SIC: Ein interaktives Deduktionssystem zur Überprüfung der Erfüllbarkeit von Integritätsbedingungen in Datenbanken*

François Bry, Norbert Eisinger, Heribert Schütz, Sunna Torge

Universität München, Institut für Informatik, Oettingenstr. 67, D-80538 München
{ bry | eisinger | hschuetz | torge }@informatik.uni-muenchen.de
http://www.pms.informatik.uni-muenchen.de

Zusammenfassung SIC (Akronym für *Satisfiable Integrity Constraints*) ist ein Prototyp eines interaktives Deduktionssystems, das den Entwurf endlich erfüllbarer Integritätsbedingungen für Datenbanken unterstützt. SIC behandelt also das Erfüllbarkeitsproblem für Integritätsbedingungen während der Schemaentwurfsphase einer Datenbank. SIC besteht aus zwei Teilsystemen, einer Inferenzkomponente und einer interaktiven graphischen Benutzerschnittstelle. Das Papier gibt einen Überblick über die Funktionalität beider Komponenten sowie über den theoretischen Hintergrund und Implementierungsaspekte der Inferenzkomponente.

1 Einführung und Motivation

Eine Datenbank besteht aus einem *Schema* und den eigentlichen Daten, dem *Zustand*. Das Schema besteht aus einer Definition der Struktur der Daten und den *Integritätsbedingungen*, also Bedingungen, die von jedem zulässigen Zustand der Datenbank erfüllt werden müssen.[1] Typischerweise sind der Schemaentwurf und das Füllen der Datenbank mit Daten zwei verschiedene Phasen in der Entwicklung einer Datenbank, die oft von verschiedenen Personen bearbeitet werden [AHV95,Dat95].

Integritätsbedingungen bringen zweierlei Fragestellungen mit sich. Die eine, nämlich die nach der *Erfülltheit* der Integritätsbedingungen, wurde ausführlich untersucht: Wie kann man überprüfen, ob alle Integritätsbedingungen in einem gegebenen Datenbank-Zustand erfüllt sind, bzw. ob sie in dem neuen Zustand, der sich aus einer Änderungstransaktion ergibt, erfüllt bleiben?

* Eine Version dieses Papiers ist erschienen in *Proc. 6th Int. Workshop on Deductive Databases and Logic Programming (DDLP'98)* bei JICSLP 1998, Manchester, UK, Juni 1998 (in englischer Sprache).

[1] Zusätzlich können sogenannte *Regeln* Views definieren. In diesem Papier werden Regeln wie Integritätsbedingungen behandelt, und es wird davon ausgegangen, dass Views auch extensionale Daten enthalten dürfen. (Die übliche strenge Trennung zwischen extensionaler und intensionaler Datenbank wird also nicht vorausgesetzt.)

Auf die andere Fragestellung, nämlich die nach der *Erfüllbarkeit* der Integritätsbedingungen, wurde in [BM86,BM87,BDM88,Man90,NML93,DTU96,MP96] hingewiesen: Wie kann sichergestellt werden, dass es für eine Menge von Integritätsbedingungen überhaupt Datenbank-Zustände – und insbesondere interessante Datenbank-Zustände – gibt, die sie erfüllen?

Die Erfüllbarkeit der Integritätsbedingungen ist eine wesentliche Voraussetzung für das Füllen einer Datenbank mit Daten. Wenn der Schema-Designer unerfüllbare Integritätsbedingungen spezifiziert hat, dann wird jede mögliche Änderungstransaktion zurückgewiesen und es können keine Daten in die Datenbank eingetragen werden. Der Benutzer (oder der Schema-Designer selbst in einer Testphase) erhält keinen Hinweis darauf, dass der tiefere Grund dafür in einem Fehler im Schema liegt und nicht in seiner Unfähigkeit, geeignete Transaktionen zusammenzustellen. Bereits der Schema-Designer sollte also unterstützt werden, Fehler dieser Art zu entdecken und zu beseitigen.

Andere Fehler im Schema führen zu Integritätsbedingungen, die zwar erfüllt werden können, aber nur durch (hypothetische) unendlich große Datenbank-Zustände. Offensichtlich kann kein tatsächlicher Datenbank-Zustand solche Integritätsbedingungen erfüllen, und wieder sollte die Entdeckung und Beseitigung solcher Fehler unterstützt werden.

Die Erfüllbarkeitsfrage für Integritätsbedingungen ist keineswegs eine unrealistische Fragestellung. Es gibt erstaunlich einfache und realistische Beispiele, in denen einer der erwähnten Fehler im Schemaentwurf auftritt (vgl. Abschnitt 2). Wenn mehrere Personen an verschiedenen Teilen der Integritätsbedingungen arbeiten, sind solche Entwurfsfehler nahezu unvermeidbar.

Was dann gebraucht wird, ist ein Hilfsmittel, um während der Schemaentwurfsphase zu überprüfen, ob die Integritätsbedingungen *endlich erfüllbar* oder *unerfüllbar* sind, oder ein sogenanntes *Unendlichkeitsaxiom* darstellen, d. h. erfüllbar sind, aber nur durch unendliche Datenbank-Zustände.

Nun sind oder entsprechen Integritätsbedingungen Formeln der Prädikatenlogik erster Stufe, und leider sind die relevanten Fragen für solche Formeln nicht entscheidbar. Genauer gesagt ist die Frage, ob eine Menge von logischen Formeln erster Stufe unerfüllbar ist, semi-entscheidbar: Es gibt Prozeduren, die eine Formelmenge als Argument haben und nach endlicher Zeit „unerfüllbar" melden, wenn die Formelmenge tatsächlich unerfüllbar ist. Für erfüllbare Mengen kann die Terminierung jedoch nicht garantiert werden [Smu68]. Ebenso ist die Frage, ob eine Menge von logischen Formeln erster Stufe endlich erfüllbar ist, semi-entscheidbar: Es gibt Prozeduren, die eine Formelmenge als Argument haben und nach endlicher Zeit „endlich erfüllbar" melden, wenn die Formelmenge diese Eigenschaft besitzt. Besitzt sie sie aber nicht, kann die Terminierung solcher Prozeduren nicht garantiert werden [Tra50].

Natürlich ist es wünschenswert, die Fähigkeit zum Erkennen sowohl der Unerfüllbarkeit als auch der Erfüllbarkeit im Endlichen in einem einzigen System zu vereinigen, das dann dem Schema-Designer in der Entwurfsphase zur Verfügung steht. Falls die Menge von Integritätsbedingungen endlich erfüllbar ist, sollte also ein solches System dies in endlicher Zeit melden. Außerdem sollte es die Analyse, ob unter den Datenbank-Zuständen, die die Integritätsbedingungen erfüllen, auch „interessante“ Zustände sind, unterstützen. Falls die Menge von Integritätsbedingungen unerfüllbar ist, sollte das System dies auch nach endlicher Zeit melden, und es sollte auch den Schema-Designer bei der Suche nach den Gründen für die Unerfüllbarkeit unterstützen. Falls die Menge von Integritätsbedingungen erfüllbar, aber nicht endlich erfüllbar ist, also ein Unendlichkeitsaxiom darstellt, dann können wir nicht erwarten, dass das System immer eine Antwort liefert. Trotzdem sollten seine Berechnungsschritte dem Designer Hinweise liefern, mit deren Hilfe er erkennen oder vermuten kann, dass ein Unendlichkeitsaxiom vorliegt.

SIC ist ein prototypisches System dieser Art für relationale (oder deduktive) Datenbanken. Es vereinigt die Inferenzkomponente FINFIMO [BT98] und die interaktive Visualisierungskomponente SNARKS [KE97]. Die Inferenzkomponente implementiert eine Methode, die sowohl für die endliche Erfüllbarkeit als auch für die Unerfüllbarkeit vollständig ist. Diese Methode ist eine Verfeinerung des Modellgenerators SATCHMO [MB88,BY96], der ursprünglich für diese Anwendung [BM86,BM87,BDM88] entwickelt wurde. Die Visualisierungskomponente wurde nicht nur für diese datenbankorientierte Anwendung, sondern allgemein für SATCHMO und Varianten davon entwickelt. Ihre interaktiven Fähigkeiten bieten Unterstützung für die Untersuchung „interessanter“ Datenbank-Zustände im Fall endlicher Erfüllbarkeit und für das Erkennen der Ursachen, wenn die Menge von Integritätsbedingungen nicht endlich erfüllbar ist.

2 Beispiele

Die Beispiele dieses Abschnitts haben zwei Zwecke: Zum einen zeigen sie, dass Schemaentwurfsfehler der oben erwähnten Art tatsächlich leicht übersehen werden können. Zum anderen geben sie eine intuitive Vorstellung der Inferenztechniken, durch die SIC die Entdeckung von Fehlern im Schemaentwurf unterstützt.

Beispiel 1. Die folgende Menge von Integritätsbedingungen ist eine Variante eines Beispiels aus [Man90]. Um Klammern zu sparen, nehmen wir an, dass Quantoren sich auf möglichst große Teilformeln beziehen. $\perp$ ist in allen Interpretationen falsch. Man beachte, dass Formel (1) nichts anderes ausdrückt als „$\neg\exists X \;\; \mathit{department}(X) \wedge \mathit{employee}(X)$“.

(1) Abteilungen und Mitarbeiter sind verschieden voneinander:
$\forall X \;\; \mathit{department}(X) \wedge \mathit{employee}(X) \;\Rightarrow\; \perp$
(2) Der Leiter X einer Abteilung Y gehört zu dieser Abteilung:
$\forall X\, \forall Y \;\; \mathit{leads}(X,Y) \;\Rightarrow\; \mathit{member}(X,Y)$
(3) Eine Person X, die zu einer Abteilung Z gehört, arbeitet für den Leiter Y dieser Abteilung:
$\forall X\, \forall Y\, \forall Z \;\; \mathit{member}(X,Z) \wedge \mathit{leads}(Y,Z) \;\Rightarrow\; \mathit{works\_for}(X,Y)$
(4) Jeder Mitarbeiter X gehört zu einer Abteilung Y:
$\forall X \;\; \mathit{employee}(X) \;\Rightarrow\; \exists Y \; \mathit{department}(Y) \wedge \mathit{member}(X,Y)$

(5) Jede Abteilung X hat einen Mitarbeiter Y als Leiter:
$\forall X \;\; \mathit{department}(X) \;\Rightarrow\; \exists Y \; \mathit{employee}(Y) \wedge \mathit{leads}(Y,X)$

(6) Niemand arbeitet für sich selbst:
$\forall X \;\; \mathit{works\_for}(X,X) \;\Rightarrow\; \bot$

Diese Menge von Integritätsbedingungen kann erfüllt werden, und zwar vom leeren Datenbank-Zustand. SIC zeigt dies an. Somit scheinen zunächst die Integritätsbedingungen gut spezifiziert worden zu sein. Der leere Datenbank-Zustand ist aber ein nicht sehr interessanter Sonderfall.

Betrachten wir also, was passiert, wenn es in der Datenbank einen Mitarbeiter *john* gibt. Mit SIC können wir dies tun, indem wir interaktiv die Formel $\mathit{employee}(\mathit{john})$ einfügen. Nach dieser Änderung schließt SIC wie folgt:

Wegen $\mathit{employee}(\mathit{john})$ verlangt Bedingung (4) die Existenz einer Abteilung Y, zu der *john* gehört. Wenn Y gleich *john* ist, erhalten wir $\mathit{department}(\mathit{john})$, was mit (1) zu einem Widerspruch führt.

Es verbleibt der Fall, dass Y ein anderes Objekt ist, etwa d_1, so dass $\mathit{employee}(\mathit{john})$ und $\mathit{department}(d_1)$ und $\mathit{member}(\mathit{john}, d_1)$ gelten. Nun verlangt (5) die Existenz eines Leiters Y von d_1. Im ersten Fall ist Y gleich *john* und die einzige neue Konsequenz ist $\mathit{leads}(\mathit{john}, d_1)$, woraus zusammen mit $\mathit{member}(\mathit{john}, d_1)$ und (3) $\mathit{works\_for}(\mathit{john}, \mathit{john})$ folgt. Mit (6) führt dies zu einem Widerspruch. Im zweiten Fall ist Y gleich d_1, und wir erhalten wieder einen Widerspruch zu (1).

Es bleibt der Fall, dass Y ein anderes Objekt ist, etwa e_2, so dass $\mathit{employee}(\mathit{john})$ und $\mathit{department}(d_1)$ und $\mathit{member}(\mathit{john}, d_1)$ und $\mathit{employee}(e_2)$ und $\mathit{leads}(e_2, d_1)$ gelten. Aus letzterem erhalten wir $\mathit{member}(e_2, d_1)$ mit (2), dann $\mathit{works\_for}(e_2, e_2)$ mit (3), und schließlich einen Widerspruch mit (6).

Wenn es also einen Mitarbeiter gibt, ist es nicht möglich, die Integritätsbedingungen zu erfüllen. SIC erzeugt eine Ableitung, die den obigen Schlüssen entspricht, und stellt sie textuell-graphisch als Baum dar. Jeden Zweig des Baums kann man als Versuch betrachten, die Einfügung von $\mathit{employee}(\mathit{john})$ mit weiteren Einfügungen zu einer Transaktion zu ergänzen um einen erlaubten (also die Integritätsbedingungen erfüllenden) Datenbank-Zustand zu erhalten. Da jeder Zweig fehlschlägt und der Baum systematisch alle möglichen Versuche abdeckt, weiß der Designer der Integritätsbedingungen sicher, dass jede tatsächliche Transaktion zurückgewiesen würde.

Die Visualisierungskomponente von SIC ermöglicht es, den Baum in verschiedener Weise zu beschriften und die Abhängigkeiten zwischen einer Formel und ihren Prämissen und Konsequenzen hervorzuheben. Diese und andere Eigenschaften von SIC geben dem Designer eine gute Möglichkeit, den Entwurfsfehler zu erkennen: Im Falle des obigen Beispiels erscheint die Bedingung (3) als zu streng, falls X und Y übereinstimmen. Der Designer hat diesen Grenzfall übersehen.

Wenn (3) verbessert wird zu

$$\forall X \; \forall Y \; \forall Z \; \mathit{member}(X,Z) \wedge \mathit{leads}(Y,Z) \;\Rightarrow\; X{=}Y \vee \mathit{works\_for}(X,Y),$$

dann können die Bedingungen auch dann erfüllt werden, wenn es einen Mitarbeiter gibt, und SIC zeigt dies an. Tatsächlich erzeugt es Beispiele für Datenbank-Zustände, die die Integritätsbedingungen erfüllen. Somit hilft SIC dem Schema-Designer dabei, Testdaten zu erfinden.

Beispiel 2. Wir erläutern das Unendlichkeitsproblem mit folgenden Integritätsbedingungen:

(1) Jeder Mitarbeiter arbeitet für jemanden:
$\forall X \;\; employee(X) \;\Rightarrow\; \exists Y \; employee(Y) \wedge works\_for(X,Y)$
(2) Kein Mitarbeiter arbeitet für sich selbst:
$\forall X \;\; works\_for(X,X) \;\Rightarrow\; \bot$
(3) Die Relation „arbeitet für“ ist transitiv:
$\forall X \; \forall Y \; \forall Z \;\; works\_for(X,Y) \wedge works\_for(Y,Z) \;\Rightarrow\; works\_for(X,Z)$

Diese Menge von Integritätsbedingungen wird wieder vom leeren Datenbank-Zustand erfüllt. Wieder fügen wir interaktiv $employee(john)$ ein, um interessantere Fälle zu untersuchen. Es wird wie folgt geschlossen:

Wegen $employee(john)$ verlangt (1) die Existenz eines Mitarbeiters Y, für den *john* arbeitet. Falls Y gleich *john* ist, erhalten wir $works\_for(john, john)$ und mit (2) einen Widerspruch. Somit muss Y ein anderes Objekt e_1 sein, derart dass $employee(john)$ und $employee(e_1)$ und $works\_for(john, e_1)$ gelten.

Als nächstes verlangt (1) die Existenz eines Y, für den e_1 arbeitet. Falls Y gleich *john* ist, erhalten wir mit (3) $works\_for(e_1, john)$ und dann $works\_for(john, john)$ und mit (2) einen Widerspruch. Falls Y gleich e_1 ist, erhalten wir $works\_for(e_1, e_1)$ und mit (2) einen Widerspruch. Somit ist Y noch ein anderes Objekt e_2.

Wieder verlangt (1) die Existenz eines Y, für den e_2 arbeitet. Wieder führen die Fälle, in denen Y eines der vorherigen Objekte ist, zu Widersprüchen, und wir müssen ein Objekt e_3 einführen, und so weiter.

Die Integritätsbedingungen können erfüllt werden, indem unendlich viele Mitarbeiter eingeführt werden, aber nicht mit einer endlichen Anzahl. Dieses Beispiel gehört zur Klasse der Unendlichkeitsaxiome, die von keiner Prozedur alle als solche erkannt werden können. Ein Schema-Designer, der SIC benutzt, würde die Ableitung jedoch schrittweise entwickeln und analysieren. Er würde bald folgende Regelmäßigkeit erkennen können: Neben vielen irrelevanten widersprüchlichen Zweigen steht ein Zweig mit einem sich wiederholenden Muster. So kann der Entwurfsfehler gefunden werden: Bedingung (1) sollte nicht für Personen an der Spitze der Firmenhierarchie gelten. Die Nichtbeachtung dieses Grenzfalls machte diese Integritätsbedingungen zu einem Unendlichkeitsaxiom. Eine mögliche Korrektur ist

$$\forall X \; employee(X) \;\Rightarrow\; boss(X) \;\vee\; (\exists Y \; employee(Y) \wedge works\_for(X,Y)).$$

Hiermit kann SIC die Erfüllbarkeit im Endlichen zeigen.

Obwohl eine allgemeine Charakterisierung aller Arten von Unendlichkeitsaxiomen nicht möglich ist, führen sie oft zu wiederholten Mustern oder – aus der

Sicht der Anwendung – unerwarteten Konsequenzen. Durch diese Phänomene bekommt der Designer oft den Verdacht, dass ein Problem vorliegt, und SIC bietet die Unterstützung um diese Phänomene deutlich sichtbar zu machen. Man beachte die Analogie zu einem Debugger für eine imperative Programmiersprache, der trotz der Unentscheidbarkeit des Halteproblems üblicherweise dem Programmierer genügend Informationen bietet, um nichtterminierende Schleifen und ihre Ursachen zu erkennen.

In beiden Beispielen sind die Entwurfsfehler nicht unmittelbar zu erkennen, insbesondere nicht für Personen, die keine eingehende Erfahrung mit logischen oder deklarativen Formalismen haben, oder wenn die fehlerhaften Integritätsbedingungen nur ein Teil einer größeren Menge sind, die möglicherweise von mehr als einer Person entwickelt wurde. Nach unserer Meinung ist dies reichliche Motivation für ein System wie SIC.

3 Eine sowohl für die Unerfüllbarkeit als auch die Erfüllbarkeit im Endlichen vollständige Methode

Die Inferenzkomponente von SIC basiert auf der "Extended Positive Tableau"-Methode (kurz EP-Tableau-Methode) [BT98]. Diese Deduktionsmethode erweitert die PUHR-Tableau-Methode [BY96], die dem Modellgenerierer SATCHMO [MB88] zugrunde liegt. Eine wesentliche Eigenschaft der PUHR-Tableau-Methode ist, dass sie nur positive Fakten generiert und negative Information implizit lässt, was für Datenbankanwendungen ein sehr natürlicher Ansatz ist.

Wie andere herkömmliche Tableau-Methoden ist die PUHR-Tableau-Methode aber nicht vollständig für die Erfüllbarkeit im Endlichen, d. h. selbst wenn eine Menge von Integritätsbedingungen ein endliches Modell hat, erzeugt die PUHR-Tableau-Methode nicht notwendigerweise ein solches. Der Grund dafür ist, dass die Expansion von PUHR-Tableaux Terme mit beliebig tief geschachtelten Funktionssymbolen erzeugen kann, was zu unendlichen Zweigen führt. Die PUHR-Tableau-Methode kann die Verwendung von Funktionssymbolen nicht vermeiden, weil sie anstelle der in Tableau-Methoden sonst üblichen δ-Regel [Smu68] die Skolemisierung von existentiell quantifizierten Variablen (wie sie etwa bei referentiellen Integritätsbedingungen auftreten) verlangt. In unserer Anwendung für relationale Datenbanken gibt es jedoch keinen Bedarf für die Behandlung von anderen Funktionssymbolen als solchen, die durch die Skolemisierung eingeführt würden.

Im Gegensatz zur PUHR-Tableaux-Methode verlangt die EP-Tableaux-Methode keine Skolemisierung. Sie behandelt vielmehr Existenz-Quantoren explizit mit einer speziellen Expansionsregel [BM87,Hin88,Lor94]. Diese Expansionsregel garantiert Vollständigkeit bzgl. der Erfüllbarkeit im Endlichen. Die zugrunde liegende Idee ist wie folgt: Die Methode versucht, eine existentielle Formel zuerst dadurch zu erfüllen, dass sie die existentiell quantifizierte Variable an die bereits existierenden Elemente des Universums bindet, bevor sie die Variable an ein neu eingeführtes Element des Universums bindet.

Die Eingabesyntax für die EP-Tableau-Methode ist ein Fragment der Prädikatenlogik erster Stufe, das die gleiche Ausdrucksmächtigkeit hat wie die vollständige Prädikatenlogik erster Stufe [BT98]. Das betrachtete Fragment verlangt sogenannte *bereichsbeschränkte Quantifikation*: Unter anderem muss der Geltungsbereich eines All-Quantors eine Implikation und der Geltungsbereich eines Existenz-Quantors eine Konjunktion sein. Verschachtelung von Quantoren ist erlaubt.

Die betrachteten Interpretationen sind *Term-Interpretationen*, die abgesehen von ihrem Universum wie Herbrand-Interpretationen [Fit90] definiert sind (s. [BT98]). Das Universum einer Herbrand-Interpretation besteht aus allen Grundtermen der Sprache. Im Gegensatz dazu besteht das Universum einer Term-Interpretation $\mathcal{I}$ nur aus den Grundtermen (hier: Konstanten), die in den Grundatomen vorkommen, die durch $\mathcal{I}$ erfüllt werden, sowie einer weiteren Konstante.

Im Folgenden bezeichne $\mathcal{S}$ eine endliche Menge von Formeln (d. h. Integritätsbedingungen und View-Definitionen) in der verlangten Syntax.

EP-Tableaux für eine Menge $\mathcal{S}$ sind Bäume, deren Knoten Mengen geschlossener Formeln sind. Sie werden folgendermaßen induktiv definiert:

1. Der Baum, der aus dem einzigen Knoten $\mathcal{S}$ besteht, ist ein EP-Tableau für $\mathcal{S}$.
2. Sei T ein EP-Tableau für $\mathcal{S}$, L ein Blatt von T, und φ eine Formel in L, die nicht erfüllt wird von der Term-Interpretation, die durch den Zweig spezifiziert wird (d. h. durch die Menge der Grundatome, die im Zweig vorkommen). Dann ist der Baum, den man aus T erhält, indem man an L entsprechend einer auf φ anwendbaren Expansionsregel eines oder mehrere Kinder anhängt, wieder ein EP-Tableau für $\mathcal{S}$. Jedes Kind von L besteht aus L und einer weiteren Formel, oder zwei weiteren Formeln, wenn φ eine Konjunktion ist.

Die *Expansionsregeln* sind die folgenden. Der Teil über der horizontalen Linie beschreibt die Formel φ, auf die die Regel angewandt wird. Unter der horizontalen Linie stehen die zusätzlichen Formeln, die zu den Kind-Knoten hinzugefügt werden. Die senkrechten Striche trennen Alternativen, die verschiedenen Kindern entsprechen.

$\wedge$-Regel:

$$\frac{E_1 \wedge E_2}{\begin{array}{c} E_1 \\ E_2 \end{array}}$$

$\vee$-Regel:

$$\frac{E_1 \vee E_2}{E_1 \mid E_2}$$

PUHR-Regel:[2]

$$\frac{\forall \bar{x}(R(\bar{x}) \ \Rightarrow \ F)}{F[\bar{c}/\bar{x}]}$$

wenn $R[\bar{c}/\bar{x}]$ durch die vom Zweig spezifizierte Interpretation erfüllt wird.

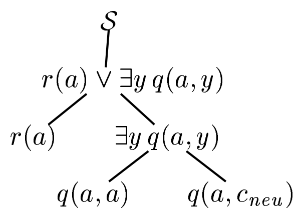
$\exists$-Regel:

$$\frac{\exists x E(x)}{E[c_1/x] \mid \ldots \mid E[c_k/x] \mid E[c_{neu}/x]}$$

wobei $\{c_1, \ldots, c_k\}$ die Menge aller im Knoten auftretenden Konstanten und c_{neu} eine von $c_1, \ldots, c_k$ verschiedene Konstante ist.

[2] Wenn $\bar{c}$ ein Tupel von Konstanten und $\bar{x}$ ein Tupel von Variablen ist, dann bezeichnet $F[\bar{c}/\bar{x}]$ die Formel, die man aus F erhält, wenn jedes freie Auftreten einer Komponente x_i von $\bar{x}$ durch die entsprechende Komponente c_i von $\bar{c}$ ersetzt wird.

Als Beispiel betrachten wir ein EP-Tableau für die Formelmenge $\mathcal{S} = \{p(a),\ \forall x(p(x) \Rightarrow r(x) \vee \exists y\, q(x,y))\}$. Der Kürze halber beschriften wir (im Gegensatz zur obigen Definition) alle Knoten außer der Wurzel nicht mit Formelmengen, sondern nur mit der gegenüber dem Elternknoten hinzukommenden Formel.



Gemäß der Definition von EP-Tableaux kann eine Expansionsregel nur auf eine Formel angewandt werden, die von der durch den Zweig spezifizierten Term-Interpretation nicht erfüllt wird. Diese Bedingung verhindert die weitere Expansion eines Zweigs, der bereits ein Modell spezifiziert.

Ein Zweig eines EP-Tableaus heißt *geschlossen*, wenn er $\perp$ enthält. Ansonsten heißt er *offen*. Ein EP-Tableau heißt *offen*, wenn wenigstens einer seiner Zweige offen ist. Andernfalls heißt es *geschlossen*. Weiterhin wird der Begriff eines *fairen* EP-Tableaus gebraucht: Ein EP-Tableau heißt fair, wenn in jedem offenen Zweig jede mögliche Anwendung einer Expansionsregel auf eine Formel stattfindet.

Wir bezeichnen ein Term-Modell einer Formelmenge $\mathcal{S}$ als *minimal*, wenn keine echte Teilmenge der von ihm erfüllten Grundatome ein Term-Modell von $\mathcal{S}$ spezifiziert [BT98]. Dieser Minimalitätsbegriff ist verschieden von dem von Lorenz [Lor94] (aufbauend auf Hintikka [Hin88]) verwendeten Begriff, der sich auf die Größe der Domain bezieht.

Die EP-Tableau-Methode hat folgende Eigenschaften [BT98]:

- *Korrektheit für die Unerfüllbarkeit:* Wenn ein geschlossenes EP-Tableau für $\mathcal{S}$ existiert, dann ist $\mathcal{S}$ unerfüllbar.
- *Korrektheit für die Erfüllbarkeit:* Wenn ein faires EP-Tableau für $\mathcal{S}$ einen offenen Zweig hat, dann spezifiziert dieser Zweig ein Modell von $\mathcal{S}$.
  Eine unmittelbare Konsequenz dieser Eigenschaft ist die
- *Vollständigkeit für die Unerfüllbarkeit:* Wenn $\mathcal{S}$ unerfüllbar ist, dann ist jedes faire EP-Tableau für $\mathcal{S}$ geschlossen.
- *Vollständigkeit für die Erfüllbarkeit im Endlichen:* Wenn $\mathcal{I}$ ein endliches minimales Modell von $\mathcal{S}$ ist, dann hat jedes faire EP-Tableau für $\mathcal{S}$ einen offenen Zweig $\mathcal{B}$, der bis auf eine Konstantenumbenennung das Modell $\mathcal{I}$ repräsentiert. Diese Eigenschaft von EP-Tableaux ist wesentlich für Datenbankanwendungen. Ihr Beweis ist komplexer als die Beweise der anderen Eigenschaften.

Es ist eine Besonderheit der EP-Tableau-Methode im Vergleich zu anderen Deduktionsmethoden, dass sie nicht nur für die Unerfüllbarkeit *oder* die Erfüllbarkeit im Endlichen vollständig ist, sondern für beide Eigenschaften.

4 Eine Implementierung in Prolog: FINFIMO

Das Prolog-Programm in den Abbildungen 1 and 2 ist eine vereinfachte Variante der Deduktionskomponente von SIC. Sie generiert die Zweige eines fairen EP-Tableaus mit Tiefensuche. Das Rücksetzen über `satisfiable` liefert Term-Modelle, die durch eine Menge $\mathcal{G}$ von Grundatomen in der dynamischen Datenbasis des Prolog-Systems repräsentiert werden. Elemente von $\mathcal{G}$ werden durch das Prädikat `assume` eingefügt und beim Rücksetzen wieder entfernt. Jede andere Formel wird in der Prolog-Datenbasis als Argument des Prädikats `axiom` repräsentiert. Das aktive Universum aller Konstanten, die in der gegebenen Formelmenge auftreten oder durch die $\exists$-Regel eingeführt wurden, wird durch das Prädikat `dom` repräsentiert. $\perp$ wird als `false` dargestellt. Eine universell quantifizierte Formel wie $\forall X\ \forall Y\ \forall Z\ \ \mathit{member}(X,Z) \wedge \mathit{leads}(Y,Z) \Rightarrow X{=}Y \vee \mathit{works\_for}(X,Y)$ wird dargestellt als `all([X,Y,Z], (member(X,Z), leads(Y,Z)) => (X=Y; works_for(X,Y)))`. Entsprechend der Prolog-Konvention werden Kommata für Konjunktionen und Strichpunkte für Disjunktionen verwendet. Eine existentiell quantifizierte Formel wie $\exists Y\ \mathit{department}(Y) \wedge \mathit{member}(X,Y)$ wird dargestellt als `exists(Y, (department(Y), member(X,Y)))`. Universelle und existentielle Quantifikation können geschachtelt werden, wie z. B. in `all([X], employee(X) => exists(Y, (department(Y), member(X,Y))))`.

Fairness wird wie bei SATCHMO durch die Verwendung des Prolog-Prädikats `setof` erreicht, das alle Lösungen auf eine Anfrage liefert. Das Prädikat `violated_instance` liefert diejenigen Instanzen von (Teil-)Formeln, die erfüllt werden müssen. Das Prädikat `satisfy` fügt Grundatome in die Prolog-Datenbank ein, damit der entstehende Datenbank-Zustand die gegebene Formel erfüllt. `satisfy` führt eine Fallunterscheidung nach der syntaktischen Form der gegebenen Formel durch. Im Fall einer Disjunktion oder existentiellen Formel gibt es verschiedene Möglichkeiten, die Formel zu erfüllen. Das Prädikat `satisfy` führt beim Rücksetzen alle Möglichkeiten durch. Im Fall einer existentiellen Formel wird die Wiederverwendung „alter“ Konstanten durch Rücksetzen über das Prädikat `dom` erreicht, während eine neue Konstante durch das Prädikat `new_constant` eingefügt wird. Im Fall einer atomaren Formel werden „eingebaute“ Prädikate (Prädikate mit vordefinierter Bedeutung, die von FINFIMO nicht verändert werden kann, wie etwa die syntaktische Gleichheit) von anderen Prädikaten unterschieden. Da in einem einzelnen Schritt von `satisfiable` eine Menge von Formeln erfüllt wird, kann folgendes passieren: Dadurch, dass einige Formeln erfüllt wurden, könnte eine weitere Formel bereits ebenfalls erfüllt worden sein. Daher enthält die erste Klausel für `satisfy` einen Test, ob die Formel tatsächlich noch unerfüllt ist.

5 Visualisierung: SNARKS

Die Visualisierungskomponente SNARKS [KE97] wurde für verschiedene Zwecke entworfen: für die Entwicklung und das Debugging von Modellgeneratoren à la

```
satisfiable :-
        setof(F, violated_instance(F), F_List),
        !,
        satisfy_list(F_List),
        satisfiable.
satisfiable.

violated_instance(A) :-
        axiom(A),
        atomic_formula(A),
        not A.
violated_instance(A) :-
        axiom((A, _B)),
        not A.
violated_instance(B) :-
        axiom((_A, B)),
        not B.
violated_instance((A ; B)) :-
        axiom((A ; B)),
        not (A ; B).
violated_instance(exists(Var, (R, F))) :-
        axiom(exists(Var, (R, F))),
        not (R, F).
violated_instance(F) :-
        axiom(all(_VarList, R => F)),
        R,
        not F.

satisfy_list([]).
satisfy_list([H | T]) :-
        satisfy(H),
        satisfy_list(T).

satisfy(F) :-
        F,
        !.
satisfy(exists(Var, (R, F))) :-
        dom(Var),
        satisfy(R),
        satisfy(F).
satisfy(exists(Var, (R, F))) :-
        !,
        new_constant(Var),
        assume(dom(Var)),
        satisfy(R),
        satisfy(F).
```

Abbildung1. Das FINFIMO-Programm, Teil 1

```
satisfy(all(VarList, R => F)) :-
        !,
        assume(axiom(all(VarList, R => F))),
        findall(F, (R, not F), L),
        satisfy_list(L).
satisfy((E1, E2)) :-
        !,
        satisfy(E1),
        satisfy(E2).
satisfy((E1 ; E2)) :-
        !,
        (satisfy(E1) ; satisfy(E2)).
satisfy(Atom) :-
        not Atom = false,
        functor(Atom,F,A),
        (built_in(F,A) ->
            Atom
        ;   assume(Atom)).

built_in(=,2).
% ...

assume(Atom) :-
        Atom,
        !.
assume(Atom) :-
        asserta(Atom).
assume(Atom) :-
        once(retract(Atom)),
        fail.

new_constant(X1) :-
        retract(next_constant(X1)),
        X2 is X1 + 1,
        asserta(next_constant(X2)).

next_constant(1).

all(_VarList, R => F) :-
        not (R, not F).
exists(_Var, (R, F)) :-
        (R, F).

atomic_formula(E) :-
        not E = (R => F),
        not E = all(_VarList, R => F),
        not E = exists(_Var, (R, F)),
        not E = (A, B),
        not E = (A ; B).
```

Abbildung2. Das FINFIMO-Programm, Teil 2

SATCHMO, für die Entwicklung und das Debugging logischer Axiomensysteme, für den Vergleich von Inferenz-Maschinen, und um den Prozess der Modellgenerierung zu illustrieren. SNARKS ermöglicht es, Ableitungen als Bäume zu visualisieren, und auch die Expansion dieser Bäume interaktiv zu kontrollieren.

Das im vorigen Abschnitt beschriebene Programm implementiert eine einfache Steuerung, indem es Prologs Tiefensuche übernimmt. Die Inferenzkomponente von SIC besteht tatsächlich aus einer modifizierten Variante dieses Programms, die eine interaktive Steuerung durch den Benutzer über SNARKS ermöglicht. Der Benutzer kann aus einem Sortiment vordefinierter Strategien mit vielen Optionen wählen, um die Entscheidungen während einer Ableitung zu beeinflussen. Dies geht von der Angabe von Tiefen-Grenzen bis zur Auswahl einzelner Ableitungsschritte.

Abbildung 3 zeigt eine der möglichen Darstellungen einer Ableitung. Es handelt sich um die in Abschnitt 2 gegebene Ableitung für Beispiel 1. Diese Ableitung kann mit einer beliebigen Steuerungstrategie erzeugt worden sein.

In diesem Baum ist der Blattknoten des mittleren Zweiges mit der Menge aller atomaren Formeln beschriftet, die dem Datenbank-Zustand in dieser Situation entspricht. Die sonstigen Knoten sind unbeschriftet. Die Kanten sind mit den atomaren Formeln beschriftet, die im jeweiligen Schritt erfüllt wurden. Verschiedene andere Beschriftungen für Knoten und Kanten sind möglich und können jederzeit geändert werden. Insbesondere können die im jeweiligen Schritt verwendeten Integritätsbedingungen angegeben werden (in Abbildung 3 nicht gezeigt).

Verglichen mit Beispiel 1 sind die meisten Identifikatoren etwas gekürzt; sie sollten aber zu erkennen sein. In den Wurzelknoten wurde interaktiv das Atom $employee(john)$ (zusammen mit $dom(john)$ aus dem im vorigen Abschnitt erläuterten Grund) eingefügt. Mit dem Symbol $\perp$ markierte Blattknoten repräsentieren Widersprüche. Wie in Abschnitt 2 erläutert, ist in dieser Ableitung jeder Zweig widersprüchlich.

Natürlich fragt sich der Benutzer, warum dies so ist. In Abbildung 3 hat der Benutzer den Widerspruch im mittleren Zweig duch ein helles Quadrat markiert und eine Abhängigkeitsanalyse angefordert. SNARKS hat nach unten zeigende Dreiecke der gleichen Farbe hinzugefügt um zu zeigen, wie es zu dem Widerspruch in dem Zweig kam: Die neu eingefügte Konstante 1 (d_1 in Abschnitt 2) wäre sowohl eine Abteilung als auch ein Mitarbeiter. Genauso werden die Gründe für den Widerspruch im rechten Zweig durch eine dunklere Farbe angegeben.

Weitere Eigenschaften von SNARKS sind u. a.:

- Die Möglichkeit, versuchsweise die Formelmenge in einem Knoten zu ändern, um Auswirkungen der Korrektur der Integritätsbedingungen vorwegzunehmen, ohne den ganzen Modellgenerierungs-Prozess zu wiederholen;
- Mechanismen um die „Granularität“, mit der ein Baum angezeigt wird, zu beeinflussen, und um ihn in einer kompakten Form darzustellen, die von gerade nicht interessanten Details abstrahiert.
- Alternative Anzeige-Modi und Baum-Layouts;

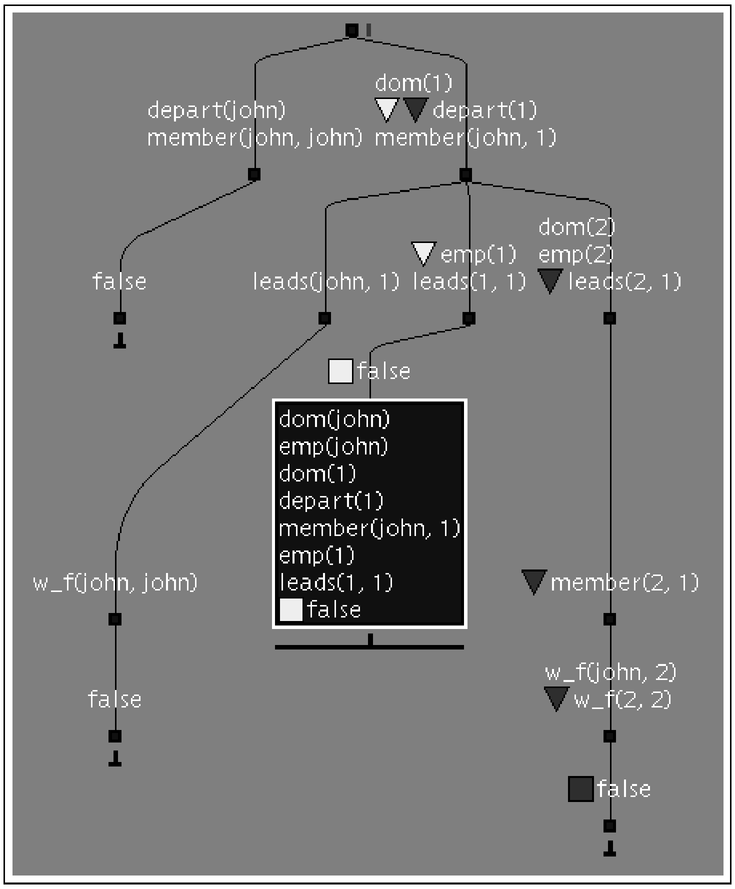


Abbildung3. Ableitungsbaum für Beispiel 1 ausgegeben von SNARKS

– Hilfsmittel um Ableitungen, die von externen Inferenz-Maschinen generiert wurden, zu „importieren“.

SNARKS ist in Java implementiert und kann von geeigneten WWW-Browsern unter `http://www.pms.informatik.uni-muenchen.de/software/` aufgerufen werden.

6 Ausblick

Der Forschungsprototyp SIC [BEST98] wird derzeit mit praktischen Beispielen getestet. Verbesserungen sowohl der Inferenz- als auch der Visualisierungskomponente sind vorstellbar.

Die offensichtlichste Verbesserung wäre eine bequemere Art, Integritätsbedingungen und Regeln auszudrücken. Dies könnte mit einer anderen logischen

Syntax geschehen, oder auch einer kontrollierten Teilsprache einer natürlichen Sprache, wie sie etwa im ATTEMPTO-Project [FS96] verwendet wird. Dieser Punkt war kein zentrales Anliegen bei der Entwicklung des SIC-Prototyps, da er keine wesentlichen Auswirkungen auf die Inferenzkomponente hätte.

Auch eine Anpassung an andere Datenmodelle als das relationale (deduktive) Modell wäre nützlich. Insbesondere eine Anpassung an ein objekt-orientiertes Datenmodell wäre wünschenswert. Dies könnte auf Vorarbeiten wie F-logic [KLW95], Chimera [CM93] oder DEL [FGVLV93] basieren.

Die gegenwärtige Inferenzkomponente findet sicher alle minimalen Modelle, aber sie kann auch nicht-minimale Modelle finden. Für SATCHMO gibt es eine Variante [BY96], die genau die minimalen Modelle generiert. Die Techniken, auf denen diese Variante beruht, sollten mit FINFIMO kombiniert werden.

Modellgeneratoren wie SATCHMO und FINFIMO sind für den erwünschten Fall optimiert, in dem die Integritätsbedingungen erfüllbar sind. Im Fall unerfüllbarer Integritätsbedingungen können die erzeugten Erklärungen (d. h. Unerfüllbarkeitsbeweise) komplexer werden als wünschenswert. Techniken für die Verkleinerung der Beweise sollten untersucht werden. Dazu gehören insbesondere die Verwendung von Lemmata und die Elimination irrelevanter Tableau-Expansions-Schritte.

Integritätsbedingungen mit arithmetischen Berechnungen und Vergleichen können von SIC nur eingeschränkt behandelt werden. Eine allgemeinere Behandlung erfordert die Integration von Techniken der Constraint-Logikprogrammierung wie in [AS97,AS98].

Wie in der Einführung erwähnt verlangt SIC nicht die verbreitete strenge Unterscheidung zwischen extensionaler und intensionaler Datenbank. Unterstützung für solch eine Trennung sollte jedoch in SIC auch integriert werden.

Literatur

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AS97] Slim Abdennadher and Heribert Schütz. Model generation with existentially quantified variables and constraints. In *6th Int. Conf. on Algebraic and Logic Programming*. Springer LNCS 1298, 1997.

[AS98] Slim Abdennadher and Heribert Schütz. $\mathrm{CHR}^{\vee}$: A flexible query language. In *Proc. Int. Conf. on Flexible Query Answering Systems (FQAS).* to appear in Springer LNAI, 1998.

[BDM88] François Bry, Hendrik Decker, and Rainer Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proc. Int. Conf. Extending Data Base Technology (EDBT)*, pages 488–505. Springer LNCS 303, 1988.

[BEST98] François Bry, Norbert Eisinger, Heribert Schütz, and Sunna Torge. SIC: An interactive tool for the design of integrity constraints (system description). In *EDBT'98 Demo Session Proceedings*, pages 45–46, 1998.

[BM86] François Bry and Rainer Manthey. Checking consistency of database constraints: A logical basis. In *Proc. 12th Int. Conf. on Very Large Data Bases (VLDB)*, pages 13–20, 1986.

[BM87] François Bry and Rainer Manthey. Proving finite satisfiability of deductive databases. In *1st Workshop on Computer Science Logic (CSL)*, pages 44–55. Springer LNCS 329, 1987.

[BT98] François Bry and Sunna Torge. A deduction method complete for refutation and finite satisfiability. In *Proc. 6th Europ. Workshop on Logics in AI (JELIA)*, LNAI. Springer, 1998.

[BY96] François Bry and Adnan Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In *5th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 143–159. Springer LNAI 1071, 1996.

[CM93] Stefano Ceri and Rainer Manthey. Chimera: A model and language for active DOOD systems. In *Proceedings of the 2nd East-West Database Workshop*, Workshops in Computing, pages 3–16. Springer, 1995, 1993.

[Dat95] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition, 1995.

[DTU96] Hendrik Decker, Ernest Teniente, and Toni Urpí. How to tackle schema validation by view updating. In *5th Int. Conf. on Extending Database Technology (EDBT)*, pages 535–549. Springer LNCS 1057, 1996.

[FGVLV93] Oris Friesen, Gilles Gauthier-Villars, Alexandre Lefebvre, and Laurent Vieille. Applications of deductive object-oriented databases using DEL. In *Workshop on Programming with Logic Databases, Int. Logic Programming Symposium (ILPS)*, pages 1–22, 1993.

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

[FS96] Norbert E. Fuchs and Rolf Schwitter. Attempto Controlled English (ACE). In *Proc. First Int. Workshop on Controlled Language Applications (CLAW)*, 1996.

[Hin88] Jaakko Hintikka. Model minimization – an alternative to circumscription. *Journal of Automated Reasoning*, 4:1–14, 1988.

[KE97] Mathias Kettner and Norbert Eisinger. The tableau browser SNARKS (system description). In *14th Int. Conf. on Automated Deduction (CADE)*, pages 408–411. Springer LNAI 1249, 1997.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 1995.

[Lor94] Sven Lorenz. A tableau prover for domain minimization. *Journal of Automated Reasoning*, 13:375–390, 1994.

[Man90] Rainer Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *Proc. 2nd Int. Workshop on Foundations of Models and Languages for Data and Objects*, 1990.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In *9th Int. Conf. on Automated Deduction (CADE)*, pages 415–434. Springer LNCS 310, 1988.

[MP96] Sophie Montes and Pierre-Yves Policella. Checking consistency in oodbms. 1996.

[NML93] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB Journal*, 2(2):173–213, 1993.

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

[Tra50] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. In *Dokl. Acad. Nauk.*, SSSR 70, 1950. In Russian; translated into English in Amer. Soc. Trans., Series 2, 23, 1963.