Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

310

E. Lusk R. Overbeek (Eds.)

9th International Conference on Automated Deduction

Argonne, Illinois, USA, May 23–26, 1988
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

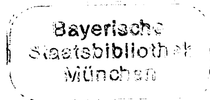
Z 74. 576 / 370

Editorial Board

D. Barstow W. Brauer P. Brinch Hansen D. Gries D. Luckham
C. Moler A. Pnueli G. Seegmüller J. Stoer N. Wirth

Editors

Ewing Lusk
Ross Overbeek
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA


Bayerische
Staatsbibliothek
München

CR Subject Classification (1987): I.2.3

ISBN 3-540-19343-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-19343-X Springer-Verlag New York Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in other ways, and storage in data banks. Duplication of this publication or parts thereof is only permitted under the provisions of the German Copyright Law of September 9, 1965, in its version of June 24, 1985, and a copyright fee must always be paid. Violations fall under the prosecution act of the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1988
Printed in Germany

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

Program Committee:

- Peter Andrews, *Carnegie Mellon University*
- W. W. Bledsoe, *University of Texas*
- Alan Bundy, *University of Edinburgh*
- Robert Constable, *Cornell University*
- Seif Haridi, *Swedish Institute of Computer Science*
- Lawrence Henschen, *Northwestern University*
- Deepak Kapur, *State University of New York at Albany*
- Dallas Lankford, *Louisiana Technological University*
- Jean-Louis Lassez, *T. J. Watson Research Center, IBM*
- Ewing Lusk, *Argonne National Laboratory*
- Michael McRobbie, *Australian National University*
- Hans-Jürgen Ohlbach, *University of Kaiserslautern*
- Ross Overbeek, *Argonne National Laboratory*
- William Pase, *I. P. Sharp Associates, Ltd.*
- Jörg Siekmann, *University of Kaiserslautern*
- Mark Stickel, *SRI International*
- James Williams, *The MITRE Corporation*

Table of Contents

Session 1

First-Order Theorem Proving Using Conditional Rewrite Rules
Hantao Zhang and Deepak Kapur 1

Elements of Z-Module Reasoning
T.C. Wang 21

Session 2

Learning and Applying Generalised Solutions Using Higher Order Resolution
Michael R. Donat and Lincoln A. Wallen 41

Specifying Theorem Provers in a Higher-Order Logic Programming Language
Amy Felty and Dale Miller 61

Query Processing in Quantitative Logic Programming
V. S. Subrahmanian 81

Session 3

An Environment for Automated Reasoning About Partial Functions
David A. Basin 101

The Use of Explicit Plans to Guide Inductive Proofs
Alan Bundy 111

LOGICALC: An Environment for Interactive Proof Development
D. Duchier and D. McDermott 121

Session 4

Implementing Verification Strategies in the KIV-System
M. Heisel, W. Reif, and W. Stephan 131

Checking Natural Language Proofs
Donald Simon 141

Consistency of Rule-based Expert Systems
Marc Bezem 151

Session 5

A Mechanizable Induction Principle for Equational Specifications
Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy 162

Finding Canonical Rewriting Systems Equivalent to a Finite Set of Ground Equations in Polynomial Time
Jean Gallier, Paliath Narendran, David Plaisted, Stan Raatz, and Wayne Snyder 182

Session 6

Towards Efficient "Knowledge-Based" Automated Theorem Proving for Non-Standard Logics
Michael A. McRobbie, Robert K. Meyer, and Paul B. Thistlewaite 197

Propositional Temporal Interval Logic is PSPACE Complete
A. A. Aaby and K. T. Narayana 218

Session 7

Computational Metatheory in Nuprl
Douglas J. Howe 238

Type Inference in Prolog
H. Azzoune 258

Session 8

Procedural Interpretation of Non-Horn Logic Programs
Jack Minker and Arcot Rajasekar 278

Recursive Query Answering with Non-Horn Clauses
Shan Chi and Lawrence J. Henschen 294

Session 9

Case Inference in Resolution-Based Languages
T. Wakayama and T.H. Payne 313

Notes on Prolog Program Transformations, Prolog Style, and Efficient Compilation to the Warren Abstract Machine
Ralph M. Butler, Rasiah Loganantharaj, and Robert Olson 323

Exploitation of Parallelism in Prototypical Deduction Problems
Ralph M. Butler and Nicholas T. Karonis 333

Session 10

A Decision Procedure for Unquantified Formulas of Graph Theory
Louise E. Moser 344

Adventures in Associative-Commutative Unification (A Summary)
Patrick Lincoln and Jim Christian 358

Unification in Finite Algebras is Unitary(?)
Wolfram Büttner 368

Session 11

Unification in a Combination of Arbitrary Disjoint Equational Theories
Manfred Schmidt-Schauss 378

Partial Unification for Graph Based Equational Reasoning
Karl Hans Bläsius and Jörg H. Siekmann 397

Session 12

SATCHMO: A theorem prover implemented in Prolog
Rainer Manthey and François Bry 415

Term Rewriting: Some Experimental Results
Richard C. Potter and David A. Plaisted 435

Session 13

Analogical Reasoning and Proof Discovery
Bishop Brock, Shaun Cooper, and William Pierce 454

Hyper-Chaining and Knowledge-Based Theorem Proving
Larry Hines 469

Session 14

Linear Modal Deductions
L. Fariñas del Cerro and A. Herzig 487

A Resolution Calculus for Modal Logics
Hans Jürgen Ohlbach 500

Session 15

Solving Disequations in Equational Theories
Hans-Jürgen Bürckert 517

On Word Problems in Horn Theories
Emmanuel Kounalis and Michael Rusinowitch 527

Canonical Conditional Rewrite Systems
Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar 538

Program Synthesis by Completion with Dependent Subtypes
Paul Jacquet 550

Session 16

Reasoning about Systems of Linear Inequalities
Thomas Käufl 563

A Subsumption Algorithm Based on Characteristic Matrices
Rolf Socher 573

A Restriction of Factoring in Binary Resolution
Arkady Rabinov 582

Supposition-Based Logic for Automated Nonmonotonic Reasoning
Philippe Besnard and Pierre Siegal 592

Session 17

Argument-Bounded Algorithms as a Basis for Automated Termination Proofs
Christoph Walther 602

Two Automated Methods in Implementation Proofs
Leo Marcus and Timothy Redmond 622

Session 18

A New Approach to Universal Unification and Its Application to AC-Unification
Mark Franzen and Lawrence J. Henschen 643

An Implementation of a Dissolution-Based System Employing Theory Links
Neil V. Murray and Erik Rosenthal 658

Session 19

Decision Procedure for Autoepistemic Logic
Ilkka Niemelä 675

Logical Matrix Generation and Testing
Peter K. Malkin and Errol P. Martin 685

Optimal Time Bounds for Parallel Term Matching
Rakesh M. Verma and I.V. Ramakrishnan 694

Session 20

Challenge Equality Problems in Lattice Theory
William McCune 704

Single Axioms in the Implicational Propositional Calculus
Frank Pfenning 710

Challenge Problems Focusing on Equality and Combinatory Logic: Evaluating Automated Theorem-Proving Programs
Larry Wos and William McCune 714

Challenge Problems from Nonassociative Rings for Theorem Provers
Rick L. Stevens 730

System Abstracts

An Interactive Enhancement to the Boyer-Moore Theorem Prover
Matt Kaufmann 735

A Goal Directed Theorem Prover
David A. Plaisted 737

m-NEVER System Summary
Bill Pase and Sentot Kromodimoeljo 738

EFS – An Interactive Environment for Formal Systems
Timothy G. Griffin 740

Ontic: A Knowledge Representation System for Mathematics
David McAllester 742

Some Tools for an Inference Laboratory (ATINF)
Thierry Boy de la Tour, Ricardo Caferra, and Gilles Chaminade 744

QUANTLOG: A System for Approximate Reasoning in Inconsistent Formal Systems
V. S. Subrahmanian and Zerksis D. Umrigar 746

LP: The Larch Prover
Stephen J. Garland and John V. Guttag 748

The KLAUS Automated Deduction System
Mark E. Stickel 750

A Prolog Technology Theorem Prover
Mark E. Stickel 752

Lambda Prolog: An Extended Logic Programming Language
Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov 754

SYMEVAL: A Theorem Prover Based on the Experimental Logic
Frank M. Brown and Seung S. Park 756

ZPLAN: An Automatic Reasoning System for Situations
Frank M. Brown, Seung S. Park, and Jim Phelps 758

The TPS Theorem Proving System
Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning 760

MOLOG: A Modal PROLOG
Pierre Bieber, Luis Fariñas del Cerro, and Andreas Herzig 762

PARTHENON: A Parallel Theorem Prover for Non-Horn Clauses
P. E. Allen, S. Bose, E. M. Clarke, and S. Michaylov 764

An nH-Prolog Implementation
B. T. Smith and D. W. Loveland 766

RRL: A Rewrite Rule Laboratory
Deepak Kapur and Hantao Zhang 768

GEOMETER: A Theorem Prover for Algebraic Geometry
David A. Cyrluk, Richard M. Harris, and Deepak Kapur 770

Isabelle: The next seven hundred theorem provers
Lawrence C. Paulson 772

The CHIP System: Constraint Handling in Prolog
M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and A. Herold 774

SATCHMO: a theorem prover implemented in Prolog

Rainer Manthey and François Bry

ECRC
Arabellastr. 17, D-8000 Muenchen 81
West Germany

Abstract

Satchmo is a theorem prover consisting of just a few short and simple Prolog programs. Prolog may be used for representing problem clauses as well. SATCHMO is based on a model-generation paradigm. It is refutation-complete if used in a level-saturation manner. The paper provides a thorough report on experiences with SATCHMO. A considerable amount of problems could be solved with surprising efficiency.

1. Introduction

In this article we would like to propose an approach to theorem proving that exploits the potential power of Prolog both as a representation language for clauses and as an implementation language for a theorem prover. SATCHMO stands for 'SATisfiability CHecking by MOdel generation'. It is a collection of fairly short and simple Prolog programs to be applied to different classes of problems. The programs are variations of two basic procedures: the one is incomplete, but allows to solve a wide range of problems with considerable efficiency; the other is based on a level-saturation organization thus achieving completeness but partly sacrificing the efficiency of the former.

Horn clause problems can be very efficiently solved in Prolog provided they are such that the Prolog-specific limitations due to missing occurs check and unbounded depth-first search are respected. As an example we mention Schubert's Steamroller [WAL 84], a problem recently discussed with some intensity: the problem consists of 27 clauses, 26 of which can be directly represented in Prolog without any reformulation and is checked for satisfiability within a couple of milliseconds by any ordinary Prolog interpreter. The idea of retaining Prolog's power for Horn clauses while extending the language in order to handle full first-order logic has been the basis of Stickel's "Prolog Technology Theorem Prover" (PTTP) [STI 84]. Stickel proposes to overcome

Prolog's limitation to non-Horn clauses by augmenting its backward reasoning mechanism by the model-elimination reduction rule. In addition he employs unification with occurs check and consecutively bounded depth-first search for achieving a complete inference system.

We propose a different way of overcoming Prolog's limitations. We introduce a second type of rules in order to be able to represent those clauses that cannot be handled by Prolog. SATCHMO can be viewed as an interpreter for this kind of rules treating them as forward rules in view of generating a model of the clause set as a whole.

A crucial point with respect to the feasibility of the approach was the observation that range-restriction of clauses may be favorably exploited when reasoning forward. A clause is called *range-restricted* if every variable occurs in at least one negative literal. (The notion 'range-restricted' was first introduced in the context of logic databases in [NIC 79]). When reasoning forward, all negative literals are resolved first. For range-restricted clauses this leads to a complete instantiation of the remaining positive literals. Ground disjunctions can be split into their component literals as has been done, e.g., in early proof procedures based on tableaux calculus [SMU 68]. The case analysis-like style of treating non-Horn clauses introduced by splitting can lead to very elegant solutions as compared with the way they are handled by PTTP. Range-restriction in combination with forward reasoning overcomes Prolog's deficencies for Horn clauses as well: infinite generation due to recursive clauses can be prevented by a subsumption test for ground atoms that is very cheap compared to 'ancestor test' and bounded search. No occurs check is needed because at least one of two literals to be unified is always ground.

A major advantage of our solution is the fact that one can afford to implement the necessary additions very easily on top of Prolog itself, whereas the additional features for the PTTP have to be implemented by extending a Prolog system. This does not prevent our approach from being implemented on a lower level as well.

A principle drawback of the approach lies in the fact that clauses which are not range-restricted may require a full instantiation of certain variables over the whole Herbrand universe. In the presence of functions this may lead to the well-known combinatorial explosion of instances. Recursive problems with functions in particular will hardly be solvable efficiently in presence of non-range-restricted clauses. As arbitrary clause sets can be transformed into range-restricted form while preserving satisfiability, one can handle instantiation naturally within the framework of range-restriction.

There are, however, much more cases than one might expect where the limitations mentioned do not harm. The efficiency SATCHMO obtains in such cases is remarkable and surprising. In order to give evidence to this claim we devote a major part of the paper to reporting about our experience with a fairly huge collection of example problems taken from recent publications. The full Steamroller, e.g., has been solved in 0.3 secs.

In an earlier paper [MB 87] we have described our model generation approach on the basis of forward rules only. In this context, model generation can be explained and justified on the basis of hyperresolution. This paper is an informal one in the sense that we don't give proofs or formal definitions. Instead we thoroughly motivate our proposal and provide full Prolog code as a specification.

Throughout the paper we employ the Prolog style of notation: Variables are represented by uppercase letters, Boolean connectives and/or by means of ',' and ';' respectively. Only a very basic knowledge about Prolog is required; as an introduction refer, e.g., to chapter 14 in [WOS 84].

In order to represent the new kind of rules inside Prolog we assume that a binary operator '--->' has been declared. Clauses that are not represented as Prolog rules may then be represented in implicational form as (A_1, ... ,A_m ---> C_1; ... ;C_n) where ~A_1 to ~A_m are the negative, C_1 to C_n the positive literals in the clause. Completely positive clauses are written as (true ---> C_1; ... ;C_n), while completely negative clauses are implicationally represented by (A_1, ... ,A_m ---> false). Thus negation never occurs explicitly. We call the left-hand side of an implication its *antecedent* and the right-hand side its *consequent.*

2. Model Generation in Prolog

2.1 A basic procedure

It is well-known that every model of a set of clauses can be represented by a set M of positive ground atoms. The elements of M are those ground literals that are satisfied in the model, the remaining ones are - by default - assumed violated. A ground conjunction/disjunction is satisfied in M, if M contains all/some of its components. A clause (A ---> C) is satisfied in M, if $C\sigma$ is satisfied for every substitution σ such that $A\sigma$ is satisfied. Conversely, (A ---> C) is violated in M if there is a substitution σ such that $A\sigma$ is satisfied in M, but $C\sigma$ is not. 'True' is satisfied, 'false' is violated in every model.

It seems to be a very natural choice to implement model construction by asserting facts into Prolog's internal database and the test for satisfaction in a model by means of Prolog goal evalution over the "program" that consists of the facts asserted. Consider, e.g., a Prolog database containing

```
p(1).        q(1,2).
p(2).        q(2,1).
p(3).        q(2,2).
```

The clause (p(X),q(X,Y) ---> p(Y)) is satisfied in this database, because evaluation of the Prolog goal 'p(X),q(X,Y),not p(Y)' fails. On the other hand, the clause (q(X,Y),q(Y,Z) ---> q(X,Z)) is violated, as the goal 'q(X,Y),q(Y,Z),not q(X,Z)' succeeds and returns bindings X=1,Y=2,Z=1. These bindings represent a substitution such that the corresponding ground instance (q(1,2),q(2,1) ---> q(1,1)) is not satisfied in the current database.

Violated clauses can be applied as generation rules in order to further extend the database so far constructed. Initially, the database is empty and therefore clauses of the form (true ---> C) are the only ones that are violated. If there is a clause (A ---> C) and a substitution σ such that $A\sigma$ is satisfied and $C\sigma$ is violated, the clause can be satisfied by satisfying $C\sigma$, i.e., by

- asserting $C\sigma$, if $C\sigma$ is an atom
- creating a choice point, chosing a component atom of $C\sigma$ and adding it to the database, if $C\sigma$ is a disjunction.

Generation of 'false' indicates that the current database contradicts at least one of the completely negative clauses and thus cannot be extended into a model. One has to backtrack to a previously established choice point (if any) and to choose a different atom there (if any remains). All facts asserted between this choice point and the point where 'false' has been generated have to be retracted from the database on backtracking.

If all possible choices lead to a contradiction, the process terminates with an empty database and reports that a model could not be created. The clause set under consideration is unsatisfiable. If on the other hand a database has been constructed in which every clause is satisfied, this database represents a model of the clause set and satisfiability has been shown. In certain cases generation will never stop, because no finite database satisfies all clauses, but a contradiction does not arise either. This is due to the undecidability of satisfiability.

The model generation process outlined above can be implemented in Prolog by means of the following very simple program:

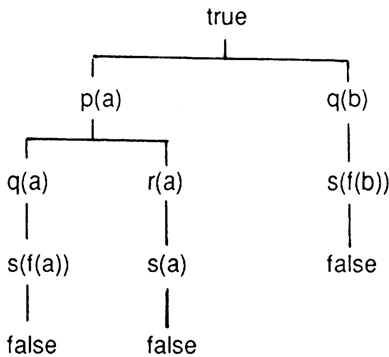
```
satisfiable :-
    is_violated(C), !,
    satisfy(C),
    satisfiable.
satisfiable.

is_violated(C) :-
    (A ---> C),
    A, not C.

on_backtracking(X).
on_backtracking(X) :-
    X, !, fail.

satisfy(C) :-
    component(X,C),
    asserta(X),
    on_backtracking(retract(X)),
    not false.

component(X,(Y;Z)) :-
    !, (X=Y ; component(X,Z)).
component(X,X).
```

Note that 'true' is a built-in Prolog predicate that always succeeds, whereas 'false' does not succeed unless asserted.

The following example is intended to illustrate model generation in Prolog. Consider the clause set S1:

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

The choices and assertions made during execution of 'satisfiable' can be recorded in form of a tree:



When executing 'satisfiable', this tree is traversed in a left-to-right, depth-first manner. Every possible path towards a model of S1 is closed because 'false' is finally generated. Thus, S1 has been shown unsatisfiable. If the clause (p(X),s(X) ---> false) were missing, 'false' could not be generated along the middle branch and the database constructed along this branch - consisting of p(a), r(a) and s(a) - would represent a model of the reduced clause set.

2.2 How to achieve soundness for unsatisfiability

If 'satisfiable' fails - when applied to a particular set S of clauses - this should always coincide with S being unsatisfiable. Conversely, if 'satisfiable' terminates successfully, S should in fact be satisfiable. While the latter is achieved, the former aim is not always reached: There are cases where 'satisfiable' fails although a model of S exists. This happens when a disjunction is generated that still contains uninstantiated variables shared by different components of the

disjunction. Consider, e.g, the following clause set S2:

```
true ---> p(X) ; q(X)
p(a) ---> false
q(b) ---> false
```

Initially 'p(X) ; q(X)' is generated. If 'p(X)' is asserted, the disjunction is satisfied, but 'false' will be generated in the next step. The same is the case if 'q(X)' is asserted. Thus, 'satisfiable' fails. However, the set {p(b),q(a)} represents a finite model of S2 which the program was unable to find.

Soundness for unsatisfiability can be guaranteed if all disjunctions generated are completely instantiated. This is the case iff all clauses are range-restricted, i.e., if every variable in the consequent of a clause occurs in its antecedent as well. In particular, completely positive clauses - those having 'true' as their antecedent - have to be variable-free in order to be range-restricted. The example set S1 given above is range-restricted, while S2 is not. Range-restriction requires that for every variable in a clause the subset of the universe over which the variable ranges is explicitly specified inside the clause. Variables implicitly assumed to range over the whole universe are not allowed. One can expect many clauses to be range-restricted if the problem domain is somehow naturally structured. This is in particular the case if a problem is (inherently) many-sorted.

If a set S contains clauses that are not range-restricted, S nevertheless can be transformed into a set S^* that is range-restricted and that is satisfiable iff S is so. For this purpose an auxiliary predicate 'dom' is introduced and the following transformations and additions are performed:

- every clause (true ---> C) that contains variables X_1 to X_n is transformed into (dom(X_1),...,dom(X_n) ---> C)
- every other clause (A ---> C) such that C contains variables Y_1 to Y_m not occuring in A is transformed into (A,dom(Y_1),...,dom(Y_m) ---> C).
- for every constant c occurring in S, a clause (true ---> dom(c)) is added; if S does not contain any constant a single clause (true ---> dom(a)) is added where 'a' is an artificial constant
- for every n-ary function symbol f occurring in S one adds a clause (dom(X_1),...,dom(X_n) ---> dom(f(X_1,..,X_n)))

The 'dom' literals added to non-range-restricted clauses explicitly provide for an instantiation of the respective variables over the Herbrand universe of S. The transformation of S into its range-restricted form S^* can be compared with the transformation of a formula into its Skolemized form: although the transformed set is not equivalent to the initial set in the strict sense, a kind of weak equivalence can be observed. If the relation assigned to 'dom' (the functions assigned to the Skolem function symbols, resp.) is removed from any model of the transformed set, a model of

the initial set is obtained. There is a one-to-one correspondence between the models of both sets of clauses up to the relation (functions, resp.) assigned to the additional predicate (function symbols, resp.). Therefore the transformation described preserves satisfiability. Transformation of S2 into range-restricted form yields S2*:

```
true ---> dom(a)          p(a) ---> false
true ---> dom(b)          q(b) ---> false
      dom(X) ---> p(X) ; q(X)
```

If 'satisfiable' is applied to S2* the program terminates successfully with the facts dom(a), dom(b), q(a) and p(b) in the database.

If applied to range-restricted clauses only, 'satisfiable' will be sound for unsatisfiability as well. For the rest of the paper we assume that all problems mentioned have been transformed into range-restricted form prior to checking them for satisfiability.

2.3 How to achieve refutation-completeness

As satisfiability is undecidable no theorem prover is able to successfully terminate for every satisfiable set of clauses. Unsatisfiability, however, is semi-decidable. Therefore, our program should terminate with failure for every unsatisfiable clause set. The following example set S3 is unsatisfiable:

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

When applied to S3, 'satisfiable' will generate an infinite sequence of p-atoms: p(a), p(f(a)), p(f(f(a))), ... etc. The example shows that 'satisfiable' is not complete for unsatisfiability!

This is due to the fact that the Prolog-like search strategy employed by the program is inherently unfair: at each recursive call of 'satisfiable' the database of problem clauses is searched from the top and the first violated clause found is tried to satisfy. There are problems - like S3 - where some violated clauses are never considered and thus a contradiction is never reached. Sometimes a proper ordering of clauses suffices for controlling the order in which facts are generated. In our example, however, ordering does not help!

Completeness can be achieved if clauses are generated systematically level by level. First, all atoms/disjunctions that can be generated from a given database are determined without modifying the database. Then all facts needed in order to satisfy these atoms/disjunctions are

asserted altogether. If S3 would have been treated this way, the following facts would have been asserted:

```
level 1:  p(a)
level 2:  p(f(a))      q(f(a))
level 3:  p(f(f(a)))   p(f(g(a)))   p(g(f(a))   p(g(g(a))   false
```

A contradiction would be detected on level 3 and the attempt to generate a model for S3 would fail. The following program implements model generation on a level-saturation basis:

```
satisfiable_level :- satisfiable_level(1).

satisfiable_level(L) :-
      is_violated_level(L), !,
      on_backtracking(clean_level(L)),
      satisfy_level(L),
      L1 is L+1,
      satisfiable_level(L1).
satisfiable_level(L).

satisfy_level(L) :-
      generated(L,C),
      not C, !,
      satisfy(C),
      satisfy_level(L).
satisfy_level(L).

is_violated_level(L) :-
      is_violated(C),
      not generated(L,C),
      assert(generated(L,C)),
      fail.
is_violated_level(L) :-
      generated(L,X).

clean_level(L) :-
      retract(generated(L,X)),
      fail.
clean_level(L).
```

The program works with an intermediate relation 'generated' used for storing atoms/disjunctions violated on a level in order to be able to satisfy them after the generation process for that level has been finished. Generation of a level can be efficiently organized by means of a backtracking loop, whereas levels have to be satisfied recursively. This is because the choice points created when satisfying a disjunction have to be kept "open" to backtracking.

Although 'satisfiable' - as opposed to 'satisfiable_level' - is not complete, most of the problems to be considered in the following section will in fact turn out to be solvable by 'satisfiable'. In cases where both programs are applicable one will usually observe that the former is much more efficient than the latter.

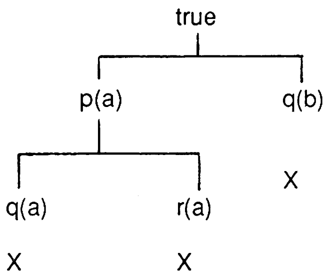
2.4 Enhancing efficiency through Prolog derivation rules

Prolog is known to be a powerful interpreter for Horn clauses. Some design decisions made for the sake of efficieny, however, prevent Prolog from being able to handle arbitrary Horn problems.

The missing occurs check requires to avoid certain unification patterns, the unbounded depth-first strategy prevents certain recursive problems from being tractable in Prolog. For the remaining cases Prolog's power can very well be exploited if those problem clauses that are in the scope of Prolog are represented as Prolog rules, i.e., by (C :- A) instead of (A ---> C). The four Horn clauses in example set S1, e.g., can all be treated this way:

```
true ---> p(a) ; q(b)          s(X) :- p(X)
p(X) ---> q(X) ; r(X)          s(f(X)) :- q(X)
                               false :- p(X) , s(X)
                               false :- q(X) , s(Y)
```

When applied to this representation of S1, 'satisfiable' will initially work as before, i.e., p(a) and q(a) are asserted into the database in order to satisfy the two '--->'-clauses. As soon as q(a) has been asserted, however, 'false' becomes derivable. The test 'not false' performed after assertion of q(a) fails and backtracking is immediately initiated: s(f(a)) and 'false' need no more be asserted in order to run into a contradiction. Similarly the two other branches can be cut earlier due to derivability of 'false'. The tree of facts asserted has become considerably smaller than before:



Crosses 'X' indicate that the respective branch has been closed because 'false' has become derivable.

If part of a clause set is directly represented in Prolog, 'satisfiable' can be applied without any change, provided inconsistency of the Prolog part of the problem has been tested before. This can be done by simply evaluating 'false' once before calling 'satisfiable'. The same applies to 'satisfiable_level'.

What has changed, however, is the way the model under construction is represented and satisfaction of clauses is determined. The problem clauses that have been directly represented in Prolog now serve as derivation rules. If the requirement for clauses to be range-restricted is respected, only ground literals will be derivable through these rules. Ground atoms that are required for satisfying the '--->'-clauses need not be explicitly asserted anymore, if they are

derivable from the already existing facts. Thus, the model under construction is no longer represented by explicitly stored facts alone, but by all facts derivable via those problem clauses that have been represented directly in Prolog. As the Prolog goal evaluation mechanism solves goals not only over facts but through rules as well, nothing has to be changed in the programs given.

In every clause set at least the positive ground units and the completely negative clauses can always be formulated directly in Prolog. There are even problems where all clauses can be represented this way. In this case satisfiability checking reduces to a single 'false' evaluation. When using Prolog rules for clause representation one nevertheless has to be very careful in order to avoid recursion and occurs check problems. (A set of clauses is recursive, if its connection graph contains a cycle that has a unifier.) Although a complete syntactic characterization of such cases is not easy, there are relatively simple sufficient conditions, like, e.g., to avoid literals with more than one occurrence of the same variable (static occurs check) in order to avoid dynamic occurs check problems. If in doubt, the option to represent a Horn clause as a generation rule always remains. Because of range-restriction this representation will never lead to any occurs check problem.

2.5 Further variations and optimizations

The basic model generation paradigm - as outlined above - may, of course, exhibit serious inefficiencies in special situations. However, one can easily incorporate several variations into the basic procedure that may lead to considerable optimizations in certain undesirable cases. In the following we will shortly discuss three such variations that are optionally available in SATCHMO. They are intended to speed up the search for violated clauses, or permit to derive contradictions earlier. Whereas the first - called clause-set compaction in [BUT 86] - will always result in some benefit, others will pay off only if applied to problems that in fact exhibit the inefficiency they are intended to cure. Otherwise these variations may even lead to some overhead compared with the basic procedures.

1. clause-set compaction:

   Search for violated clauses may be fairly expensive in cases where the clausal formulation of a problem is highly redundant, as is the case, e.g., with

```
p(X,Y) , q(Y,Z)  ---> h(X)
p(X,Y) , q(Y,Z)  ---> h(Y)
p(X,Y) , s(Y,Z)  ---> h(X)
p(X,Y) , s(Y,Z)  ---> h(Y)
```

   The p- and h-relations have to be searched four times, the q- and s-relations twice in

order to determine all instances of the clauses that are violated in a given database. If we would allow ';' to occur in the antecedent, and ',' to occur in the consequent of a rule as well, these four clauses could be compactified into the single rule

```
p(X,Y) , (q(Y,Z) ; s(Y,Z))  --->  h(X) , h(Y)
```

If this expression is tested for violation, the p-, q-, and s-relation are searched only once, the h-relation twice. In order to handle non-clausal generation rules as well, the following has to be added on top of 'satisfy':

```
satisfy((A,B))  :- !, satisfy(A), satisfy(B).
```

2. clause compilation:

When testing for contradictions by evaluating 'false', the test is performed globally over the whole database of facts. One does not take into consideration the specific fact asserted just before. As 'false' has not been derivable before this update, it can be derivable now only if the most-recently introduced literal is able to participate in a derivation of 'false'. Therefore it is possible to "focus" the contradiction test by precompiling the completely negative problem clauses into local test clauses. Consider, e.g., a set of clauses containing

```
false  :- p(X,Y) , q(Y,Z).
q(A,B)  :- s(B,A).
```

Precompilation would result in the following local test clauses being generated:

```
incompatible(p(X,Y))  :- q(Y,Z).        incompatible(q(Y,Z))  :- p(X,Y).
incompatible(p(X,Y))  :- s(Z,Y).        incompatible(s(Z,Y))  :- p(X,Y).
```

Once negative clauses have been compiled this way one can exploit them by slightly modifying the 'satisfy' predicate again: instead of 'assert(X),..,not false' one performs 'not incompatible(X),assert(X),..'. Thus, facts the assertion of which would directly lead to a contradiction are never asserted. Precompilation of the incompatibility rules can easily and efficiently be programmed in Prolog. A similar precompilation idea may be applied for the remaining clauses as well possibly speeding-up the search for violated clauses. We have reported about this in [BRY 87].

3. complement splitting:

When the assertion of an atom A_i chosen from a disjunction has resulted in a contradiction one may try to benefit from the information thus obtained while trying the

remaining components. This can be achieved by temporarily asserting the rule (false :- A_i). This way any attempt to re-assert A_i while trying to satisfy the respective disjunction are immediately blocked. The size of the search tree may be considerably cut down in case several occurrences of big subtrees are avoided this way. Exploiting information about attempts that have already failed for avoiding redundant work has already been suggested by Davis and Putnam in their early proof procedure. The feature can be implemented by modifying the 'component' predicate as follows:

```
component(X,(Y;Z)) :- !,
     (X=Y ;
      assert((false :- Y)),
      on_backtracking(retract((false :- Y))),
      component(X,Z)).
```

3. Experiences with SATCHMO

When reporting about experiences with a new method, one of course tends to start with "showcase" examples that are particularly well handled by the approach proposed. In our case, it turns out that several examples recently discussed in the literature are suitable for this purpose. Thus, we begin this section with discussing these examples in some detail. Then we shortly address combinatorial puzzles - a class of problems preferably taken for demonstrating the power of a theorem prover. The third and most comprehensive section will be devoted to Pelletier's "Seventy-five Problems for Testing Theorem Provers" [PEL 86]. As this collection covers a wide range of problem classes, we regard it as particularly well suited for exhibiting the potential power as well as the limits of the approach suggested.

All examples discussed have been run under interpreted CProlog Vers. 1.5 on a VAX 11/785. The solution times that will be given have been measured using the built-in predicate 'cputime'. The authors are very much aware of the fact that comparing theorem provers on the basis of cpu times is hardly ever able to do justice to the particular approaches except if all conditions are respected absolutely fairly. Reporting cpu times in this paper is not intended to compete with others, but to show which kind of examples are hard for model generation and which are easy. Moreover, we would like to demonstrate this way that theorem proving in Prolog is feasible and that the efficiency obtained when doing so may be remarkable.

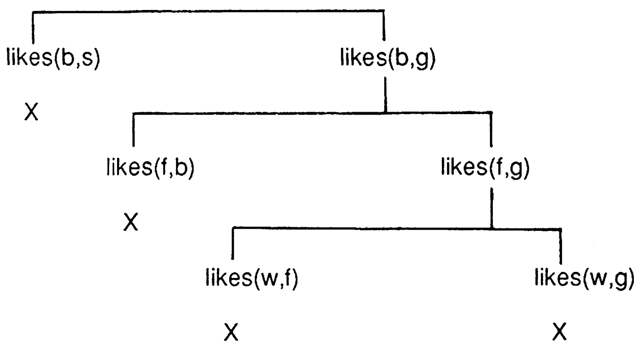
3.1 Schubert's Steamroller

The steamroller problem has been presented by Len Schubert nearly a decade ago. Mark Stickel's article "Schubert's Steamroller Problem: Formulations and Solutions" [STI 86] provides

an excellent overview of various attempts to solve this problem. We use the (unsorted) formulation that Stickel has proposed as a standard. As already mentioned in the introduction, this problem can be completely expressed in Prolog except for one non-Horn clause:

```
wolf(w).                                      animal(X) :- wolf(X).
fox(f).                                       animal(X) :- fox(X).
bird(b).                                      animal(X) :- bird(X).
snail(s).                                     animal(X) :- snail(X).
caterpillar(c).                               animal(X) :- caterpillar(X).
grain(g).                                     plant(X) :- grain(X).

smaller(X,Y) :- fox(X) , wolf(Y).             plant(i(X)) :- snail(X).
smaller(X,Y) :- bird(X) , fox(Y).             likes(X,i(X)) :- snail(X).
smaller(X,Y) :- snail(X) , bird(Y).           plant(h(X)) :- caterpillar(X).
smaller(X,Y) :- caterpillar(X) , bird(Y).     likes(X,h(X)) :- caterpillar(X).

likes(X,Y) :- bird(X) , caterpillar(Y).

false :- wolf(X) , (fox(Y) ; grain(Y)) , likes(X,Y).
false :- bird(X) , snail(Y) , likes(X,Y).
false :- animal(X) , animal(Y) , likes(X,Y) , grain(Z) , likes(Y,Z).

animal(X) , animal(Y) , smaller(Y,X) , plant(W) , likes(Y,W) , plant(Z) ---> likes(X,Y) ; likes(X,Z).
```

Satisfiability of the Prolog part of the problem formulation can be demonstrated within 0.05 secs by evaluation of the goal 'false'. Unsatisfiability of the whole problem is proved by means of 'satisfiable' after 0.3 secs. (The best time reported in Stickel's paper was 6 secs for the unsorted version of the problem.) The choices and assertions performed during execution are as follows:



The way 'satisfiable' solves the problem corresponds pretty well to the natural language solution given by Stickel.

3.2 Lewis Carroll's "Salt-and-Mustard problem"

This problem - which stems from Lewis Carroll's "Symbolic Logic" of 1897 - has been discussed in a publication by Lusk and Overbeek in the problem corner of the Journal of Automated Reasoning [LO 85]. It is about five friends who have imposed certain rules governing which condiment - salt or mustard - to take when the five are having beef together. The problem is to check whether these rules are compatible, i.e., satisfiable. The major part of the problem consists of if-and-only-if conditions for each friend and each condiment. Each of these can be expressed by means of a non-Horn generation rule and a Horn derivation rule. Our formulation is as follows (note that some 'false'-rules have been compactified):

```
friend(barry).                 salt(X) :- both(X).
friend(cole).                  mustard(X) :- both(X).
friend(dix).
friend(lang).                  salt(X) , mustard(X) ---> both(X).
friend(mill).

salt(barry)     :-  oneof(cole)    ;  oneof(lang).
mustard(barry)  :-  neither(dix)   ;  both(mill).
salt(cole)      :-  oneof(barry)   ;  neither(mill).
mustard(cole)   :-  both(dix)      ;  both(lang).
salt(dix)       :-  neither(barry) ;  both(cole).
mustard(dix)    :-  neither(lang)  ;  neither(mill).
salt(lang)      :-  oneof(barry)   ;  oneof(dix).
mustard(lang)   :-  neither(cole)  ;  neither(mill).
salt(mill)      :-  both(barry)    ;  both(lang).
mustard(mill)   :-  oneof(cole)    ;  oneof(dix).

salt(barry)     --->  oneof(cole)    ;  oneof(lang).
mustard(barry)  --->  neither(dix)   ;  both(mill).
salt(cole)      --->  oneof(barry)   ;  neither(mill).
mustard(cole)   --->  both(dix)      ;  both(lang).
salt(dix)       --->  neither(barry) ;  both(cole).
mustard(dix)    --->  neither(lang)  ;  neither(mill).
salt(lang)      --->  oneof(barry)   ;  oneof(dix).
mustard(lang)   --->  neither(cole)  ;  neither(mill).
salt(mill)      --->  both(barry)    ;  both(lang).
mustard(mill)   --->  oneof(cole)    ;  oneof(dix).

oneof(X)  ---> salt(X) ; mustard(X).               false  :- oneof(X) ,(both(X) ; neither(X))
friend(X) ---> both(X) ; neither(X) ; oneof(X).    false  :- oneof(X) , salt(X) , mustard(X).
                  false  :- neither(X) , (both(X) ; salt(X) ; mustard(X)).
```

The problem has a single model:

```
salt(barry)   mustard(barry)   neither(dix)    oneof(lang)    both(barry)
salt(mill)    mustard(lang)    neither(cole)   oneof(mill)
```

'Satisfiable' finds it within 1.1 secs. Lusk and Overbeek have judged this problem to be especially hard as their theorem prover has produced 32 000 clauses for solving the problem. Although the tree to be searched before a solution is found is considerably bigger than for the Steamroller the problem is still a simple one for a model generation approach.

The two other problems discussed in the article by Lusk and Overbeek are in fact much easier: "truthtellers and liars" are dismantled within 0.1 secs, while a model for the "schoolboys problem" is found after 0.2 secs.

3.3 A non-obvious problem

Pelletier and Rudnicki [PR 86] have recently discussed a problem that is simple to state, but hard to prove. Because of its brevity we give their problem formulation as well: "Suppose there are two relations, P and Q. P is transitive, and Q is both transitive and reflexive. Suppose further the 'squareness' of P and Q: any two things are either related in the P manner or in the Q manner. Prove that either P is total or Q is total."

Our formalization of the problem - requiring a transformation into range-restricted form - is as follows:

```
dom(a).        p(X,Y) , p(Y,Z)  ---> p(X,Z).
dom(b).        q(X,Y) , q(Y,Z)  ---> q(X,Z).
dom(c).        q(X,Y)  ---> q(Y,X).
dom(d).        dom(X),dom(Y)  ---> p(X,Y) ; q(X,Y).
               false  :-  p(a,b).
               false  :-  q(c,d).
```

The tree to be searched by 'satisfiable' is already quite big: 348 facts are asserted and subsequently retracted again, 52 choice points are established and 53 branches are closed. The theorem is proved within 16 secs.

Pelletier/Rudnicki report about a solution time just under 2 mins, McCune reports just under 1 min [McC 86]. Just recently Walther has obtained 31.1 secs [WAL 88]. If complement splitting is applied for this example, the search tree can be considerably reduced such that the solution time goes down to 9.5 secs.

3.4 Some combinatorial puzzles

In Pelletier's collection, to be discussed below, there is another problem (number 55) posed by Len Schubert. This problem is a simple combinatorial puzzle where the murderer of aunt Agatha has to be found. Equality (or identity) is used in order to express that the murderer has to be found among the three people living in the house. We conjecture that the following formulation of the problem (not using '=') is simpler and more natural than Pelletier's:

```
lives(agatha).                false :- killed(X,Y) , richer(X,Y).
lives(butler).                false :- hates(agatha,X) , hates(charles,X).
lives(charles).               false :- hates(X,agatha) , hates(X,butler) , hates(X,charles)

hates(agatha,agatha).         hates(X,Y) :- killed(X,Y).
hates(agatha,charles).        hates(butler,X) :- hates(agatha,X).

true ---> killed(agatha,agatha) ; killed(butler,agatha) ; killed(charles,agatha).
lives(X) ---> richer(X,agatha) ; hates(butler,X).
```

After 0.05 secs a model for the problem is found indicating that aunt Agatha has killed herself. After 0.1 secs all other possibilities have been ruled out proving that Agatha indeed must have committed suicide.

Using this style of formulation, i.e., expressing the different choices that are existing over the finite domain of the puzzle by means of disjunctions and expressing the remaining conditions directly in Prolog, one can solve quite a lot of similar puzzles efficiently. We just would like to mention as examples

- Lewis Carroll's "lion-and-unicorn" puzzle (discussed recently in the Journal of Automated Reasoning [OHL 85]): 20 Prolog clauses, 1 non-Horn clause - solved in 0.1 secs

or a more substantial one:

- the full "jobs" puzzle taken from [WOS 84]: 31 Prolog clauses, 2 non-Horn clauses - solved in 4.5 secs

Again the incomplete program 'satisfiable' has been sufficient.

3.5 Pelletier's seventy-five problems

All problems discussed in the following have been solved with 'satisfiable' as well, unless stated differently.

The propositional problems 1-17 are all very easy once clausal form has been obtained. Eight of them can be completely represented in Prolog and are all solved under 0.01 secs. Problem 12 is the hardest among the remaining ones - a solution requires 0.15 secs.

The monadic problems 18-33 (34 has been omitted because the authors did not want to perform the "exercise" of computing 1600 clauses) are simple as well, with one exception namely problem 29. This problem - consisting of 32 clauses - requires 33 secs! (Attention: Pelletier's clausal version contains two typing mistakes.) A clause-set compaction as described above results in 23 compactified clauses and unsatisfiability of the compactified set can be shown within 4.3 secs. If in addition complement splitting is applied, the time needed goes down to 1.1 secs.

Problems 19,20,27,28, and 32 are completely expressable in Prolog and solvable in less than 0.02 secs (problem 28 is satisfiable!).

The full predicate logic problems without identity and (non-Skolem) functions 35-47 do not impose particular problems either. Problem 44 is the only one that can be completely represented in Prolog (solved under 0.01 secs). Problem 35 is the first problem for which 'satisfiable' would run forever. 'Satisfiable_level' solves it within 0.07 secs. In problem 46 the clause 'f(X) ---> f(f(X)) ; g(X)' has to be "hidden" at the end of the clause list in order to maintain applicability of 'satisfiable'. The most difficult problem in this section is problem 43, requiring 0.65 secs. Problem 47 is the Steamroller discussed earlier.

Among the remaining problems 48-69 there are nine functional problems without identity. Problems 57,59 and 60 (Pelletier's faulty clausal form corrected) can be solved by 'satisfiable' under 0.1 secs, problem 50 requires 0.55 secs (0.28 with complement splitting).

For problem 62 once again the clausal form given by Pelletier does not correspond to the non-clausal form of the theorem. If corrected the clausal form can immediately be shown satisfiable as no completely positive clauses exist.

Problems 66-69 cannot be solved by any of the two SATCHMO programs! These problems are variations of a hard recursive Horn-problem with functions. There is a single predicate ranging over the whole domain. As the problems are not range-restricted instantiation over the Herbrand universe has to be provided through the 'dom'-predicate. Consider, e.g., problem 66:

```
t(i(X),i(Y,X)) :- dom(X) , dom(Y).
t(i(i(X,i(Y,Z)),i(i(X,Y),i(X,Z))) :- dom(X) , dom(Y) , dom(Z).
t(i(i(n(X),n(Y)),i(Y,X)) :- dom(X) , dom(Y).
t(i(X,Y)),t(X) ---> t(Y)
false :- t(i(a,n(n(a)))).

dom(X) ---> dom(n(X)).
dom(X) , dom(Y) ---> dom(i(X,Y)).
```

Satchmo fails because the generation of new Herbrand terms via the two 'dom'-rules interferes with the generation of the necessary 't'-facts. The number of 'dom'-facts generated explodes and the comparatively few 't'-facts that can be generated on each level are "buried" by them. The only

way towards possibly solving problems of this kind seems to be a careful control of Herbrand term generation: 'dom'-rules should not be applied before the other rules have not been exhaustived. As such a control feature has not yet been implemented, we do not further elaborate on this point.

Prolog's implementation of '=' cannot be used for correctly representing logical identity (except in very restricted cases). In order to represent the remaining problems with identity there are two possibilities:

1. to introduce a special equality predicate and to add the necessary equality axioms (transitivity, substitutivity etc.): this has been done for problems 48,49,51-54,56, and 58
2. to recode the problems without explicitly using identity as done in the original formulation of the three group theory problems 63-65 by Wos; problem 61 has been çoded this way too

Of the problems thus augmented or recoded, 'satisfiable' was able to solve problems 48, 49, 61, 64, and 65 in less than 1 sec each. For the remaining problems 'satisfiable_level' had to be employed: of these, problem 58 was solved in 0.15 secs and problem 63 in 0.75 secs; problem 55 has been discussed above. Problems 51-54 and 56 could not be solved by either programs, due to deficiencies very similar to those responsible for failure in case of 66-69.

The last section in Pelletier's collection provides problems for studying the complexity of a proof system. The following figures are given without further comment as we have not really studied their relevance yet. Pigeonhole problems (72,73):

n	1	2	3	4	5
secs	0.05	0.13	0.7	3.8	25.5 ...

Times are given for our formulation of the predicate logic version (73): the times clearly indicate exponential growth. The expository arbitrary graph problem 74 is solved in 0.18 secs.

For U-problems (71) coded as arbitrary graph problems (75) growth seems to be at most cubic:

n	1	2	3	4
secs	0.05	0.2	0.75	2.15 ...

4. Conclusion

In this paper SATCHMO, a theorem prover based on model generation, is presented and experiences are described. Prolog has been used as a representation language for expressing problem clauses as well as for the implementation of SATCHMO. The approach extends Prolog while retaining its efficiency for Horn clauses as has been done by Stickel's PTTP. The additions we are proposing are, however, considerably different from Stickel's. As a consequence, SATCHMO can be implemented on top of Prolog without causing too severe inefficiencies by doing so.

As an extension of the work reported here, we would like to investigate more deeply how to benefit from further compilations of problem clauses and how to control term generation. Some considerable gain in efficiency can also be expected from investigations in more sophisticated solutions to controlling recursive Prolog-rules. Apart from this, we would like to know how SATCHMO behaves when implemented in up-to-date Prolog-systems. The simplicity of its code should make it extremely portable. Due to the splitting feature especially forthcoming parallel implementations of Prolog should be promising.

5. Acknowledgement

We would like to thank Hervé Gallaire and Jean-Marie Nicolas as well as our colleagues at ECRC for providing us with a very stimulating research ambience. The work reported in this article has benefited a lot from it. We also would like to thank the Argonne team for their interest in our work and their encouragement to present our results to the automated deduction community.

References:

[BRY 87] Bry, F. et. al., *A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*, ECRC Techn. Rep. KB-16, 1987 (to appear in Proc. Int. Conf. Extending Database Technology EDBT 88)

[BUT 86] Butler, R. et. al., *Paths to High-Performance Automated Theorem Proving*, Proc. 8th CADE 1986, Oxford, 1986, 588-597

[LO 85] Lusk, E. and Overbeek, R., *Non-Horn problems*, Journal of Automated Reasoning 1 (1985), 103-114

[MB 87] Manthey, R. and Bry, F., *A hyperresolution-based proof procedure and its implementation in PROLOG*, Proc. of GWAI-87 (11th German Workshop on Artificial Intelligence), Geseke, 1987, 221-230

[McC 86] McCune, B., *A Proof of a Non-Obvious Theorem*, AAR Newsletter No. 7, 1986, 5

[NIC 79] Nicolas, J.M., *Logic for improving integrity checking in relational databases*, Tech. Rep., ONERA-CERT, Toulouse, Feb. 1979 (also in Acta Informatica 18,3, Dec. 1982)

[OHL 85] Ohlbach, H.J. and Schmidt-Schauss, M., *The Lion and the Unicorn*, J. of Automated Reasoning 1 (1985), 327-332

[PEL 86] Pelletier, F.J., *Seventy-five Problems for Testing Automatic Theorem*

Provers, J. of Automated Reasoning 2 (1986), 191-216

[PR 86] Pelletier, F.J. and Rudnicki, P., *Non-Obviousness*, AAR Newsletter No. 6, 1986, 4-5

[SMU 68] Smullyan, R., *First-Order Logic,* Springer-Verlag, 1968

[STI 84] Stickel, M., *A Prolog Technology Theorem Prover*, New Generation Computing 2, (1984), 371-383

[STI 86] Stickel, M., *Schubert's steamroller problem: formulations and solutions,* J. of Automated Reasoning 2 (1986), 89-101

[WAL 84] Walther, C., *A mechanical solution of Schubert's steamroller by many-sorted resolution*, Proc. of AAAI-84, Austin, Texas, 1984, 330-334 *(Revised version in Artificial Intelligence, 26, 1985, 217-224)*

[WAL 88] Walther, C., *An Obvious Solution for a Non-Obvious Problem*, AAR Newsletter No. 9, Jan. 1988, 4-5

[WOS 84] Wos, L. et. al., *Automated Reasoning*, Prentice Hall, 1984