kompakte Darstellung großer Klauselmengen

Watched Literals: Datenstrukturen



Moskewicz *et al*. (2001)

Watched Literals: Beispiel

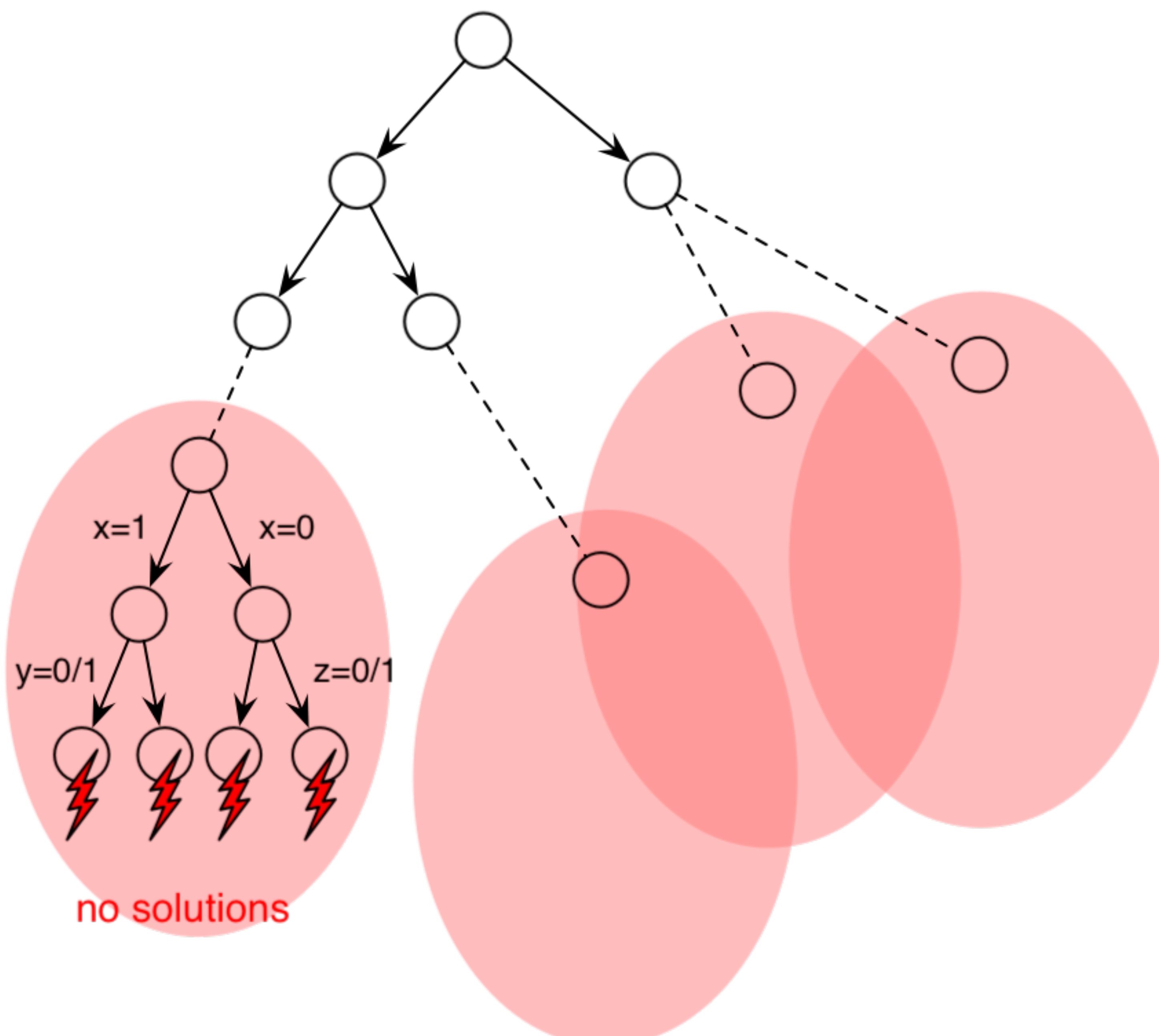


Literalauswahl-Strategien

- Welches Literal soll im Fallunterscheidungs-Schritt ausgewählt werden?
  - Größe des Suchraums (und damit die Laufzeit des DPLL-Algorithmus) kann sehr stark von der Literalauswahl-Strategie abhängen.
  - Strategie stark problemabhängig; keine allgemeine “beste” Strategie.
- Ideen im Zhg. mit Auswahlheuristik:
  1. Maximale Vereinfachung; z.B. maximiere die Anzahl der subsumierten (d.h. zu „löschenden“) Klauseln
  2. Versuche, handhabbare Teilklasse von SAT zu erreichen; z.B. 2-SAT, Horn-SAT, nur positive Klauseln
  3. Basierend auf Konfliktanalyse bzw. Klausel-Lernen (Präferenz für Literale in kürzlich gelernten Klauseln)

Konfliktanalyse: Motivation

{¬x,y},
{¬x,¬y},
{x,¬z}.
{x,z}

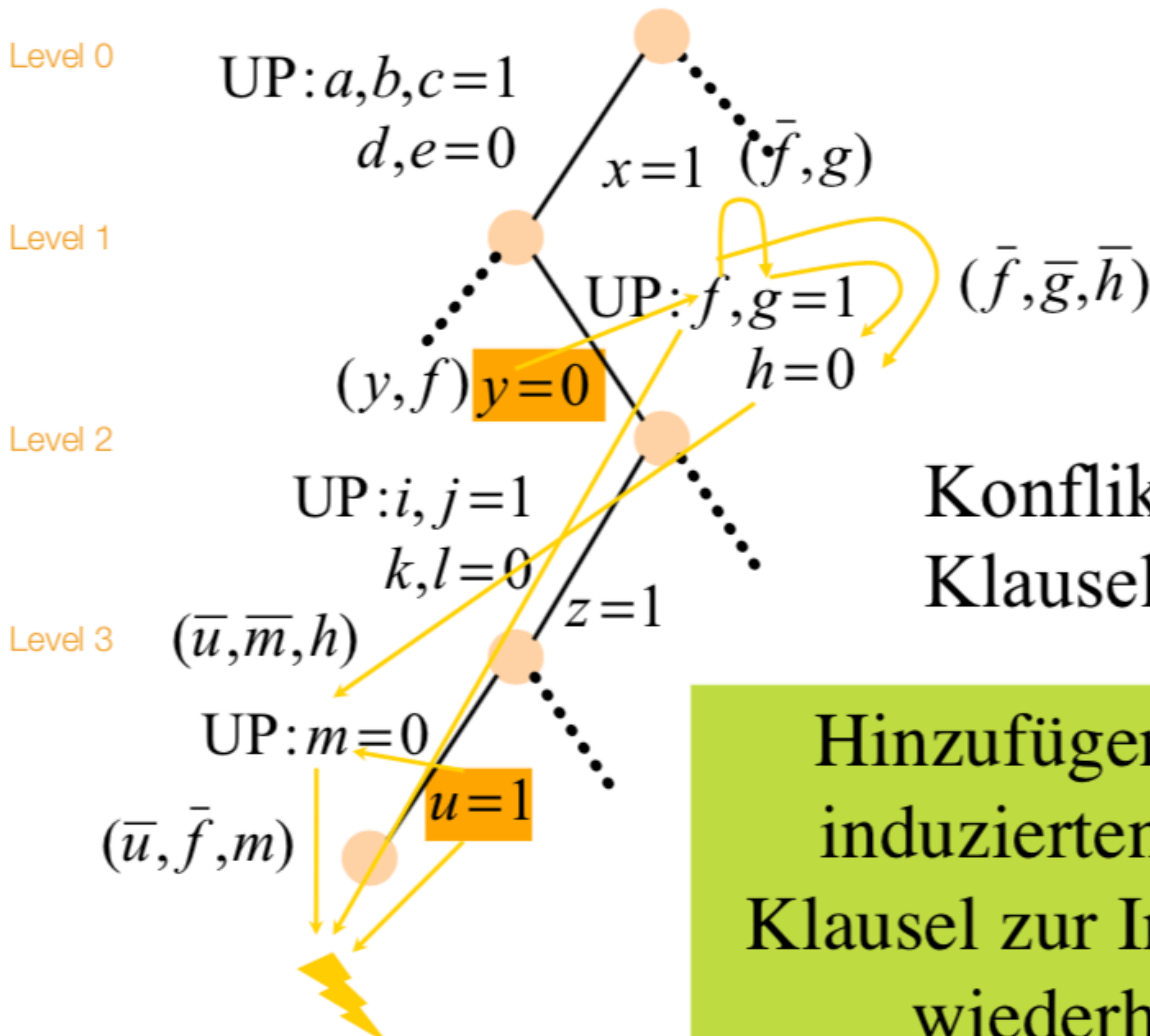


Konfliktanalyse & Klausel-Lernen

- Versuche wiederholte Untersuchung von Teilen des Suchraums ohne Lösungen zu vermeiden.
  - Dies kann eine „schlechte“ Variablenauswahl-Heuristik kompensieren
- Methode: finde schwächste Annahme (weakest precondition), unter der ein Widerspruch auftritt.
  - Jedes für die Fallunterscheidung gewählte Literal gilt als „atomarer“ Grund.
  - Finde minimal erforderliche Bedingung (d.h. kleinste Literalmenge), die denselben Konflikt hervorruft.
- Klausel-Lernen wird auch als „no-good-learning“ bezeichnet (CSP).

Konfliktklausel-Generierung (Marques-Silva, Sakallah, 1996)

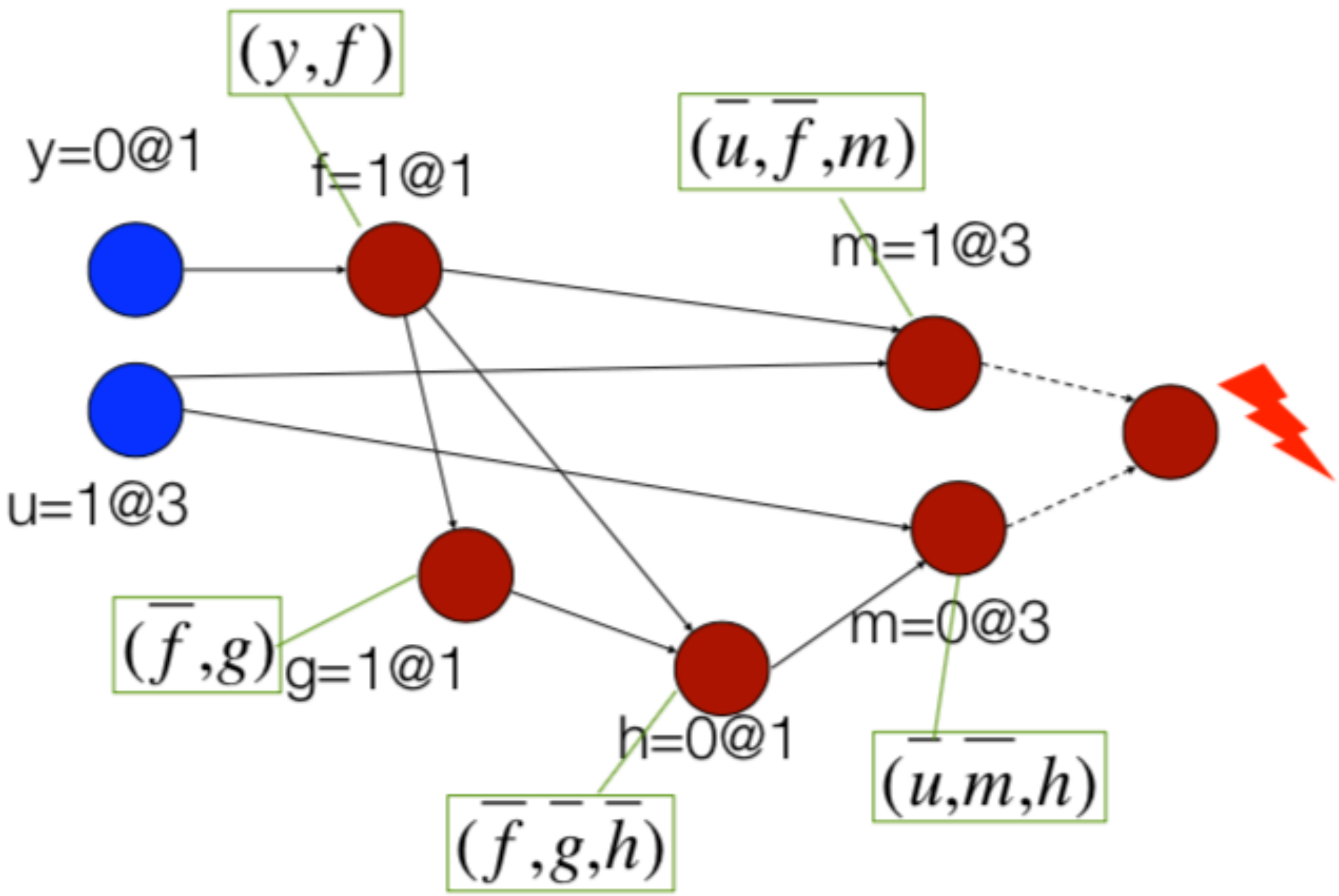
$(\bar{u},\bar{m},h)$
$(\bar{u},\bar{f},m)$
$(\bar{f},\bar{g},\bar{h})$
(y,f)
$(\bar{f},g)$
$\vdots$
$(\bar{x},a)$
$(\bar{x},b,\bar{a})$
$(\bar{x},\bar{a},\bar{b},c)$
$(\bar{a},\bar{d})$
$(\bar{b},d,\bar{e})$
$\vdots$



Konflikt-induzierte Klausel: $(y,\bar{u})$

Hinzufügen der Konflikt-induzierten („gelernten") Klausel zur Instanz vermeidet wiederholte Suche.

Konfliktgraph (I)

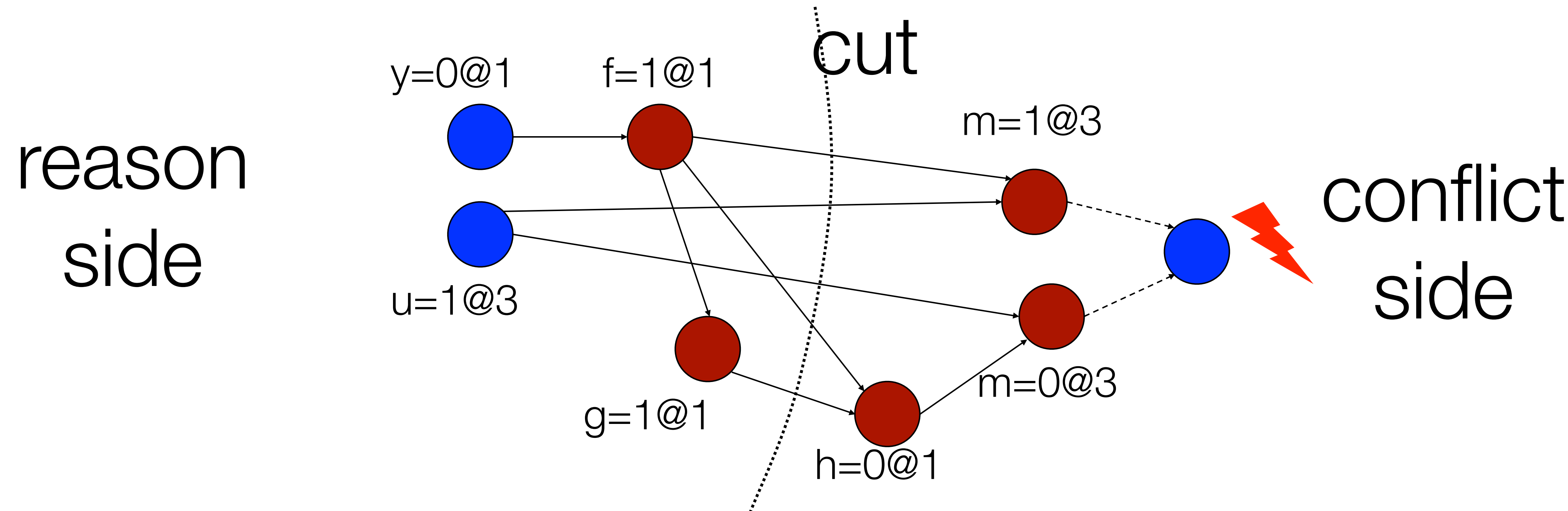


y=0 (auf Level 1) impliziert:
f=1, g=1, h=0

u=1 (auf Level 3) impliziert:
m=0, m=1, Widerspruch

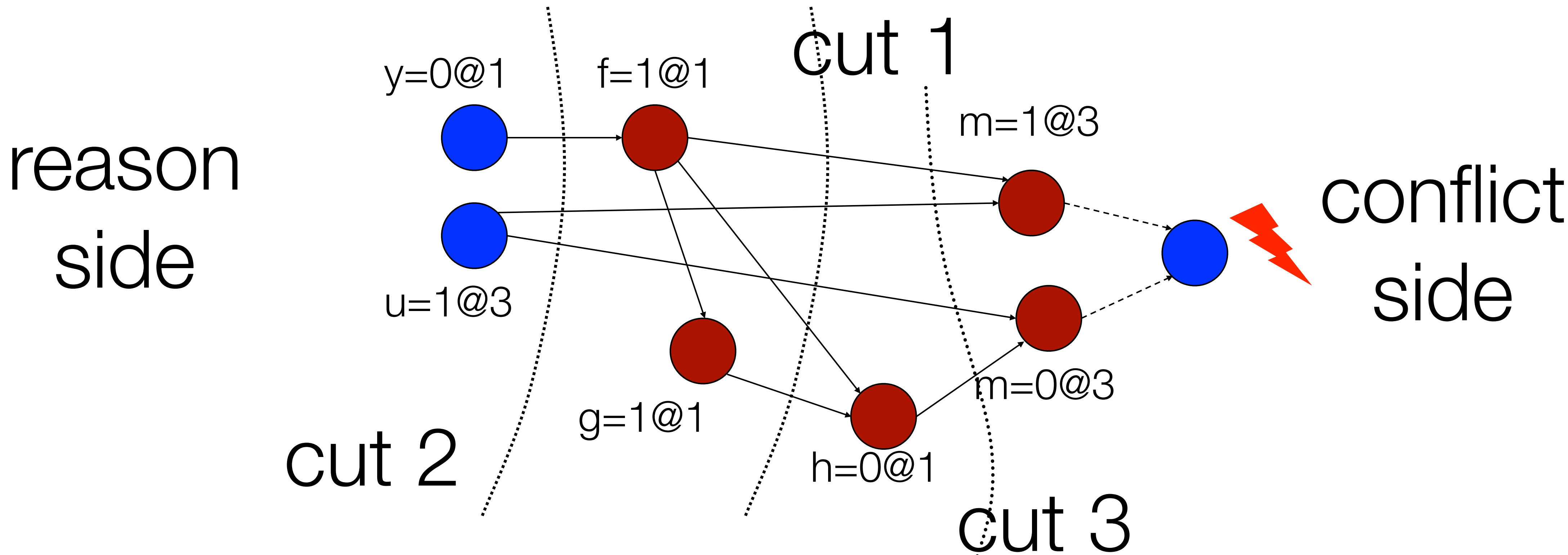
$(\overline{u},\overline{m},h)$	$(\overline{x},a)$
$(\overline{u},\overline{f},m)$	$(\overline{x},b,\overline{a})$
$(\overline{f},\overline{g},\overline{h})$	$(\overline{x},\overline{a},\overline{b},c)$
(y,f)	$(\overline{a},\overline{d})$
$(\overline{f},g)$	$(\overline{b},d,\overline{e})$
$\vdots$	$\vdots$

Konfliktgraph (II)



- Jede Konfliktklausel ist durch einen Cut durch den Konfliktgraphen bestimmt (Knoten-Partitionierung in *reason side* und *conflict side*)
    - *Decision nodes* sind auf der *reason side.*
    - *Widerspruch* ist auf der *conflict side.*
    - Konfliktklausel ergibt sich aus Negation aller Literale, von denen eine

Konfliktgraph (III)



Cut 1: Konfliktklausel {¬f, ¬u, ¬g}

Cut 2: Konfliktklausel {y, ¬u}

Cut 3: Konfliktklausel {¬f, ¬u, h}

Jede Konfliktklausel (oder auch mehrere) kann bei einem Konflikt zur Klauselmenge hinzugefügt werden.

Welche Konfliktklausel hinzufügen?

- Decision clause
  - enthält nur decision nodes (cut 2)
- First new cut clause
  - enthält nur sich widerspr. Literale (komplementäres Literalpaar) auf conflict side (cut 3)
- 1UIP clause
  - UIP (unique implication point): Knoten, über den alle Pfade von *conflict side* zu *reason side* laufen
    - Anmerkung: alle decision nodes sind UIPs
  - 1: Ausgehend von *first new cut clause*, gehe „zurück“ im Graph solange Knoten-Level nicht kleiner wird, bis UIP erreicht.
- 1UIP wird in den meisten Solvern verwendet.

CDCL: Conflict-Driven Clause Learning

```
boolean CDCL
{
   forever {
       do {
        ok = propagate_units(); // returns false iff conflict occurred
        if (!ok) {              // conflicting assignment
            generate_and_add_conflict_clause();
            new_level = backtrack();
            if (new_level < 0) return false;
            }
       } while(!ok);
       if no more open variables return true;
       decide();            // select open literal, assign value to it
    }
}
```

Konfliktklausel-basierte Variablenauswahl-Heuristik: VSIDS

- VSIDS:
  - Ordne jeder Variablen einen *score* zu.
    - Initial 0 (oder Anzahl der Vorkommen)
  - Wird eine Konfliktklausel C erzeugt, so wird der *score* eines jeden Literals aus C um einen Betrag a erhöht.
  - Nach n Konflikten werden alle *scores* durch einen konstanten Faktor f geteilt (*ageing*).
  - Die Heuristik wählt immer das Literal mit dem höchsten *score*.

Weitere Techniken im SAT-Solving

- **Restarts**
  - Breche Suche nach einer vorgegebenen Anzahl von Schritten k ab und starte diese neu
- **Phase memorization**
  - Wähle nach Backtracking in `decide()` gleiche Phase (d.h. gleiches „Vorzeichen“) wie bei letzter Zuweisung
- **Preprocessing**
  - Vereinfache Formel in einem Vorverarbeitungsschritt

SAT-Codierung: Einführendes Beispiel

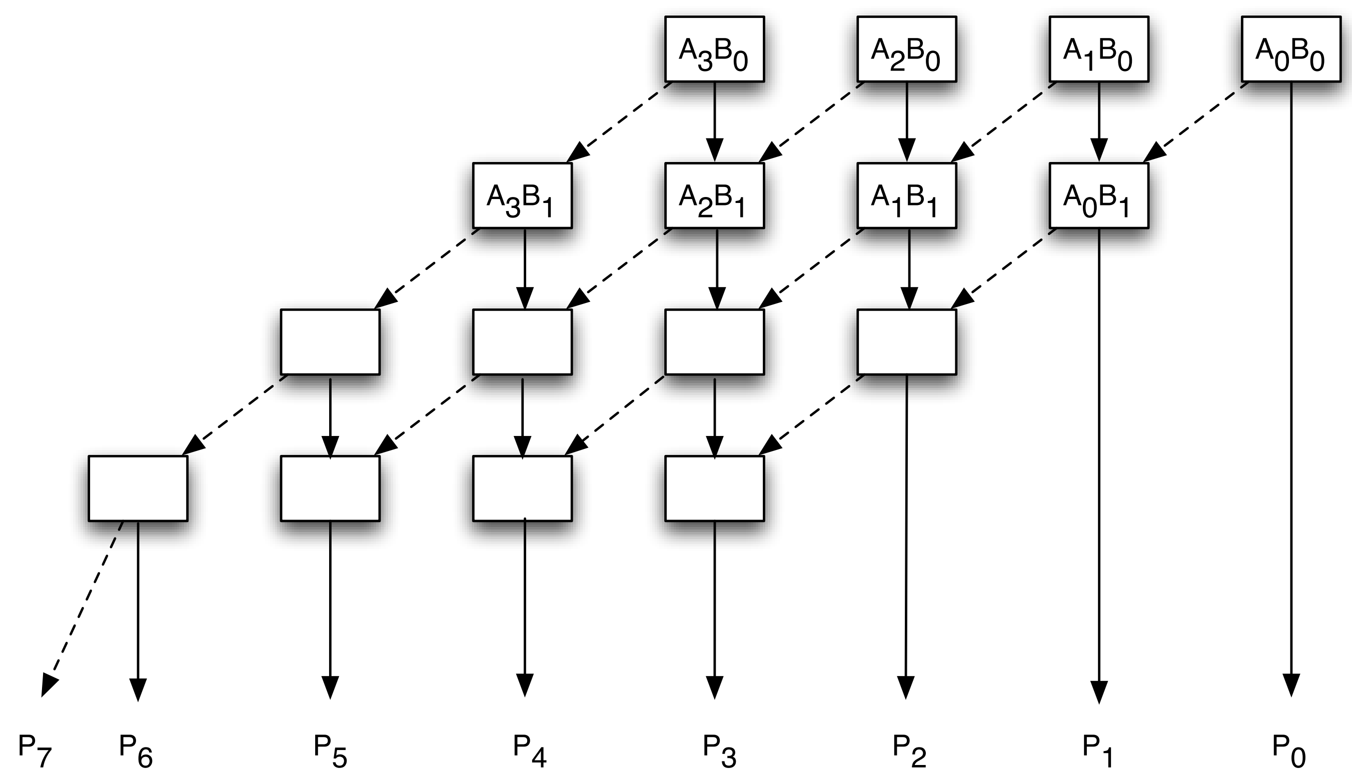
- Problem: Ganzzahl-Faktorisierung
  - Gegeben eine ganze Zahl p, gibt es eine Zerlegung $p = x \cdot y$ mit $1 < x, y < p$?
- Programm:

```
void factorize(unsigned int x, unsigned int y) {
  unsigned int p = 1023;
  if (!(x > 1 && y > 1 && x < p && y < p))
    return;
  if (p == x * y)
    assert(0); // error
}
```

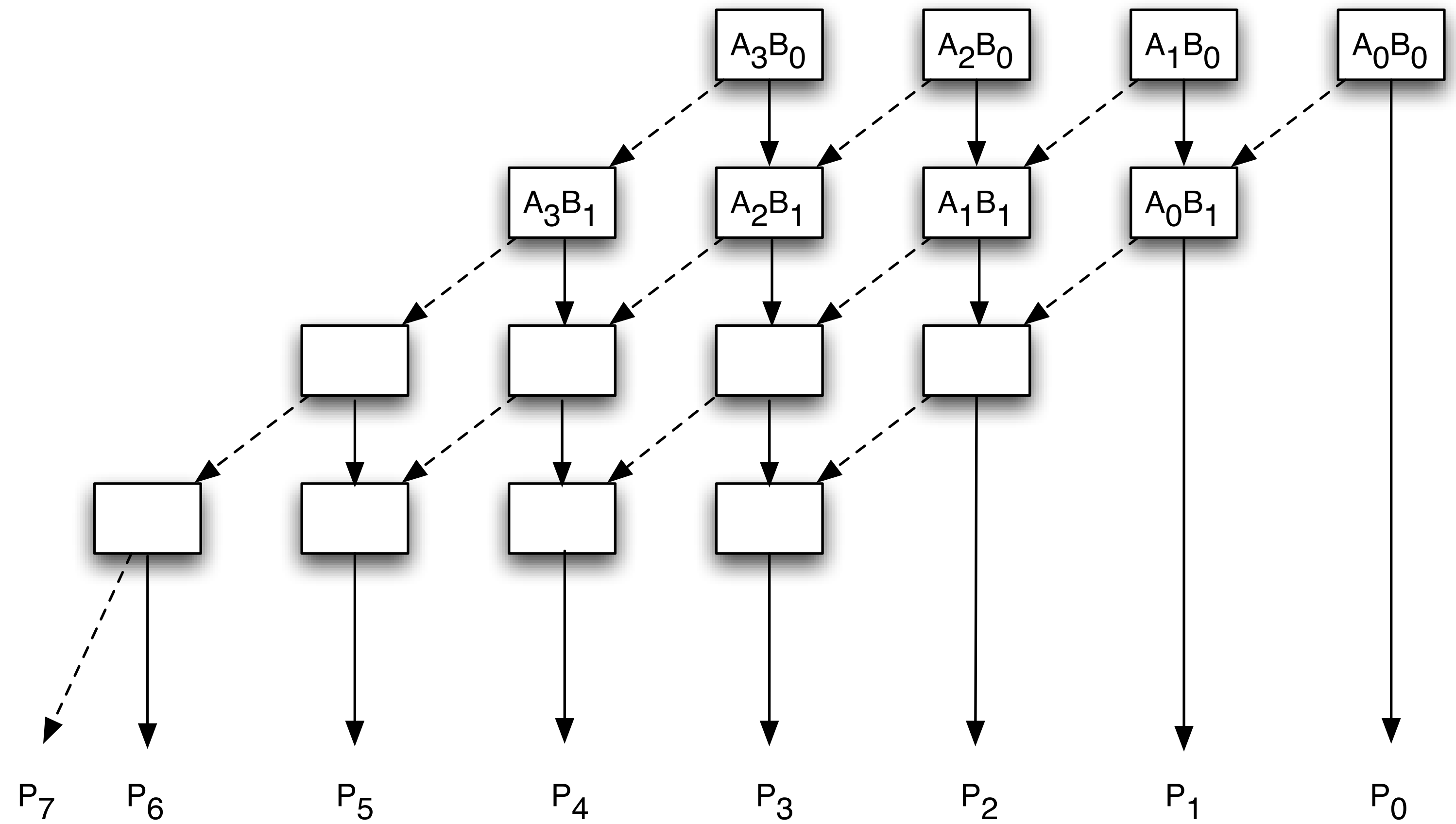
- Wenn es eine Faktorisierung von p gibt, ist die „error"-Zeile erreichbar
- Wie können wir das in SAT codieren?

Modellierung in Hardware

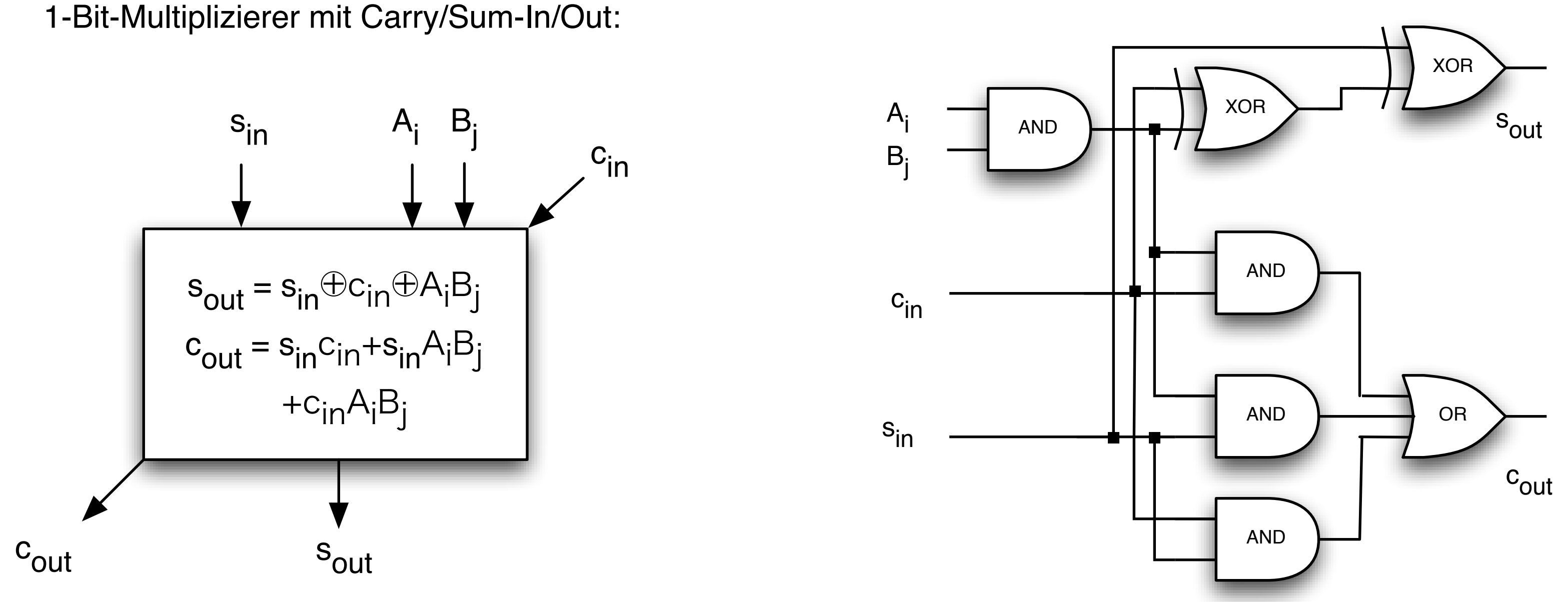
- **Formel zusammengesetzt aus:**
  - n-Bit Multiplizierer zur Multiplikation `x*y`
  - Ausgabe wird auf `p` festgehalten
  - Zusätzliche Klauseln zur Einschränkung der Eingabe (`1 < x, y < p`)
  - Bit-Vektoren für `x` und `y` sind Eingabe-Variablen
- **Multiplizierer (4 Bit):**



Hardware-Multiplizierer



1-Bit-Multiplizierer mit Carry/Sum-In/Out:



Tseitin-Codierung

- Regeln zur Tseitin-Codierung (mit Optimierungen von Plaisted-Greenbaum):

$$\mathcal{T}(F) = d_F \wedge \mathcal{T}^1(F)$$

$$\mathcal{T}^p(F) = \begin{cases} \mathcal{T}^p_{\mathsf{def}}(F) \wedge \mathcal{T}^p(G) \wedge \mathcal{T}^p(H), & \mathit{if}\ F = G \circ H\ \ \mathit{and}\ \circ \in \{\wedge, \vee\} \\ \mathcal{T}^p_{\mathsf{def}}(F) \wedge \mathcal{T}^{p \oplus 1}(G), & \mathit{if}\ F = \neg G \\ \top, & \mathit{if}\ F \in \mathcal{V} \end{cases}$$

$$\mathcal{T}^1_{\mathsf{def}}(F) = \begin{cases} (\neg d_F \vee d_G) \wedge (\neg d_F \vee d_H), & \mathit{if}\ F = G \wedge H \\ (\neg d_F \vee d_G \vee d_H), & \mathit{if}\ F = G \vee H \\ (\neg d_F \vee \neg d_G), & \mathit{if}\ F = \neg G \end{cases}$$

$$\mathcal{T}^0_{\mathsf{def}}(F) = \begin{cases} (d_F \vee \neg d_G \vee \neg d_H), & \mathit{if}\ F = G \wedge H \\ (d_F \vee \neg d_G) \wedge (d_F \vee \neg d_H), & \mathit{if}\ F = G \vee H \\ (d_F \vee d_G), & \mathit{if}\ F = \neg G \end{cases}$$