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

Ludwig
Maximilians-
Universität
München

SIC: Satisfiability Checking for Integrity Constraints

François Bry, Norbert Eisinger, Heribert Schütz, Sunna Torge

appeared in *Proc. Deductive Databases and Logic Programming (DDLP'98)*, workshop at JICSLP 1998
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1998-3, April 1998

SIC: Satisfiability Checking for Integrity Constraints

François Bry, Norbert Eisinger, Heribert Schütz, Sunna Torge
{ bry | eisinger | hschuetz | torge }@informatik.uni-muenchen.de
http://www.pms.informatik.uni-muenchen.de

Abstract

SIC is an interactive prototype to assist in the design of finitely satisfiable integrity constraints. Thus SIC addresses the constraint satisfiability problem during the schema design phase of a database. SIC combines two systems, a reasoning component and an interactive visual interface. This paper outlines the functionality of both components and the theoretical background and implementation aspects of the reasoning component.

1 Introduction and Motivation

A database consists of a *schema* and the actual data, the *state*. The schema defines the structure of the data and the *integrity constraints*, conditions that have to be satisfied by all valid database states.[1] Typically, designing the schema and populating the database with data are two different phases of the database development and often involve different people [3, 11].

Integrity constraints raise two issues. One of them, *constraint satisfaction*, is well-studied: How to check whether all integrity constraints are satisfied by a given database state, or whether they remain satisfied by the new state resulting from a transaction.

The other issue, *constraint satisfiability*, was pointed out and discussed in [6, 7, 4, 20, 12]: How to ensure that the integrity constraints can be satisfied by any database state at all and in particular by "interesting" ones. This may be prevented by a design flaw in the schema. Constraint satisfiability is an essential prerequisite for the population phase. If the schema designer has specified integrity constraints that cannot be satisfied, every possible transaction will be rejected and the database cannot be populated. The user (or, more likely, the schema designer during a test phase) has no indication that the reason for that really is a design flaw in the schema rather than the lack of skill in putting the transactions together. So there ought to be support for the schema designer to detect and eliminate such flaws.

Other design flaws may result in integrity constraints that can be satisfied, but only by hypothetical, infinitely large database states. Obviously no actual database state will satisfy such integrity constraints and again there should be support to detect and eliminate this kind of flaws.

The problem addressed by constraint satisfiability is a real one. There are amazingly simple and realistic examples where one of the design flaws above occurs and can easily escape the designer's attention. See the examples section below. If several designers work on different parts of the integrity constraints, the problem becomes almost inevitable.

[1] There may also be rules defining views. For the purpose of this paper we consider them as integrity constraints, too. Moreover, we assume that views may also contain extensional data. We do not assume a strict separation between extensional and intensional database.

What is required, then, is a way to check during the schema design phase whether a set of integrity constraints is *finitely satisfiable*, or is *unsatisfiable*, or, in the remaining case, is an *infinity axiom*, i. e., is satisfiable but not finitely satisfiable.

Now such a set is just a set of logical formulas, and unfortunately the relevant questions are not decidable. More precisely, the question whether a set of first-order logical formulas is unsatisfiable, is semi-decidable: there are procedures that take a set of formulas as argument and report "unsatisfiable" in finite time if the set is indeed unsatisfiable; for satisfiable sets, however, termination cannot be guaranteed [22]. Likewise, the question whether a set of first-order logical formulas is finitely satisfiable, is semi-decidable: there are procedures that confirm the finite satisfiability of a set in finite time, but may run forever on other sets [23]. As a consequence, the question whether a set of formulas is satisfiable, but not finitely satisfiable, is not semi-decidable.

Of course it would be desirable to integrate such capabilities into one system that supports the designer during the schema design phase. If the set of integrity constraints is finitely satisfiable, the system should report this in finite time, and should moreover support an analysis whether "interesting" database states are among the satisfying ones. If the set is unsatisfiable, the system should again report this in finite time, but should also support the designer in finding the reasons of the unsatisfiability. If the set is satisfiable, but not finitely satisfiable, we cannot expect the system always to come up with an answer, but nevertheless its steps should provide sufficient hints for the designer to recognise that and why the set has these properties.

SIC is a prototypical system of this kind for relational (or deductive) databases. It combines a reasoning component called FINFIMO [8] and an interactive visualisation component called SNARKS [17]. The reasoning component is based on a method that is both complete for finite satisfiability and complete for unsatisfiability. This method is based on the ideas in [6, 7, 4, 20][2] and is a refinement of the method underlying the model generator SATCHMO [21, 9], which was originally developed for this kind of applications [4]. The visualisation component was originally developed as a debugging tool for SATCHMO and variants thereof. Its interactive features can provide support for investigating "interesting" database states in the case of finite satisfiability, and for identifying the reasons if the set of integrity constraints is not finitely satisfiable.

2 Examples

The examples in this section serve two purposes: they illustrate that design flaws of the kind above can indeed be easily overlooked, and they give an intuitive idea of the kind of reasoning by which SIC helps detect the design flaws.

Example 1 The following set of integrity constraints is adapted from [20]. In order to save parentheses we assume quantifiers to have maximal scope. $\perp$ evaluates to false in all interpretations. Note that (1) is our way to express "$\neg\exists X \;\; \mathit{department}(X) \wedge \mathit{employee}(X)$".

(1) Departments and employees are different entities:
$\forall X \;\; \mathit{department}(X) \wedge \mathit{employee}(X) \;\Rightarrow\; \perp$

(2) The manager X of a department Y is also a member of this department:
$\forall X \, \forall Y \;\; \mathit{leads}(X,Y) \;\Rightarrow\; \mathit{member}(X,Y)$

[2] But it differs from the approach in [12], which is based on abduction rather than model generation.

(3) A member X of a department Z works for the manager Y of this department:
$\forall X\ \forall Y\ \forall Z\ \ member(X,Z) \wedge leads(Y,Z)\ \Rightarrow\ works\_for(X,Y)$

(4) Every employee X is a member of some department Y:
$\forall X\ \ employee(X)\ \Rightarrow\ \exists Y\ department(Y) \wedge member(X,Y)$

(5) Every department X is managed by some employee Y:
$\forall X\ \ department(X)\ \Rightarrow\ \exists Y\ employee(Y) \wedge leads(Y,X)$

(6) Nobody works for herself/himself:
$\forall X\ \ works\_for(X,X)\ \Rightarrow\ \bot$

This set of integrity constraints can be satisfied, namely by the database with an empty state, and SIC does indicate this. Thus, at the start of the population phase everything is fine. However, this is not a very interesting case.

The notion of "interesting" is application-dependent. Users of SIC can modify states interactively and let the reasoning resume from modified states. This amounts to saying, "we are interested in states in which this and that holds, now what would be the consequences for such states?" In the example above we are interested in nonempty states, especially in ones in which there is an employee. The simplest way to investigate such states is to insert interactively the formula $employee(john)$. Given this modification, SIC will reason as follows.

Since $employee(john)$ holds, integrity constraint (4) requires the existence of some department Y of which $john$ is a member. In case Y is $john$, we get $department(john)$, which leads to a contradiction by (1). There remains the case that Y is another entity, say d_1, such that $employee(john)$ and $department(d_1)$ and $member(john,d_1)$ hold.

Now (5) requires the existence of a manager Y of d_1. In the first case Y is $john$ and the only new consequence is $leads(john,d_1)$, which together with $member(john,d_1)$ and (3) implies $works\_for(john,john)$. By (6) this results in a contradiction. In the second case Y is d_1, and we get again a contradiction by (1). There remains the case that Y is another entity, say e_2, such that $employee(john)$ and $department(d_1)$ and $member(john,d_1)$ and $employee(e_2)$ and $leads(e_2,d_1)$ hold.

From the latter we get $member(e_2,d_1)$ by (2), then $works\_for(e_2,e_2)$ by (3), and finally a contradiction by (6).

Thus, if there is an employee, there is no way to satisfy the integrity constraints. SIC constructs a derivation that reflects the reasoning above and visualises it as a tree. Each branch of the tree can be regarded as simulating an attempt to combine the insertion of $employee(john)$ with additional insertions into a transaction producing a valid database state. Since each branch fails and the tree systematically covers all possible such attempts, the designer of the integrity constraints knows for sure that any actual transaction would be rejected.

The visualisation component of SIC makes it possible to label the tree with the relevant arguments or with other information and to highlight the dependencies between a formula and its premises and consequences. These and other features of SIC give the designer a good chance to identify the design flaw: constraint (3) is too strong if X and Y happen to be the same. The designer overlooked this borderline case.

If (3) is corrected to $\forall X\ \forall Y\ \forall Z\ member(X,Z) \wedge leads(Y,Z)\ \Rightarrow\ X{=}Y \vee works\_for(X,Y)$, the constraints can be satisfied even if there is an employee, and SIC will show this. In fact it constructs sample or "dummy" states satisfying the integrity constraints, thus helping the schema designer in the need to invent test data.

Example 2 We illustrate the infinity problem with the following integrity constraints:

(1) Every employee works for somebody:
$\forall X \;\; employee(X) \;\Rightarrow\; \exists Y \; employee(Y) \wedge works\_for(X, Y)$

(2) No employee works for herself/himself:
$\forall X \;\; works\_for(X, X) \;\Rightarrow\; \bot$

(3) The *works_for* relation is transitive:
$\forall X \; \forall Y \; \forall Z \;\; works\_for(X, Y) \wedge works\_for(Y, Z) \;\Rightarrow\; works\_for(X, Z)$

This set of integrity constraints is again satisfied by a database with an empty state, and again we insert $employee(john)$ interactively in order to investigate more interesting cases. Here the reasoning is as follows:

Since $employee(john)$ holds, (1) requires the existence of an employee Y for whom *john* works. If Y is *john*, we get $works\_for(john, john)$ and a contradiction by (2). Thus Y must be another entity e_1, such that $employee(john)$ and $employee(e_1)$ and $works\_for(john, e_1)$ hold.

Next, (1) requires the existence of a Y for whom e_1 works. In case Y is *john*, we get $works\_for(e_1, john)$, then $works\_for(john, john)$ by (3) and a contradiction by (2). In case Y is e_1, we get $works\_for(e_1, e_1)$ and a contradiction by (2). Thus Y is yet another entity e_2.

Again, (1) requires the existence of a Y for which e_2 works. Again, the cases where Y is one of the former entities are contradictory and we must introduce an entitiy e_3, and so on.

The integrity constraints can be satisfied by introducing infinitely many employees together, but not with any finite number. This example belongs to the class of infinity axioms, for which no procedure can guarantee termination. However, a designer using SIC would gradually develop and analyse the derivation. The designer would soon recognise the overall regularity with many irrelevant contradictory branches and one branch with a reoccuring pattern and thus find the design flaw: constraint (1) does not hold for the people on top of the company hierarchy. The omission of this borderline case actually turned the integrity constraints into an infinity axiom.

A suitable repair is $\forall X \; employee(X) \;\Rightarrow\; boss(X) \;\vee\; \exists Y \; employee(Y) \wedge works\_for(X, Y)$, and with that SIC can demonstrate the finite satisfiability.

Although a general characterisation of all kinds of infinity axioms is not possible, they often involve some reappearing pattern or unexpected consequences. Such phenomena help the designer to suspect that there may be a problem, and SIC provides the support to make the phenomena clearly visible. Note the analogy to a debugger for an imperative programming language, which in spite of the undecidability of the halting problem can usually provide sufficient information for the programmer to identify nonterminating loops and their causes.

In both examples the design flaws are not straightforward to recognise, especially by people who are not thoroughly experienced in logic formalisms and if the given integrity constraints are just part of a larger set that was possibly developed by more than one person. In our opinion this is ample motivation for the need for a system in the style of SIC.

3 A Method Complete for Unsatisfiability and for Finite Satisfiability

The reasoning component of SIC is based on the Extended Positive Tableau method (short EP Tableau method) [8]. This method extends the PUHR Tableau method [9] underlying the model generator SATCHMO [21]. A major feature of the PUHR Tableau method is that it generates only positive facts, leaving negative information implicit, which is a most natural approach for database applications.

On the other hand, the PUHR Tableau method is not complete for finite satisfiability, i.e., even if a set of integrity constraints has a finite model, the PUHR Tableau method does not necessarily generate one. The reason for that is that PUHR Tableau expansion may produce terms with function symbols nested to arbitrary depths, thus leading to infinite branches. The PUHR Tableau method cannot avoid supporting function symbols, because it requires Skolemization of existentially quantified variables, which frequently introduces function symbols. In our application for relational databases, however, there is no need for function symbols other than those Skolemization would introduce.

In contrast to the PUHR Tableau method, the EP Tableau method does not require Skolemization. Instead, it handles existential quantifiers explicitly with a special expansion rule [7, 16, 19]. This ensures completeness with respect to finite satisfiability. The underlying idea is as follows: The method tries to satisfy an existential formula first by binding the existentially quantified variable to already existing domain elements before it binds the variable to a newly introduced domain element.

The input syntax for the EP Tableau method is a fragment of first-order logic, which has been shown to have the same expressive power as full first-order logic [8]. Quantification is range-restricted: Among other things, the scope of a universal quantifier must be an implication and the scope of an existential quantifier must be a conjunction. In both cases nesting of quantifiers is allowed. The interpretations considered are *term interpretations*, which, except for their domains, are defined like Herbrand interpretations [13]. The domain of an Herbrand interpretation consists of all possible ground terms. In contrast, the domain of a term interpretation $\mathcal{I}$ contains only those ground terms (here constants) that occur in the ground atoms satisfied by $\mathcal{I}$, plus one additional constant.

In the following, $\mathcal{S}$ denotes a finite set of formulas (i.e., the integrity constraints and view definitions) in the required syntax. *EP tableaux* for a set $\mathcal{S}$ are trees whose nodes are sets of closed formulas. They are inductively defined as follows:

1. The tree consisting of the single node $\mathcal{S}$ is an EP tableau for $\mathcal{S}$.

2. Let T be an EP tableau for $\mathcal{S}$, L a leaf node of T, and φ a formula in L that is not satisfied in the term interpretation specified by the branch, i.e., by the set of ground atoms in L. Then the tree obtained from T by appending one or more children to L according to an expansion rule applicable to φ is again an EP tableau for $\mathcal{S}$. Each child of L consists of L and one more formula, or two more formulas if φ is a conjunction.

The expansion rules are as follows. The part above the horizontal line describes the formula φ to which the rule is applied. Below the horizontal line are the additional formulas to be added to the child nodes. Vertical bars separate alternatives corresponding to different children.

$\wedge$ rule:	$\vee$ rule:	PUHR rule:[3]	
$\dfrac{E_1 \wedge E_2}{\begin{array}{c}E_1\\ E_2\end{array}}$	$\dfrac{E_1 \vee E_2}{E_1 \mid E_2}$	$\dfrac{\forall \bar{x}(R(\bar{x}) \;\Rightarrow\; F)}{F[\bar{c}/\bar{x}]}$	where $R[\bar{c}/\bar{x}]$ is satisfied by the interpretation specified by the branch.

[3] If $\bar{c}$ is a tuple of constants and $\bar{x}$ a tuple of variables, then $F[\bar{c}/\bar{x}]$ denotes the formula F where every free occurrence of a member of $\bar{x}$ is replaced by the corresponding member of $\bar{c}$.

$\exists$ rule:

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

where $\{c_1 \ldots c_k\}$ is the set of all constants occurring in the node and c_{new} is a constant distinct from $c_1, \ldots, c_k$.

As an example, we give an EP tableau for $\mathcal{S} = \{p(a),\ \forall x(p(x) \Rightarrow r(x) \vee \exists y\, q(x,y))\}$. For the sake of brevity, the non-root nodes are (in contrast to the definition above) not labelled with sets of formulas, but only with the single formula added w.r.t. the node's parent.

$$\begin{array}{c}
\mathcal{S} \\
| \\
r(a) \vee \exists y\, q(a,y) \\
\swarrow \qquad \searrow \\
r(a) \qquad\qquad \exists y\, q(a,y) \\
\qquad\qquad \swarrow \qquad \searrow \\
\qquad\qquad q(a,a) \qquad q(a, c_{new})
\end{array}$$

According to the definition of EP tableaux, an expansion rule may be applied only to a formula that is not satisfied in the term interpretation specified by the branch. This condition prevents further expansion of a branch already specifying a model.

A branch of an EP tableau is *closed* if it contains $\perp$. Otherwise, it is *open*. An EP tableau is *open* if at least one of its branches is open; otherwise, it is *closed*. Furthermore the notion of a *fair* EP tableau is needed, which means that in every open branch every possible application of an expansion rule to a formula in the branch takes place after finitely many expansion steps.

The EP Tableau method has the following properties [8]: *soundness for unsatisfiability* (if there exists a closed EP tableau for $\mathcal{S}$, then $\mathcal{S}$ is unsatisfiable) and *soundness for satisfiability* (if a fair EP tableau for $\mathcal{S}$ has an open branch, the open branch specifies a model of $\mathcal{S}$); an immediate consequence of the latter is *completeness for unsatisfiability* (if $\mathcal{S}$ is unsatisfiable, then every fair EP tableau for $\mathcal{S}$ is closed). Finally, the method also enjoys *completeness for finite satisfiability*: If $\mathcal{I}$ is a finite minimal[4] model of $\mathcal{S}$, then every fair EP tableau for $\mathcal{S}$ has a finite open branch $\mathcal{B}$ such that, up to a renaming of constants, the branch $\mathcal{B}$ represents $\mathcal{I}$. This property of EP tableaux is essential with regard to database applications. Its proof is more complex than those of the other results. Thus the EP Tableau method is not only complete for either unsatisfiability or finite satisfiability, but for both of them.

4 A Prolog Implementation: FINFIMO

The Prolog program given in Figure 1 constructs the branches of fair EP tableaux using a depth-first strategy. Backtracking over `satisfiable` returns term models, which are represented by a set $\mathcal{G}$ of ground atoms in the dynamic database of the Prolog system. Elements of $\mathcal{G}$ are inserted and removed on backtracking by the predicate `assume`. Any other formula is represented in the Prolog database as an argument of the predicate `axiom`. The active domain of all constants occurring in the given set of formulas and the new constants introduced by the $\exists$-rule is represented by the predicate `dom`. $\perp$ is represented by `false`. A universally quantified formula like $\forall X\ \forall Y\ \forall Z\ member(X,Z) \wedge leads(Y,Z) \Rightarrow X{=}Y \vee works\_for(X,Y)$ is

[4] A term model of $\mathcal{S}$ is *minimal*, if no proper subset of the ground atoms it satisfies specifies a term model of $\mathcal{S}$.

```
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).

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).
```

Figure 1: The FINFIMO program

represented as `all([X,Y,Z], (member(X,Z), leads(Y,Z)) => (X=Y; works_for(X,Y)))`. Following the Prolog convention, commas are used for conjunctions and semicolons for disjunctions. An existentially quantified formula like $\exists Y\ \mathit{department}(Y) \wedge \mathit{member}(X,Y)$ is represented as `exists(Y, (department(Y), member(X,Y)))`. Universal and existential quantifications may be nested as in `all([X], employee(X) => exists(Y, (department(Y), member(X,Y))))`.

Fairness is ensured by the call to the Prolog built-in all-solutions predicate `setof`. The predicate `violated_instance` returns those instances of (parts of) formulas that need to be satisfied. The predicate `satisfy` causes the insertion of ground atoms in order to satisfy the violated instance in the database state thus obtained. `satisfy` performs a case analysis on the syntactical form of the given instance. In case of a disjunction or an existential formula, there are different possibilities to satisfy the formula. The predicate `satisfy` returns all possibilities on backtracking. In the existential case, the reuse of "old" constants is performed by a backtrackable call to the `dom` predicate, whereas the introduction of a new constant is performed by the predicate `new_constant`. In the atomic case, built-in predicates (these are all predicates with predefined semantics that may not be altered by FINFIMO) are distinguished from other predicates. Since in a single `satisfiable` step a set of instances is satisfied, the following might happen: After having satisfied some violated instances, another one might have become satisfied as well. Therefore the first clause for `satisfy` is a test whether the formula is still unsatisfied.

5 Visualisation: SNARKS

The visualisation component SNARKS [17] was designed as a tool for several purposes: for developing and debugging model generators à la SATCHMO, for developing and debugging logical theories, for comparing inference engines, or for illustrating the model generation process. It allows to visualise derivations as trees and also to control interactively the expansion of those trees.

The program described in the previous section implements one simple control strategy by reusing Prolog's depth-first search. SIC's reasoning component actually consists of a modified version of this program that supports interactive control by the user through SNARKS. The user may choose from a variety of predefined strategies with many options to influence the decisions during a derivation ranging from depth limits up to the selection of individual derivation steps.

Figure 2 shows one of the possible visualisations of a derivation. The derivation is the one explained in the examples section for Example 1 and may have been obtained using whichever control strategy.

In this tree the leaf node of the middle branch is labelled with the set of all atomic formulas corresponding to the database state in this situation. The other nodes are displayed without labels. The edges are labelled with the atomic formulas that are newly derived in each step. A number of other labellings for nodes and edges are possible and can be changed any time, in particular labellings showing the integrity constraints used in each step, which are not displayed in Figure 2.

Compared to Example 1 most of the identifiers are slightly abbreviated but ought to be recognisable. The root node was interactively changed to contain $\mathit{employee}(\mathit{john})$ (together with $\mathit{dom}(\mathit{john})$ for the purpose explained in the previous section). Leaf nodes marked with

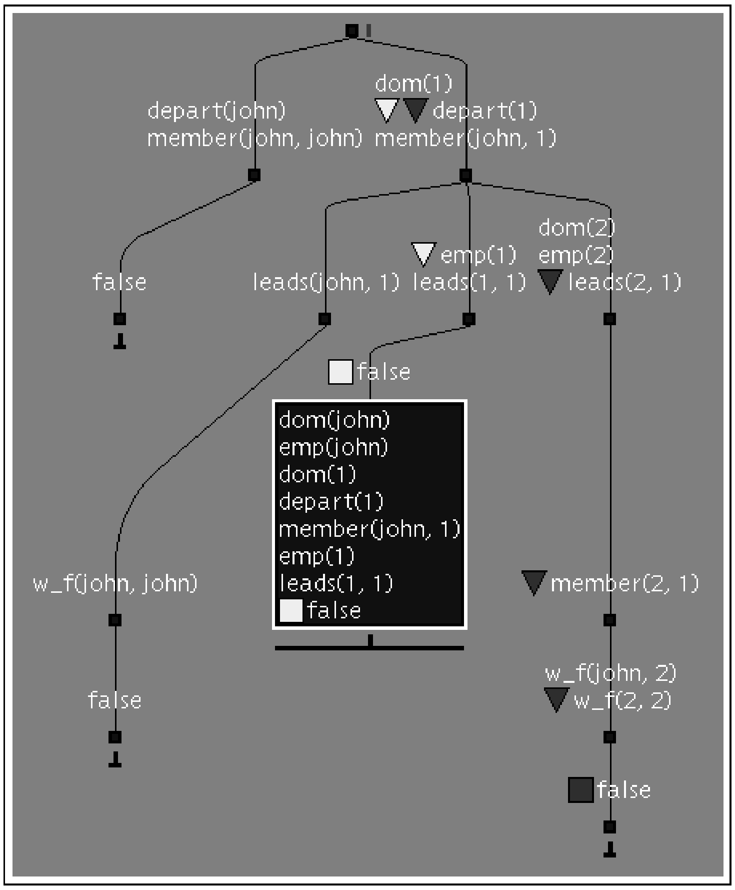

Figure 2: Derivation tree for Example 1 as visualised by SNARKS

the $\perp$ symbol represent contradictions. As explained in the examples section, every branch of this derivation happens to be contradictory,

It is natural for the user to ask why this is so. In Figure 2 the user marked the contradiction in the middle branch with a light-coloured square and requested a dependency analysis. SNARKS added the downward pointing triangles of the same colour, which indicate how the contradiction in this branch came about: because the newly introduced constant 1 (called d_1 in the examples section) would be both a department and an employee. Likewise, the reasons for the contradiction in the rightmost branch are indicated by a darker colour.

Other features of SNARKS include: possibilities to modify tentatively the set of formulas in a node in order to anticipate the effects of corrections of the integrity constraints without reperforming the model generation process from scratch; mechanisms to influence the "granularity" with which a tree is displayed and to show it in a compacted form that abstracts from details outside the user's current focus of interest; alternative display styles and layouts of the tree; means to "import" derivations constructed by external inference engines.

SNARKS is implemented in Java and can be called from appropriate WWW browsers at `http://www.pms.informatik.uni-muenchen.de/software/`.

6 Perspectives

The SIC research prototype [5] is currently being tested on practical examples. Improvements of both the reasoning and the visualisation component can be considered.

The most obvious improvement would be a more convenient form to express integrity constraints and rules. There could be a different logical syntax or even a controlled subset of a natural language such as used, for example, in the ATTEMPTO project [15]. This point was not a major concern in developing the SIC prototype, because it would not affect the reasoning component in a significant way.

Also, an adaptation to data models other than the relational (deductive) one would be useful. Especially, an adaptation to an object-oriented data model would be desirable. This could be based on existing work such as F-logic [18], Chimera [10] or DEL [14].

The present reasoning component is guaranteed to find all minimal models, but may also generate non-minimal ones. In the case of SATCHMO there is a variant that generates exactly the minimal models [9]. The techniques used to achieve that should be combined with FINFIMO.

Model generators like SATCHMO and FINFIMO are optimised for the case of satisfiable integrity constraints. In the case of unsatisfiable integrity constraints, however, the generated explanations (i. e., proofs) for unsatisfiability can become more complex than necessary. Techniques to be investigated for reducing proof sizes include the use of lemmata and a-posteriori elimination of irrelevant tableau expansion steps.

Integrity constraints containing arithmetic comparisons can be handled by SIC only in a restricted way. A more general treatment requires the integration of constraint logic programming techniques into our method as in [1, 2].

As has been said in the introduction, we do not assume the usual strict separation between extensional and intensional database. Support for such a separation should be integrated into SIC.

References

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

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

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] F. Bry, H. Decker, and R. 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.

[5] F. Bry, N. Eisinger, H. Schütz, and S. Torge. SIC: An interactive tool for the design of integrity constraints (system description). In *EDBT'98 Demo Session Proceedings*, pages 45–46, 1998.

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

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

[8] F. Bry and S. Torge. Model generation for applications – a tableaux method complete for finite satisfiability. Research report PMS-FB-1997-15, Institut für Informatik, Ludwig-Maximilians-Universität München, 1997.

[9] F. Bry and A. 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.

[10] S. Ceri and R. 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.

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

[12] H. Decker, E. Teniente, and T. 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.

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

[14] O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. 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.

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

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

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

[18] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 1995.

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

[20] R. 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.

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

[22] R. Smullyan. *First-Order Logic.* Springer, 1968.

[23] 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.