INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67 D–80538 München

Ludwig–
Maximilians–
Universität
München

Variationen über ein Thema: Suchstrategien und Datenstrukturen für SATCHMO-Beweiser

François Bry **Rainer Manthey**

Appeared in Proc. 11th Workshop Logische Programmierung, A. Krall und U. Geske Hrsg.,
GMD-Studien Nr. 270, Sept. 1995
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1995-2
September 1995

Variationen über ein Thema: Suchstrategien und Datenstrukturen für SATCHMO-Beweiser

François Bry Rainer Manthey*

September 1995

Abstract

Mit der kleinen Sammlung SATCHMO von Prolog-Programmen ist ein Ansatz zum automatischen Beweisen in der Prädikatenlogik erster Stufe eingeführt worden, dem ein besonders effizientes Modellgenerierungsverfahren zugrunde liegt. Dieses Verfahren ist in seinem Prinzip der Logikprogrammierung sehr nahe, was eine extrem kurze Implementierung in Prolog ermöglicht. In diesem Artikel werden die Implementierungen in Prolog von alternativen Suchstrategien und Datenstrukturen für SATCHMO-Programme erläutert. Dabei werden die Affinität des Modellgenerierungsverfahrens zur Logikprogrammierung sowie einige Mängel der Sprache Prolog erörtert.

1 Einleitung

Mit den Artikeln [MB87, MB88] wurde gegen Ende der 80er Jahre die Sammlung SATCHMO von Prolog-Programmen und damit ein neues Modellgenerierungsverfahren für die Prädikatenlogik erster Stufe eingeführt. Merkmale, die die Sammlung kennzeichnen, sind einerseits die Einfachheit der Programme, andererseits die Effizienz des Modellgenerierungsverfahrens. Dank dieser Eigenschaften wurde das Verfahren zur Lösung verschiedener Probleme unter anderem zum Entwurf von Datenbankschemata, zur Programmverifikation und -synthese [HFF91], zum juristischen Schließen und zur modellbasierten Diagnose [FN90] angewendet sowie weiterentwickelt [FHKF92, HKF92, Ram91, LRW93].

Die Grundprogramme der SATCHMO-Sammlung sind aber oft misverstanden worden, andere Programme sind bisher noch nicht veröffentlicht worden. Ziel dieses Artikels ist es, die Implementierung von bisher nicht veröffentlichen Programmen systematisch zu erläutern, die in Suchstrategien oder Datenstrukturen von den in den Artikeln [MB87, MB88] eingeführten Programmen abweichen. Dieser Artikel ist ein Teil eines ausführlicheren Berichts [Bry95], der auch andere Themen (die inkrementelle Auswertung von Klauseln, Syntaxerweiterungen, eine Implementierung des Tableauverfahrens, und ein Programm zur Überprüfung der Erfüllbarkeit im Endlichen) behandelt.

Der Artikel setzt sich wie folgt zusammen. Der erste Abschnitt ist diese Einleitung. Das Grundprogramm der SATCHMO-Sammlung wird im Abschnitt 2 wiederholt. Die Suchstrategie dieses Grundprogramms, die Tiefensuche, und ihre Implementierung in Prolog werden im Abschnitt 3 behandelt. Alternative Implementierungen der Tiefensuche und des Grundprogramms, die Akkumulationslisten anstatt von Nebeneffekten verwenden, werden im Abschnitt 4 vorgestellt. Im Abschnitt 5 werden die zwei eingeführten Versionen des Grundprogramms

*Also: Institut für Informatik III, Universität Bonn, Römerstr. 164, 53117 Bonn

verglichen. Die Breitensuche und ihr Einbau in einen SATCHMO-Beweiser werden im Abschnitt 6 behandelt. Eine alternative Datenstruktur zur Darstellung der Klauselmenge wird im Abschitt 7 gegeben. Der Abschnitt 8 ist der Implementierung von fairen Suchstrategien gewidmet. Schlußbemerkungen bilden den Abschnitt 9.

2 Das Grundprogramm

SATCHMO erbt von der Logikprogrammierung ihre effiziente Behandlung von Hornklauseln. Um allgemeine Klauseln zu behandeln, stützt sich SATCHMO auf das Vorwärtsschließen und auf Fallunterscheidungen. Die Eingabesprache von SATCHMO besteht aus Klauseln in Implikationsform, das heißt ohne explizites Vorkommnis des Negationssymbols. Die Klausel $\{p(x,y), \neg q(y,z), \neg r(x,z), t(z)\}$ wird zum Beispiel als `q(Y,Z), r(X,Z) ---> p(X,Y) ; t(Z)` dargestellt. Diese Darstellung setzt voraus, daß das Symbol `--->` als Prolog-Infixoperator deklariert wird (mittels `:- op(1200, xfx, --->)`). Der linke Teil einer Implikation wird Prämisse oder Rumpf genannt, der rechte Teil Konklusion oder Kopf. Eine Klausel, die kein negatives Literal enthält, wird als Implikation mit dem Rumpf `true` dargestellt. Eine Klausel, die kein positives Literal enthält, wird als Implikation mit Kopf `false` dargestellt. Das nullstellige Relationssymbol `true` ist in Prolog immer erfolgreich. Das Relationssymbol `false` wird im Programm nicht weiter definiert und scheitert deswegen. Variablen in Klauseln werden durch Variablen der Implementierungssprache, nämlich Prolog, dargestellt. Die Prolog-Darstellung der Konjunktion (durch `,`) und der Disjunktion (durch `;`) wird ebenfalls in die SATCHMO-Eingabesprache übernommen. Relationssymbole, Konstantensymbole bzw. Funktionssymbole, die in den Klauseln vorkommen, werden als Prolog-Relationssymbole, -Konstantensymbole bzw. -Funktionssymbole dargestellt.

Das Vorwärtsschließen aus einer Klausel `A1, A2, ..., An ---> B1 ; B2 ; ... ; Bm` findet im SATCHMO-Verfahren erst statt, wenn eine Teilinterpretation aufgebaut wurde, die den Rumpf `A1, A2, ..., An` der Klausel erfüllt. Sei dieser Rumpf für eine Substitution σ erfüllt, dann wird eine Fallunterscheidung über den Kopf B1σ ; B2σ ; ... ; Bmσ durchgeführt. Bei jedem Fall wird die bereits aufgebaute Interpretation um ein Atom Biσ erweitert. Ist der Kopf ein Atom, dann wird die bereits aufgebaute Interpretation um dieses Atom erweitert, es sei denn, dieses Atom ist `false`. In diesem Fall wird der Zweig des Suchraumes geschlossen. Anfangs ist lediglich das Atom `true` erfüllt. Daher wird anfangs nur aus Klauseln geschlossen, deren Rümpfe `true` sind.

Bei den meisten SATCHMO-Programmen wird angenommen, daß die Klauseln bereichsbeschränkt sind, das heißt, daß jede im Kopf vorkommende Variable auch im Rumpf vorkommt. Diese Annahme stellt sicher, daß bei einer Durchführung der Fallunterscheidung auf einem Klauselkopf alle in diesem Kopf vorkommenden Variablen mit variablenfreien Termen instanziiert sind. Wegen der Bereichsbeschränkung und des Vorwärtsschließens ist nur eine eingeschränkte Form der Unifikation, Matching genannt, erforderlich. Daher ist der in den Standardimplementierungen von Prolog fehlende Occur-check überflüssig. Die Bereichsbeschränkung ist bei der natürlichen Repräsentation der meisten Anwendungen, auch aus dem mathematischen Bereich, vorhanden. Sie kann als Erweiterung der Sortenlogik angesehen werden.

Das folgende Beispiel veranschaulicht die Generierung eines Modells, dargestellt als Atommenge, aus einer Klauselmenge. Die Verzweigungen stellen die Fallunterscheidungen, die Knoten die hergeleiteten variablenfreien Atome und Disjunktionen dar. Dieses Beispiel verdeutlicht, daß das SATCHMO-Modellgenerierungsverfahren einen wesentlich kleineren

Suchraum expandiert als ein Tableauverfahren [Smu68, Fit90].

```
true ---> p(a) ; q(b)            p(b) ---> false
p(X) ---> p(f(X)) ; q(f(X))      p(f(X)) ---> false
q(X) ---> p(X) ; r(X)            p(X), q(f(X)) ---> false
                             true
                               |
                          p(a) ; q(b)
                               |
                +--------------+---------------+
                |                              |
              p(a)                           q(b)
                |                              |
        p(f(a)) ; q(f(a))              p(b) ; r(b)
                |                              |
     +----------+----------+        +----------+----------+
     |                     |        |                     |
  p(f(a))               q(f(a))   p(b)                  r(b)
     |                     |        |
   false                 false    false
```

Die Menge $\mathcal{M}$ der Atome entlang eines Zweiges, dessen Blatt nicht `false` ist, stellt ein Modell der Klauselmenge dar: ein Atom A ist in diesem Modell genau dann wahr, wenn $A \in \mathcal{M}$. Das Modell ist also nur durch die von ihm erfüllten positiven Literale repräsentiert.

Das oben geschilderte Verfahren ist korrekt für die Erfüllbarkeit, jedoch für diese Eigenschaft nicht vollständig. Ist eine Fairneß-Bedingung (s. Abschnitt 8 unten) erfüllt, dann ist das Verfahren für die Unerfüllbarkeit vollständig [BM87]. Das Verfahren kann also als Refutationsbeweiser wie folgt angewendet werden: Um festzustellen, ob eine Formel F aus einer Formelmenge $\mathcal{F}$ folgt, kann das Verfahren auf eine aus $\mathcal{F} \cup \{\neg F\}$ berechnete Klauselmenge in Implikationsform angewendet werden.

Das folgende Programm ist das Grundprogramm der SATCHMO-Sammlung. Zusätzlich zu diesem Programm müssen alle in den SATCHMO-Klauseln vorkommenden Relationsymbole als dynamisch deklariert werden. Für ein 3-stelliges Relationssybol `p` geschieht dies durch den Prolog-Aufruf: `:- dynamic(p/3).` Einige Prolog-Systeme verlangen auch, daß das Atom `false` wie folgt bekannt gemacht wird: `:- assert(false), retract(false).`

```
satisfiable :-
   (Body ---> Head),
   violated_instance(Body ---> Head),
   !,
   satisfy(Head),
   satisfiable.
satisfiable.

component(Atom, (Atom ; _Rest)).
component(Atom, (_Beg ; Rest)) :-
   !,
   component(Atom, Rest).
component(Atom, Atom).

violated_instance(Body ---> Head) :-
   Body,
   not Head.

satisfy(Exp) :-
   component(Atom, Exp),
   not (Atom = false),
   asserta(Atom),
   on_backtracking(retract(Atom)).

on_backtracking(_X).
on_backtracking(X) :-
   once(X),
   fail.
```

Programm 1: das Grundprogramm der SATCHMO-Sammlung.

Dieses Grundprogramm repräsentiert die aufzubauende Interpretation (also eine Menge $\mathcal{M}$ im obigen Beispiel) durch Fakten in der Prolog-Datenbank. Bei einem Aufruf von `satisfiable` wird zunächst eine Instanz einer Implikation ermittelt, die von der bereits generierten Interpretation falsifiziert ist, das heißt, deren Rumpf erfüllt ist, aber deren Kopf nicht erfüllt ist. Dieser Kopf wird dann dadurch erfüllt, daß eines seiner Atome in die Prolog-Datenbank eingefügt wird. Führt solch ein Einfügen zur Herleitung von `false` in einem späteren Schritt, wird dann ein Rücksetzen ausgelöst, wodurch das eingefügte Atom wieder zurückgenommen werden kann. Das nächste Atom im Kopf wird dann eingefügt, und das Verfahren setzt sich gleichermaßen fort. Ist eine Klauselinstanz berechnet, die nicht erfüllt werden kann, dann scheitert erwartungsgemäß der Aufruf von `satisfiable`. Gibt es keine Klauselinstanz mehr, die falsifiziert ist, dann ist der Aufruf von `satisfiable` wegen der zweiten Klausel dieser Prozedur erwartungsgemäß erfolgreich. Ein dritter möglicher Fall tritt ein, wenn immer neue falsifizierte Klauselinstanzen gefunden werden. In dem Fall terminiert das Verfahren nicht.

3 Die dem Grundprogramm zugrundeliegende Tiefensuche

Die Suchmethode, die dem Programm 1 zugrundeliegt, wird im folgenden Programm 2 verdeutlicht.

```
search :-
   read_first(state(S)),
   final(S).
search :-
   read_first(state(S)),
   transition(S, Next),
   push(state(Next)),
   on_backtracking(pop(state(Next))),
   search.
```

```
read_first(Atom) :-
   once(Atom).

push(Atom) :-
   asserta(Atom).

pop(Atom) :-
   retract(Atom)
```

Programm 2: Die dem Programm 1 zugrundeliegende Implementierung der Tiefensuche.

Es wird im Programm 2 angenommen, daß die Prolog-Datenbank beim Aufruf von `search` ein `state`-Atom zur Spezifikation des Initialzustands enthält. Bei einem Aufruf von `search` wird mit dem Aufruf von `final(S)` zunächst überprüft, ob der Initialzustand ein Zielzustand ist. Ist es nicht der Fall, wird mit dem Aufruf von `transition(S, Next)` einen Nachfolgerzustand ermittelt und in einen Keller gespeichert (Aufruf von `push(state(Next))`). Die Suche wird dann mit dem rekursiven Aufruf von `search` ab diesem neuen Zustand fortgesetzt. Ist ein Zustand erreicht, der kein Zielzustand ist und keinen Nachfolger hat, dann scheitert die Suche und das Rücksetzen tritt ein. Der zuletzt in den Keller gespeicherten Zustand wird beim Rücksetzen entfernt (Aufruf von `on_backtracking(pop(state(Next)))`). Das Verfahren setzt sich mit einer alternativen Transistion fort.

Die Prolog-Relation `state` dient in diesem Programm der Implementierung eines Kellers. Die Systemprozedur `asserta` wird aufgerufen, um ein Element – das heißt ein `state`-Faktum – in diesen Keller – das heißt am Anfang der Definition von `state` – einzufügen. Mit der Systemprozedur `retract` wird das erste Element des Kellers gelöscht.

Das Programmieren mit `asserta` und `retract` ist nicht unumstritten, weil damit Programme verfaßt werden können, die sich während deren Aufrufe ändern und deswegen keine deklarative Semantik haben. Ohne gegen diese allgemeine Behauptung Einwände erheben zu wollen, wird auf die sehr eingeschränkte Nutzung von `asserta` und `retract` in den oben gegebenen Programmen Programm 1 und Programm 2 hingewiesen. Die während eines Aufrufes dieser Programme geänderte Teilprogramme, im Falle von Programm 2 die Definition von `state`, können wohl als Datenstruktur angeshen werden. Die `state`-Relation mit den Prozeduren `read_first`, `push` und `pop` stellt lediglich eine Implementierung des abstrakten Datentyps Keller dar.

4 Akkumulationslisten zur Implementierung der Tiefensuche

Die Frage stellt sich natürlich, ob der vom letzten Programm benutzten Keller nicht auch ohne `asserta` und `retract` implementiert werden kann. Das folgende Programm ist solch eine Implementierung, die den Keller mittels Akkumulationslisten implementiert.

```
search(L, L) :-
   read_first(state(S), L),
   final(S).
search(L1, L3) :-
   read_first(state(S), L1),
   transition(S, Next),
   push(state(Next), L1, L2),
   search(L2, L3).

read_first(Atom, [Atom | _Rest]).

push(Atom, L, [Atom | L]).
```

Programm 3: Akkumulationslisten zur Implementierung der Tiefensuche.

Bei dem ersten Aufruf von `search` soll das erste Argument eine Liste sein, deren einziges Element der Initialzustand ist. Das zweite Argument soll beim Aufruf eine uninstanziierte Variable sein, die bei einem Erfolg von `search` als Wert den Keller – das heißt den Zweig vom Initialzustand bis zum ermittelten Zielzustand – liefert. Da das Rücksetzen in Prolog vorhanden ist, bedarf es keine Spezifikation der Prozedur `pop`: Das Rücksetzen auf einem Aufruf von `push` hat den Effekt einer `pop`-Operation. Programm 3 unterscheidet sich vom Programm 2 lediglich in der Implementierung der Keller-Datenstruktur.

Das folgende Programm ist eine Version des im Abschnitt 2 eingefürhten SATCHMO-Grundprogramms, das Akkumulationsliste zur Implementierung des Kellers nach dem Vorbild von Programm 3 verwendet.

```
satisfiable(Int1, Int3) :-
   (Body ---> Head),
   violated_instance(Body ---> Head, Int1),
   !,
   satisfy(Head, Int1, Int2),
   satisfiable(Int2, Int3).
satisfiable(Int, Int).

violated_instance(Body ---> Head, Int) :-
   eval(Body, Int),
   not eval(Head, Int).

satisfy(Exp, Int, [Atom | Int]) :-
   component(Atom, Exp),
   not (Atom = false).

eval((Atom, Rest), Int) :-
   !,
   member(Atom, Int),
   eval(Rest, Int).
eval((Atom ; _Rest), Int) :-
   member(Atom, Int).
eval((_Atom ; Rest), Int) :-
   !,
   eval(Rest, Int).
eval(true, _Int) :-
   !.
eval(Atom, Int) :-
   member(Atom, Int).
```

Programm 4: Das SATCHMO-Grundprogramm mit Akkumulationslisten.

Bei dem ersten Aufruf von `satisfiable` soll das erste Argument die leere Liste sein, das zweite Argument eine uninstanziierte Variable sein, die bei einem Erfolg als Wert das Modell (streng genommen die Menge der im Modell erfüllten positive Literale) liefert. Die Prozedur `component` wird im Programm 4 gleich wie im Programm 1 implementiert. Die Prozedure `satisfiable`, `violated_instance` und `satisfy` von Programm 4 sind sehr ähnlich wie die gleichgenannten Prozeduren von Programm 1. Der Unterschied zwischen Programm 1 und Programm 4 liegt lediglich in Aufrufen von `eval` und in den Listenparametern zur Weitergabe der aufgebauten Interpretation in der Prozedur `satisfiable`, Unter Verwendung von unterschiedlichen Datenstrukturen implementieren sie genau dasselbe Verfahren.

Die Prozedur `eval` ist im Programm 4 unumgänglich, um Klauselrümpfe und -köpfe gegenüber einer als Liste dargestellten Interpretation auszuwerten. Im Programm 1 ist solch ein Metainterpretierer nicht nötig, weil dank der Darstellung des Kellers als Faktenmenge in der Prolog-Datenbank und der Übernahme der Prolog-Syntax für SATCHMO-Klauseln die Prolog-Auswertung unmittelbar übernommen werden kann.

In den Artikeln [MB87, MB88] wurde das Programm 4 nicht erwähnt, weil es unmittelbar aus dem Programm 1 gewinnen werden kann.[1]

5 Datenbankänderungen gegen Akkumulationslisten

Die Frage stellt sich, was die Vor- und Nachteile von Programm 1 und Programm 4 sind. Dabei können drei Aspekte betrachtet werden: der Programmierstil, die Effizienz, und die Parallelisierung.

Was den Programmierstil betrifft, kann die erste Version wegen der Nutzung von `asserta` und `retract` als nichtdeklarativ kritisiert werden. Obwohl dieser Einwand sicherlich für eine uneingeschränkte Nutzung von `asserta` und `retract` zuträffe, ist er, wie bereits argumentiert, in diesem besonderen Fall schwer vetretbar. Die Nutzung eines Hilfsprogramms als Datenstruktur ist eine interessante Programmiertechnik, von uns "Datenbank-Programmierung" genannt, die weiterer Untersuchungen bedarf. In der funktionalen Programmierung ist sie eine wohlverstandene Standardtechnik. Es ist zu bemerken, daß Programmiersprachen der Logikprogrammierung höherer Ordnung wie λProlog [MN86] und Goedel [HL94] Modulkonzepte bieten, die solch einen Programmierstil erleichtern und fordern.

Was die Effizienz betrifft, erweist sich der Stil der Datenbank-Programmierung (d.h. Anwendung von `asserta` und `retract` anstatt von Akkumulationslisten) für nichttriviale Probleme als deutlich vorteilhafter. Experimente mit komplexen Beispiele zeigen, daß Programm 1 je nach Beispiel und Prolog-System 2 bis 4 Mal so schnell ist wie Programm 4 (Die Messungen wurden mit ECLiPSe und SWI-Prolog durchgeführt). Die bessere Leistung von Programm 1 kann zwei Gründe haben. Zum einen ist es anzunehmen, daß Änderungen von Faktmengen mit den meisten Prolog-Systemen schneller als Listenänderungen erfolgen. Zum anderen kann der Mehraufwand mit Programm 4 an dem Metainterpretierer `eval` im Programm 4 liegen. In der Tat liegt die bessere Leistung von Programm 1 hauptsächlich an den schnelleren Änderungen von Faktmengen. Um den Einfluß des Metainterpretierers auf die Leistung experimentell zu messen, wurde eine Variante von Programm 1 betrachtet, die die Klauselrümpfe und -köpfe mit einem der Prozedur `eval` von Programm 4 ähnlichen Metainterpretierer zerlegt und die Atome gegenüber der Prolog-Datenbank statt gegenüber einer Liste auswertet. Die in dieser Weise geänderte Version von Programm 1 ist nur sehr geringfügig weniger effizient als Programm 1 : je nach Beispiel und Prolog-System ist sie nur um den Faktor 1.02 bis 1.4 langsamer.

Wegen der Fallunterscheidung eignen sich SATCHMO-Programme zur (or-)parallelen Auswertung besonders gut. Da die meisten parallelen Sprachen der Logikprogrammierung [Tic91] nur über eine einzige Datenbank verfügen, kann nur ein Programm parallel ausgewertet werden, das den Keller mittels Akkumulationslisten darstellt. Der Beweiser MGTP [FHKF92] ist eine Ausnahme. Um das Satchmo-Verfahren im Datenbankstil in der parallelen Sprache der Logikprogrammierung KL1 zu implementieren, wird eine besondere Datenstruktur, *ramified stack* [FH91] genannt, verwendet.

[1] Lee Naish hat gegen Ende der 80er Jahre das Programm im Internet erwähnt.

6 Implementierungen der Breitensuche

Bekanntlich entspricht die Warteschlange bei der Breitensuche, dem Keller bei der Tiefensuche. Es stellt sich also die Frage, wie die oben erwähnten Suchprogramme auf die Breitensuche angepaßt werden können. Das folgende Programm ist solch eine Anpassung von Programm 2, dem Suchprogramm im Stil der Datenbank-Programmierung.

```
search :-
   read_first(sequence([S|_T])),
   final(S).
search :-
   dequeue(sequence([S|T])),
   findall(sequence([Next,S|T]), transition(S,Next), AllSeq),
   queueall(AllSeq),
   search.
```

```
read_first(Atom) :-
   once(Atom).

queueall([]).
queueall([Atom|Rest]) :-
   assert(Atom),
   queueall(Rest).

dequeue(Atom) :-
   retract(Atom).
```

Programm 5: Implementierung der Breitensuche im Stile der Datenbank-Programmierung.

Dieses Programm speichert mit jedem Zustand alle seiner Vorgängerzustände in einem Listenargument eines `sequence`-Faktums mit. Bei einem Aufruf von `search` soll ein `sequence`-Faktum in der Prolog-Datenbank vorhanden sein, dessen Argument die Liste mit dem Initialzustand als einziges Element ist. Die Schlange selber besteht aus der Relation `sequence`. Ein Faktum `sequence([a, b, c, d])` gibt zum Beispiel einen der zuletzt expandierten Zustände wieder, nämlich `a`, mit seinem Vorgänger `b`, `c` bis zum Initialzustand `d`.

`findall` ist in Prolog eine Built-in-Prozedur: `findall(Term, Goal, List)` bindet die Variable `List` mit der Liste der Terme `Term`, die bei Auswertungen von `Goal` ermittelt werden können.

Bei der herkömmlichen Definition der Breitensuche werden die Vorgängerzustände nicht mitgespeichert. Dies ist aber für eine Anwendung in einem SATCHMO-Programm erforderlich, weil eine Interpretation durch die Menge aller Atome definiert wird, die mit den Zuständen eines Zweiges gespeichert sind. Ein durch Tiefensuche gesteuertes SATCHMO-Programm verlangt keine explizite Speicherung der Vorgängerzustände, weil der Keller die Vorgängerzustände des zuletzt expandierten Zustands enthält. Da aber bei der Breitensuche mehrere Zustände "parallel" expandiert werden, müssen die Vorgängerzustände explizit gespeichert werden. Das folgende Programm ist ein Gegenstück zu Programm 5 im Akkumulationslistenstil.

```
search(L, L) :-
   read_first(sequence([S|_T]), L),
   final(S).
search(L1, L5) :-
   queueall(L2, L3, L4),
   dequeue(sequence([S|T]), L1, L2),
   findall(sequence([Next,S|T]),transition(S,Next),L3),
   queueall(L2, L3, L4),
   search(L4, L5).
```

```
read_first(Atom, [Atom | _Rest]).

queueall(L1, L2, L3) :-
   append(L2, L1, L3).

dequeue(Atom, [Atom | L], L).
```

Programm 6: Implementierung der Breitensuche mit Akkumulationslisten.

Bei einem Aufruf von `search` soll das erste Argument eine Liste mit einem einzigen `sequence`-Term sein, dessen einziges Element der Initialzustand ist. Das zweite Argument soll eine Variable sein, die bei einer erfolgreichen Auswertung mit der Liste der Pfaden vom Initialzustand bis zu den Ziel- beziehungsweise Endzuständen instanziiert wird.

Die Programme 1 und 4 lassen sich leicht in dieser Weise auf die Breitensuche anpassen. Dabei muß beachtet werden, daß eine Transition im SATCHMO-Verfahren nicht nur aus der Auswahl einer falsifizierten Klauselinstanz, sondern auch aus der Auswahl einer Komponente des Instanzkopfes besteht. Das folgende Programm ist eine Anpassung von Programm 4 zur Breitensuche. Ein entsprechendes Programm im Stil der Datenbank-Programmierung ist unmitellbar.

```
satisfiable([], M, M) :-
   !.
satisfiable(I1, M1, M3) :-
   expandall(I1, I2, M1, M2),
   satisfiable(I2, M2, M3).
```

```
expandall([Int | I1], I4, M1, M2) :-
   (Body ---> Head),
   violated_instance(Body ---> Head, Int),
   !,
   findall(ExtInt, satisfy(Head, Int, ExtInt), I2),
   expandall(I1, I3, M1, M2),
   append(I2, I3, I4).
expandall([Int | I1], I2, M1, [Int | M2]) :-
   expandall(I1, I2, M1, M2).
expandall([], [], M, M).
```

Programm 7: Das SATCHMO-Grundprogramm mit Breitensuche.

Der erste Parameter der Prozedur `satisfiable` ist die Liste der bereits expandierten Interpretationen. Die zwei letzten Parameter dieser Prozedur sind Akkumulationslisten zur Berechnung der als Atommengen dargestellten Modelle. Der erste Aufruf von `satisfiable/3` muß also die folgende Form haben: `satisfiable([[]], [], M)`. Die Prozedur `expandall` sucht für jede bereits expandierte Interpretation eine Klauselinstanz, die in dieser Interpretation fasifiziert ist. Ist solch eine Klauselinstanz gefunden, wird die Interpretation in allen möglichen Weisen erweitert, die die Instanz erfüllen. Ist keine solche Instanz gefunden, dann ist die Interpretation ein Modell und wird in die Modelliste eingeführt. Es ist günstiger durch Rekursion anstatt mit `findall` alle Interpretationen zu überprüfen. Im Programm 7 werden die Hilfsprozeduren `violated_instance` und `satisfy` wie im Programm 4 definiert.

7 Klauselmenge als Argument

Die Klauselmenge, deren Unerfüllbarkeit von den Programmen 1, 4 und 7 überprüft wird, wird in der Prolog-Datenbank spezifiziert. Sie kann auch als Listenparameter gegeben werden. Bei der folgenden Variante des Grundprogramms Programm 4 dient die Variable `Theory` diesem Zweck.

```
satisfiable(Theory, Int1, Int3) :-
   member((Body ---> Head), Theory),
   standardize_apart((Body ---> Head), (NewBody ---> NewHead)),
   violated_instance((NewBody ---> NewHead), Int1),
   !,
   satisfy(NewHead, Int1, Int2),
   satisfiable(Theory, Int2, Int3).
satisfiable(_Theory, Int, Int).
```

```
standardize_apart(T, NewT) :-
   copy_term(T, NewT).
```

Programm 8: Das SATCHMO-Grundprogramm mit Klauselmenge als Argument.

Die als "standardization apart" bekannte Umbennung der Klauselvariablen läßt sich in Prolog mit der Systemprozedur `copy_term` besonders leicht implementieren. Bei den Programmen 1, 4 und 7 muß diese Operation nicht explizit implementiert werden werden, weil Prolog sie bei jedem Zugriff in die Datenbank (in den Programmen 1, 4 und 7 bei den Aufrufe von `(Body ---> Head)`) durchführt. Im Programm 8 werden die Hilfsprozeduren `violated_instance` und `satisfy` wie in den Programmen 4 und 7 definiert.

8 Faire Suchstrategien

Bei der Auswahl einer falsifzierten Klauselinstanz muß sichergestellt werden, daß im Lauf des Verfahrens jede solche Instanz nach endlicher Zeit berücksichtigt wird. Man sagt, daß die Suchstrategie "fair" sein soll. Das folgende Beispiel veranschaulicht die Notwendigkeit einer fairen Strategie.

```
true ---> p(a)                    p(X) ---> p(g(X))
p(X) ---> p(f(X))           p(g(f(X))) ---> false
```

Diese Klauselmenge ist nicht erfüllbar: Aus `p(a)` folgt `p(f(a))`; aus `p(f(a))` folgt `p(g(f(a)))`; aus `p(g(f(a)))` folgt `false`. Werden sie auf diese Klauselmenge angewandt, berücksichtigen die Programme 1, 4, 7 und 8 nur Instanzen der zwei ersten Klauseln und generieren die folgende unendliche Atommenge: `p(a)`, `p(f(a))`, `p(f(f(a)))`, `p(f(f(f(a))))`, usw. Sie leiten also kein Atom ab, das einen `g`-Term enthält und stellen deswegen die Unerfüllbarkeit der Klauselmenge nicht fest.

Um bezüglich der Unerfüllbarkeit vollständig zu sein, muß ein Verfahren alle falsifizierten Klauselinstanzen zunächst berechnen und erst dann, wenn möglich, erfüllen. Programm 9 implementiert solch eine sogenannte faire Strategie im Stil der Datenbank-Programmierung unter Anwendung der Prolog-Systemprozedur `findall`. In diesem Programm werden die Hilfsprozeduren `violated_instance` und `satisfy` wie im Programm 1 definiert. (Anstatt von Klauselinstanzen kann auch die Prozedur `all_violated_instances` nur Instanzen von Klauselköpfe sammeln. Obwohl etwas weniger effizient ist das obige Programm anschaulicher.) In [MB87, MB88] wurde anstatt der System-Prozedur `findall` eine eigene und schwächere Implementierung verwendet, die in dem Fall ausreicht.

```
satisfiable_level :-
   all_violated_instances(Set),
   not (Set = []),
   !,
   satisfy_all_instances(Set),
   satisfiable_level.
satisfiable_level.

all_violated_instances(Set) :-
   findall(Clause, violated_instance(Clause), Set).

satisfy_all_instances([]).
satisfy_all_instances([_Body ---> Head | Rest]) :-
   satisfy(Head),
   satisfy_all_instances(Rest).
```

Programm 9: Fair-SATCHMO im Stil der Datenbank-Programmierung.

In ähnlichen Weise wie im Programm 9 kann eine faire Strategie in den Programmen 4, 7 und 8 eingebaut werden. Ist die Klauselmenge, deren Unerfüllbarkeit überprüft werden soll, wie im Programm 8 als Parameter gegeben, kann auf den Aufruf der System-Prozedur `findall` verzichtet werden [Bry95]. Wenn die Klauselmenge als Listenparameter (wie im Programm 8) anstatt als Faktenmenge (wie in den Programmen 1, 4 und 7) definiert wird, kann eine faire Suchstrategie lediglich durch eine einfache Rotation der Klauseln in der Liste implementiert werden: eine Klausel, die eine falsifizierte Instanz geliefert hat, wird ans Ende der Liste versetzt [Bry95].

9 Schlußbemerkungen

Das Modellgenerierungsverfahren, das den SATCHMO-Programmen zugrundeliegt, hat viel Ähnlichkeiten mit der Logikprogrammierung. Zum einen gibt es den negativen Literalen keine explizite Darstellung. Zum anderen behandelt es positive und negative Literale in unterschiedlicher Weise.

Wie viele Autoren bereits in anderen Anwendungsbereichen festgestellt haben, ist es einfach andere Schluß- und Suchverfahren als die von Prolog mittels Metaprogrammen in dieser Sprache zu implementieren. Insbesondere bedarf es wenig, von der in Prolog vorhandenen Tiefensuche abzuweichen. Dank der logischen Variablen ist die Darstellung von logischen Formeln in Prolog einfach, ziemlich natürlich und effizient. Diese zwei Aspekte – Tiefensuche und logische Variablen – sind Merkmale von Prolog, die in kaum anderen Programmiersprachen vorhanden sind.

Die in diesem Artikel dargestellten Programme weisen aber auch auf folgende Nachteile von Prolog hin. Zunächst ist Prolog übermäßig anfällig auf Änderungen von Datenstrukturen. Im Abschnitt 5 wurde erwähnt, daß die Laufzeit des Grundprogramms 2 bis 4 Mal größer sein kann, wenn der Keller als Akkumulationsliste dargestellt ist. Auch die Darstellung der Theorie als Liste anstatt als Faktenmenge verursacht einen deutlichen Effizienzverlust. Weitere Forschungen im Bereich der Kompilierung von Prolog-Programmen werden möglicherweise diesen Nachteil beseitigen oder wenigstens weniger akut machen. Eine weiterer Nachteil von Prolog ist, daß die Sprache kein Modulkonzept bietet. Moduln könnten das Programmieren im Stile von Programm 1 erleichtern und formal begründen. Es ist bemerkenswert, daß zwei neue Sprache der Logikprogrammierung λProlog [MN86] und Goedel [HL94], die die Metaprogrammierung unterstützen, Moduln anbieten. Letztlich zeigt sich die Metaprogrammierung und die Behandlung von logischen Ausdrücke in Prolog in manchen Hinsichten ziemlich umständlich, weil die Sprache freie und gebundene Variablen nicht unterscheidet. Im Gegenteil zu Prolog ermöglicht λProlog diese Unterscheidung.

Die Methoden, die in diesem Artikel für SATCHMO-Beweiser dargestellt werden, sind zweifelsohne für andere Modellgenerationsverfahren anwendbar. In [Fit90] z. B. wird eine Implementierung in Prolog eines Tableauverfahrens mit Breitensuche eingeführt, das durch Änderungen von Suchstrategien oder Datenstrukture effizienter gemacht werden kann. [BP94] beschreibt solch eine Anpassung dieses Programms auf die Tiefensuche. Modellgenerierungsverfahren für nichtklassische Logiken sind in jüngster Zeit von vielen Autoren untersucht beziehungsweise implementiert worden. Es wäre interessant, die Programmiertechniken, die mit SATCHMO eingeführt wurden, zur Implementierung dieser Verfahren anzuwenden. Ebenfalls interessant wäre es, die Implementierung von weiteren Strategien und Heuristiken für Modellgenerierungsverfahren im von SATCHMO eingeführten Rahmen zu untersuchen. Der Bericht [Bry95] enthält einige Beiträge dazu. Wünschenswert wäre es auch, die Übertragung der SATCHMO-Programme auf Sprachen wie λProlog und Goedel zu untersuchen.

Danksagung: Wir danken Norbert Eisinger und Tim Geisler für nützliche Kommentare über eine frühere Fassung des Artikels.

References

[BM87] François Bry and Rainer Manthey. Proving finite satisfiability of deductive databases. In *Proc. 1st Workshop on Computer Science Logic (CSL)*. Springer-Verlag, LNCS 329, 1987.

[BP94] Bernhard Beckert and Joachim Posegga. LeanTap: Lean tableau-based theorem proving. In *Proc. 12th Conf. on Automated Deduction (CADE)*. Springer-Verlag, LNCS 814, 1994. system description.

[Bry95] François Bry. Variationen über ein Thema. Technical report, Institut für Informatik der Universität München, 1995. In Vorbereitung.

[FH91] Hiroshi Fujita and Ryuzo Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In *Proc. 8th Int. Conf. on Logic Programming*, 1991.

[FHKF92] Masayuki Fujita, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Model generation theorem proving on a parallel inference machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems*. ICOT, Tokyo, 1992.

[Fit90] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[FN90] Gerhard Friedrich and Wolfgang Nejdl. MOMO – Model-based diagnosis for everybody. In *Proc. IEEE Conf. on Artificial Intelligence Applications*, 1990.

[HFF91] Ryuzo Hasegawa, Hiroshi Fujita, and Masayuki Fujita. A parallel theorem prover in KL1 and its application to program synthesis. Technical report, ICOT, Tokyo, 1991.

[HKF92] Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. MGTP: A parallel theorem prover based on lazy model generation. In *Proc. 11th Conf. on Automated Deduction (CADE)*. Springer-Verlag, LNCS 607, 1992. system abstract.

[HL94] Pat Hill and John Lloyd. *The Goedel Programming Language*. The MIT Press, 1994.

[LRW93] Donald W. Loveland, David W. Reed, and Debra S. Wilson. SATCHMORE: SATCHMO with relevancy. Technical report, Dept. of Computer Science, Duke University, 1993.

[MB87] Rainer Manthey and François Bry. A hyperresolution-based proof procedure and its implementation in Prolog. In *Proc. 11th German Workshop on Artificial Intelligence (GWAI)*. Springer-Verlag, IFB 152, 1987.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proc. 9th Int. Conf. on Automated Deduction (CADE)*. Springer-Verlag, LNCS 310, 1988.

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In *Proc. 3rd Int. Logic Programming Conf.*, pages 448–462, 1986.

[Ram91] Allan Ramsay. Generating relevant models. *Journal of Automated Reasoning*, 7:359–368, 1991.

[Smu68] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968.

[Tic91] Evan Tick. *Parallel Logic Programming*. The MIT Press, 1991.