Constraint Propagation in Model Generation*

Jian Zhang and Hantao Zhang

Department of Computer Science, University of Iowa
Iowa City, IA 52242, USA
{ jzhang | hzhang }@cs.uiowa.edu

Abstract. Model generation refers to the automatic construction of models of a given logical theory. It can be regarded as a special case of constraint satisfaction where the constraints are a set of clauses with equality and functions. In this paper, we study various constraint propagation rules for finite model generation. We implemented these rules in a prototype system called SEM (a System for Enumerating Models). By experimenting with SEM, we try to identify a set of transformation rules that are both efficient and easy to implement. We also compare several existing model generation systems that are based on different logics and strategies.

1 Introduction

Many problems in computer science and AI can be formulated as constraint satisfaction problems (CSPs), that is, finding suitable values for a set of variables such that some constraints hold. However, the notion of "constraints" is somehow vague. In general, the CSP paradigm in AI can be regarded as a restricted logical calculus [1, 10] and various logical formalisms can be used to express the constraints. In this paper, we study a class of CSPs that are different from the conventional CSP paradigm [8, 10] or the constraint logic programming (CLP) paradigm [6]. Here we are concerned with finding structures satisfying a set of clauses in first order logic. That is, instead of finding values for variables, our goal is to define functions (and predicates) in a suitable way such that all the clauses hold. Such a problem is often called *model generation* (or *model finding*).

The conventional CSP can be expressed by the formula $\exists \mathbf{x}.C(\mathbf{x})$, where $\mathbf{x}$ is a finite set of variables (with each variable $x \in \mathbf{x}$ bounded to a specific domain) and C is a first order formula (or a conjunction of clauses). In contrast, the model generation problem can be expressed by $\forall \mathbf{x}.\exists \mathbf{f}.C(\mathbf{f}, \mathbf{x})$, where $\mathbf{f}$ is a finite set of function symbols appearing in C. In other words, we view C as a special case of second-order formula. When the domain of each variable is finite, we may remove the universal quantifier by instantiating $\mathbf{x}$ with its all possible values. For instance, suppose f is a binary function over the domain $D = \{0, 1, ..., n-1\}$ and C is $f(y, f(x, y)) = x$. To eliminate x and y, we may use the n^2 instances of C, i.e, $f(0, f(0, 0)) = 0$, $f(1, f(0, 1)) = 0$, etc.

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

Model generation is clearly very important. In fact, many open questions in mathematics have recently been solved by various model finding programs, such as FALCON [20], FINDER [14], MGTP [4, 5], LDPP [19, 15], SATO [19] and MACE [13]. These programs successfully found mathematical objects having certain properties. Besides such uses, model generators also have other applications. For example, the existence of a model implies the consistency of a theory. And, suitable models may serve as counterexamples refuting conjectures. In this sense, model generation is complementary to conventional theorem proving.

Currently, there exist a number of different approaches to generating models of first order theories, see for example, [2, 9, 15, 16]. Some of them are closely related to automated deduction, while others rely heavily on constraint satisfaction techniques. Some deal only with a certain type of logical formulas, and some can only find finite models.

The formulas considered in this paper are arbitrary clauses, but the models are restricted to *finite* ones. In such cases, decidability or completeness is guaranteed. A model can be found by exhaustive search, if it exists. However, efficiency is an important issue, because in general, the problems involve huge search spaces.

Like conventional CSPs, finite model generation problems can be solved with backtracking procedures and other algorithms (like constraint propagation) [8]. To define a function over finite domains, we can define its "multiplication table", or equivalently, determine the values for all the cells (or entries) in the table. This can be done one cell at a time. For each cell, we choose a value from a finite set of possible values; then we propagate the effects of this choice; if it does not result in contradiction, we accept it and consider the next cell; otherwise we try other values or go back to the previous cell. Just as in the case of ordinary constraint satisfaction, many issues arise here. For example, in what order shall we choose the next cell? When shall we do the propagation? How can we propagate the effect of a choice? What kind of propagation is complete or inexpensive? How much propagation is necessary? Of course, the answers to these questions depend largely on the application domains.

The purpose of this paper is to describe and analyze some rules for constraint propagation and entailment, in the context of finite model generation. To the best of our knowledge, there is little formal study of constraint satisfaction when the constraints are expressed as arbitrary clauses. In general, many notions of CSP such as k-variable constraints and arc-consistency do not apply here.

As we mentioned earlier, there exist several model generation systems. But their performances on some problems are quite different. For instance, Slaney, Stickel and Fujita [15] observed that, on the quasigroup identity problems, there is a "large difference between DDPP and FINDER in the matter of branching: DDPP generates far fewer branches than FINDER, but takes from 5 to 66 times longer to explore each one." However, no reasons were given why there exists such a big difference. We believe that our experimental results and analysis will provide some hint on this issue.

The rest of this paper is organized as follows. In the next section, we describe

some basic concepts and notations which will be used later, e.g. cells, elements and some special forms of clauses. In §3, we give some rules for transforming the constraints. Then we briefly describe our model finding program SEM, and its constraint solving mechanism. In §5, several similar systems are compared on some benchmark problems, and their differences in performances are analyzed. Finally, we discuss some of the related work and conclude the paper.

2 Basic Concepts and Notations

For the basic concepts in predicate calculus, automated theorem proving and the clausal representation language, the reader is referred to standard textbooks or the book [17] (Chap. 4). In the sequel, we use s, t (and s_i, t_j) to denote arbitrary terms; C and M to denote a set of clauses and a disjunction of literals, respectively. A positive equality literal is represented by $t_1 = t_2$ and a negative equality literal is by $t_1 \neq t_2$, where t_1 and t_2 are well-formed terms. We also use $L[t]$ to denote a literal which contains the subterm t.

Simply stated, the model generation problem is, given a set of first order clauses, find an interpretation of all the function symbols appearing in them such that all the clauses become true. Such an interpretation is called a *model*. Predicate symbols can be regarded as function symbols of the Boolean sort in a many-sorted language. For simplicity, we shall only consider the uni-sorted case with the special predicate "equality". We assume that the size of the model is a fixed finite positive number, denoted by n.

When the involved domain is finite, an interpretation can be represented by a set of *multiplication tables*, one for each function. Without loss of generality, an n-element domain is assumed to be $D_n = \{ 0, 1, \ldots, n-1 \}$. From now on, we use e_i to denote an element of a domain. Syntactically it can be regarded as a constant symbol. For any two distinct elements e_i and e_j $(i \neq j)$, it is assumed that the inequality $e_i \neq e_j$ always holds.

For an m-ary function f, each entry of its multiplication table can be represented by a ground term of the form $f(e_1, e_2, \ldots, e_m)$. Such a term is called a *cell term* or simply a *cell*, and will be denoted by ce or ce_i.

To find the models, we have to assign suitable values (i.e. elements) to the cells. For this purpose, we are interested in certain relationships between cells and elements. In particular, we distinguish the following three types of clauses:

- possible values (PV) clause: $ce = e_1 \vee \ldots \vee ce = e_i \vee \ldots \vee ce = e_k$
- value assignment (VA) clause: $ce = e_i$
- value elimination (VE) clause: $ce \neq e_i$

These clauses tell us what values a cell can take during the search. Note that, in a PV clause, all the cell terms are the same, and a VA clause is a special case of a PV clause. The left-hand side and the right-hand side of an equation (or inequality) are interchangeable. So, $e_i = ce$ is also a VA clause, and $e_j \neq ce$ is also a VE clause.

A finite model can be represented by a set of VA clauses. As a simple example, let us consider quasigroups.

Example. A quasigroup has one binary operation whose multiplication table forms a Latin square, i.e. each row and each column is a permutation of the elements. Let us denote the operation by f. A 3-element quasigroup is shown below.

f	0	1	2
0	0	2	1
1	1	0	2
2	2	1	0

This table is equivalent to the following set of VA clauses:

$$\begin{array}{lll} f(0,0)=0, & f(0,1)=2, & f(0,2)=1, \\ f(1,0)=1, & f(1,1)=0, & f(1,2)=2, \\ f(2,0)=2, & f(2,1)=1, & f(2,2)=0. \end{array}$$

In general, the constraint of a (finite) model generation problem is a set of clauses, called *axioms*. For example, quasigroups can be axiomatized by the following two clauses:

$$f(x,y) \neq f(x,z) \vee y = z, \tag{1}$$
$$f(x,z) \neq f(y,z) \vee x = y, \tag{2}$$

where all the three variables, x, y, and z, are universally quantified. One can easily verify that the above 3-element quasigroup satisfies the two axioms, by substituting i, j, k $(0 \leq i, j, k \leq 2)$ for x, y, z, respectively.

Usually the axioms contain some free variables which are assumed to be universally quantified. When the size of each domain is finite and fixed beforehand, we may consider the instantiated clauses as the constraints, instead of the original axioms. For example, to find a 3-element quasigroup, we should satisfy $3^3 \times 2 = 54$ ground (i.e. variable-free) clauses. The benefit of this instantiation is that unification is avoided in the rules for constraint propagation, and the computational costs of many inferences are reduced.

In short, when the domain is finite, both the constraint and the solution (i.e. the model) can be expressed by a set of first-order ground clauses. In the remainder of this paper, we shall assume that all the terms, literals and clauses are ground (i.e., free of variables). And, when saying a set of clauses is *satisfiable*, we mean that it has a model whose domain is D_n.

3 Rules for Constraint Transformation

To find an n-element model of a set of axioms, we start from a set of ground clauses which includes all the instantiations of the axioms (by substituting elements of D_n for the variables), and exactly one PV clause for each cell ce:

$$ce = 0 \vee ce = 1 \vee \ldots \vee ce = n-1.$$

For convenience, let us call such a set of ground clauses *n-complete*. Our goal is to derive a set of VA clauses such that all the clauses are true. Essentially this can be achieved through a sequence of transformations.

In this section, we describe some rules for transforming the clause set. These rules can be classified into the following three categories:

- **simplification rules:** A current clause is simplified or removed from the clause set. Tautology deletion and subsumption are such rules.
- **inference rules:** A new clause is generated and added into the clause set. For instance, resolution and paramodulation [17] are two well-known inference rules.
- **splitting rules:** The current clause set is split into several clause sets. Such rules are used in case-analysis reasoning.

3.1 Simplification Rules

Most simplification rules in first order theorem proving, such as tautology deletion, subsumption, and rewriting, can be used to simplify constraints. Let us start with the simplest ones.

equality resolution:

$$\textbf{(ER1)}\ \frac{C \cup \{t \neq t \vee M\}}{C \cup \{M\}} \qquad \textbf{(ER2)}\ \frac{C \cup \{e_i = e_j \vee M\}}{C \cup \{M\}}\ (i \neq j)$$

equality subsumption:

$$\textbf{(ES1)}\ \frac{C \cup \{t = t \vee M\}}{C} \qquad \textbf{(ES2)}\ \frac{C \cup \{e_i \neq e_j \vee M\}}{C}\ (i \neq j)$$

The above rules are sound, because we have, for all x, $x = x$; and $e_i \neq e_j$ is assumed to be true for any two distinct elements e_i and e_j in the domain.

Next we consider unit resolution, which is a well-known inference rule in theorem proving. In the ground case, it can be used as a simplification rule. When an empty clause is produced by either equality resolution or unit resolution, the entire set of clauses is unsatisfiable.

unit resolution:

$$\textbf{(UR1)}\ \frac{C \cup \{t_1 \neq t_2 \vee M,\ t_1 = t_2\}}{C \cup \{M,\ t_1 = t_2\}} \qquad \textbf{(UR2)}\ \frac{C \cup \{t_1 = t_2 \vee M,\ t_1 \neq t_2\}}{C \cup \{M,\ t_1 \neq t_2\}}$$

Because it is expensive to perform full subsumption in model generation, only unit subsumption is considered in our experimentation. Actually, in our experiments, we find that when splitting rule is used, the performance of the system is often better if we impose further restrictions on unit resolution and unit subsumption that the involved unit clause must be either a VA or VE clause.

unit subsumption:

$$\textbf{(US1)}\ \frac{C \cup \{t_1 = t_2 \vee M,\ \ t_1 = t_2\}}{C \cup \{t_1 = t_2\}} \qquad \textbf{(US2)}\ \frac{C \cup \{t_1 \neq t_2 \vee M,\ \ t_1 \neq t_2\}}{C \cup \{t_1 \neq t_2\}}$$

Rewriting is a powerful simplification rule for theorem proving. It can also be used in constraint propagation.

rewriting:

$$\frac{C \cup \{L[t_1] \vee M,\ \ t_1 = t_2\}}{C \cup \{L[t_2] \vee M,\ \ t_1 = t_2\}}$$

The effect of this rule is to replace t_1 in the clause $L[t_1] \vee M$ by t_2. In our experimentation, we require that t_2 be an element and t_1 be a non-element term, so that the termination of rewriting is guaranteed. In fact, it is often simple and efficient if we require that t_1 be a cell, i.e., $t_1 = t_2$ is a VA clause.

After the simplification rules are applied, some of the clauses may be reduced to VA or VE clauses, so that further simplification can be performed.

Let us consider the quasigroup example in §2. Substituting 0, 1, 2 for x, y, z respectively in clause (1), we get the following clause:

$$f(0,1) \neq f(0,2) \vee 1 = 2$$

This may be reduced by the rule **(ER2)** to:

$$f(0,1) \neq f(0,2)$$

Suppose during the search we choose $f(0,1) = 2$, then we can infer a VE clause $2 \neq f(0,2)$ by rewriting.

3.2 Inference Rules

The first inference rule we consider here is paramodulation, which is defined as follows (assuming C is a set of ground clauses):

paramodulation:

$$\frac{C \cup \{L[t_1] \vee M_1,\ \ t_1 = t_2 \vee M_2\}}{C \cup \{L[t_1] \vee M_1,\ \ t_1 = t_2 \vee M_2,\ \ L[t_2] \vee M_1 \vee M_2\}}$$

This rule, together with equality resolution, provides a decision procedure for the unsatisfiability of a set of clauses. However, our experiences show that it is not practical to apply paramodulation in its general form, because it may generate too many new clauses. Imposing an ordering on literals can help but the problem of explosion remains. However, if we impose more restrictions on this rule, we may obtain various simplification rules such as rewriting or unit resolution.

Another inference rule involving the equality predicate is called **dismodulation:**

$$\frac{C \cup \{f(t_1,...,t_m) = t_0 \vee M_1,\ \ f(s_1,...,s_m) \neq s_0 \vee M_2\}}{C \cup \{f(t_1,...,t_m) = t_0 \vee M_1,\ \ f(s_1,...,s_m) \neq s_0 \vee M_2,\ \ \bigvee_{i=0}^{m}(t_i \neq s_i) \vee M_1 \vee M_2\}}$$

The soundness of this rule can be easily established. Suppose the derived clause, $\bigvee_{i=0}^{m}(t_i \neq s_i) \vee M_1 \vee M_2$, is false in an interpretation, then both M_1 and M_2 are false, and for each i $(0 \leq i \leq m)$, $t_i = s_i$ holds. So one of the parent clauses must be false in the interpretation, since $f(t_1,...,t_m) = f(s_1,...,s_m)$ and $t_0 = s_0$.

The above rule is quite similar to negative paramodulation [18], in that new information is derived from inequalities. However, since we consider only ground clauses, the function f in our rule need not have any special property (e.g. the cancellation property).

The dismodulation rule in its general form has the similar problem as the paramodulation rule, i.e., it may generate too many new clauses. From our experiments, we find that it is often beneficial if the above rule produces only VE clauses. In this case, M_1 and M_2 are empty, one of the terms $t_1,...,t_m$, $s_1,...,s_m$ is a cell, and the rest t_i's and s_i's (including t_0 and s_0) are elements. So we have the following two rules.

VE generation:

$$\textbf{(VEG1)}\ \frac{C \cup \{f(e_1,..,e_i,..,e_m) = e_0,\ \ f(e_1,..,ce,...,e_m) \neq e_0\}}{C \cup \{f(e_1,..,e_i,..,e_m) = e_0,\ \ f(e_1,..,ce,...,e_m) \neq e_0,\ \ ce \neq e_i\}}$$

$$\textbf{(VEG2)}\ \frac{C \cup \{f(e_1,..,ce,..,e_m) = e_0,\ \ f(e_1,..,e_i,...,e_m) \neq e_0\}}{C \cup \{f(e_1,..,ce,..,e_m) = e_0,\ \ f(e_1,..,e_i,...,e_m) \neq e_0,\ \ ce \neq e_i\}}$$

Note that, to apply each of the VE generation rules, one parent must be a VA or VE clause, and the complex terms in the two parents differ only in one argument. The resulting new clause is a VE clause.

Actually, the above rules of VE generation can be generalized. Let us call a complex term *nearly evaluable* if all but one of its arguments are elements in the domain. For example, $h(1, g(f(2,4)), 3)$ is a nearly evaluable term, while $h(1, f(2,4), g(3))$ is not. We may replace the cell ce in the above rules by a nearly evaluable term t. The deduced inequality $t \neq e_i$ can participate in further similar inferences, until a VE clause is produced. The new rule can be decomposed into a sequence of VE generation steps, just as hyperresolution can be decomposed into a sequence of resolution steps. For example, suppose we have the following three clauses:

$$h(1, g(f(2,4)), 3) \neq 5$$
$$h(1,2,3) = 5$$
$$g(1) = 2$$

Then we can conclude that $f(2,4) \neq 1$ by applying **(VEG1)** twice: from the first two clauses, derive $g(f(2,4)) \neq 2$; and then from this new inequality and the third clause, derive the VE clause $f(2,4) \neq 1$.

3.3 Splitting

Since we do not use paramodulation in its full version, to compensate the incompleteness resulted from this restriction, we use splitting (or equivalently, case analysis). The effect of this rule is that the current set of clauses is split into several sets of clauses. The completeness of a splitting rule requires that the original set of clauses is satisfiable iff one of the resulting set of clauses is satisfiable. Below we give two forms of splitting rules which are often used. In each case, a non-unit PV clause is chosen to split. If there is no such a PV clause, the search is completed.

splitting:

$$\textbf{(SP1)} \quad \frac{C \cup \{ce = e_1 \vee ce = e_2 \vee \ldots \vee ce = e_k\}}{C \cup \{ce = e_1\} \qquad C \cup \{ce \neq e_1,\ ce = e_2 \vee \ldots \vee ce = e_k\}} \ (k > 1)$$

$$\textbf{(SP2)} \quad \frac{C \cup \{ce = e_1 \vee ce = e_2 \vee \ldots \vee ce = e_k\}}{C \cup \{ce = e_1\} \qquad C \cup \{ce = e_2\} \qquad \cdots \qquad C \cup \{ce = e_k\}} \ (k > 1)$$

In the first rule, two sets of clauses are generated such that the VA clause $ce = e_1$ is in one set and the VE clause $ce \neq e_1$ is in the other. In the second rule, k sets of clauses are generated, each containing a unique VA clause $ce = e_i$. The two rules are logically equivalent but may result in different performances. We shall discuss this issue in §5.3.

The soundness and completeness of the simplification rules, inference rules and the splitting rules can be easily established. Specifically, let $C \Rightarrow C'$ denote that C' is derived from C by either of equality resolution, unit resolution, rewriting by VA clause, VE generation or splitting, and let $\Rightarrow^*$ be the reflexive and transitive closure of $\Rightarrow$, then we have the following

Theorem *A set C of n-complete ground clauses has a model over D_n iff $C \Rightarrow^* C'$ in a finite number of steps, where C' is satisfiable and contains a VA clause $ce = e$ for every cell ce.*

Note that the satisfiability of C' is very easy to check when it contains a VA clause $ce = e$ for every cell ce.

4 A Prototype System

We have developed a prototype System for Enumerating finite Models of first order theories, called SEM. It accepts a set of many-sorted clauses, and produces finite models of fixed sizes, if they exist. The system is implemented in C. Internally it uses ground clauses to represent the constraints of a problem, as described in §2.

In SEM, we implemented the rules **(SP2)**, **(ER2)**, **(ES2)** and the VE generation rules. We also implemented restricted forms of other simplification rules which include: rewriting by VA clauses, **(ER1)** and **(ES1)** when the term t is

an element, unit resolution and unit subsumption when one term is a cell and the other is an element. The execution mechanism of the system is as follows. Through splitting, we get some VA clauses. With such clauses, the resolution and rewriting rules can be applied to simplify the constraints. After the simplification, some new VA (or VE) clauses might be generated, which can be used again. A VE clause may be propagated in two ways. Firstly, it reduces the number of possible values for the cell, thus simplifying the corresponding PV clause; secondly, VE generation rules might be applied to produce new VE clauses.

The overall process can be described as a search tree whose internal nodes correspond to the applications of the splitting rule. As the search proceeds, the number of VA and VE clauses increases, while the constraint clauses become simpler or even disappear. Each branch of the search tree ends in two possible states: (1) the clause set contains one VA clause for each cell, and a model is found successfully; or (2) the empty clause is generated, which means the clause set is unsatisfiable.

Let us give a simple example to illustrate the use of SEM. The problem is to decide how to color the Canadian flag with two colors, satisfying certain constraints [10]. The problem can be specified in SEM as follows. (A comment begins with '%' and extends to the end of the line.)

```
%% Sorts

( color  : red, white )
( region : X, Y, Z, U )

%% Functions

{ ne : region region -> BOOL }      % the neighborhood relation
{ f  : region -> color }

%% Variables

< x1, x2 : region >

%% Clauses

[ -ne(x1,x2) | ne(x2,x1) ]          % commutativity of 'ne'
[ -ne(x1,x2) | f(x1) != f(x2) ]     % different colors for neighbors
[ ne(X,Y) ]
[ ne(Y,Z) ]
[ ne(Y,U) ]
[ f(U) = red ]
```

With this as input, SEM will produce a model of the clauses in which $f(X) = f(Z) = f(U) = red, f(Y) = white$.

In contrast to most constraint logic programming systems [6], SEM is not restricted to Horn clauses. Lee and Plaisted give solutions to some non-Horn problems in [9]. These problems can be solved easily with our system. For example, Lee and Plaisted's prover, which was implemented in C Prolog and runs

on a DEC workstation 5000/125, spent 31.5 seconds solving the so-called "salt and mustard" problem. SEM used the same clauses as given in [9] and found a solution in 0.06 seconds, running on a SPARCstation 2.

Our system also allows the use of function symbols. Let us give one example of its uses in solving mathematical problems. In [12], McCune gives some single (equational) axioms for group theory. (A single axiom is an equation from which all the group theorems can be derived.) He reports that the search for simpler axioms (by taking instances of the known axioms) failed. There can be two reasons for such failures. One is that the equation is too weak, the other is that the proof is difficult to find. Model generators may play a role here. Consider the equation

$$f(y, g(f(y, f(f(f(z, g(z)), g(f(u, y))), y)))) = u$$

which is an instance of the single axiom (3.1) in [12]. SEM found a 4-element model of it which is not a group. Thus we know that it is *impossible* for this equation to be a single axiom.

5 Experiments and Analysis

SEM can solve a wide range of problems efficiently. Its performance is very competitive, compared with other similar systems. In this section, we describe some experiments with various rules on solving sample problems from discrete mathematics. Some issues will be discussed, based on the analysis of the results.

Besides SEM, we used the following three programs in our experiments: FINDER 3.0.1 [14], MACE 1.0.0 [13], and SATO 2.0 [19]. All of them were implemented in C, and are available to the public. Moreover, each has solved nontrivial previously open questions from mathematics.

Like SEM, FINDER also uses first-order ground clauses. Its rules include unit resolution and negative hyper-resolution, in addition to case analysis. On the other hand, MACE and SATO, as well as DDPP [19] which is written in LISP, are based on the Davis-Putnam algorithm [3] for testing satisfiability of propositional formulas. These three programs rely heavily on the unit propagation rule. But they use different data structures and search strategies.

5.1 Test Problems

Our focus is on the applications to some branches of mathematics. Two representative problems are chosen to test the aforementioned programs. The first one is to find quasigroups satisfying some additional constraints, called the QG5 problem in [4, 15].

The QG5 problem. The axioms for this problem are[2]:

$$f(x,y) \neq f(x,z) \vee y = z$$
$$f(x,z) \neq f(y,z) \vee x = y$$
$$f(x,x) = x$$
$$f(f(f(y,x),y),y) = x$$
$$f(y,f(f(x,y),y)) = x$$
$$f(f(y,f(x,y)),y) = x$$

The second problem asks to find finite noncommutative groups. It is suggested in [17] that, to test the performance of automated theorem provers, one may let them prove certain properties (e.g. commutativity) of finite groups. Such problems are good candidates for model generators.

The NCG problem. The axioms are:

$$f(0,x) = x$$
$$f(x,0) = x$$
$$f(g(x),x) = 0$$
$$f(x,g(x)) = 0$$
$$f(f(x,y),z) = f(x,f(y,z))$$
$$f(1,2) \neq f(2,1)$$

Here f and g denote respectively the multiplication and inverse operations of groups. Without loss of generality, we choose 0 as the identity element of the group. It can be proved that, in a noncommutative group, if there are two elements, a and b, for which $f(a,b) \neq f(b,a)$, then a and b should be different from each other, and neither of them can equal the identity. Thus we assume the two elements are 1 and 2.

We shall use QG5.n and NCG.n to denote the problem of finding n-element models of QG5, and n-element non-commutative groups, respectively. It should be noted that, in experimenting with the different programs, we use the same representation for the same problem. On the NCG problem, we do not use any method for rejecting isomorphism[3]. For each problem, the programs are asked to search for all solutions. Table 1 gives the numbers of models of different sizes.

5.2 Experimental Results

The following four tables show the performances of the four programs on the test problems. The columns of these tables are explained as follows.

- c: number of propositional clauses (for MACE and SATO)

[2] An additional clause for eliminating some isomorphic models is not shown here. See [4, 15].

[3] With the least number heuristic [20] for isomorphism elimination, SEM's performance is much better than described in this paper. On the QG5 problem, FINDER also has a more elaborate method for dealing with isomorphism [14].

QG5	order	7	8	9	10	11	12
	models	3	1	0	0	5	0
NCG	order	4	5	6	7	8	9
	models	0	0	18	0	480	0

Table 1. Number of Models

- v: number of propositional variables (for MACE and SATO)
- b: number of branches of the search tree, or, number of backtracks (for FINDER)
- t_1: time spent on generating the clauses
- t_2: time spent on search
- m: memory used (for MACE and SEM)

In addition, b' and t_2' denote respectively the number of branches and the search time, when SEM does not use the VE generation rules. The execution times are measured in seconds, and the memory is in K bytes. The data were obtained on a SPARCstation 2. FINDER did not complete the search for NCG.8 within an hour, and MACE aborted the search for NCG.9 due to the limit of memory.

	b	t_1	t_2
QG5.7	3	0.53	0.07
.8	13	1.07	0.10
.9	46	1.98	0.33
.10	341	3.42	3.70
.11	1728	5.53	20.27
.12	11047	19.98	139.92
NCG.4	5	0.01	0.03
.5	44	0.02	0.43
.6	727	0.10	12.23
.7	24237	1.83	539.02

Table 2. Performance of FINDER

5.3 First-Order vs. Propositional Clauses

Finite model generation problems in first-order logic, as well as many finite constraint satisfaction problems, can be transformed to problems in propositional logic. (See, for example, [10, 7].) In this approach, we introduce some propositional variables to represent the values of the cells. For example, if f is a binary

	b	b'	t_1	t_2	t_2'	m
QG5.7	6	12	0.03	0.02	0.02	95
.8	11	66	0.04	0.03	0.06	127
.9	29	509	0.05	0.06	0.32	159
.10	250	2329	0.06	0.62	2.48	223
.11	1231	20351	0.08	2.71	18.96	287
.12	8636	381787	0.13	20.08	365.24	351
NCG.4	30	30	0.01	0.00	0.00	31
.5	123	123	0.01	0.03	0.03	63
.6	734	734	0.03	0.22	0.19	63
.7	4893	4893	0.05	1.89	1.87	95
.8	54288	54288	0.06	24.58	25.07	159
.9	598871	598871	0.10	347.66	343.24	191

Table 3. Performance of SEM

	v	c	b	t_1	t_2	m
QG5.7	392	10508	4	0.39	0.08	300
.8	576	17949	8	0.65	0.12	596
.9	810	28792	14	1.09	0.24	894
.10	1100	43946	37	1.63	0.58	1193
.11	1452	64428	112	2.45	2.38	1786
.12	1872	91363	369	3.52	7.25	2674
NCG.4	108	8976	1	0.37	0.00	294
.5	190	32921	2	1.34	0.05	295
.6	306	96471	18	4.13	1.08	884
.7	462	240771	120	10.44	3.04	2645
.8	664	533162	1110	23.51	175.57	6751
.9	918	1076541		47.12		>16000

Table 4. Performance of MACE

n	v	c	b	t_1	t_2
QG5.7	343	10461	5	0.16	0.06
.8	512	17887	8	0.32	0.09
.9	729	28714	11	0.55	0.17
.10	1000	43849	21	0.91	0.39
.11	1331	64311	43	1.41	1.06
.12	1728	91223	277	2.18	4.97

Table 5. Performance of SATO

function symbol, then we may use the propositional variable f_{ijk} to denote that $f(i,j) = k$. And, a first-order ground clause like $f(2, f(0,1)) = 0$ can be translated into a set of propositional clauses, i.e., $\{ \neg f_{01k} \vee f_{2k0} \mid 0 \leq k < n \}$.

Thus decision procedures for propositional logic (like the Davis-Putnam procedure [3]) may be used to find finite models of first-order theories. But in some cases, a large number of propositional clauses are needed to represent the problem. (See Table 4 and Table 5.) The memory requirements restrict the uses of propositional clauses in model generation. However, when the number of clauses is not too large, programs based on this approach are very efficient.

One can see from the four tables that, the search trees of MACE and SATO have much fewer branches than those of FINDER and SEM. This is because different splitting rules are used (see §3.3). Let us compare SEM with SATO. The former uses (**SP2**) for splitting, while the latter uses (**SP1**). Suppose during the search, there are k possible values e_i $(1 \leq i \leq k)$ for the cell ce. Then SEM divides the search space into k subspaces, each containing a different assignment $ce = e_i$. In contrast, SATO first considers the case $ce = e_1$; if this assignment leads to a contradiction, then we know $ce \neq e_1$, and the program propagates the effect of this negative unit clause. Such a propagation may reveal the unsatisfiability immediately, eliminating the need for considering the cases $ce = e_2, \ldots, ce = e_k$.

The difference can be illustrated with a simple example. Suppose we want to find QG5.6, and assume that the multiplication table is constructed row by row. After making three choices, both SEM and SATO can determine all the values at the first row, which are:

$$f(0,0) = 0,\ f(0,1) = 2,\ f(0,2) = 1,\ f(0,3) = 4,\ f(0,4) = 5,\ f(0,5) = 3.$$

The next cell is $f(1,0)$ which has three possible values: 3, 4, or 5. SEM assigns each value to the cell, then deduces contradiction from each assignment. On the contrary, SATO first assumes $f(1,0) = 3$ and then assumes $f(1,0) \neq 3$. In each case, contradiction results.

Now let us compare SEM and SATO on constraint propagation. The unit propagation rules in propositional logic are flexible enough to simulate many rules discussed in this paper. However, there are still some differences between the behaviors of the two programs. Suppose we have an assignment $f(1,0) = 3$, then SEM simply replaces all occurrences of $f(1,0)$ by 3. SATO not only performs such propagations, but also propagate the negative unit clauses $f(1,0) \neq k$, for every $k \neq 3$.

In general, it is important to eliminate as many candidate values as possible, before the splitting rule is used. This is crucial for some problems which have many negative constraints (like QG5). From Table 3, we can see the difference in performances when SEM uses and does not use the VE generation rules. For problems like NCG in which most of the clauses are positive equalities, using VE rules has some overhead, but that is negligible.

When choosing the next cell for splitting, a good heuristic is to choose the cell with the fewest possible values. Our experiments confirm that it is indeed helpful. But this is not the focus of this paper.

6 Related Work

Many researchers have worked on constraint satisfaction problems, and a number of algorithms and heuristics have been proposed. (See for example, [8].) Some authors have also discussed them from the deductive point of view [1, 10, 15]. But the constraints studied in this paper are more complicated than those of conventional CSPs. On the other hand, most work on constraint logic programming [6] focus on such specialized domains as Boolean algebra and linear arithmetic.

Bourely, Caferra and Peltier [2] proposed some inference rules for building Herbrand models. Their method is based on constrained formulas which usually contain variables. Infinite models as well as finite ones can be found. The rules discussed in this paper are restricted to ground clauses. But they are very simple, computationally inexpensive and easy to implement.

Slaney, Stickel and Fujita [15] described some experiments with three model generation programs, i.e. FINDER, MGTP, and DDPP, on solving the quasigroup problems. Since the three programs are based on different approaches, implemented in different languages, and accept different specifications, it is hard to compare the constraint propagation mechanisms of these systems.

FINDER and SEM are quite similar in that both of them reason with first-order ground clauses. But there are also some differences. For example, SEM keeps only a limited amount of information. In contrast, one feature of FINDER is that, it derives (using negative hyper-resolution) and stores secondary constraints during the search process, so that the program does not backtrack twice for the same reason [4, 14, 15]. This may be beneficial on some problems. But unrestricted use of such a strategy can result in too many clauses. We believe this is one reason why FINDER is not so efficient on the NCG problem.

The program MGTP [5, 15] uses (range-restricted) clauses with variables. One benefit of such a formalism is that less memory is required. But our experiences tell us, even though ground clauses are used, memory is not a serious problem in most applications.

DDPP, like SATO, is an implementation of the Davis-Putnam algorithm [19]. From the comparison of SEM and SATO made in §5.3, we may see why there is a "large difference between DDPP and FINDER in the matter of branching" [15].

7 Conclusion

The subject of this paper is to study constraint propagation rules for finding finite models. Our experiments show that first-order ground clauses are suitable to represent the constraints in such applications as finding finite algebras. For the purpose of model generation, we defined several special forms of clauses, and studied some rules for transforming the ground clauses. These rules have been implemented in our model generation system SEM. Experiments with SEM show that the transformation rules are quite effective. For example, the VE generation rules help to prune the search tree in some applications.

We also made some comparison between several existing model generators. Our experimental results exhibit the differences between systems based on the propositional logic (like DDPP, SATO and MACE) and those based on first-order ground clauses (like FINDER and SEM). Comparing SEM with FINDER, we find that, it is often beneficial to use a limited number of rules in a restricted manner. This is similar to the case of propositional logic. For example, in practical implementations of the Davis-Putnam algorithm, the pure literal rule is rarely used, and the subsumption rule turns out to be expensive in most cases [13, 19].

Our experimental evaluation is by no means complete. We will investigate which rules and which data structures work best for which problems. We intend to conduct additional experiments with more problems, and compare SEM with more systems, such as SATCHMO [11] and MGTP [5].

References

1. Bibel, W., "Constraint satisfaction from a deductive viewpoint," *Artificial Intelligence* 35 (1988) 401–413.
2. Bourely, C., Caferra, R., and Peltier, N., "A method for building models automatically: Experiments with an extension of OTTER," *Proc. 12th Conf. on Automated Deduction* (*CADE-12*), Springer *LNAI* 814 (1994) 72–86.
3. Davis, M., and Putnam, H., "A computing procedure for quantification theory," *J. ACM* 7 (1960) 201–215.
4. Fujita, M., Slaney, J., and Bennett, F., "Automatic generation of some results in finite algebra," *Proc. 13th IJCAI* (1993) 52–57.
5. Hasegawa, R., Koshimura, M., and Fujita, H., "MGTP: A parallel theorem prover based on lazy model generation," *Proc. 11th Conf. on Automated Deduction* (*CADE-11*) Springer *LNAI* 607 (1992) 776–780.
6. Jaffar, J., and Maher, M.J., "Constraint logic programming: A survey," *J. of Logic Programming* 19/20 (1994) 503–581.
7. Kim, S., and Zhang, H., "ModGen: Theorem proving by model generation," *Proc. AAAI-94*, Seattle (1994) 162–167.
8. Kumar, V., "Algorithms for constraint satisfaction problems: A survey," *AI Magazine* 13 (1992) 32–44.
9. Lee, S.-J., and Plaisted, D. A., "Problem solving by searching for models with a theorem prover," *Artificial Intelligence* 69 (1994) 205–233.
10. Mackworth, A.K., "The logic of constraint satisfaction," *Artificial Intelligence* 58 (1992) 3–20.
11. Manthey, R., and Bry, F., "SATCHMO: A theorem prover implemented in Prolog," *Proc. 9th Conf. on Automated Deduction* (*CADE-9*), *LNCS* 310 (1988) 415–434.
12. McCune, W., "Single axioms for groups and Abelian groups with various operations," *J. of Automated Reasoning* 10 (1993) 1–13.
13. McCune, W., "A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems," Technical Report ANL/MCS-TM-194, Argonne National Laboratory (1994).
14. Slaney, J., "FINDER: Finite domain enumerator. Version 3.0 notes and guide," Australian National University (1993).

15. Slaney, J., Stickel, M., and Fujita, M., "Automated reasoning and exhaustive search: Quasigroup existence problems," *Computers and Mathematics with Applications* 29 (1995).
16. Tammet, T., 'Using resolution for deciding solvable classes and building finite models,' *Baltic Computer Science: Selected Papers*, Springer *LNCS* 502 (1991) 33–64.
17. Wos, L., *Automated Reasoning: 33 Basic Research Problems*, Prentice-Hall, Englewood Cliffs, New Jersey (1988).
18. Wos, L., and McCune, W., "Negative paramodulation," *Proc. 8th Conf. on Automated Deduction (CADE-8)*, Springer *LNCS* 230 (1986) J.H. Siekmann (ed.) 229–239.
19. Zhang, H. and Stickel, M., "Implementing the Davis-Putnam algorithm by tries," Technical Report, University of Iowa (1994).
20. Zhang, J., "Constructing finite algebras with FALCON," *J. of Automated Reasoning*, to appear.