JSatchmo
Eine Implementierung des Modellgenerierungsverfahrens PUHR-Tableau in Java

Martin Okršlar

3. August 1998

Inhaltsverzeichnis

1 Einleitung und Aufgabenstellung

JSatchmo ist eine Implementierung der PUHR-Tableau-Methode in Java. Die PUHR-Tableau-Methode ist ein Verfahren zur Generierung von Modellen. Die PUHR-Tableau-Methode wurde von Bry und Yahya [BY96] entwickelt und ist eine Formalisierung des Modellgenerators Satchmo [MB88].

Anwendungsbereiche für Modellgenerierungsverfahren umfassen bestimmte Planungs- und Konfigurationsprobleme; im Bereich des Theormebeweisens und der Programmverifikation werden Modellgeneratoren zur Suche nach Gegenbeispielen benutzt.

Aufgabe dieses Fortgeschrittenenpraktikums war es, die PUHR-Tableau-Methode in Java zu implementieren. Merkmale der erfolgten Implementierung sind:

- Generierung minimaler Modelle
- Complement-Splitting
- Fairness bei der Auswahl verletzter Instanzen
- Auswahl verletzter Instanzen nach Kopflänge
- Generierung von Trace-Dateien für SNARKS [KE97]

2 Die PUHR-Tableau-Methode

Der Name PUHR ist ein Akronym für *positve unit hyper-resolution*. Die PUHR-Tableau-Methode ist ein Tableauverfahren zur Generierung von Modellen. Die folgende Darstellung folgt im wesentlichen [SG97]. Sei $\mathcal{S}$ eine Menge von Klauseln in Implikationsform. Klauseln in Implikationsform werden

$$A_1 \wedge \ldots \wedge A_n \longrightarrow B_1 \vee \ldots \vee B_m$$

geschrieben, wobei die A_i und B_j Atome sind. $A_1 \wedge \ldots \wedge A_n$ nennt man den Rumpf, $B_1 \vee \ldots \vee B_m$ den Kopf der Klausel. Der leere Rumpf wird $\top$, der leere Kopf $\perp$ geschrieben. Die Klauseln in $\mathcal{S}$ müssen bereichsbeschränkt sein, d.h. es darf im Kopf einer Klausel keine Variable vorkommen, die nicht auch im Rumpf dieser Klausel vorkommt.

Ein PUHR-Tableau-Baum beginnt mit $\top$, und wird mit Hilfe folgender Expansionsregeln erweitert (die A_i und B_j sind wieder Rumpf- bzw. Kopfatome einer Klausel):

PUHR-Regel:

$$\frac{\begin{array}{c} A_1 \\ \vdots \\ A_n \end{array}}{B_1 \vee \ldots \vee B_m}$$

Splitting-Regel:

$$\frac{B_1 \vee \ldots \vee B_m}{B_1 \mid \cdots \mid B_m}$$

Zur PUHR-Regel muß noch angemerkt werden, daß die $A_1 \ldots A_n$ Grundinstanzen der $A_1 \wedge \ldots \wedge A_n$ sein müssen. Da die Klausel bereichsbeschränkt ist, sind die $B_1 \vee \ldots \vee B_m$ damit natürlich grund.

Der Zweig eines PUHR-Tableaus wird geschlossen, wenn auf ihm eine Klausel mit leerem Kopf $\perp$ zur Anwendung kam. Ein nicht geschlossener Zweig heißt offen. Ein offener Zweig heißt erfüllt, wenn für jede Klausel in $\mathcal{S}$ die Instanzen einiger Rumpfatome auf dem Zweig nicht vorkommen, oder aber die Instanzen einiger Kopfatome darauf vorkommen.

Die Konjunktion der Atome eines offenen erfüllten Zweiges eines PUHR-Tableaus ist ein Modell für die Klauselmenge $\mathcal{S}$. Ist $\mathcal{S}$ unerfüllbar, so hat das Tableau keinen erfüllten offenen Zweig.

Beispiel 1 Sei $\mathcal{S}$ die Menge folgender Klauseln:

$$\begin{array}{ll} \top \longrightarrow P(a) \vee P(b) & P(a) \longrightarrow P(b) \vee P(d) \\ \top \longrightarrow P(a) \vee P(c) & P(b) \longrightarrow P(a) \vee P(d) \\ P(d) \wedge P(c) \longrightarrow \bot & \end{array}$$

Abbildung 1 zeigt ein PUHR Tableau für die Spezifikation $\mathcal{S}$ aus Beispiel 1. Es werden die Modelle $\{P(a), P(b)\}, \{P(a), P(d)\}$ und $\{P(b), P(c), P(a)\}$ generiert. Man beachte, daß $\{P(a), P(b)\}$ zweimal generiert wird, und $\{P(b), P(c), P(a)\}$ ein sog. nicht-minimales Modell ist, da es das Modell $\{P(a), P(b)\}$ als Teilmenge enthält.

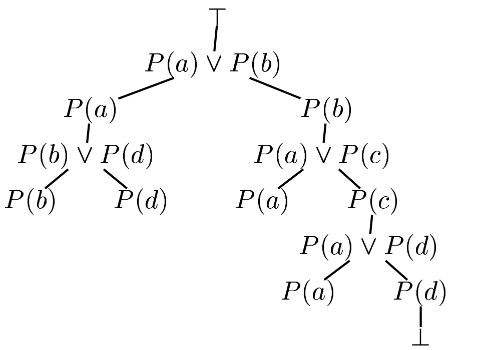

Abbildung 1: PUHR-Tableau für Beispiel 1.

Wie man aus Beispiel 1 sehen kann, generiert die PUHR-Tableau-Methode manche Modelle doppelt. Um das zu vermeiden, wurde die Complement-Splitting-Regel eingeführt (die B_i sind die Kopfatome einer Klausel):

Complement-Splitting-Regel:

$$\frac{B_1 \vee B_2 \vee \ldots \vee B_n}{\begin{array}{c|c|c|c} B_1 & & & \\ \neg B_2 & B_2 & & \\ \vdots & \vdots & & \\ \neg B_n & \neg B_n & \ldots & B_n \end{array}}$$

Nach dieser Regel wird die Negation aller $B_i (1 < i \leq m)$ einer Disjunktion den Zweigen hinzugefügt, die von dem Atom B_j ausgehen, wobei gilt $j < i$. Dadurch erreicht man, daß kein Modell doppelt generiert wird.

Eine weitere Möglichkeit ist die Generierung nur minimaler Modelle: Kein Modell enthält ein bereits gefundenes Modell als Teilmenge. Dabei wird die Negation jedes gefundenen Modells der Menge $\mathcal{S}$ der Klauseln hinzugefügt. Auch bei der Generierung minimaler Modelle wird die Complement-Splitting-Regel benutzt.

3 Syntax der logischen Spezifikationen

Eine Spezifikation besteht aus einer Menge von Regeln. Regeln sind Klauseln in Implikationsform $A_1 \wedge \ldots \wedge A_n \longrightarrow B_1 \vee \ldots \vee B_m$ mit atomaren Formeln A_i und B_j. Im folgenden wird die Syntax von Spezifikationen für JSatchmo beschrieben.

Die Atome der logischen Sprache sind Terme in Prolog-Syntax (d.h. Konstanten, Prädikat- und Funktionssymbole werden klein, Variablen groß geschrieben). Für $\longrightarrow$ schreibt man '`--->`', für $\wedge$ '`,`', und für $\vee$ '`;`'. Nach jeder Klausel steht ein Punkt '`.`'.

Die Spezifikation $\mathcal{S}$ aus Beispiel 1 muß demnach in folgende Form gebracht werden, um von JSatchmo akzeptiert zu werden:

```
---> p(a), p(b).
---> p(a), p(c).
p(a) ---> p(b); p(d).
p(b) ---> p(a); p(d).
p(d), p(c) ---> .
```

Man kann Klauseln einen Namen geben, indem man diesen vor die Klausel schreibt und mit zwei Doppelpunkten '`::`' von ihr trennt. Namen dürfen Groß- und Kleinbuchstaben, und außerdem den Unterstrich '_' enthalten:

```
Guildo_Horn:: piep, piep, piep ---> guildo(hat, Euch, lieb).
```
[1]

Weiterhin wurde der Parser so geschrieben, daß eine Infixnotation der Prädikatsymbole (aber nicht der Funktionssymbole!) möglich ist. Statt

```
haßt(agatha, Mörder) ---> findet(polizei, Mörder).
```

kann man also auch

```
agatha haßt Mörder ---> polizei findet Mörder.
```

schreiben.
Kommentare beginnen mit dem Prozentzeichen '`%`' und reichen bis zum Zeilenende.

Um die Kompatibilität mit Spezifikationen für die Prolog-Implementierungen der PUHR-Tableau-Methode in [BY96, MB88] zu erhalten, ist es auch möglich, `true ---> p(a), p(b).` bzw. `p(d), p(c) ---> false.` zu schreiben.

4 Implementierung

Ein großes Versprechen objektorientierter Programmierung, nämlich die Wiederverwendbarkeit von Code, wird von JSachtmo erfüllt: Die Klassen Atom, AtomList, Term, Cipher, Compound, DataBase, FkSymbol und Var lassen sich unabhängig vom restlichen Code benutzen [2]. Damit werden einige Eigenschaften von Prolog zur Verfügung gestellt, auf die zukünftige Projekte, die diese Eigenschaften benötigen, zurückgreifen können. Im einzelnen wären dies:

- Nachbildung der logischen Struktur von Prolog-Termen.
- Matching von Atomen.
- Bilden aller Instanzen und Substitutionen, wenn ein Atom mehrmals matcht.

[1] Diese Klausel würde natürlich niemals den Parser passieren, da sie nicht bereichsbeschränkt ist!

[2] Falls eine Datenbasis nicht benötigt wird, kann DataBase auch weggleassen werden.

- Operationen auf der Datenbasis wie assert und retract.

Für die Wiederverwendung der Atom-Klasse ist es wichtig, sich mit der Methode internVars() auszukennen. Darauf gehe ich in Abschnitt 4.1.2 näher ein.

4.1 Aufbau und Struktur von Klausel-Objekten

Klauseln sind die oberste logische Einheit der Elemente einer Spezifikation. Deren Repräsentation in JSatchmo soll im folgenden dargestellt werden.

Die Tabelle zeigt den Aufbau der Klausel-Klasse aus spezielleren Klassen:

Klasse	Clause	besteht aus	Body und Head
Klasse	Body/Head	besteht aus	Liste von Atomen
Klasse	Atom	besteht aus	Compound
Klasse	Compound	besteht aus	Funktionssymbol + Liste von Termen
Interface	Term	wird implementiert von	Cipher, Compound und Var
	Konstanten	bestehen aus	Compound mit leerer Term-Liste

Zu weiteren Einzelheiten des Codes gibt es eine ausführliche HTML-Dokumentation im Verzeichnis `jsatchmo/documentation/`.

4.1.1 Aufbau einer Beispielklausel

Abbildung 2 zeigt den Aufbau der Klausel

```
p(A,f(f(A,X),B)) ---> p(B); p(f(X)).
```

aus den jeweiligen Klassen-Objekten.

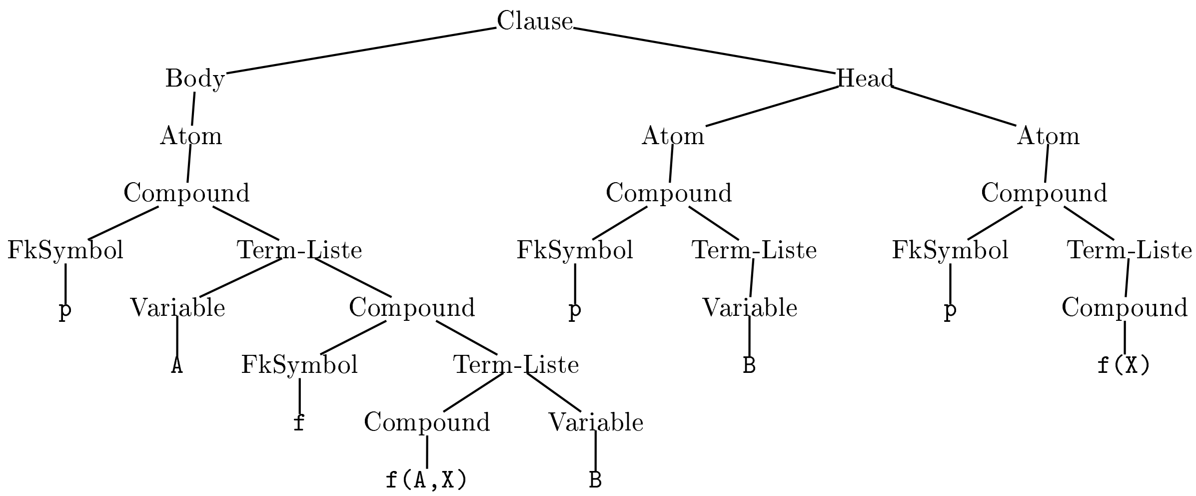

Abbildung 2: Aufbau einer Beispielklausel

4.1.2 `internVars()`

Da JSatchmo objektorientiert ist, war es notwendig, in den Compounds einer Klausel alle gleichnamigen Variablenvorkommnisse durch ein und die selbe Instanz der Klasse Var (im folgenden auch Variablen-Instanz) zu ersetzen. Dadurch wird ermöglicht, daß bei Bindung einer Variablen alle anderen Variablenvorkommnisse der Kausel auch gebunden sind, da ja in allen Compounds die selbe Variablen-Instanz steht. Diese Variablenersetzung wird nach dem Parsen erledigt, indem die Funktion internVars() jeder Klausel aufgerufen wird.

Dabei legt Body eine anfangs leere Liste an, die an alle Atome übergeben wird. Da Atom nur eine Wrapper-Klasse um Compound ist, wird diese Liste sofort an die Compound des Atoms weitergegeben. Die Variablen-Objekte der Compound, die ein gleichnamiges Pendant in der übergebenen Liste haben, werden durch das Variablen-Objekt aus der Liste ersetzt. Hat eine Variable allerdings kein Gegenstück in der Liste, so wird diese in die Liste aufgenommen. Die Variablenliste wird an alle Sub-Compounds weitergereicht. Sind alle Variablen und Compounds des Atoms abgeprüft, wird die Liste an Body zurückgegeben.

Nachdem Body an allen Atomen internVars() aufgerufen hat, wird die Liste an Head weitergereicht. Head übergibt die Liste ebenso wie Body an alle seine Atome. Danach findet die Überprüfung auf Bereichsbeschränkung statt: Enthält die Variablenliste nun mehr Elemente, als zu dem Zeitpunkt, zu dem sie Head übergeben wurde, ist die Bereichsbeschränkungs-Bedingung nicht eingehalten worden.

Eine weitere Möglichkeit, an allen Positionen die selbe Variablen-Instanz zu erhalten, an denen laut Spezifikation die selbe Variable stehen soll, wäre, diese Aufgabe vom Parser erledigen zu lassen. Dort könnte man ebenso eine Variablenliste anlegen, durch deren Inhalt bestimmt ist, ob die gerade geparste Variable durch eine aus der Liste ersetzt, oder aber ein neues Variablen-Objekt erzeugt werden soll.

Damit wäre sicherlich ein gewisser Effizienzgewinn verbunden, da nun keine Variablen-Objekte mehr erzeugt würden, die später gelöscht werden. Allerdings würde dadurch auch der Wert der wiederverwendbaren Klassen dieses Projekts sinken, da die Klassen selbst keine Möglichkeit bieten würden, die obige Variablenersetzung vorzunehmen.

4.1.3 Substitutionen

Was sind Substitutionen? Beim Matching, das in den das Interface Term implementierenden Klassen in der Methode `matches` implementiert ist, werden Instanzen von Rumpf-Atomen gebildet. Die in den Atomen vorkommenden Variablen werden dabei an andere Terme gebunden. Diese Zuordnung von Termen zu Variablen nennt man eine Substitution. Da man die in einer Substitution enthaltene Information (nämlich welcher Term an welche Variable gebunden wird) auch nach dem Matching zur Verfügung haben möchte, muß man eine Implementierung entwickeln, die das leistet.

Eine solche Substitution sollte sowohl die Eigenschaft haben, auf einfache Weise von dem gematchen Atom erhalten zu werden, als auch sich leicht auf die Atome einer Klausel anwenden lassen. Da Atome eine sinnvolle Einheit für diese Aufgabe sind, bildet JSatchmo nur Substitutionen von Atomen (denkbar wäre ja auch Matching und Substitutionsbildug von ganzen Klauseln).

Was aber soll heißen, daß Atome eine *sinnolle Einheit* sind? Betrachten wir dazu folgende Spezifikation:

```
          ---> p(a).
          ---> p(b).
          ---> q(a).
          ---> q(b).
p(X), q(Y) ---> r(X,Y).
```

Es ist klar, daß man vier Instanzen vom Rumpf der Regel bilden kann, nämlich {p(a), q(a)}, {p(a), q(b)}, {p(b), q(a)} und {p(b), q(b)}. Das ist am leichtesten möglich, indem für jedes Atom des Rumpfes von DataBase alle Substitutionen ausgegeben werden. Diese Substitutionen (nämlich jeweils {a,b} für p(X) und q(Y)) werden dann in der Klasse Satchmo auf die jeweiligen Atome angewandt. Dabei ist zu berücksichtigen, daß durch Prolog-artiges Zurücksetzen (backtracking) über die Atome beim Binden der jeweiligen Terme an ihre Variablen alle Instanzen der Regel gebildet werden.

Das Erzeugen von Substitutionen erledigen die Objekte in der Atom-Klasse in drei Schritten mit dem Aufruf dreier Methoden an der Compound des Atoms:

1. `cloneYourValue()` Jede Variable in der Term-Liste muß einen 'Klon' ihres Wertes (d.h. des Objekts, an das sie gebunden ist) erzeugen:

   - Ist die Variable an eine Zahl (Cipher) gebunden, erzeugt die Variable eine Kopie der Cipher.
   - Ist die Variable an eine andere Variable gebunden, erzeugt die Variable eine Kopie des Wertes dieser anderen Variable (für PUHR nicht relevant).
   - Ist die Variable an eine Compound gebunden, ruft sie bei dieser cloneYourValue() auf.

   Da die Klauseln bereichsbeschränkt sind, kann es nicht passieren, daß in einem matchenden Atom eine Variable nicht an einen Wert gebunden ist. Besitzt eine Variable bereits einen 'Klon' ihres Wertes, so tut sie nichts.

2. `getValuesCopy()` In der Compound wird eine leere Liste angelegt, die die Länge der Term-Liste der Compound hat. In der Kopie der Term-Liste werden an die Positionen, an denen in der ursprünglichen Term-Liste Variablen stehen, die durch cloneYourValue() erzeugten Kopien der Variablenwerte geschrieben. An die Positionen, an denen Compounds stehen, werden die Kopien der Sub-Compounds geschrieben.

3. `unbind()` Die Variablen löschen ihre Referenz auf die Klone, und lösen die Bindung an ihre Werte.

Als Substitution wird die Kopie der Term-Liste der obersten Compound zurückgegeben. Zum Programmieren sollte man die Methode `getSubstitutions(Compound)` benutzen, denn sie versucht nicht nur die übergebene Compound zu matchen, sondern gibt, falls das gelingt, gleich noch die Substitution aus.

Der Vorteil bei dieser Art der Substitutionenerzeugung liegt darin, daß die von internVars() geschaffene Variablenverteilung (d.h. alle gleichnamigen Variablen eines Atoms werden von einer Variableninstanz repräsentiert) erhalten bleibt.

4.2 Andere Klassen

Die Klasse Satchmo ist die zentrale Klasse dieses Projekts. In Satchmo ist der Inferenzmotor untergebracht. Er besteht aus den beiden Methoden satisfiable und satisfy_all.

4.2.1 Beschreibung des Inferenzmotors

Nachdem JSatchmo die Spezifikation mit den Regeln und Fakten übergeben wurde, beginnt er mit dem Modellgenerierungsprozess:

1. Im ersten Schritt schreibt JSatchmo alle Fakten in seine Datenbasis.

2. Dann sucht er eine Regel, deren linke Seite (Body) durch die Fakten in der Datenbasis erfüllt ist, und deren rechte Seite (Head) aber kein Atom enthält, das bereits in der Datenbasis enthalten ist. Findet er eine solche Regel (man nennt sie auch verletzte Instanz einer Regel), dann wertet JSatchmo die rechte Seite aus. Hier gibt es drei Möglichkeiten:

3. (a) die rechte Seite enthält genau ein Atom. Das Atom wird in die Datenbasis geschrieben.

   (b) die rechte Seite enthält mehr als ein Atom. Dann wird das erste Atom in die Datenbasis geschrieben; während die anderen Atome der Disjunktion warten, bis alle Ableitungen mit dem ersten Atom erledigt sind. Dann kommen sukzessive alle 'wartenden' Atome an die Reihe.

   (c) die rechte Seite enthält gar kein Atom; dann ist die Regel eine Negation, und das Atom, das zuletzt in die Datenbasis geschrieben wurde, wird gelöscht.

4. Nach jeder Anwendung der Schritte 3a-3c wird solange bei Schritt 2 weitergemacht, bis JSatchmo keine Regel mehr findet. Hier gibt es wiederum zwei Möglichkeiten:

5. (a) Der letzte Schritt war 3c. Dann gilt die Ableitung als falsch, da sie zu einem Widerspruch geführt hat (man spricht auch von einem geschlossenen Tableau).

   (b) Der letzte Schritt war 3a oder 3b. Dann heißt der Ast offen, und die Ableitung war erfolgreich. Es wurde ein Modell gefunden.

6. Gibt es noch Disjunktionen mit 'wartenden' Atomen, kehrt JSatchmo an diese Stelle zurück und löscht dabei alle von dort bis zum Ende der Ableitung eingetragenen Atome aus der Datenbasis. Nun fährt JSatchmo mit dem ersten wartenden Atom bei Schritt 2 fort.

4.3 Eine kurze Geschichte über das Auffinden eines Fehlers

In grauer Vorzeit, als JSatchmo noch in der Testphase lag ... konnte JSatchmo bei einer einzigen bestimmten Spezifikation kein Modell finden, obwohl es sicher war, daß es eines gab.

Die Schwierigkeit bestand nun darin, festzustellen, welchen Fehler JSatchmo denn mache. Diese bestimmte Testspezifikation (es war 'implikanien', eines der mit dem JSatchmo-Paket mitgelieferten Beispiele) war sehr groß. Um festzustellen, daß er kein Modell finden kann, brauchte JSatchmo über zwei Stunden. Bei dieser Größe war es natürlich unmöglich, den Fehler zu suchen, indem man jeden Schritt einzeln nachvollzieht. Auch eine Visualisierung des Suchbaumes durch SNARKS war nicht weiter hilfreich.

Beim Auffinden des Fehlers wurde folgender Weg eingeschlagen: Da die Lösung der Testspezifikation bekannt war, war es leicht, die Klauseln in der Spezifikation so umzuformen, daß der Zweig am weitesten links außen des Tableaus das gesuchte Modell enthalten würde. Nun war SNARKS natürlich von großer Nützlichkeit, da man alle weiteren Schritte nach dem Ableiten des Modells leicht verfolgen konnte.

Der Fehler selbst war recht interessant und soll deshlab auch ausführlicher erläutert werden. Bei folgender (komprimierter) Spezifikation

$$
\begin{array}{l}
\longrightarrow \texttt{p(a).} \\
\longrightarrow \texttt{p(b).} \\
\longrightarrow \texttt{r(c).} \\
\texttt{r(X), p(X)} \longrightarrow \texttt{q(X).}
\end{array}
$$

gab JSatchmo als Ergebnis als Modell p(a), p(b), r(c) und q(b) aus. Das Erstaunliche dabei ist, daß q(b) aus der Spezifikation gar nicht ableitbar ist Wie kam es dazu?

Wie bereits erwähnt, steht ja an jeder Stelle der Klausel die selbe Variablen-Instanz. Beim Matching wurde im ersten Schritt das X in r(X) an c gebunden. Dann wurde versucht, p(c) zu beweisen, was nicht gelang, da p(c) nicht zu den Axiomen gehört (d.h. nicht in der Datenbasis enthalten war). Da der Versuch des Beweisens scheiterte, wurde ein neuer Versuch gestartet: p(c) wurde von der Konstanten c gelöst, und es wurde versucht p(X) zu beweisen. Aber genau das war der Fehler! Das X in p(X) wurde von r(X) an c gebunden, und deshalb hätte auch nur r(c) die Variable X von c lösen dürfen. Der oben erwähnte zweite Beweisversuch gelang, p(X) wurde an b gebunden, und damit auch r(X) und q(X).

Der Fehler wurde behoben, indem es nun nur noch derjenigen Compound gestattet ist, die Variablenbindung zu lösen, in der die Variable auch gebunden wurde. Für das obige Beispiel bedeutet das, daß nur r(X) berechtigt ist, die Variable X von einer Bindung zu lösen.

4.4 Weitere Merkmale der Implementierung

Fairness: JSatchmo ist fair, bezüglich der Auswahl und der Anwendung der verletzten Instanzen, d.h. daß an jedem offenen Zweig jede mögliche Anwendung einer Expansionsregel mindestens einmal durchgeführt wird.

Auswahl nach Kopflänge: Um die Effizienz zu steigern, wendet JSatchmo von allen verletzten Instanzen immer erst die mit dem kürzesten Kopf (den wenigsten Disjunktionen) an.[3] Dadurch wird erreicht, daß nach der Verzweigung nicht auf jeden Ast ein Atom geschrieben werden muß, das man vor der Verzweigung in das Tableau hätte schreiben können (und zwar nur einmal).

5 Benutzung und Installation

5.1 Installation

Entpacken Sie das Paket mit

```
tar -xf jsatchmo.tar
```

Sie erhalten ein Verzeichnis `jsatchmo`. Darin befinden sich die Klassen, mit denen man das Programm ausführen kann, das Verzeichnis `documentation` mit der HTML-Dokumentation zum Code, und das Verzeichnis `src` mit den Quelltexten.

Außerdem befinden sich noch drei Beispielsspezifikationen (nämlich `implikanien`, `mm` und `worf.prolog`) in diesem Verzeichnis.

5.2 Benutzung

JSatchmo ist ein Kommandozeilentool. Da es komplett in Java geschrieben ist, läuft es auf jedem Rechner, der eine Java Virtual Machine der Version 1.1.3 (oder höher) hat.[4] Der Aufruf ist wie folgt:

```
java Satchmo [Optionen] Datei [Dateien]
```

Die Optionen werden später genauer erläutert. Um einen Überblick über die Optionen zu gewinnen, tippt man nur `java Satchmo`, und es wird ein kurzer Hilfetext ausgegeben.
Hier eine Beispielsitzung:

```
java@boxhamster:/home/java/uni/satchmo > cat worf.prolog

%Beispiel in Prolog-Notation

learning(Person) ---> successful(Person).
student(Person), at(Person, uni) ---> playing(Person); learning(Person).
lecturer(Person), at(Person, uni) ---> teaching(Person); researching(Person).

computerscientist(Person), playing(Person) ---> false.
```

[3] Da JSatchmo nicht inkrementell arbeitet, zerstört die Auswahl nach Kopflänge nicht die Fairness von Satchmo.

[4] JSatchmo dürfte auch mit künftigen Versionen des JDK problemlos funktionieren (und sich kompilieren lassen), da lediglich Klassen der Pakete `java.lang` und `java.util` benutzt wurden. Anders als bei Pakten wie z.B. `java.awt` sind an diesen Kernbestandteilen der Sprache keine wesentlichen Änderungen zu erwarten.

```
true ---> student(worf).
true ---> at(worf, uni).

%Mit dieser Zeile gibt es nur noch ein Modell:
%true ---> computerscientist(worf).

%Mit der vorherigen und dieser Zeile gibt es gar kein Modell mehr:
%successful(worf) ---> false.

java@boxhamster:/home/java/uni/satchmo > java Satchmo -cs worf.prolog


__________Saturate: worf.prolog__________

model 1:
        at(worf, uni)
        playing(worf)
        student(worf)

model 2:
        at(worf, uni)
        student(worf)
        successful(worf)
        learning(worf)
```

5.2.1 Kommandozeilenoptionen

`-mm` Generiert nur minimale Modelle: kein generiertes Modell hat ein anderes generiertes Modell als Teilmenge.

`-cs` Complement-Splitting: kein Modell wird doppelt generiert.

`-trace` Erzeugt Tracefiles für SNARKS. JSatchmo speichert die Datei dann unter dem Dateinamen der 'getracten' Datei mit dem Anhängsel `.trace`.

Zu SNARKS siehe auch [KE97]. Zur Benutzung der Tracedatei in SNARKS gibt es ein eigenes Kapitel im Handbuch zu SNARKS (`http://pms.informatik.uni-muenchen.de/~snarks/doc/handbook/how-to-connect-an-external-deduction-system.html`).

`-maxmodels <n>` JSatchmo bricht nach dem n-ten Modell den Modellgenerierungsvorgang ab.

5.3 Kompilierung

Zur Kompilierung benötigt man auf jeden Fall den JavaCC Parsergenerator Version 0.7 (oder höher) und das Java Development Kit ab Version 1.1.

Wenn Sie 'make' installiert haben, ist alles ganz einfach: Begeben Sie sich in das Verzeichnis `jsatchmo/src/`, in dem sich auch der JSatchmo-Quelltext befindet. Tippen Sie:

```
make
```

Haben Sie kein 'make' zur Verfügung, tippen sie

```
javacc SatchParser.jj
javac Satchmo.java
```

Mit dem ersten Schritt erzeugt JavaCC den Quelltext für den Parser. Da `javac` die Abhängigkeiten zwischen den Klassen selbst herausfindet, wird so das gesamte Projekt kompiliert.

Der JavaCC Parsergenerator ist unter `http://www.suntest.com/JavaCC/` kostenlos zu beziehen.

6 Ausblick

Die PUHR-Tableau-Methode wurde in JSatchmo vollständig implementiert. Dennoch lassen sich einige Erweiterungen des Programms denken.

Man könnte zum Beispiel eingebaute Prädikate implementieren, wie $\neq$ (syntaktische Ungleichheit) oder Arithmetik. Die gegenwärtige Auswahl der verletzten Instanzen nach Kopflänge könnte man (wahlweise) durch andere Auswahlkriterien ersetzen.

Für die Zukunft ist geplant, die Prolog-Komponente durch JSatchmo zu ersetzen.

Literatur

[BY96] François Bry and Adnan Yahaya. Minimal model generation with positive unit hyper-resolution tableaux. *Tableaux*, 1996.

[KE97] M. Kettner and N. Eisinger. The tableau browser snarks (system description). *Proc. 14. Conf. on Automated Deduction (CADE)*, 1997.

[MB88] R. Manthey and François Bry. Satchmo: a theorem prover implemented in prolog. *Proc. 9. Conf. on Automated Deduction (CADE)*, 1988.

[SG97] Heribert Schütz and Tim Geisler. Efficient model generation through compilation. *Information and Computation*, 1997.