Entscheidungsverfahren für die Software-Verifikation

3 – Aussagenlogische Erfüllbarkeit

Definition SAT-Problem

- Gegeben: Formel F in CNF.
- Frage: Ist F erfüllbar, d.h. gibt es ein Modell α von F?
- Beispiel: F = {{x,¬y}, {¬x,¬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:
  - $\neg$m kann aus allen anderen Klauseln in F gestrichen werden (da $\neg$m immer unter α mit falsch belegt ist).
  - Wenn dadurch die leere Klausel ($\square$, 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 ¬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;
}
```

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

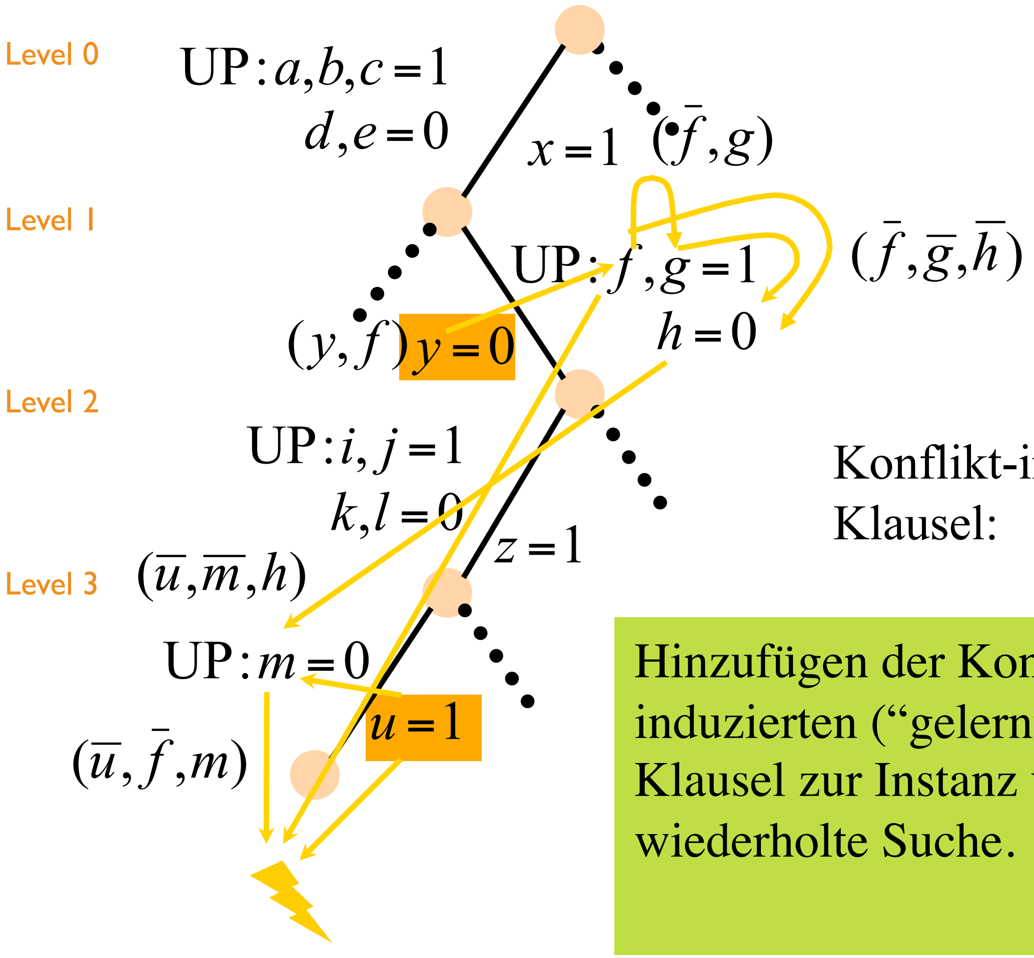
Ergebnis: S erfüllbar!

```
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;
}
```

Lemma-Generierung

(Marques-Silva, Sakallah, 1996)

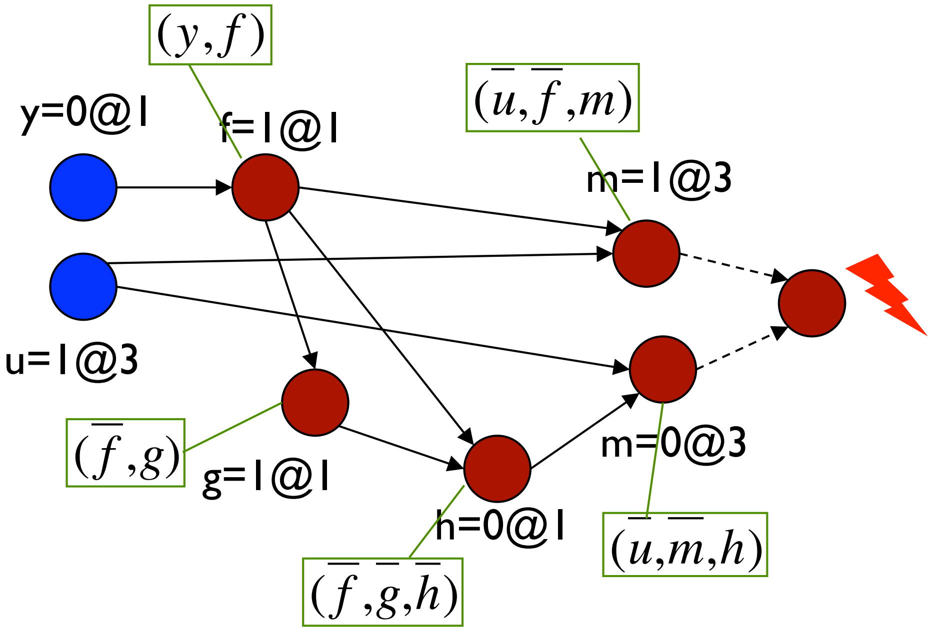
$(\overline{u}, \overline{m}, h)$
$(\overline{u}, \overline{f}, m)$
$(\overline{f}, \overline{g}, \overline{h})$
(y, f)
$(\overline{f}, g)$
$\vdots$
$(\overline{x}, a)$
$(\overline{x}, b, \overline{a})$
$(\overline{x}, \overline{a}, \overline{b}, c)$
$(\overline{a}, \overline{d})$
$(\overline{b}, d,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



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

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

Konfliktgraph (I)

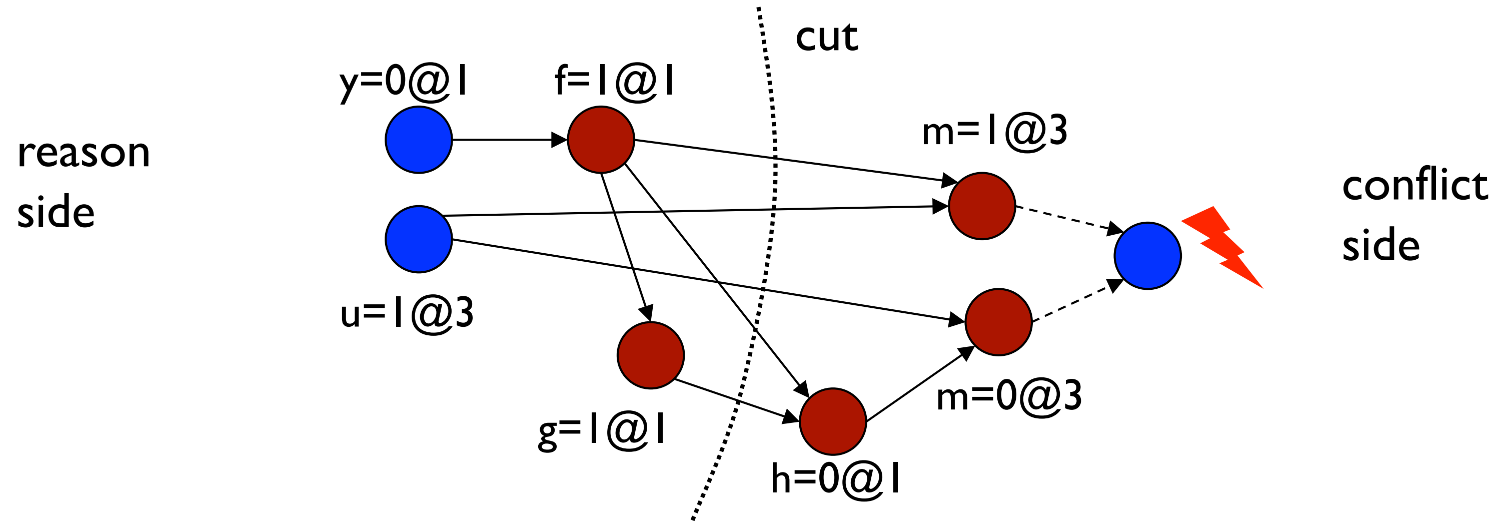


y=0 (auf Level 1) impliziert:
f=1, g=1, h=0

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

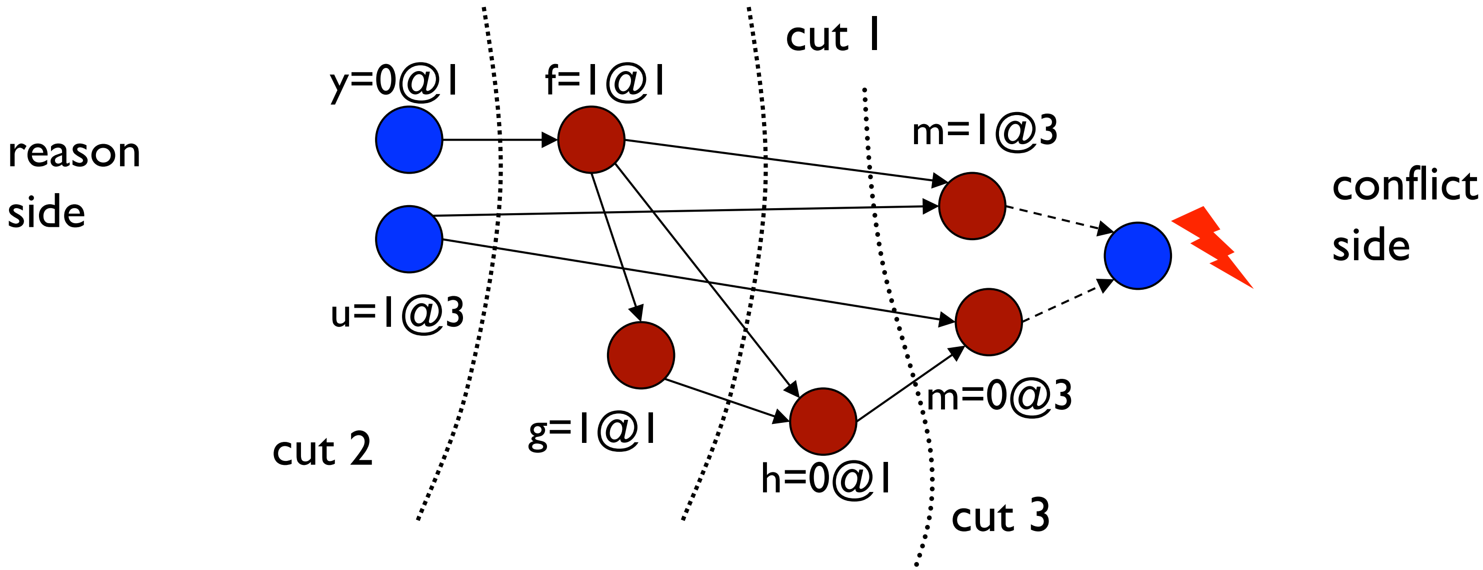
$(\overline{u},\overline{m},h)$	$(\overline{x},a)$
$(\overline{u},\overline{f},m)$	$(\overline{x},b,\overline{a})$
$(\overline{f},\overline{g},\overline{h})$	$(\overline{x},\overline{a},\overline{b},c)$
(y,f)	$(\overline{a},\overline{d})$
$(\overline{f},g)$	$(\overline{b},d,\overline{e})$
$\vdots$	$\vdots$

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 Kante von der *reason side* ausgeht zur *conflict side*.

Konfliktgraph (III)



Cut 1: Konfliktklausel {¬f, ¬u, ¬g}

Cut 2: Konfliktklausel {y, ¬u}

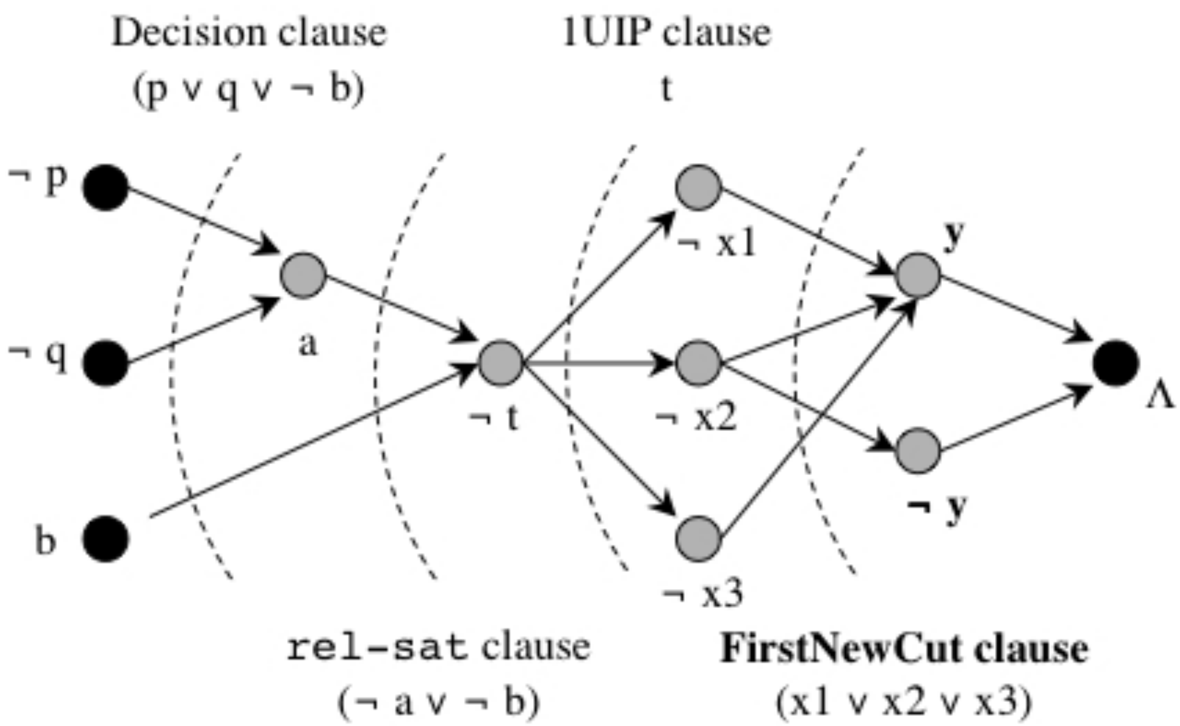
Cut 3: Konfliktklausel {¬f, ¬u, h}

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.

Beispiel: Konfliktgraph mit verschiedenen Konfliktklauseln



Quelle: Beame et al.: Understanding the Power of Clause Learning (2003)

Erweiterter DPLL Algorithm mit Klausel-Lernen

```
boolean DPLL-Enhanced
{
    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();                           // assign value to open literal
    }
}
```

Konfliktklausel-basierte Variablenauswahl-Heuristiken: VSIDS und Berkmin-Heuristik

- 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*.

Resolution (I)

- Kalkül (= formales Regelsystem) für (aussagenlogische) Formeln in CNF
- Entwickelt 1965 von Alan Robinson
  - Ursprünglich für Prädikatenlogik erster Stufe
- Zielstellung: Beweise Unerfüllbarkeit von (aussagenlogischen) Formeln
- Methode:
  - Wende Inferenzregel(n) zur Herleitung der leeren Klausel an.
  - Axiome: Klauseln der Ausgangsformel
  - Falls sich leere Klausel (□) herleiten lässt, ist ein Beweis gefunden.

Resolution (II)

- (Einzige) Inferenzregel:

Prämissen

$$\frac{C \qquad\qquad D}{(C \setminus \{l\}) \cup (D \setminus \{\neg l\})}\ \text{Res}$$

Konklusion

wobei $l \in C$, $\neg l \in D$ und C, D Klauseln.

- Beispiele:

$$\frac{\{x,y,\neg z\} \quad \{u,\neg v,z\}}{\{x,y,u,\neg v\}} \qquad \frac{\{x,u,\neg v\} \quad \{\neg u\}}{\{x,\neg v\}}$$

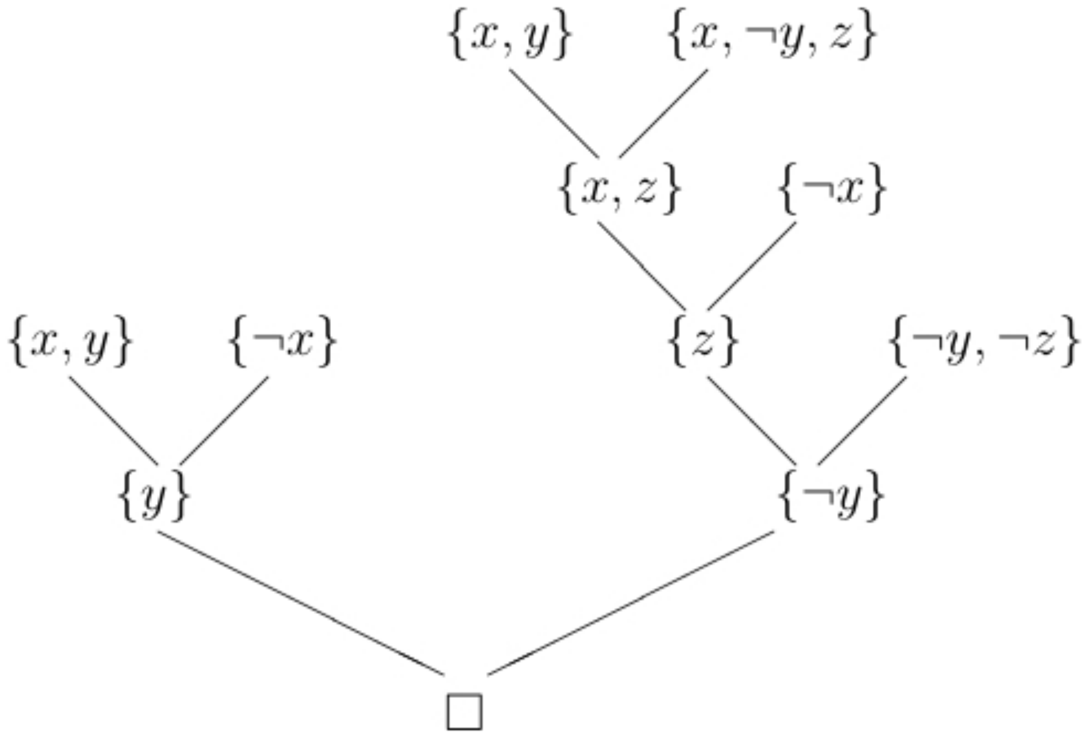
Resolution (III)

$$\frac{C \quad D}{E} \mathrm{Res} \quad \text{mit} \quad E = (C \setminus \{l\}) \cup (D \setminus \{\neg l\})$$

- **Begriffe:**
  - E: Resolvente
  - C, D: Eltern-Klauseln
  - „E entsteht durch Resolution über l aus C und D“

Resolutionsbeweis: Beispiel

$$F_3 = \{\{x, y\}, \{x, \neg y, z\}, \{\neg x\}, \{\neg y, \neg z\}\}$$



Resolution (IV)

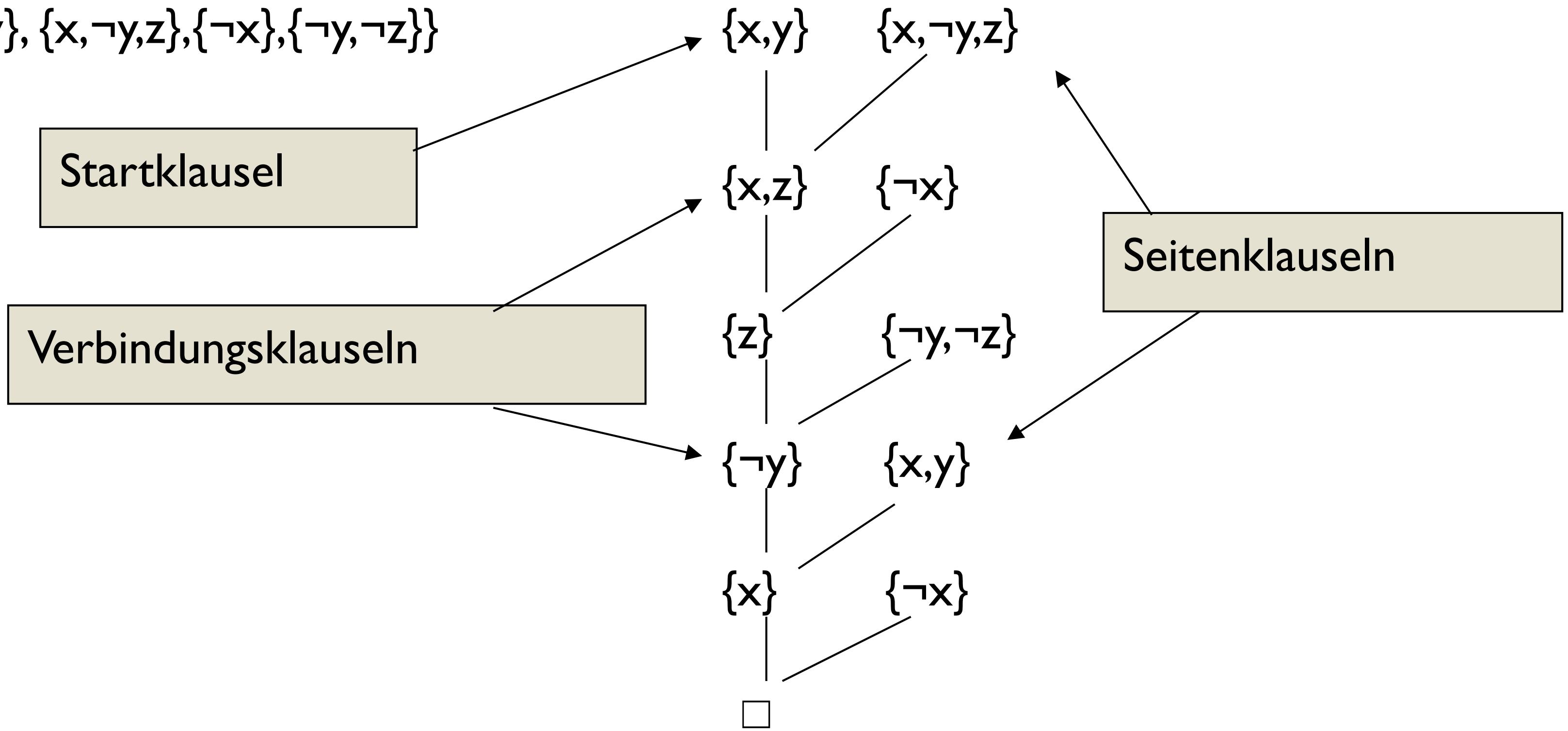
- Ableitungsbegriff: $F \vdash C$
  - Aus der Klauselmenge F lässt sich durch eine endliche Anzahl von Anwendungen der Resolutionsregel die Klausel C herleiten.
  - $\vdash$: Folgerungsoperator (des Resolutionskalküls)
- Resolution ist
  - korrekt: Nur für unerfüllbare Formeln lässt sich die leere Klausel herleiten ($F \vdash \square$ impliziert F unerfüllbar).
  - (widerlegungs-)vollständig: Für jede unerfüllbare Formel gibt es einen Resolutionsbeweis (F unerfüllbar impliziert $F \vdash \square$).

Varianten der Resolution

- Lineare Resolution:
  - linearer Beweis, d.h. letzte hergeleitete Konklusion ist immer Prämisse des nächsten Resolutionssschrittes.
- Geordnete Resolution:
  - zusätzlich: strikte, totale Ordnung auf den Variablen
  - Einschränkung in Anwendbarkeit der Resolutionsregel:
    - Literal, über das resolviert wird, muss in jeder Elternklausel (Prämisse) maximal sein.
- Hyper-Resolution
  - mehrere Resolutionsschritte auf einmal

Lineare Resolution

$F = \{\{x,y\}, \{x,\neg y,z\},\{\neg x\},\{\neg y,\neg z\}\}$

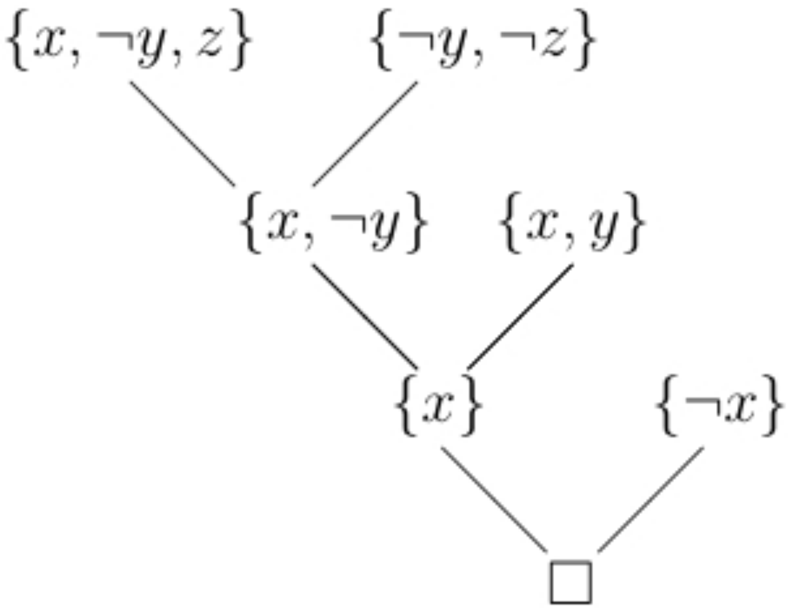


Lineare Resolution ist widerlegungsvollständig.

Geordnete Resolution: Beispiel

$F_3, x \prec y \prec z$

$F_3 = \{\{x, y\}, \{x, \neg y, z\}, \{\neg x\}, \{\neg y, \neg z\}\}$



Resolution nur über maximale Literale.

Geordnete Resolution ist widerlegungs-vollständig.

Variante der geordneten Resolution: Bucket-Elimination

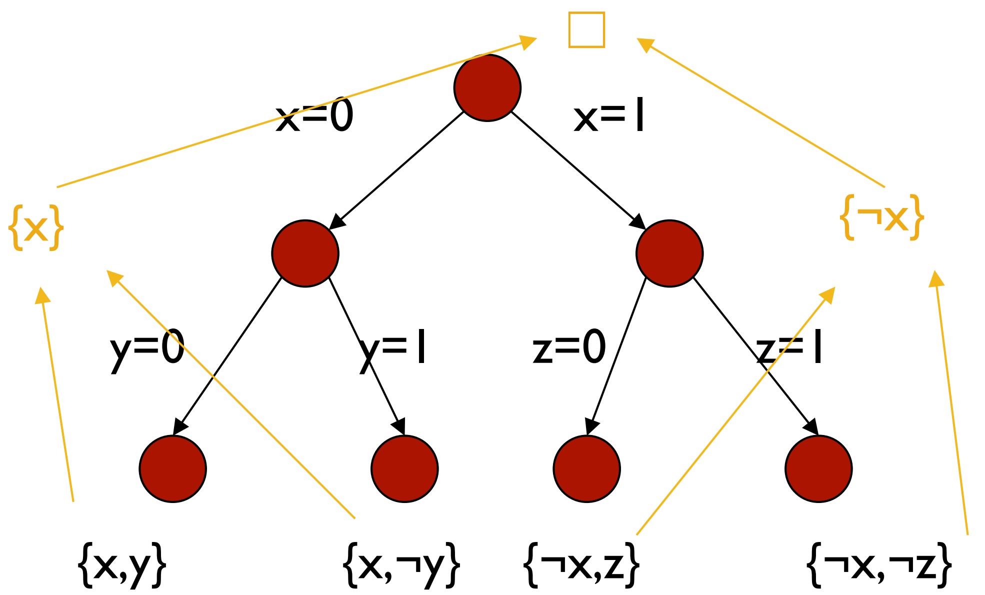
- Jede Klausel wird nach der größten Variable in einen „Bucket“ (Eimer) einsortiert.
- Resolventen werden wieder einsortiert.
- Resolution in Reihenfolge „absteigender Buckets“.

$F = \{\{x,y\}, \{x,\neg y,z\},\{\neg x\},\{\neg y,\neg z\}\}, \quad x < y < z$

B.-Var	Bucket	Resolventen
z	$\{x,\neg y,z\}, \{\neg y,\neg z\}$	
y	$\{x,y\}$	$\{x,\neg y\}$
x	$\{\neg x\}$	$\{x\}$
-		$\square$

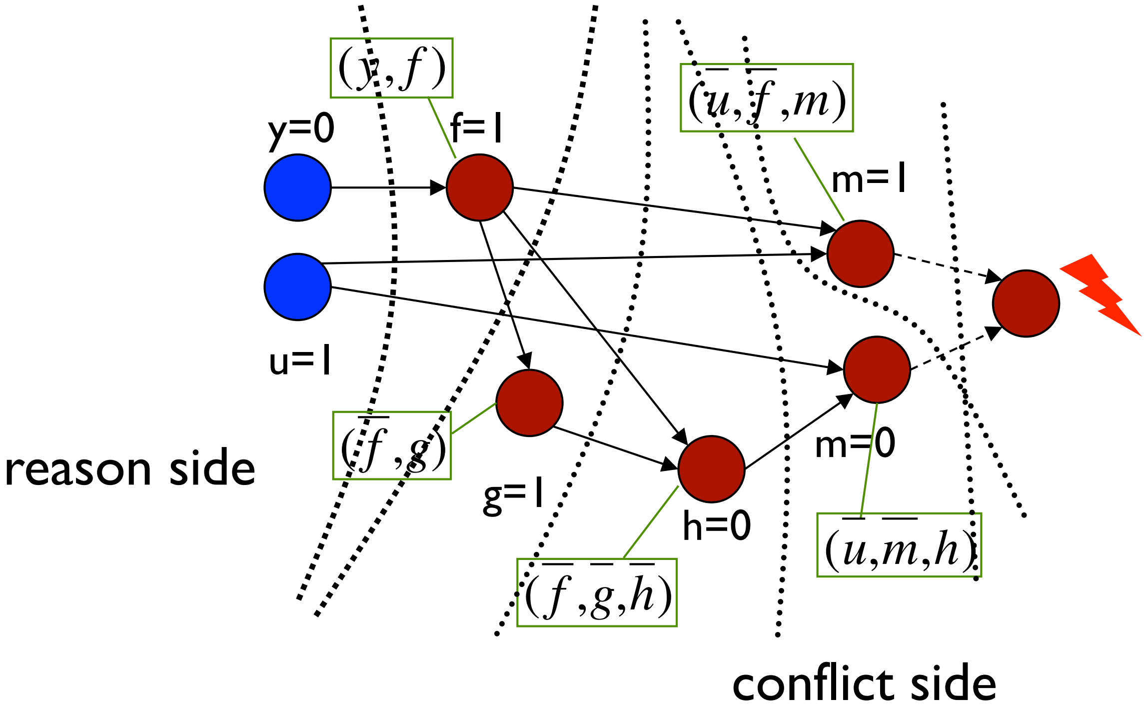
Resolution und DPLL

- Beweise aus Search-Tree
- F = {{x,y},{x,¬y},{¬x,z},{¬x,¬z}}



Konflikt:

Resolutionsbeweise für gelernte Klauseln



Resolutions-Herleitung der Konfliktklausel:

$(\bar{u}, \bar{f}, m)$ $\quad$ $(\bar{u}, \bar{m}, h)$

$(\bar{u}, \bar{f}, h)$ $\quad$ $(\bar{f}, \bar{g}, \bar{h})$

$(\bar{u}, \bar{f}, \bar{g})$ $\quad$ $(\bar{f}, g)$

$(\bar{u}, \bar{f})$ $\quad$ (y, f)

$(\bar{u}, y)$

(y, f) $\quad$ $(\bar{f}, g)$ $\quad$ $(\bar{f}, \bar{g}, \bar{h})$ $\quad$ $(\bar{u}, \bar{m}, h)$ $\quad$ $(\bar{u}, \bar{f}, m)$

Ausblick:
SAT zur Verifikation von C-Programmen

- Zwei verbreitete Verfahren:
  1. Abstraktion (CEGAR: counter-example-guided abstraction refinement)
  2. Bit-Blasting / Bounded Model Checking
- Abstraktion
  - Abstraktion zu Programm mit ausschließlich Booleschen Variablen
    - Partitioniere Variablenbereiche (z.B. int x $\rightarrow$ { $x<0$, $x\geq 0$ })
    - Verwende Model-Checking-Algorithmen

```
int x = -1;
if(x < 0)
    x++;
```

Abstraktion

```
Bool P = false; // P: x≥0
if(!P)
    P = *;
```

Bounded Model Checking: Introductory Example

- What does this function compute?

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8; i *= 2)
    x = x | (x >> i);
  return x+1;
}
```

- next_power_of_two(5)=8
- next_power_of_two(1024)=1024
- How can we verify that this function is correct?

C Bounded Model Checking: Outline

0. Add specification using *assume*/*assert* statements
1. Inline functions (up to a fixed bound *b*)
2. Unroll loops (up to a fixed bound *b*)
   - All program paths are now finite
3. Convert program to *single static assignment* (SSA) form (by renaming variables)
   - Each variable (besides arrays) is written at most once
4. Convert program to set of bit-vector equations
5. Pass equations to an SMT solver *or*
   Convert equations to propositional logic and use a SAT solver

Software BMC by Example: Add Specification

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8; i *= 2)
    x = x | (x >> i);
  return x+1;
}

int main(void)
{
  int x;
  assume(4096 < x && x <= 8192);
  assert(next_power_of_two(x) == 8192);
}
```

Software BMC by Example: Inline Functions

```
int next_power_of_two(int x)
{
  int i;
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  return x+1;
}

int main(void)
{
  int x;
  assume(4096 < x && x <= 8192);
  assert(next_power_of_two(x) ==
                           8192);
}
```

```
int main(void)
{
  int x, i, rt;
  assume(4096 < x && x <= 8192);
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  rt = x+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Unroll Loops

```
int main(void)
{
  int x, i, rt;
  assume(4096 < x && x <= 8192);
  x--;
  for(i=1; i < sizeof(int)*8;
      i *= 2)
    x = x | (x >> i);
  rt = x+1;
  assert(rt == 8192);
}
```

```
int main(void)
{
  int x, rt;
  assume(4096 < x &&
              x <= 8192);
  x--;
  x = x | (x >> 1);
  x = x | (x >> 2);
  x = x | (x >> 4);
  x = x | (x >> 8);
  x = x | (x >> 16);
  rt = x+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Convert to Singe Static Assignment Form

```
int main(void)
{
  int x, rt;
  assume(4096 < x &&
                x <= 8192);
  x--;
  x = x | (x >> 1);
  x = x | (x >> 2);
  x = x | (x >> 4);
  x = x | (x >> 8);
  x = x | (x >> 16);
  rt = x+1;
  assert(rt == 8192);
}
```

```
int main(void)
{
  int x0, x1, x2, x3, x4, x5,
      x6, rt;
  assume(4096 < x0 && x0 <= 8192);
  x1 = x0-1;
  x2 = x1 | (x1 >> 1);
  x3 = x2 | (x2 >> 2);
  x4 = x3 | (x3 >> 4);
  x5 = x4 | (x4 >> 8);
  x6 = x5 | (x5 >> 16);
  rt = x6+1;
  assert(rt == 8192);
}
```

Software BMC by Example: Convert to Bit-Vector Equations

```
int main(void)
{
  int x0, x1, x2, x3, x4, x5,
      x6, rt;
  assume(4096 < x0 &&
                x0 <= 8192);
  x1 = x0-1;
  x2 = x1 | (x1 >> 1);
  x3 = x2 | (x2 >> 2);
  x4 = x3 | (x3 >> 4);
  x5 = x4 | (x4 >> 8);
  x6 = x5 | (x5 >> 16);
  rt = x6+1;
  assert(rt == 8192);
}
```

⇒

```
bv32lt(4096, x0)
bv32le(x0, 8192)
x1 = bv32sub(x0, 1)
x2 = bv32or(x1, bv32ashr(x1, 1))
x3 = bv32or(x2, bv32ashr(x2, 2))
x4 = bv32or(x3, bv32ashr(x3, 4))
x5 = bv32or(x4, bv32ashr(x4, 8))
x6 = bv32or(x5, bv32ashr(x5, 16))
rt = bv32add(x6, 1)
rt ≠ 8192
```