SEM: a System for Enumerating Models*

Jian Zhang* and Hantao Zhang
{jzhang, hzhang}@cs.uiowa.edu
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
U. S. A.

Abstract

Model generation can be regarded as a special case of the Constraint Satisfaction Problem (CSP). It has many applications in AI, computer science and mathematics. In this paper, we describe SEM, a System for Enumerating finite Models of first-order many-sorted theories. To the best of our knowledge, SEM outperforms any other finite model generation system on many test problems. The high performance of SEM relies on the following two techniques: (a) an efficient implementation of constraint propagation which requires little dynamic allocation of storage; (b) a powerful heuristic which eliminates many isomorphic partial models during the search. We will present the basic algorithm of SEM along with these two techniques. Our experimental results show that general purpose finite model generators are indeed useful in many applications.

1 Introduction

A large number of problems in AI and computer science can be viewed as special cases of the constraint-satisfaction problem (CSP). Some examples are machine vision, belief maintenance, scheduling, temporal reasoning, graph problems, and the satisfiability problem. In this paper, we are interested in one special case of CSPs, i.e., model generation, where the constraints are expressed in predicate logic.

By *model generation* we mean, given a set of axioms (which are first order formulas), find their models automatically. A model is an interpretation of the function and predicate symbols over some domain, which satisfies all the axioms. The scope of this paper is restricted to finite models, whose domains are finite sets. Model generation is very important to the automation of reasoning. For example, the existence of a model implies the consistency of a theory. A suitable model can also serve as a counterexample which shows some conjecture does not follow from some premises. In this sense, model generation is complementary to classical theorem proving. Finite models help people understand a theory, and they can also guide conventional theorem provers in finding proofs. In recent years, several programs have been developed for generating finite models, such as FALCON [Zhang, 1994a] FINDER [Slaney, 1993], MGTP-G [Fujita *et al,* 1993], LDPP, SATO [Zhang and Stickel, 1994], ModGcn [Kim and Zhang, 1994], and MACE [McCune, 1994b]. They have been used to solve a large number of open questions in mathematics (see for example, [Fujita *et a*l., 1993; Slaney *et a*l., 1995; Zhang and Stickel, 1994; Zhang, 1994b]).

Theoretically speaking, any approach of constraint satisfaction can be used for finite model generation. For instance, a simple backtracking algorithm can always find a finite model (if it exists). Of course, a brute-force search procedure is too inefficient to be of any practical use. In general, the performances of backtracking algorithms can be improved in a number of ways such as forward checking and lookahead. There are many other search procedures and heuristics proposed in the AI literature; see [Kumar, 1992] for a good survey. However, in the special case of model generation, we have to address the following two issues:

- How can we implement constraint propagation and consistency checking efficiently? In particular, what data structures should be used?
- How can isomorphism be eliminated effectively during the search? Two models are isomorphic if one can be obtained from the other by permuting element names.

The first issue may be trivial for some constraint satisfaction algorithms because the constraints they accept are often assumed to be unary or binary. It is true that n-ary constraints can be converted into an equivalent set of binary constraints; but this conversion usually entails the introduction of new variables and constraints, and hence an increase in problem size. This issue is particularly important to model generation because in this case, the constraints are represented by complicated formulas. Experience tells us that a careful implementation can improve the performance of a program by several orders of magnitude. For instance, SATO [Zhang and Stickel, 1994] is a careful implementation of the Davis-Putnam algorithm, a well-known method for satisfiability testing

* Partially supported by the National Science Foundation under Grants CCR-9202838 and CCR-9357851.

'On leave from Chinese Academy of Sciences (Beijing).

in propositional logic. Because of its efficiency, SATO has been used by the second author to solve several dozens of open cases of quasigroup problems that can hardly be solved by other constraint satisfaction systems.

The second issue is obviously very important, because one model may be represented in many ways, which result in much redundancy in the search space. For propositional satisfiability, isomorphism elimination has been studied via the symmetric property of variables [Benhamou and Sais, 1994]. In the study of quasigroup identity problems, a key technique used in [Fujita *et* al., 1993; Slaney *et al,* 1995] is to impose extra constraints as part of the input, so that some isomorphic models need not be considered. It appears that the method implemented in FALCON [Zhang, 1994a] is better for handling isomorphism. It works dynamically and does not require extra constraints. FALCON has been used by the first author to solve a variety of open problems in abstract algebra.

In this paper, we describe a new model generating tool called SEM (a System for Enumerating Models) that combines the strengths of FALCON and SATO. Like FALCON, SEM is based on first-order ground clauses, and uses the powerful isomorphism eliminating technique. It also incorporates some data structures and inference mechanisms which are similar to those of SATO. As expected, SEM outperforms any other finite model generator that we know. SEM not only reproduced many results obtained by SATO and FALCON, but also solved several new problems. In particular, SEM works very well when the axiom set contains long formulas — the propositional satisfiability based systems like LDPP, SATO [Zhang and Stickel, 1994], ModGen [Kim and Zhang, 1994] and MACE [McCune, 1994b] cannot handle these cases because it is too expensive to convert such formulas into propositional clauses.

In the next section, we present some basic concepts, the abstract algorithm for finding finite models, as well as some features of SEM. In Section 3, we compare SEM with other similar systems on various test problems, and illustrate the applications of SEM with several examples. We hope that these experiments will be used as a basis for further comparison between finite model generators.

2 Search for Finite Models

2.1 Basic Concepts

SEM accepts problem specifications in many-sorted first-order logic. But only finite sorts are allowed and formulas should be in clause form. Let *(S, F)* be a *signature* where *S* is a set of *sorts* and *F* is a set *of function symbols*. We shall use the letter *s* (or s,) to denote a sort, and / to denote a function symbol. Each sort *s E S* consists of a finite number of elements, which may be designated by a list of names. Alternatively, one can simply give a positive integer (say *n)* as the cardinality of s, and the elements of s are assumed to be 0, 1, ..., *n* — 1. In the sequel, an element of some sort will be denoted by *e* or ei. Syntactically these elements may be regarded as constant symbols. The built-in sort BOOL has two elements: TRUE and FALSE.

Function symbols are sorted. Each *f E F* has some fixed arity *k* > 0 and is specified by / : *s1* ... *sk* —► s, where s, *s1*,..., *sk* £ *S*. The special predicate 'equality' is specified by EQ: *s s* —► BOOL, where *s* is an arbitrary sort. We assume that, for any element e, EQ(e,e) evaluates to TRUE; and for any two distinct elements ei,-Cj *(i not= j)* of the same sort, EQ(ej,ej) evaluates to FALSE. There can be other predefined functions and predicates, but for the sake of brevity, they will be neglected in this paper.

Terms are built from (sorted) variables and function symbols in the usual way. For convenience, let us call a term of the form /(e1,..., ek) (ei, E SI) *cell terms* or simply *cells.* (Intuitively, such a term corresponds to an entry or a cell in the "multiplication table" of the function /.) We shall denote a cell term by *ce* or cei;. Assigning an element of some appropriate sort to each cell, we obtain an *interpretation* of the function symbols.

The specification of a problem consists of a set of clauses. Each clause is a disjunction of literals, and a literal is a term of the sort BOOL or its negation. All the variables in the specification are universally quantified. Our goal is to find an interpretation which makes all the clauses true, i.e., to find a (finite) model of the clauses.

2.2 Model Generation as CSP

A constraint satisfaction problem (CSP) consists of a set of variables, { *Vi* I 1 < *i* < *m* }; a domain of values, Di, for each variable Vi; and a collection of constraints. A solution to a CSP is an assignment of values to all the variables such that no constraint is violated. Finite model generation may be considered as a kind of CSP where the "variables" are the cells in the multiplication tables and the "constraints" are specified by the clauses.

The model finding process can be described by the recursive procedure in Figure 1. The procedure uses the following parameters:

- *A:* assignments (cells and their assigned values), {(ce,e) | c E *sort{ce)}*\
- *B:* unassigned cells and their possible values, *{(ce,D)* | DC *sort(ce)};*
- *C:* constraints (i.e. the clauses).

Initially *A* is empty, and *B* contains all the cells: {(ce,D) | *D = sort(ce)}.* Since it is relatively expensive and inconvenient to check the consistency of a set of clauses containing free variables, in the implementation of SEM, before the first call of the procedure search, the clauses are instantiated once for all, with each variable replaced by each element of the same sort.

The procedure propa(A, *B, C)* propagates assignment *A* in *C:* it simplifies *C* and may force some variables in *B* to be assigned. In the next subsection, we give more details about this procedure.

2.3 Constraint Propagation

The procedure propa(A, *B, C)* is essentially a closure operation (with respect to a set of sound inference rules). It repeatedly modifies *A, B,* and *C* until no further changes can be made. When it exits, it returns the modified triple *(A,B,C).* The basic steps of this procedure can be described as follows.

```
proc search(A, B, C)
{
   if B = ∅ then /* a model is found */
        { print(A); return; }
   choose and delete (ce_i, D_i) from B;
   if D_i = ∅ then return; /* no model */
   for e ∈ D_i do
   {
      (A', B', C') := propa(A ∪ {(ce_i, e)}, B, C);
      if C' = false
        then continue; /* contradiction */
        else  search(A', B', C');
   }
}
```

Figure 1: The abstract search procedure

1. For each assignment (ce, e) in A, replace the occurrence of ce in C by e.
2. If there exists an empty clause in C (i.e., each of its literals becomes **FALSE** during the propagation), replace C by **FALSE**, and exit from the procedure. Otherwise, for every unit clause in C (i.e. all but one of its literals become **FALSE**), examine the remaining literal l.
   - If l is a cell term ce, and $(ce, D) \in B$, then delete (ce, D) from B and add $(ce, \mathbf{TRUE})$ to A; similarly, if l is the negation of a cell term in B, i.e. $\neg ce$, delete (ce, D) from B and add $(ce, \mathbf{FALSE})$ to A.
   - If l is of the form $\mathbf{EQ}(ce, e)$ (or $\mathbf{EQ}(e, ce)$), and $(ce, D) \in B$, delete (ce, D) from B and add (ce, e) to A; similarly, if l is of the form $\neg\mathbf{EQ}(ce, e)$ (or $\neg\mathbf{EQ}(e, ce)$), and $(ce, D) \in B$, then delete e from D.
3. For each pair $(ce, D) \in B$, if $D = \emptyset$, then replace C by **FALSE**, and exit from the procedure; if $D = \{e\}$ (i.e. $|D| = 1$), then delete the pair (ce, D) from B, and add the assignment (ce, e) to A.

Because the data *(A,B,C)* will be used in many recursive calls, it would be much simpler to implement the procedure search if we could save a copy of *(A,B,C)* before modifying them in each recursive call. However, this would require a lot of storage. Instead, we record in a stack all the modifications done to *(A,B,C)* in each call and undo them when backtracking.

In our implementation, a term in a clause is represented by a tree in the usual way. A value is associated with each leaf node (called *value node)* and a function symbol is associated with each internal node. An internal node has a counter to record the number of its children nodes that are internal nodes; if this number is zero, this node corresponds to a cell, called a *cell node.*

Associated with each cell, there is a list of cell nodes that represent all the occurrences of that cell in the constraints *C.* When a cell is assigned a value, we simply mark each cell node in the list as a value node. If a cell node in the list has a parent node, we decrease the parent's counter of internal children nodes by one. When that number becomes zero, the parent node is inserted into the cell node list of the corresponding cell. This kind of modifications on tree nodes is stored in a stack so that we can undo these modifications when backtracking.

Similarly, a clause has a counter to record the number of uninterpreted literals of the clause. When a literal becomes false, this number is decreased by one. When a literal becomes true, this clause is marked as "inactive". This technique of counter-manipulating has been used in LDPP [Zhang and Stickel, 1994] for propositional clauses. Because of space limitation, we shall not give more details about the implementation.

2.4 The Least Number Heuristic

In the procedure **search**, there are several places where we can use heuristics to improve its efficiency. For example, how to choose a variable (ce_i, D_i) from B for the next assignment; and, when (ce_i, D_i) is chosen, in what order should the values in D_i be assigned to ce_i. These issues have been studied extensively in the AI literature. They have also been considered in model generation programs like FINDER [Slaney, 1993] and MGTP [Fujita *et al.*, 1993]. In this paper, we focus on another kind of heuristics, i.e., how to reduce the number of values in D_i. If we can delete some values from D_i (without losing the completeness), then there will be fewer recursive calls to **search** and the search space will become smaller.

In a typical model generation problem, there can be many isomorphic models. When two values e and e' are symmetric in the input constraints, given one model M, we can obtain another model M' which is isomorphic to M, by exchanging e with e'. In an exhaustive search procedure like **search**, it is too expensive to enumerate all the possible models. Thus isomorphism rejection is crucial to the efficiency of finite model generation.

FALCON uses the so-called least number heuristic (LNH) in the search to avoid exploring isomorphic subspaces [Zhang, 1994a]. It is based on the observation that, at the early stages of the search, many element names (which have not yet been used in the search) are essentially the same. Thus when choosing a value for a new cell, we need not consider all the elements of a given sort. It makes no difference to assign the element e or e' to the cell, if neither of them has been assigned to some cell. To make this concise, we say that two elements e and e' are *symmetric* with respect to a set of ground clauses if the clause set remains the same when e and e' are interchanged.

Theorem. The procedure $\mathbf{search}(A, \{(ce_i, D_i)\} \cup B, C)$ can find a model iff $\mathbf{search}(A, \{(ce_i, Nonsymm(D_i, C))\} \cup B, C)$ can find one, where $Nonsymm(D_i, C)$ is obtained from D_i such that if two values in D_i are symmetric with respect to C, then one of the two values is deleted from D_i.

In general, it is expensive to repeatedly check symmetry of elements. But when the input clauses (before the instantiation of free variables) do not use the element names (i.e., 0, 1, ...) or their properties (e.g., $0 < 1$), then any two elements are symmetric with respect to the ground clauses obtained by instantiating the input clauses with element names. In this case, we can show

that in search$(A, \{(ce_i, D_i)\} \cup B, C)$, any two values in *Di* that are not used In *A* are symmetric with respect to *C*. We can simply delete all the values from Di, that do not appear in *A*, except one (the smallest one in our implementation). This technique of exploiting partially the symmetry of values is called *the least number heuristic* (LNH). As we will see in the next section, it is very effective in reducing isomorphic subspaces.

2.5 The System

SEM was implemented in C. The YACC tool is used for parsing. The current version has about 4000 lines of source code. The major algorithm is essentially the same as search, except that, for the sake of efficiency, we use an iterative procedure instead of a recursive one.

In SEM, the input specification for a problem consists of the following four parts:

Sorts; Functions; Variables; Clauses.

As an example, we give the specification for the pigeonhole problem.

Example Pigeonhole problem (10 pigeons, 9 holes). (The character 7,' starts a comment in a line.)

```
% Sorts
( pigeon [10] )
( hole [9] )

% Functions
{ h  : pigeon -> hole }
{ in : pigeon hole -> BOOL }

% Variables
< x, y : pigeon >
< z : hole >

% Clauses.  '-' is 'not'; '|' is 'or'.
[ in(x,h(x)) ]
[ -in(x,z) | -in(y,z) | EQ(x,y) ]
```

3 Experimental Results

In this section, we describe some of our experiments with SEM, and illustrate the applications of finite model generation with various examples. We compare SEM with other similar systems like FALCON-2 [Zhang, 1994a], FINDER [Slaney, 1993] (version 3.0), SATO 2 [Zhang and Stickel, 1994] and MACE [McCune, 1994b] (version 1.0.0). (All these systems were implemented in C) Since different specifications may result in different execution times, we use the sample input files provided by the designers of the systems, whenever possible. For the same problem, we use the same set of axioms. All of our experiments were carried out on a Sparc 2 workstation. The following tables show the execution times (in seconds) of the programs on various problems. Each row corresponds to one problem instance. Its first column is of the form *name.n* or simply n, where *name* is the name of a problem, and *n* is the size of the model. In all but the first table, we also give the number of models (denoted by m).

n	BS	SEM	
		create	search
14	4.80	0.21	0.01
16	7.73	0.28	0.02
18	13.50	0.42	0.02
20	22.36	0.56	0.03
22	37.11	0.79	0.03
24	55.58	1.00	0.03
26	96.00	1.30	0.06
28	122.00	1.58	0.06
30	184.00	2.03	0.08

Table 1: Pigeonhole Problems.

3.1 The Pigeonhole Problem

As a simple example of many-sorted applications, let us consider the pigeonhole problem, which has been included as a benchmark in most of the aforementioned systems. When not exploiting symmetry of the clauses, these systems can handle at most 11 pigeons in a reasonable amount of time (say 10 minutes).

Benhamou and Sais [Benhamou and Sais, 1994] proposed a method for exploiting symmetry in the propositional case. In the first-order case, we can use the least number heuristic to solve the problem more quickly. Table 1 gives the execution times of the two programs (n pigeons and *n*- 1 holes). The SEM specification for this problem was given in §2.5. Benhamou and Sais' algorithm (BS, in short) was implemented in Pascal on a SUN4/110 and the run times are taken from [Benhamou and Sais, 1994].

3.2 Quasigroups and Latin Squares

Recently a large number of open cases of quasigroup identity problems have been solved by various programs [Zhang, 1991; Fujita *et* a/., 1993; Slaney *et* al., 1995; Zhang and Stickel, 1994; McCune, 1994b]. A quasigroup has only one binary function, denoted by . or juxtaposition. In the multiplication table of this function, each row and each column is a permutation of all the elements.

In this subsection, we compare SEM with FINDER and SATO on the following 3 problems:

1. IQG: enumerate idempotent quasigroups, i.e. quasigroups satisfying the identity *xx — x*.
2. QG5: find idempotent quasigroups satisfying the identity *{yx.y)y = x*. For the sake of efficiency, we also use two additional equations, i.e. *y(xy.y) = x* and *(y.xy)y = x*.
3. RLS: enumerate reduced latin squares. A Latin square is called *reduced* if in the first row and column its elements occur in natural order. Thus a reduced latin square corresponds to a quasigroup satisfying the identities: *xO = x* and *Ox = x*.

For the first two problems, each program uses a different method for handling isomorphism. SATO uses the last-column constraint [Fujita *et* al., 1993; Slaney *et* al., 1995], FINDER introduces the 'cycle' function in the specification [Slaney, 1993], and SEM relies on the LNH.

	FINDER		SATO		SEM	
	m	t	m	t	m	t
IQG.4	1	0.03	1	0.02	1	0.02
5	10	0.05	10	0.03	8	0.02
6	488	1.17	728	0.63	448	0.37
QG5.6	0	0.27	0	0.13	0	0.04
7	3	0.72	3	0.21	1	0.12
8	1	1.33	1	0.43	1	0.17
9	0	2.53	0	0.71	0	0.25
10	0	6.95	0	1.32	0	0.50
11	5	20.72	5	2.46	1	1.22
12	0	82.50	0	6.82	0	5.28
RLS.4	4	0.02	4	0.05	4	0.03
5	56	0.12	56	0.08	56	0.06
6	9408	16.47	9408	7.38	9408	4.44

Table 2: Quasigroup problems

	m	FINDER	MACE	SEM
NCG.4	0	0.07	0.34	0.04
5	0	0.52	1.30	0.08
6	18	12.75	4.84	0.21
7	0	543.17	12.84	1.81
8	480	> 1000	200.75	22.95
9	0		*	318.44
RU.4	6	0.45	3.69	0.09
5	6	6.02	18.37	0.32
6	24	> 1000	81.20	2.41
7	120		*	33.10
BA.4	1	0.08	3.47	0.05
5	0	1.07	17.83	0.07
6	0	> 1000	*	0.30
7	0			1.41
8	120			15.61

Table 3: Equational problems (no heuristics)

For the last problem, no isomorphism-rejecting heuristic is used by any program.

From Table 2, one can see that the LNH is the most powerful heuristic for eliminating isomorphism. When this heuristic is used, fewer (or the same number of) models are generated. For a fixed size, all the models are essentially the same. Take QG5.7 for example, each of the 3 models produced by the other two programs is isomorphic to the one found by SEM, under certain permutations of the elements.

3.3 Some Equational Problems

Tables 3 and 4 give some data comparing the performances of several systems on generating some finite algebras: non-commutative groups (NCG), rings with unit (RU), and Boolean algebras (RA). No isomorphism eliminating heuristic is used by the programs in Table 3. A V indicates that MACE runs out of memory.

From these tables, one can see that isomorphism rejection is very important. Without such mechanisms, it is very difficult to generate moderately large algebras. The use of LNH enables FALCON and SEM to complete the search very quickly. It can also be seen that constraint propagation in SEM is very efficient. In many cases, SEM is about 10 times faster than FALCON.

3.4 Comparison

Programs like ModGen and MACE are based on decision procedures for the propositional logic. They perform very well on quasigroup problems, since the problems can be represented concisely in the propositional logic. But in general, it seems inappropriate to use them to find finite models of first-order theories because too many propositional variables and clauses are needed to represent the problems. On the other hand, FINDER, FALCON and SEM are based on (ground) first-order clauses and can deal with more complicated axiom sets. However, FINDER does not have any built-in mechanism for isomorphism rejection. In contrast, the LNH is quite general and powerful. Both FALCON and SEM rely on this heuristic to reduce the search space. The underlying algorithm of the two programs are virtually the same. However, FALCON was designed mainly for

	m	FALCON	SEM
NCG.4	0	0.05	0.04
5	0	0.17	0.06
6	3	0.48	0.09
7	0	1.30	0.10
8	4	3.37	0.30
9	0	7.70	0.57
RU.4	5	0.27	0.10
5	1	1.30	0.16
6	1	3.53	0.38
7	1	9.45	0.72
BA.4	1	0.08	0.04
5	0	0.27	0.06
6	0	0.92	0.14
7	0	2.72	0.23
8	4	6.67	0.46

Table 4: Equational problems (with LNH)

equational theories, and its implementation does not use so sophisticated indexing techniques like those used by the other programs.

3.5 Newly Solved Problems

SEM obtained several nontrivial results, some of which seem quite difficult for other systems to reproduce.

- On Tarski's high school identities [Burris and Lee, 1993], we completed the search for Gurevič algebras of size 8, without finding a solution.
- To prove that the fragment $\{B, M\}$ of combinatory logic does not possess the fixed point property, we searched for countermodels of size up to 7. (No model has been found.)
- R. Padmanabhan asked if there is a finite model of the following set of clauses:

```
g(f(x,y),f(x,x)) = x
f(g(x,y),g(x,x)) = x
g(g(x,y),z) = g(g(y,z),x)
f(f(x,y),z) = f(f(y,z),x)
g(a,f(b,a)) != f(a,g(b,a))
```

Here / and g are two binary function symbols, a and b are two constant symbols. We completed the search for models of size up to 20, but did not find any one.1

* In the study of orthogonal arrays, given three positive integers g, h and k, we'd like to know if there exists an Abelian group G of size g, which has a subgroup H of size h, and on which we can define a function m : G —► G, satisfying the following two conditions: (1) for any a, b E G, m.(ab) = m(a)m(6); and (2) for 0 < i < k and a E H, mk(a) = a, m1(a) not= a (where m°(a) = a and mi+1(a) — m(ml(a))). SEM successfully found (G,H,m) for (g,h,k) = (32, 2, 3), (56, 2, 3), (27, 3, 3), (36, 3, 3), (48, 3, 3), (57, 3, 3).

- In group theory, we know that (xy)2 = x2y2 implies xy = yx. This can be proved with a conventional theorem prover like OTTER [McCune, 1994a]. In contrast, with SEM, we can show that (xy)k — xkyk does not imply the commutativity law, for k — 3,4,5. SEM found the appropriate countermodels of sizes 27, 8, 8, respectively.

- SEM also found some idempotent quasigroups satisfying two identities simultaneously, for example, a 13-element model of QG7 and QG9, a 13-element model of QG8 and QG9, and a 21-element model of QG5 and QG7. (The related identities are: QG7. (yx)y = x(yx); QG8. x(xy) = yx; and QG9. {(xy)y)y = x.)

For the first two problems,2 it took several days for SEM to complete the search. However, the last two problems are not so difficult as they appear — the execution times range from less than one second to several seconds.

4 Concluding Remarks

We have described SEM, a System for Enumerating finite Models, and compared its performance with those of other similar systems. Clearly, SEM can solve a wide range of problems efficiently. As shown in this paper and other papers [Fujita et al, 1993; Slaney et al, 1995; Zhang and Stickel, 1994; Zhang, 1994a; 1994b], finite model generators are very useful tools. So far, they have been mainly used to solve problems in mathematics. But we believe that the related techniques will find applications in AI and computer science as well.

Finite model generation is closely related to search and reasoning, which are two of the most important subjects in AI, and which have been studied extensively. Most existing programs are based solely on backtracking procedures. In the future, we shall experiment with more search heuristics and other non-exhaustive methods. The reliability of non-existence results is also worth studying.

1 Later we proved that this set of clauses is unsatisfiable.
2See [Zhang, 1994b] for the clauses of these two problems.

References

[Benhamou and Sais, 1994] B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. J. of Automated Reasoning, 12(1):89 102, 1994.

[Burris and Lee, 1993] Stanley Burris and Simon Lee. Tarski's high school identities. The American Mathematical Monthly 100(3):231-236, 1993.

[Fujita et al, 1993] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic generation of some results in finite algebra. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, pages 52-57, Chambery, France, 1993.

[Kim and Zhang, 1994] Sun Kim and Hantao Zhang. ModGen: Theorem proving by model generation. In Proceedings of the Twelfth National Conference on Artificial Intelligence, pages 162-167, Seattle, August 1994. American Association for Artificial Intelligence.

[Kumar, 1992] V. Kumar. Algorithms for constraint satisfaction problems: A survey. AI Magazine, 13(1):32-44, 1992.

[McCune, 1994a] William McCune. OTTER 3.0 Reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, January 1994.

[McCune, 1994b] William McCune. A Davis-Putnam program and its application to finite first-order mode] search: Quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, September 1994.

[Slaney, 1993] John Slaney. FINDER: Finite Domain Enumerator. Version 3.0 notes and guide. Australian National University, 1993.

[Slaney et al, 1995] John Slaney, Mark Stickel, and Masayuki Fujita. Automated reasoning and exhaustive search: Quasigroup existence problems. Computers and Mathematics with Applications, to appear.

[Zhang and Stickel, 1994] Hantao Zhang and Mark Stickel. Implementing the Davis-Putnam algorithm by tries. Technical Report, University of Iowa, 1994.

[Zhang, 1991] Jian Zhang. Search for idempotent models of quasigroup identities. Unpublished typescript, Institute of Software, Academia Sinica, Beijing, 1991.

[Zhang, 1994a] Jian Zhang. The generation and applications of finite models. PhD thesis, Institute of Software, Academia Sinica, Beijing, 1994.

[Zhang, 1994b] Jian Zhang. Problems on the generation of finite models. In Proceedings of the Twelfth International Conference on Automated Deduction, pages 753-757, Nancy, France, 1994. Springer Lecture Notes in Artificial Intelligence, Vol. 814.