

**FACHHOCHSCHULE WEDEL**  
**SEMINARARBEIT**

in der Fachrichtung

Wirtschaftsinformatik

Thema:

**Implementierung eines Resolutionsbeweisers**

Eingereicht von:	Daniel Dittmann Dieselstraße 30 22307 Hamburg Tel. (040) 32 03 40 43
Erarbeitet im:	6. Semester
Abgegeben am:	8. Juni 2005
Referent:	Prof. Dr. Sebastian Iwanowski Fachhochschule Wedel Feldstraße 143 22880 Wedel Tel. (0 41 03) 80 48-63

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>II</b>
<b>Darstellungsverzeichnis .....</b>	<b>III</b>
<b>Abkürzungsverzeichnis .....</b>	<b>IV</b>
<b>Symbolverzeichnis.....</b>	<b>V</b>
<b>1. Einführung.....</b>	<b>6</b>
1.1. Inhaltliche Anknüpfungspunkte.....	6
1.2. Eingrenzung und Erkenntnisziele.....	6
1.3. Vorgehensweise .....	7
<b>2. Überblick über das System .....</b>	<b>8</b>
<b>3. Die Grundalgorithmen im Detail.....</b>	<b>10</b>
<b>3.1. Der Unifikationsalgorithmus .....</b>	<b>10</b>
3.1.1. Anforderungen und Funktionalität .....	10
3.1.2. Funktionsweise .....	10
<b>3.2. Der Faktorisierungsalgorithmus.....</b>	<b>12</b>
3.2.1. Anforderungen und Funktionalität .....	12
3.2.2. Funktionsweise .....	13
<b>3.3. Der Resolutionsalgorithmus .....</b>	<b>14</b>
3.3.1. Anforderungen und Funktionalität .....	14
3.3.2. Funktionsweise .....	15
<b>3.4. Der Widerlegungsalgorithmus.....</b>	<b>16</b>
3.4.1. Anforderungen und Funktionalität .....	16
3.4.2. Funktionsweise .....	16
<b>3.5. Berechnung von Klauselmengen .....</b>	<b>18</b>
3.5.1. Anforderungen und Funktionalität .....	18
3.5.2. Funktionsweise .....	18
<b>3.6. Der Beweisalgorithmus .....</b>	<b>19</b>
3.6.1. Anforderungen und Funktionalität .....	19
3.6.2. Funktionsweise .....	19
<b>4. Optimierungsansätze.....</b>	<b>21</b>
4.1. Löschregeln und Ableitungsstrategien.....	21
4.2. Klauselgraphen.....	22
4.3. Splitting .....	25
<b>5. Zusammenfassung und Ausblick .....</b>	<b>27</b>
<b>Literaturverzeichnis .....</b>	<b>29</b>

## **Darstellungsverzeichnis**

Darst. 1: Datenfluss im Gesamtsystem .....	8
Darst. 2: Ablaufbeispiel Unifikationsalgorithmus.....	12
Darst. 3: Paarweiser Test auf Resolutionsmöglichkeiten.....	15
Darst. 4: Funktionsweise von Klauselgraphen.....	24
Darst. 5: Vererbung in Klauselgraphen.....	25

## **Abkürzungsverzeichnis**

gdw.                      genau dann wenn

## Symbolverzeichnis

$\mathcal{F}$	Menge aller Formeln der Prädikatenlogik erster Stufe
$\mathcal{L}$	Klauselsprache bzw. Menge aller Klauseln
$\mathcal{T}$	Menge aller Terme aus $\mathcal{L}$
$\mathcal{AT}$	Menge aller Atome aus $\mathcal{L}$ ohne "falsch"
$\mathcal{V}$	Menge der Variablen aus $\mathcal{L}$
$\mathcal{V}(t)$	Menge aller Variablen aus Term $t$
$\text{mg}\mathcal{U}(D)$	Menge der allgemeinsten Unifikatoren der Menge $D$
$\square$	leere Klausel
<b>assume</b>	Vorbedingung im verwendeten Pseudocode

# 1. Einführung

## 1.1. Inhaltliche Anknüpfungspunkte

Ein Teilgebiet der Informatikdisziplin der künstlichen Intelligenz ist die Theorie des automatischen Beweisens, welche sich mit der Frage beschäftigt, wie die Folgerung einer Aussage aus einer Menge von Aussagen - aufgefasst als Axiomsystem - algorithmisch nachgewiesen werden kann.

Ein automatischer Beweiser lässt sich dabei als spezielle Ausprägung eines *Deduktionssystems* verstehen, definiert als "programmierbares Verfahren zur Erkennung von Folgerungsbeziehungen zwischen Aussagen"<sup>1</sup>. Weitere Anwendungen für Deduktionssysteme bestehen z.B. im logischen Programmieren oder in der Programmverifikation.<sup>2</sup>

Deduktionssysteme bedienen sich dabei bestimmter Kalküle, die festlegen, wie aus gegebenen Aussagen neue Aussagen durch rein syntaktische Operationen hergeleitet werden können.<sup>3</sup> In der Praxis besonders erfolgreich ist der Resolutionskalkül<sup>4</sup>, welcher als Grundlage des hier zu implementierenden Systems verwendet werden soll.

## 1.2. Eingrenzung und Erkenntnisziele

Ziel dieser Arbeit ist die Skizzierung einer einfachen Implementierung eines Deduktionssystems zum automatischen Beweisen prädikatenlogischer Formeln erster Stufe unter Nutzung des Resolutionskalküls.

Im Zentrum der Betrachtung stehen dabei konkrete Algorithmen, wobei die primären Erkenntnisziele in deren Verständnis und der Schaffung eines Problembewusstseins für die zugrunde liegenden Implementierungskonzepte bestehen.

Grundlagen der Theorie der Deduktionssysteme und des automatischen Beweisens sind damit nicht Kern der Betrachtung und werden nur ggf. zur Unterstreichung wichtiger Implementierungskonzepte berücksichtigt. Ebenfalls wird auf eine

---

<sup>1</sup> Eisinger, Norbert / Ohlbach, Hans Jürgen: Grundlagen, 1992, S. 1.

<sup>2</sup> Vgl. Eisinger, Norbert / Ohlbach, Hans Jürgen: Grundlagen, 1992, S. 1.

<sup>3</sup> Vgl. Siekmann, Jörg H: Geschichte, 1992, S. 17f.

Darstellung der Hintergründe des Resolutionskalküls verzichtet, wenn auch einige Begriffe und Formeln definiert werden. Des Weiteren werden logische Grundlagen - insbesondere die Prädikatenlogik erster Stufe und die Terminologien der Klauselsprache - sowie die Kenntnis allgemeiner Programmierkonzepte vorausgesetzt.

### **1.3. Vorgehensweise**

Als Einstiegspunkt in die Betrachtungen dient ein Überblick über die Zerlegung des Systems in Teilalgorithmen, wobei insbesondere die Datenflüsse und gegenseitigen Abhängigkeiten der Module von Interesse sein sollen.

Vor diesem Hintergrund erfolgt die detaillierte Betrachtung dieser Module, wobei neben einem Überblick über Anforderungen und Funktionalität der Schwerpunkt auf der Grundidee der Implementierung liegen soll. Auf dieser Grundlage wird eine vollständige Darstellung des jeweiligen Algorithmus in Pseudocode entwickelt.

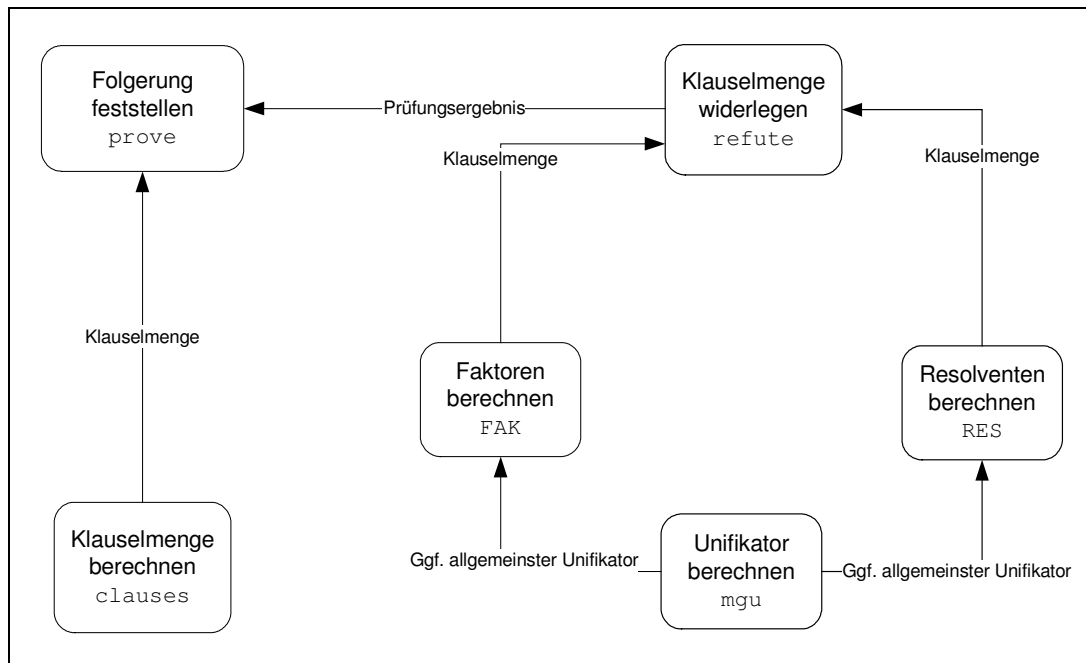
Wie zu zeigen sein wird bedürfen die somit dargestellten Grundversionen der Algorithmen unter dem Blickwinkel der Praxistauglichkeit teils erheblicher Optimierungen. Ansatzpunkte für derartige Effizienzsteigerungen und die beispielhafte, vertiefende Darstellung ausgewählter Konzepte bilden den letzten Abschnitt des Hauptteils dieser Arbeit.

---

<sup>4</sup> Vgl. Walther, Christoph: Beweisen, 2003, S. 199.

## 2. Überblick über das System

Die hier dargestellte Implementierung besteht aus den in Darst. 1 skizzierten sechs Modulen:



Darst. 1: Datenfluss im Gesamtsystem

Der `prove`-Algorithmus fungiert als Schnittstelle für das gesamte System und hat die Aufgabe, die Folgerung der zu beweisenden Formel aus der übergebenen Formelmenge zu überprüfen. Seine Inputgrößen sind dargestellt in Prädikatenlogik erster Stufe, während der im System verwendete Resolutionskalkül lediglich mit Klauseln und Klauselmengen arbeitet. Die sich hieraus ergebende Forderung einer Umformungsmöglichkeit erfüllt das Modul `clauses`, welches aus prädikatenlogischen Formeln Klauselmengen berechnet. Der `prove`-Algorithmus verwendet die so erzeugten Klauseln und übergibt sie an den Algorithmus `refute`, welcher die eigentliche Implementierung des Resolutionskalküls darstellt. Er versucht, die ihm übergebene Klauselmenge zu widerlegen und übermittelt das Ergebnis dieses Versuchs an den Beweisalgorithmus `prove`. Die beiden Schlussregeln des Resolutionskalküls sind in den Routinen `FAK` und `RES` implementiert, welche von `refute` genutzt werden und die berechneten Ergebnisse in Form von Klauselmengen zurückgeben. Beide benötigen zur Herleitung der Klauseln die Prüfung auf Unifizierbarkeit von Termen und Atomen und ggf. die



Berechnung eines allgemeinsten Unifikators, beides geleistet von dem Algorithmus `mg_u`.

### 3. Die Grundalgorithmen im Detail

#### 3.1. Der Unifikationsalgorithmus

##### 3.1.1. Anforderungen und Funktionalität

Der Unifikationsalgorithmus ist als Hilfsalgorithmus für die Routinen `FAK` und `RES` zu verstehen, der diesen den Test auf Unifizierbarkeit und ggf. die Suche eines allgemeinsten Unifikators zur Verfügung stellt. Die Funktionalität beschränkt sich dabei auf den Test von genau zwei Termen oder Atomen, da die Operation für größere Mengen von zu überprüfenden Ausdrücken auf diesen elementaren Fall zurückführbar ist, was ggf. in den aufrufenden Algorithmen zu implementieren ist.<sup>5</sup> Zurückgegeben wird bei Fehlschlag der Suche eine entsprechende Meldung an den Aufrufer, ansonsten die berechnete Substitution.

##### 3.1.2. Funktionsweise

Zunächst sei der Begriff der Unifizierbarkeit von zwei Termen oder Atomen  $t_1$  und  $t_2$  präzisiert, wobei im Folgenden immer von Termen gesprochen wird, die Ausführungen aber analog für Atome zu verstehen sind:

Die Terme sind nicht unifizierbar, wenn einer der beiden eine Variable ist, die im anderen enthalten ist, also  $t_1=x$ ,  $t_2=...f(..x..)...$  Unabhängig von der verwendeten Ersetzung  $\{x/s\}$  wird  $s$  in  $t_1$  immer alleine und in  $t_2$  innerhalb der Funktion stehen, so dass  $t_1 \neq t_2$ . Diese Konstellation wird als *occur failure* bezeichnet.

Ein *clash failure* tritt dann auf wenn die Terme Ausdrücke der Form  $t_1=g(q_1 \dots q_n)$   $t_2=h(r_1 \dots r_m)$  mit  $g \neq h$  sind. Offenbar kann durch Variablenersetzung nie eine Identität von  $g$  und  $h$  erreicht werden, die aber für die Unifizierbarkeit der Terme erforderlich wäre. Insgesamt können im Vergleich zweier Terme folgende Fälle auftreten:

1. Die Terme sind gleich. In diesem Fall wird die Substitution  $\epsilon$  zurückgegeben, da nicht substituiert werden muss.
2. Mindestens ein Term ist eine Variable. Es sind weitere Fälle zu unterscheiden:
  - Die betreffende Variable kommt im anderen Term vor, so dass ein *occur failure* auftritt und die Funktion mit einer entsprechenden Meldung abbricht.

---

<sup>5</sup> Dies ist z.B. der Fall in der Implementierung des Faktorisierungsalgorithmus. Siehe dazu auch S. 13f.

- Andernfalls ist der gesuchte Unifikator derjenige, der die Variable durch den zweiten Term ersetzt, z.B.  $t_1=x$ ,  $t_2=f(y)$  daraus folgt  $\sigma = \{x/f(y)\}$ .
3. Die Terme sind geschachtelte Ausdrücke der Form  $t_1=g(q_1 \dots q_n)$   $t_2=h(r_1 \dots r_m)$ , dies kann bedeuten:
- Es tritt ein clash failure auf, denn  $g \neq h$  oder
  - die Unifizierbarkeit ergibt sich aus derjenigen der Funktionsparameter  $q_i$  und  $r_i$ . In diesem Fall wird über die Parameter iteriert, Teillösungen werden gesucht und diese schrittweise zum gesuchten allgemeinsten Unifikator zusammengeführt.

```

function mgu( $t_1, t_2 \in \mathcal{T} \cup \mathcal{AT}^1$ ) : SUB  $\cup$  {failed}
  if  $t_1 = t_2$  then return {} fi                                // 1. Fall
  if  $t_1 \in \mathcal{V}$  then                                              // 2. Fall
    if  $t_1 \in \mathcal{V}(t_2)$  then
      return failed                                              // "occur failure"
    else
      return { $t_1/t_2$ }
    fi
  fi
  if  $t_2 \in \mathcal{V}$  then return mgu( $t_2, t_1$ ) fi
  assume  $t_1 = \mathcal{G}(q_1 \dots q_n)$ ,  $t_2 = \mathcal{H}(r_1 \dots r_m)$           // 3. Fall
  if  $\mathcal{G} \neq \mathcal{H}$  then return failed fi                            // "clash failure"
  assume  $t_1 = f(q_1 \dots q_n)$ ,  $t_2 = f(r_1 \dots r_n)$ ,  $n > 0$ 
   $\sigma := \epsilon$ 
  for  $i := 1$  to  $n$  do
     $\theta := \text{mgu}(\sigma(q_i), \sigma(r_i))$ 
    assume  $\sigma = \{x_1/s_1, \dots, x_n/s_n\}$ 
     $\sigma := \theta \cup \{x_1/\theta(s_1), \dots, x_n/\theta(s_n)\}$ 
  done
  return  $\sigma$ 
end

```

Der erste Teil der Funktion implementiert die beiden ersten, elementaren Fälle, wobei davon ausgegangen wird, dass ggf. Term  $t_1$  die Variable ist. Im umgekehrten Fall folgt wegen der Symmetrie der Funktion<sup>6</sup> ein rekursiver Aufruf mit entsprechend umgestellten Eingabeparametern.

<sup>6</sup> Vgl. Walthers, Christoph: Beweisen, 2003, S. 215.

Im dritten Fall können bereits bestimmte Annahmen über die Form der Terme gemacht werden, bzw. kann bei Nicht-Auftreten eines clash failures von Funktionen oder Prädikaten mit gleicher "formaler Parameterliste" ausgegangen werden, weshalb wie bereits angedeutet die Unifizierbarkeit von den einzelnen  $r_i$  bzw.  $q_i$  abhängt.

Bei der Iteration über die Parameter wird der gesuchte Unifikator  $\sigma$  schrittweise aufgebaut, indem er zunächst mit der leeren Substitution  $\epsilon$  initialisiert wird. Pro Iterationsschritt wird ein allgemeinsten Unifikator  $\theta$  für den betrachteten Parameter gesucht und der bisherigen Lösung durch Funktionskomposition mit  $\sigma$  hinzugefügt, wobei  $\theta$  selbst auch Teil der Lösung ist. Darst. 2 zeigt den Ablauf für die Beispiel-Terme  $t_1 = f(x\ g(y\ h(y)))$  und  $t_2 = f(z\ g(z\ h(a)))$ . Die Tabelle ist dabei zeilenweise von links nach rechts zu lesen und bezieht sich auf die Anweisungen innerhalb der for-Schleife.

$\sigma$	$\sigma(q_i)$	$\sigma(r_i)$	$\theta$
$\epsilon$	x	z	{x/z}
{x/z}	g(y h(y))	g(z h(a))	{z/a, y/a}
$\epsilon$	y	z	{y/z}
{y/z}	h(z)	h(a)	{z/a}
$\epsilon$	z	a	{z/a}
{z/a}			
{z/a, y/a}			
{z/a, y/a, x/a}			

Darst. 2: Ablaufbeispiel Unifikationsalgorithmus

## 3.2. Der Faktorisierungsalgorithmus

### 3.2.1. Anforderungen und Funktionalität

Der Algorithmus `FAK` implementiert die Schlussregel der Faktorisierung für die Nutzung durch `refute`. Seine Aufgabe besteht in der Berechnung sämtlicher für eine gegebene Klauselmenge möglicher Faktoren. Die Menge dieser neu gebildeten Klauseln vereint mit der Eingabemenge wird zurückgegeben.

### 3.2.2. Funktionsweise

Da Faktoren immer aus genau einer Klausel abgeleitet werden, lässt sich die Faktorisierung von Klauselmengen auf einen Hilfsalgorithmus  $\text{fak}$  zurückführen, der die Menge aller Faktoren genau einer gegebenen Klauseln berechnet. Die Aufgabe von  $\text{FAK}$  beschränkt sich somit auf die Iteration über alle Elemente der Eingabemenge und des jeweiligen Aufrufs dieses Hilfsalgorithmus.

```
function  $\text{FAK}(\mathcal{S} \in 2^{\mathcal{L}}) : 2^{\mathcal{L}}$ 
   $\mathcal{F} := \emptyset$ 
  for all  $C \in \mathcal{S}$  do
     $\mathcal{F} := \mathcal{F} \cup \text{fak}(C)$ 
  done
  return  $\mathcal{F}$ 
end
```

Die zunächst leere Menge  $\mathcal{F}$  wird schrittweise erweitert um die berechneten Faktoren jeder Klausel  $C$  und nach vollständiger Iteration zurückgegeben.

Die Arbeitsweise von  $\text{fak}$  wird durch die Definition der Faktorisierungsoperation  $\text{Fak}$  für eine Klausel  $C$  vorgegeben:

$$\text{Fak}(C, \sigma) = \sigma(C) \text{ gdw. } \sigma \in \text{mg}\mathcal{U}(D) \text{ für ein } D \subset C \quad (1)$$

Die einzelnen Faktoren einer Klausel unterscheiden sich somit lediglich durch den verwendeten Unifikator  $\sigma$ . Die Aufgabe besteht also in der Suche nach allen allgemeinsten Unifikatoren für Teilmengen der Klausel, welche durch paarweisen Test zwischen allen Literalen der Klausel umgesetzt wird. Bei Auffinden eines allgemeinsten Unifikators wird dieser auf die Klausel angewendet und der Algorithmus für den so berechneten Faktor erneut aufgerufen. Dies ist notwendig damit auch Teilmengen  $D$  mit Kardinalität größer zwei überprüft werden können, deren Unifikatoren Faktoren ermöglichen, die sonst nicht gefunden würden. So könnte z.B.  $\{P(x)\} = \text{Fak}(\{P(x), P(y), P(z)\}, \{y/x, z/x\})$  ohne die Rekursion nicht gefunden werden, da immer nur Unifikatoren für zwei Literale ermittelt würden.

```
function  $\text{fak}(C \in \mathcal{L}) : 2^{\mathcal{L}}$ 
   $\mathcal{S} := \emptyset$ 
   $\mathcal{D} := C$ 
```

```

for all  $\mathcal{L} \in C$  do
   $\mathcal{D} := \mathcal{D} - \mathcal{L}$ 
  for all  $\mathcal{K} \in \mathcal{D}$  do
    if not  $\mathcal{L}$  komplementär  $\mathcal{K}$  then
       $\sigma := \text{mgu}(|\mathcal{L}|, |\mathcal{K}|)$ 
      if  $\sigma \neq \text{failed}$  then
         $S := S \cup \text{fak}(\sigma(C))$ 
      fi
    fi
  done
done
return  $S \cup \{C\}$ 
end

```

Die Menge  $S$  fungiert als Akkumulator für die berechneten Faktoren, während  $D$  lediglich temporär für den paarweisen Test der Literale benötigt wird. Pro Literal aus  $C$  wird das entsprechende Literal aus  $D$  gestrichen und die paarweise Verbindung mit den verbliebenen Literalen durchgeführt. Der Test auf Komplementarität gewährleistet, dass Unifikatoren nur für Atome von Literalen mit gleichem "Vorzeichen" gesucht werden. Dies ist erforderlich, da `mgu` als Parameter nur Terme oder Atome akzeptiert. Bei Unifizierbarkeit wird die entsprechende Substitution direkt ausgeführt ebenso wie der rekursive Aufruf und die Menge der berechneten Faktoren in  $S$  gespeichert. Insofern erklärt sich auch, weshalb die ursprüngliche Klausel Teil des Ergebnisses sein muss, denn berechnete Faktoren werden direkt an den rekursiven Aufruf übergeben und würden sonst nicht berücksichtigt.

### 3.3. Der Resolutionsalgorithmus

#### 3.3.1. Anforderungen und Funktionalität

Wie noch zu zeigen sein wird benötigt der Widerlegungsalgorithmus eine Berechnungsroutine für die Menge aller Resolventen, die sich jeweils zwischen Klauseln zweier bestimmter Mengen bilden lassen<sup>7</sup>. Der Algorithmus `RES` als Implementierung der Schlussregel der Resolution hat somit diese Funktionalität zu erfüllen. Zudem soll aus Effizienzgründen die Berechnung abgebrochen werden sobald die leere Klausel resolviert werden konnte, da in diesem Fall der

---

<sup>7</sup> Siehe dazu auch S. 18f.

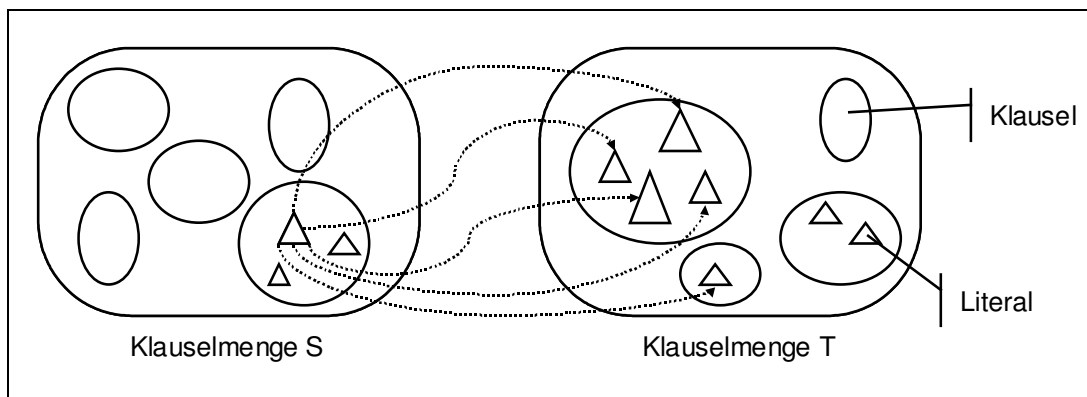
Beweisvorgang abgeschlossen und die Herleitung weiterer Klauseln nicht mehr nötig ist.

### 3.3.2. Funktionsweise

Für die Resolution zweier Klauseln müssen für die Literale, über die resolviert wird, zwei Voraussetzungen gelten:

1. Die Literale müssen komplementär sein.
2. Die den Literalen zugehörigen Atome müssen unifizierbar sein.

Diese Vorbedingungen werden für jede mögliche paarweise Verbindung von Literalen der einen mit Literalen der anderen übergebenen Klauselmenge getestet wie Darst. 3 visualisiert:



Darst. 3: Paarweiser Test auf Resolutionsmöglichkeiten

Sind beide erfüllt liegt eine Resolutionsmöglichkeit vor, wobei gefundene potenzielle Resolventen sofort gebildet und in einem Ergebnisbehälter  $R^*$  gesammelt werden. Bei Herleitung der leeren Klausel wird sofort gestoppt.

```

function RES( $\mathcal{S}, \mathcal{T} \in 2^{\mathcal{L}}$ ) :  $2^{\mathcal{L}}$ 
   $\mathcal{R}^* := \emptyset$ 
   $\mathcal{T}' := \mathcal{T}$ 
  for all  $C \in \mathcal{S}$  do
     $\mathcal{T}' := \mathcal{T} \setminus \{C\}$ 
    for all  $\mathcal{L} \in C$  do
      for all  $\mathcal{D} \in \mathcal{T}'$  do
        for all  $\mathcal{K} \in \mathcal{D}$  do
          if  $\mathcal{L}$  komplementär  $\mathcal{K}$  then
             $\sigma := \text{mgu}(|\mathcal{L}|, |\mathcal{K}|)$ 

```

```

        if  $\sigma \neq \text{failed}$  then
             $\mathcal{R} := \sigma(C - \mathcal{L}) \cup \sigma(D - \mathcal{K})$ 
             $\mathcal{R}^* := \mathcal{R}^* \cup \{\mathcal{R}\}$ 
            if  $\mathcal{R} = \square$  then return  $\mathcal{R}^*$  fi
        fi
    fi
done
done
done
done
return  $\mathcal{R}^*$ 
end

```

Der Algorithmus iteriert über jede Klausel der Menge  $S$ , streicht diese aus  $T$  und testet pro enthaltenem Literal die Bedingungen für jedes Literal jeder verbliebenen Klausel der Menge  $T'$ . Nach vollständiger Iteration enthält  $\mathcal{R}^*$  die Menge aller Resolventen und wird zurückgegeben.

### 3.4. Der Widerlegungsalgorithmus

#### 3.4.1. Anforderungen und Funktionalität

Der Algorithmus `refute` stellt die Implementierung des Resolutionskalküls dar. Während der Beweisalgorithmus als Überbau die Rolle einer Benutzerschnittstelle übernimmt, leistet der Widerlegungsalgorithmus die eigentliche Arbeit des Beweisers. Die Schnittstelle ist dabei auf den Kalkül zugeschnitten; so wird direkt eine Klauselmengende als Eingabeparameter erwartet, die der Algorithmus versucht, zu einem Widerspruch zu führen. Gelingt dies, wird der Wert "true" zurückgegeben, sonst "false".

Dabei ist der Aufgabenbereich des Moduls beschränkt auf die systematische Anwendung der Schlussregeln des Kalküls; diese selbst sind durch die Algorithmen `FAK` und `RES` implementiert.

#### 3.4.2. Funktionsweise

Der Algorithmus berechnet iterativ Mengen von Resolventen bis keine Resolution mehr möglich oder ein Widerspruch gefunden ist, was genau dann der Fall ist, wenn die leere Klausel  $\square$  hergeleitet werden konnte.

Die Implementierung nutzt zu diesem Zweck zwei Datencontainer:



1. einen Akkumulator S für sämtliche Klauseln der bisherigen Herleitung sowie
2. einen Behälter T für die zuletzt berechneten Resolventen.

Dabei wird pro Iteration die Menge aller zwischen Klauseln aus S und Klauseln aus T möglichen Resolventen gebildet, diese wird neuer Inhalt von T und der alte Inhalt von T wird mit S vereinigt und bildet dessen neuen Inhalt.

S und T werden dabei initialisiert mit der um die Menge aller ihrer Faktoren ergänzte Eingabeklauselmenge; zudem wird der Inhalt von T vor jedem Resolutionsvorgang (außer dem ersten) wiederum ergänzt um alle seine Faktoren, wodurch die Vollständigkeit des Verfahrens gesichert wird.

```

function refute( $S \in 2^{\mathcal{L}}$ ) : bool
   $\mathcal{T} := \text{FAK}(S)$ 
   $S := \mathcal{T}$ 
  while  $\mathcal{T} \neq \emptyset$  and  $\square \notin \mathcal{T}$  do
     $\mathcal{T} := \text{RES}(S, \mathcal{T})$ 
    if  $\square \notin \mathcal{T}$  then
       $\mathcal{T} := \text{FAK}(\mathcal{T})$ 
       $S := S \cup \mathcal{T}$ 
    fi
  done
  return not ( $\mathcal{T} = \emptyset$ )
end

```

Jede Klausel der Herleitung lässt sich einer Generation zuordnen. Dabei werden die der Funktion übergebenen Klauseln als Klauseln der Generation 0 bezeichnet, während alle weiteren Klauseln zur Generation t gehören, wenn sie entstanden sind aus der Resolution zwischen Klauseln aus S und Klauseln der Generation t-1.

Nach Initialisierung der Datenstrukturen werden iterativ solange die nachfolgenden Klauselgenerationen erzeugt bis keine Resolutionsmöglichkeit mehr besteht (dies ist genau dann der Fall, wenn  $\mathcal{T} = \{ \}$  also die letzte Iteration keine neuen Klauseln mehr hervorgebracht hat) oder die leere Klausel hergeleitet ist.

### 3.5. Berechnung von Klauselmengen

#### 3.5.1. Anforderungen und Funktionalität

Bedingung für die Nutzbarkeit des Resolutionskalküls durch das System ist die durch den Algorithmus `clauses` geleistete Umformung der übergebenen prädikatenlogischen Formelmengen in entsprechende Klauseldarstellung. Dabei ist zu beachten, dass aufgrund der geringeren Darstellungsmächtigkeit der Klauselsprache die Äquivalenz der Formeln nicht immer erhalten werden kann<sup>8</sup>. Die verwendeten Verfahren (z.B. Skolemisierung) leisten jedoch eine Erhaltung der Unerfüllbarkeit, d.h. für alle endlichen  $\Phi \subset \mathcal{F}$  gilt  $\Phi \models \Box \text{gdw. } \text{clauses}(\Phi) \models \Box$ .

#### 3.5.2. Funktionsweise

Die Umformung verläuft in drei Schritten. Zunächst wird durch die Anwendung syntaktischer Äquivalenzregeln (z.B. der Kommutativität / Distributivität oder der de Morgan'schen Regeln) die prenex Normalform mit Matrix in konjunktiver Normalform hergestellt. Formeln in dieser Darstellungsform können Existenzquantoren beinhalten, welche jedoch in Klauselform nicht darstellbar sind. Insofern wird anschließend eine Skolemisierung der Formel durchgeführt, welche die durch Existenzquantoren gebundene Variablen durch Skolemfunktionen ersetzt, wodurch zwar eine Äquivalenz zu der ursprünglichen Formel nicht mehr gewährleistet werden kann, die Unerfüllbarkeit jedoch erhalten wird (z.B.  $\forall x \exists y [P(x, y)]$  wird zu  $\forall x [P(x, f(x))]$ ). Der letzte Schritt besteht in der Umsetzung der so gewonnen Formel in die für Klauseln übliche Mengendarstellung. Dies bedeutet eine rein syntaktische Umformung, da sich die Formel zuvor bereits in der durch die Klauseldarstellung implizit angenommenen Form befindet. So steht beispielsweise die Klausel  $\{P(x), Q(y)\}$  für die prädikatenlogische Formel  $\forall x, y [P(x) \vee Q(y)]$

```
function clauses( $\Phi \in 2\mathcal{F}$ ) :  $2\mathcal{L}$ 
  assume  $\Phi = \emptyset$  or  $\Phi = \{\varphi_1, \dots, \varphi_n\}$ 
   $S := \emptyset$ 
  if  $\Phi = \emptyset$  then return ( $S$ ) fi
  for  $i := 1$  to  $n$  do
    while "eine Regel auf  $\varphi_i$  anwendbar" do
```

---

<sup>8</sup> Vgl. Walthers, Christoph: Beweisen, 2003, S. 212f.

```

    "wende Regel höchster Priorität auf  $\varphi_i$  an"
  done
  while " $\exists \in$  Präfix von  $\varphi'_i$ " do
    "skolemisiere  $\varphi'_i$ "
  done
   $S := S \cup S(\varphi''_i)$  done
  return S
end

```

In der Vorbedingung werden Bezeichnungen für die Elemente der übergebenen Klauselmengen vereinbart, so dass im Folgenden bei nichtleerer Menge über die Einzelformeln iteriert und pro Formel die angeführten Umformungsschritte angewendet werden können. Die so erzeugten Klauseln  $\varphi''_i$  werden zur Ergebnismenge  $S$  hinzugefügt und diese nach erfolgter Verarbeitung aller Einzelformeln zurückgegeben.

## 3.6. Der Beweisalgorithmus

### 3.6.1. Anforderungen und Funktionalität

Die Routine `prove` implementiert die Schnittstelle des Systems und führt den Beweis auf die durch die Algorithmen `refute` und `clauses` bereitgestellte Funktionalität zurück. Ein- und Ausgabedaten entsprechen dabei denjenigen des Gesamtsystems. Da es sich bei den Eingabeparametern um Formeln der Prädikatenlogik erster Stufe handelt, ist die Nutzung des Resolutionskalküls für den Anwender transparent, das intern genutzte Beweisverfahren ließe sich also in verschiedenen Versionen des Systems austauschen.

### 3.6.2. Funktionsweise

Nach der Definition der Folgerungsbeziehung muss für  $\Phi \in 2^{\mathcal{F}}, \varphi \in \mathcal{F}$  und  $\Phi \models \varphi$  gelten, dass  $\varphi$  für jede Belegung wahr ist, die alle Formeln aus  $\Phi$  erfüllt. Somit muss bei Vorliegen einer Folgerungsbeziehung gelten, dass  $\neg\varphi$  für alle wahren Belegungen von  $\Phi$  falsch ist,  $\Phi \cup \{\neg\varphi\}$  also unerfüllbar sein muss. Umgekehrt gilt, dass bei Unerfüllbarkeit dieser Vereinigungsmenge die ursprüngliche Folgerungsbeziehung gelten muss.

Somit ermöglicht die W-Vollständigkeit des Resolutionskalküls die Implementierung eines vollständigen Beweisalgorithmus, der sich diese Beziehung zu Nutze macht und in drei logischen Schritten abläuft:

1. Umformung der prädikatenlogischen Eingabegrößen in Klauseldarstellung
2. Hinzufügen der negierten zu beweisenden Formel zum Axiomensystem
3. Versuch, die gebildete Menge zu widerlegen

```
function prove( $\Phi \in 2^{\mathcal{F}}, \varphi \in \mathcal{F}$ ) : bool  
     $\mathcal{S}_{\mathcal{A}} := \text{clauses}(\Phi)$                                 // 1. Stufe  
     $\mathcal{S}_{\mathcal{T}} := \text{clauses}(\{\neg\varphi\})$                     // 1./2. Stufe  
     $r := \text{refute}(\mathcal{S}_{\mathcal{A}} \cup \mathcal{S}_{\mathcal{T}})$                         // 2./3. Stufe  
    return  $r$   
end
```

Da die Folgerung genau dann gilt, wenn die erzeugte Menge widerlegt werden konnte, entspricht die Ausgabe des Beweisalgorithmus derjenigen von `refute`.

## 4. Optimierungsansätze

Die bisher dargestellten Algorithmen implementieren die Grundfunktionalität eines Resolutionsbeweisers, sind jedoch bewusst einfach gehalten und bedürfen teilweise für den Einsatz in der Praxis signifikanter Effizienzsteigerungen, wie im Folgenden anhand einiger diesbezüglicher Ansätze vorgestellt sei.

### 4.1. Löschregeln und Ableitungsstrategien

Da der Algorithmus `refute` in jedem Iterationsschritt alle Resolventen zwischen den bisher bestehenden Klauseln bildet, was die Gesamtanzahl der Klauseln der bisherigen Herleitung und damit die Resolutionsmöglichkeiten des nächsten Schrittes erhöht, muss i. Allg. von einem exponentiellen Klauselzuwachs bei der Suche nach der leeren Klausel ausgegangen werden. Unter Effizienzkriterien kann es folglich sinnvoll sein, die Bildung bestimmter Resolventen zu verbieten, um die Anzahl der bis zum Auffinden des Widerspruchs benötigten Resolutionsvorgänge zu verringern.

Löschregeln und Ableitungsstrategien implementieren in diesem Zusammenhang Verbotskriterien zur Bildung neuer Resolventen. Dabei kontrollieren Löschregeln die Bildung neuer Klauseln unabhängig von deren Entstehungsgeschichte, während Ableitungsstrategien diese Entwicklung explizit beachten. Beiden Verfahren ist gemeinsam, dass eine bestimmte Kategorie von Resolventen definiert wird, deren Bildung untersagt ist. Im Folgenden sei beispielhaft je ein Verfahren skizziert.

Eine typische Löschregel betrifft die Bildung von Tautologieklauseln, wobei eine Klausel  $C$  genau dann tautologisch genannt wird wenn  $\{A, \neg A\} \subseteq C$ ,  $C$  also für jede Belegung zu "wahr" ausgewertet wird. Da Klauseln in einer Menge implizit konjunktiv verknüpft sind und "wahr" das neutrale Element der Konjunktion ist, stellt jede Tautologieklausel ein neutrales Element dar und kann weggelassen werden. Bei Implementierung dieser Löschregel muss `refute` derart modifiziert werden, dass Tautologieklauseln aus der initialen Klauselmenge entfernt werden, und in `RES` muss eine Anpassung erfolgen, welche die Herleitung von Tautologieklauseln unterbindet.

Als Beispiel für eine Ableitungsstrategie lässt sich die *set-of-support* Methodik anführen, welche in der initialen Klauselmenge unterscheidet, ob eine Teilmenge

aus dem Axiomsystem oder der zu negierenden Formel hervorgegangen ist. Als set-of-support einer Menge  $S$  wird dabei eine Menge  $T \subset S$  genau dann bezeichnet, wenn die Menge  $S \setminus T$  erfüllbar ist. Die Ableitungsstrategie verbietet die Bildung von Resolventen aus Klauseln der Menge  $S \setminus T$ . Die Zulässigkeit dieses Vorgehens lässt sich durch die Tatsache plausibilisieren, dass erfüllbare Mengen niemals einen Widerspruch hervorbringen können, also Resolventen der Menge  $S \setminus T$  keinen Beitrag zur Suche der leeren Klausel leisten. Konkret lässt sich das dem Beweiser übergebene Axiomsystem als  $S \setminus T$  auffassen und die zu beweisende bzw. zu widerlegende Formel in Klauseldarstellung als set-of-support  $T$ ; die Strategie würde also die Resolution zwischen Axiomen verbieten. Da die Menge der Axiome zunächst den größten Teil der gesamten Klauselmengen ausmacht, wird die Bildung der ersten Generation von Resolventen i. Allg. sehr viel effizienter. Allerdings bringt die Strategie im späteren Verlauf keine Verbesserung mehr, da Klauseln nachfolgender Generationen immer zumindest einen durch Resolution entstandenen Vorgänger besitzen.

## 4.2. Klauselgraphen

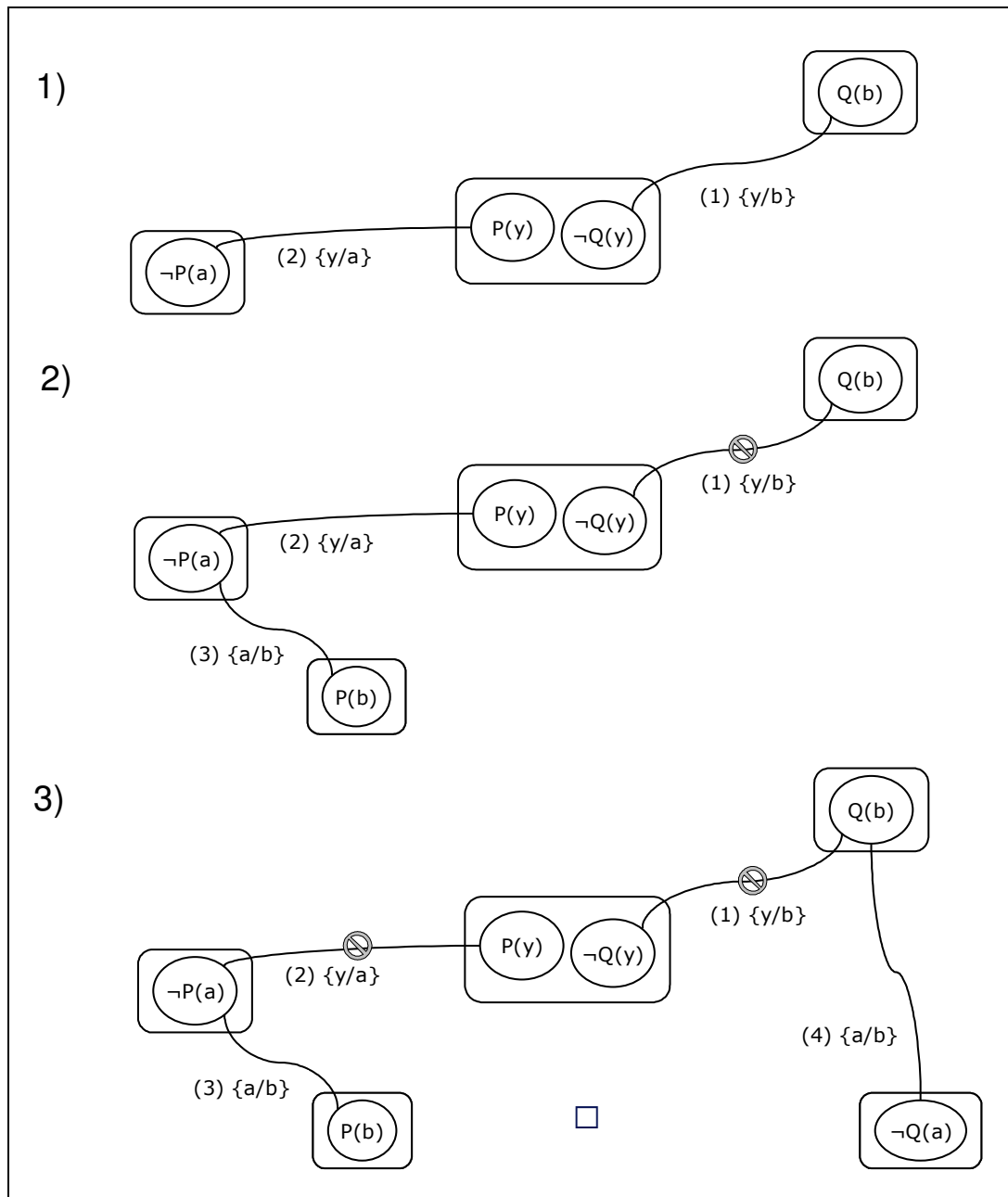
Die im Resolutionsalgorithmus `RES` implementierte Suche nach bildbaren Resolventen hat eine Komplexität von  $O(n*m)$ , da über alle Verbindungen zwischen Literalen der beiden Eingabemengen iteriert werden muss. Dabei sind  $n$  und  $m$  jeweils die Anzahl der Literale der Klauselmengen. In der Regel kann jedoch nur ein relativ kleiner Teil der getesteten Verbindungen resolviert werden.<sup>9</sup> Offenbar böte sich also ein Optimierungsansatz, wenn man die Menge der zu prüfenden Verbindungen aufgrund bestimmter Überlegungen bereits im Voraus einschränken könnte, was durch die Benutzung der Klauselgraphen ermöglicht wird.

Ein Klauselgraph ist ein ungerichteter Graph mit Literalen als Knoten und Resolutionsmöglichkeiten als Kanten, d.h. zwei Literale sind durch eine Kante verbunden, wenn die jeweils zugehörigen Klauseln über sie resolvierbar sind. Jede Kante ist dabei mit dem entsprechenden allgemeinsten Unifikator markiert und mit einer fortlaufenden Nummer versehen. Zu beachten ist ferner, dass Kanten nur zwischen Literalen unterschiedlicher Klauseln bestehen dürfen.

---

<sup>9</sup> Vgl. Walthers, Christoph: Beweisen, 2003, S. 232f.

Die Resolventen werden in der Reihenfolge der Nummerierung gebildet und in den Graphen eingefügt, wobei neue Kanten jeweils die nächste freie Nummer erhalten und die initiale Nummerierung willkürlich gebildet wird. Abgearbeitete Kanten werden gesperrt. Zum Finden neuer Kanten ist es zunächst noch erforderlich, dass der neue Knoten mit allen bereits bestehenden auf Resolutionsmöglichkeiten untersucht wird - dieses Verfahren entspricht der vollständigen Suche des bisherigen RES-Algorithmus. Darst. 4 zeigt einen beispielhaften Klauselgraphen sowie die Schritte der ersten Resolutionen, wobei die leere Klausel als Ergebnis des dritten, nicht mehr explizit aufgeführten Resolutionsschrittes entsteht.

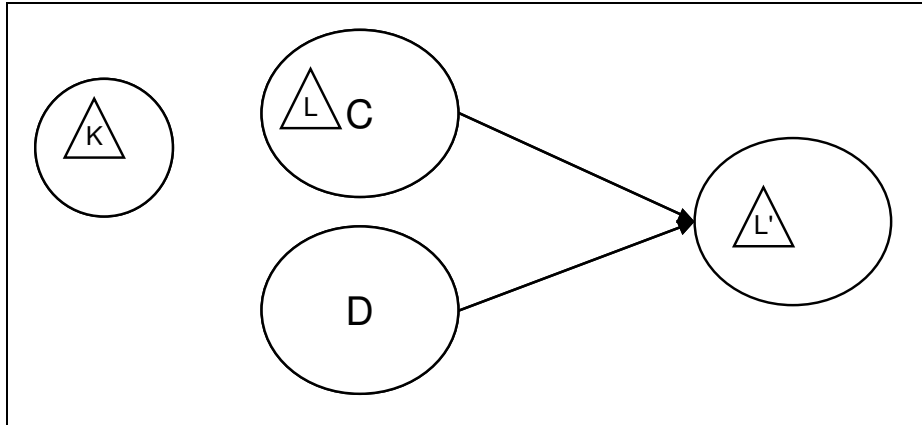


Darst. 4: Funktionsweise von Klauselgraphen

Die Idee des Einsparens von Suchschritten basiert auf dem *Vererbungskonzept* von Kanten: Jedes Literal  $L'$  eines zwischen zwei Klauseln  $C$  und  $D$  gebildeten Resolventen ist in einer der *Elternklauseln*  $C$  und  $D$  enthalten, wobei sich enthaltene Variablen nur durch die bei der Resolution verwendete Substitution  $\sigma$  unterscheiden können. Das Literal in der Elternklausel wird auch als *Elternliteral* bezeichnet. Besteht zwischen dem Elternliteral  $L$  und einem Literal  $K$  einer weiteren Klausel keine Kante, bedeutet dies, dass eine der Vorbedingungen für die Resolvierbarkeit nicht gegeben ist, d.h. entweder  $L$  und  $K$  sind nicht komplementär oder nicht



unifizierbar. Unabhängig von  $\sigma$  folgt daraus, dass  $L'$  diese Eigenschaften beibehält und ebenfalls nicht mit  $K$  resolvierbar sein wird. Die entsprechenden Zusammenhänge sind in Darst. 5 skizziert:



Darst. 5: Vererbung in Klauselgraphen

Aus diesen Überlegungen heraus lässt sich durch die Nutzung von Klauselgraphen die Resolventensuche dahingehend vereinfachen, dass die Suche nach neuen Kanten bei Einfügen neuer Knoten eingeschränkt wird auf

1. Partner der Elternliterale und
2. Literale der Elternklauseln.

Die zweite Bedingung ist notwendig, da mögliche Verbindungen zwischen neuen Literalen und denjenigen der Elternklauseln nicht vererbt werden können, denn Kanten bestehen immer nur zwischen Literalen verschiedener Klauseln.

### 4.3. Splitting

Das Splitting-Konzept setzt am Beweisalgorithmus `prove` an, indem die zu beweisende Formel  $\varphi$  ggf. in ihre durch Konjunktionen verknüpfte Bestandteile zerlegt wird, welche einzeln bewiesen werden. Dieses Vorgehen basiert auf der Überlegung, dass für  $\varphi = \varphi_1 \wedge \dots \wedge \varphi_n$  gilt  $\Phi \models \varphi$  genau dann wenn  $\Phi \models \varphi_1$  und ... und  $\Phi \models \varphi_n$ . I. Allg. wird dadurch nicht nur, wie Walther feststellt, die Gesamtanzahl der hergeleiteten Klauseln und Literale reduziert<sup>10</sup>, sondern zudem müssen im Misserfolgsfall nur durchschnittlich die Hälfte der  $\varphi_i$  überprüft werden,

denn bei Nicht-Widerlegbarkeit nur einer der Teilformeln ist der gesamte Beweis fehlgeschlagen und das Verfahren kann abgebrochen werden.<sup>11</sup>

Insofern wird die Funktion `prove` wie folgt erweitert:

```
function prove( $\Phi \in 2^{\mathcal{F}}$ ,  $\varphi \in \mathcal{F}$ ) : bool
  assume  $\varphi = \varphi_1 \wedge \dots \wedge \varphi_k, k \geq 1$ 
   $S_{\mathcal{A}} := \text{clauses}(\Phi)$                                 // 1. Stufe
   $i := 0$ 
  repeat
     $i := i + 1$ 
     $S_{\mathcal{T}} := \text{clauses}(\{\neg \varphi_i\})$                     // 1./2. Stufe
     $r := \text{refute}(S_{\mathcal{A}} \cup S_{\mathcal{T}})$                         // 2./3. Stufe
  until not  $r$  or  $i = k$ 
  return  $r$ 
end
```

Anders als in der ursprünglichen Version des Algorithmus entsteht  $S_{\mathcal{T}}$  nicht mehr aus der gesamten Formel  $\varphi$  sondern nur aus einer Teilformel. Im Gegenzug muss über sämtliche Teilformeln  $\varphi_i$  iteriert werden bzw. bis zum Auftreten eines eventuellen Fehlschlags des Beweises.

---

<sup>10</sup> Vgl. Walther, Christoph: Beweisen, 2003, S. 231.

<sup>11</sup> Wobei zu beachten ist, dass der Algorithmus bei Misserfolg nicht zwingend terminiert. Vgl. Walther, Christoph: Beweisen, 2003, S. 204f.

## 5. Zusammenfassung und Ausblick

Diese Arbeit zeigt die Grundstruktur einer Implementierung eines automatischen Beweisers auf Basis des Resolutionskalküls unter Angabe von beispielhaften, einfachen Algorithmen für die wichtigsten Module wie z.B. die Unifikationseinheit und die Widerlegungsfunktion.

Der Grundgedanke der Implementierung besteht in der systematischen Erweiterung einer Menge von Klauseln durch Herleitung neuer Klauseln mittels Anwendung der Schlussregeln des Resolutionskalküls. bis zur Herleitung der leeren Klausel. Dabei ist sicherzustellen, dass diese - wenn vorhanden - in endlicher Zeit und darüber hinaus in möglichst wenig Ableitungsschritten gefunden wird.

Wie gezeigt wurde sind die entwickelten Algorithmen für eine praktische Anwendung noch stark optimierungsbedürftig, wobei als Ansatzpunkte für mögliche Effizienzsteigerungen insbesondere die Algorithmen `RES`, `refute` und `prove` infrage kommen.

Als Optimierungsansatz für den Widerlegungsalgorithmus wurde das Konzept der Ableitungsstrategien und Löschregeln vorgestellt, welche durch Implementierung von Verbotskriterien für die Herleitung bestimmter Klauseln eine Reduktion des Wachstums der Klauselmenge innerhalb des Algorithmus erreichen. Als Beispiel für eine Löschregel wurde das Weglassen von Tautologieklauseln angeführt und die `set-of-support` Strategie diente als Beispiel für eine Ableitungsstrategie. Weiterhin wurde gezeigt wie und warum Klauselgraphen den Suchaufwand nach neu zu bildenden Resolventen verringern können, indem sie durch Beachtung des Vererbungskonzeptes die Menge der potenziellen Resolutionsmöglichkeiten zwischen bestehenden und neu gebildeten Klauseln verkleinern. Schließlich wurde der Algorithmus `prove` dahingehend erweitert, dass er die zu beweisende Formel in ihre konjunktiv verknüpften Teile zerlegt und diese einzeln prüft, so dass im Vergleich zum Grundalgorithmus sowohl die Anzahl der Klauseln und Literale als auch der durchschnittliche Rechenaufwand bei Misserfolg reduziert ist.

Insgesamt bleibt festzuhalten, dass die dargestellten Algorithmen im Wesentlichen einer Verständnisförderung des generellen Aufbaus und der Implementierungskonzepte für Resolutionsbeweiser dienlich sind, und praktische Implementierungen Randbedingungen besonders der Effizienz zu beachten haben.

Besonders auf dem Gebiet der Suchstrategien im Widerlegungsalgorithmus bzw. der Ableitungsstrategien wurde in diesem Zusammenhang intensive Forschungsarbeit geleistet. Dabei sind neben den Chancen auch Risiken zu verzeichnen, z.B. kann die parallele Benutzung verschiedener Ableitungsstrategien zu Problemen führen und bestimmte Ableitungsstrategien können die Vollständigkeit von `refute` gefährden, so dass die Anwendbarkeit der Ableitungsstrategien bestimmten Randbedingungen gehorchen muss.

## Literaturverzeichnis

- Eisinger, Norbert / Ohlbach, Hans Jürgen, 1992: [Grundlagen] und Beispiele, in: Bläsius, Karl-Hans / Bürckert, Hans-Jürgen (Hrsg.): Deduktionssysteme, Automatisierung des logischen Denkens, Internet <http://www.dfki.uni-sb.de/~hjb/Deduktionssysteme/I-Geschichte-und-Anwendungen.pdf>, Abruf 2005-06-07
- Siekman, Jörg H., 1992: [Geschichte] und Anwendungen, in: Bläsius, Karl-Hans / Bürckert, Hans-Jürgen (Hrsg.): Deduktionssysteme, Automatisierung des logischen Denkens, Internet <http://www.dfki.uni-sb.de/~hjb/Deduktionssysteme/II-Grundlagen-und-Beispiele.pdf>, Abruf 2005-06-07
- Walther, Christoph, 2003: Automatisches [Beweisen], in: Görz, Günther et al (Hrsg.): Handbuch der Künstlichen Intelligenz, 4., korrigierte Auflage, München, Wien: Oldenbourg, 2003, S. 199-236