MACE4 and SEM: A Comparison of Finite Model Generators*

Hantao Zhang[1] and Jian Zhang[2]

[1] Department of Computer Science
The University of Iowa
Iowa City, IA 52242, U.S.A.
hantao-zhang@uiowa.edu
[2] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zj@ios.ac.cn

Abstract. This article has three objectives: (1) Promote Mace4, a program developed by Bill McCune that searches for finite models of first-order formulas and that is the best way to remember Bill. (2) Promote the research on model generation of first-order formulas. Mace4 remains one of the best model generation programs and we need newcomers who can take over Bill's torch, because model generation is very important to automated reasoning and has many applications. (3) Compare Mace4 with SEM in detail so that the users of these tools or new model generator developers will understand the strengths and weaknesses of both systems and take advantage from this study.

Keywords: Finite models, constraint propagation, backtracking search.

1 Introduction

If automated reasoning tools are regarded as race cars, then the people who created these tools are regarded not only as car designers, but also as car engineers. Engineers pay more attention to details and have their own "small talks". Bill McCune was such an engineer and so are the two authors. In fact, each of us read the source code written by the other two. Sato [48], a tool developed by the first author, still uses some code from Otter [34].

In the fall of 1995, the first author of this article spent four months at Argonne National Laboratory and met Bill daily. At that time, Bill just finished the implementation of the Knuth-Bendix completion procedure modulo AC. We talked about the RRL experience [26] and how to make the AC completion faster [13]. The next year, using the AC completion, Bill successfully solved the long standing open Robbins algebra problem using his EQP (Equational Prover) [37].

In October 2001, Bill drove the second author to the home of Larry Wos, and Larry invited us for lunch. In 2006, Bill was invited to give a speech at the 8th

* Supported in part by NSFs of China and USA.

M.P. Bonacina and M.E. Stickel (Eds.): McCune Festschrift, LNAI 7788, pp. 101–130, 2013.

International Conference on Artificial Intelligence and Symbolic Computation (AISC) which was held in Beijing. Bill gave an excellent talk and discussed with the second author during the conference.

The last invited talk given by Bill was on March 30, 2007, at the University of Iowa. After the talk, Bill and the first author went to a famous steak restaurant by the Iowa river. We had a great time there and Bill really liked local beer and drank two bottles. We expressed concern about his early retirement from Argonne and he said that he had enough saving for the rest of his life. Indeed, we no longer need to be concerned about his retirement but also no longer have a chance to offer him the beer he liked.

The objective of this article is three-fold. Bill was not only a great designer of automated reasoning tools, but also a great software engineer. He created many popular tools including Otter, a resolution-based theorem prover, and Mace4, a program that searches for finite models of first-order formulas. Bill is gone but his tools remain. The best way to remember him is to continue to use the tools he created. There are no publications introducing Mace4, except two Argonne National Laboratory technical reports. This article will promote the use of Mace4 and that is the first objective of this article.

The second objective is to promote research on model generation of first-order formulas. If we can generate a model for a formula, we have proved that the formula is satisfiable, which is an important property of logical formulas. The model generation problem, regarded as a special case of the Constraint Satisfaction Problem (CSP), has many applications in AI, computer science and mathematics. For instance, it has been shown in [42,53,50] that model generators can solve various problems from algebra, number theory, and design theory. 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. Thus, it was a major objective of Bill to create Mace4 as a companion of the theorem prover Otter, using the same input files for both Otter and Mace4. Mace4 remains one of the best model generation programs. However, unlike research in propositional satisfiability where we see new solvers every year, we saw very few new model generators for first-order logic. We need newcomers who can take over Bill's torch.

The third objective is to provide a comprehensive comparison of Mace4 and our program SEM (*a System for Enumerating Models*) [59]. From the viewpoint of a theoretician, Mace4 and SEM are identical as they are based on the same principle and use roughly the same strategies. From the viewpoint of an engineer, Mace4 and SEM are different in many technical aspects. We will discuss these differences and their impact on the performance of these programs. Although their development was completed approximately ten years ago, Mace4 and SEM are still two of the most efficient finite model generators. The source codes of

Mace4 and SEM have been made available to the public since they were created. Since then, both tools have been used by other researchers and ourselves to solve various problems.

Before Mace4, there was a model generator called Mace2 [33] which is based on propositional satisfiability, like ModGen [27] and Paradox [17]. Mace4 is the first released program that has been constructed with LADR (Library for Automated Deduction Research) [36], a library of C routines for building automated deduction tools and a larger project developed by Bill. We will give an overview of model generation for first-order formulas, including the design, implementation and main capabilities, as well as some of its applications. We also give an outline of some future improvements.

2 Model Generation: Basic Concepts and Notations

The satisfiability problem (SAT) in propositional logic, i.e., deciding whether an arbitrary propositional formula is satisfiable, is a well-known NP-complete problem. In general, it is undecidable to decide if a set of first-order formulas is satisfiable. However, this satisfiability problem is so important that it cannot be ignored. Consider traditional automated theorem proving (ATP) in first order logic, an important class of theorem provers work by refutation. That is, the negation of the conjecture is added to the hypothesis and the axioms. Then the resulting set of formulas is shown to be unsatisfiable. It may occur that the conjecture is not valid, in which case a conventional theorem prover may not terminate or terminate but cannot give a clear answer. If we can find a model of the formulas including the negation of the conjecture, it will serve as a counterexample showing that the conjecture does not hold. With this application in his mind, McCune developed Mace4 as a companion of the theorem prover Otter and both tools can use the same input files.

Due to the undecidability in the general case and the high complexity in the case of domains of finite size, there was not much progress in creating effective model generators for decades. In the late 1980's and early 1990's, several serious attempts were made to tackle the problem in first-order logic. Some methods are based on first-order reasoning [31,16,46,23,15,40,5,6]; some are based on constraint satisfaction, e.g., Finder [41], Falcon [53] and SEM [59]; while others are based on propositional logic, e.g., ModGen [27] and Mace2 [33]. Later, other powerful tools have been developed, such as Gandalf [47], Mace4 [35], Paradox [17]. These tools have been used to solve a number of challenging problems in discrete mathematics [42,33,53]. The two programs studied in this article, Mace4 and SEM, are still two of the most powerful model generators, especially on problems with deeply-nested terms.

The models of a formula are interesting in themselves, because they may reveal interesting properties of the formula. This can be very useful, especially when the formula is complicated and people do not have a good understanding of it yet. See, for instance, [53].

Given the undecidability of the problem, we can only solve some subcases by putting certain restrictions on the original problem. Alternatively, we may check the satisfiability of the given formula in certain finite domains. If the size of the domain is a fixed positive integer n, we are certainly able to decide whether a formula has a model of size n or not. If we can find a model of fixed size for a first-order formula, we can say that the formula is satisfiable.

A finite model of first-order formulas is an interpretation of the constants, function symbols and predicates over a finite domain of elements that satisfies the formulas [41]. If no such model exists, then the formulas are unsatisfiable in the given domain. Although the decidability in this case is no longer an issue, it is still a very hard problem to find non-trivial finite models, as propositional satisfiability (SAT) is a special finite model generation problem where the domain size is two. In theory, every finite model generation problem can be encoded as an instance of SAT. However, despite the fact that we have very powerful SAT solvers today, many model generation problems cannot be solved by SAT solvers, because converting them into an SAT instance takes too much computer memory.

In the following, we briefly review some relevant concepts and notations in first-order predicate logic. We have function symbols and predicate symbols, variables and constants. We can construct *terms* and *formulas*. As in resolution-based theorem proving, it is often more convenient to transform first-order formulas into clausal form. A *clause* is a disjunction of literals, where a *literal* is a predicate (applied to as many terms as required by its arity) or the negation of a predicate.

A *model* of a set of clauses is an interpretation of the function symbols (including constant names) and predicate symbols in some (nonempty) domain, such that every clause is evaluated to true. A finite model can be conveniently represented by a set of tables, each of which corresponds to the definition of a function/predicate. If the arity of a function symbol is two, its table is two dimensional; if the arity is three, its table is three dimensional, and etc. Suppose the arity of f is k, each entry in the k-dimensional table can be represented by a unit positive clause $f(a_1, a_2, ..., a_k) = a_0$, where $a_i \in D_n$, called *VA (value assignment) clause* and the ground term $f(a_1, a_2, ..., a_k)$ is called a *cell term*. On the other hand, we call unit negative clause $f(a_1, a_2, ..., a_k) \neq a_0$ a *VE (value elimination) clause*, where a_i is an element of the domain.

2.1 Two Small Examples

Example 2.1 A *Latin square* of size n is an $n \times n$ square whose elements are D_n, the integers in the interval $[0..(n-1)]$. Each row and each column of the square is a permutation of the n integers. Latin squares satisfy the following formulas:

$$\forall x, y, z \ (y \neq z \ \rightarrow \ f(x, y) \neq f(x, z));$$
$$\forall x, y, z \ (x \neq y \ \rightarrow \ f(x, z) \neq f(y, z)).$$

where $\rightarrow$ stands for implication, $f(x, y)$ denotes the cell on the x'th row and the y'th column of the Latin square. In abstract algebra, a *quasigroup* is (D_n, f) sat-

isfying the above formulas, and it is well-known that the "multiplication table" of a quasigroup is a Latin square.

Without loss of generality, we can denote an n-element domain by the integer set $D_n = \{0, 1, 2, \ldots, n-1\}$. The following is a 3-element model of the above axioms, over the domain D_3.

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

This model may be represented by the following VA clauses:

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

In many cases, it is more convenient to use a *many-sorted* language. Sorts are similar to types in programming languages. In such a case, variables, terms, functions are all sorted. The following is a simple example.

Example 2.2 Let us look at the formulas

$$\begin{array}{l} \forall x, y : Dog\,,\ mating(x, y) \rightarrow gender(x) \neq gender(y) \\ \forall x, y : Dog\,,\ mating(x, y) \rightarrow mating(y, x) \end{array}$$

where *mating* is a binary predicate, meaning that the two dogs are the mates in a mating, *gender* is a function that maps *Dog* to *Sex*. So we have the sort *Dog* and the sort $Sex = \{\text{Male, Female}\}$.

One model of the above formula is given below on the domain $Dog = \{\text{Benji, Bailey}\}$ and the function/predicate symbols are interpreted as follows:

gender	Bailey	Benji
	Female	Male

mating	Benji	Bailey
Benji	**F**	**T**
Bailey	**T**	**F**

Here **F** and **T** are `false` and `true`, respectively. Thus the model says that Benji is male, Bailey is female, and one is the mate of the other.

2.2 Input of Mace4 and SEM

The input of Mace4 usually consists of parameter settings and a list of untyped first-order formulas or clauses, which is compatible with the input syntax of Otter. For Example 2.1, the input of Mace4 may look like this:

```
assign(domain_size, 3).
formulas(theory).
y = z | f(x,y) != f(x,z).
x = y | f(x,z) != f(y,z).
end_of_list.
```

SEM's input file for Example 2.1 is very similar to that of Mace4:

```
3.

y = z | f(x,y) != f(x,z).
x = y | f(x,z) != f(y,z).
```

That is, the first line specifies the size of the domain, and the other lines give the clauses. This syntax applies to problems with one sort of data. In general, the input of SEM consists of the following four parts:

$$Sorts \quad Functions \quad Variables \quad Clauses$$

where *Sorts* specify the size of each sort and *Functions* specify the signatures of each function. The *Clauses* often contain some *Variables* which are assumed to be universally quantified over the given finite domains. The third part of the input, i.e., *Variables*, can be omitted. In this case, SEM will try to infer the sorts of the variables from the context.

For Example 2.2, the input of SEM looks like this:

```
% Sorts
( Sex : Female, Male )
( Dog : Bailey, Benji )

% Functions
{ gender: Dog -> Sex }
{ mating: Dog Dog -> BOOL }

% Clauses
[ -mating(x,y) | gender(x) != gender(y) ]
[ -mating(x,y) | mating(y,x) ]
```

Here a line beginning with the character '`%`' is a comment. The symbol "`-`" denotes NOT; "`|`" denotes OR; and the symbol "`!=`" denotes "not-equal".

To facilitate the use of SEM, the second author of this article wrote (in 2000) some Perl scripts [57]. Given a set of (uni-sorted) first-order formulas, the scripts will call Otter to generate a set of clauses, and call SEM to search for models of increasingly larger sizes. This set of scripts can help Otter users to use SEM, even though SEM and Otter are not tightly integrated.

Mace4 accepts only sortless first-order logic formulas (clauses). For Example 2.2, the user has to provide the predicates for `Dog` and `Sex`. On the other hand, because of the simplicity of its input syntax and the popularity of the Otter prover, Mace4's input syntax is more appealing to the community of automated reasoning than that of SEM. Moreover, Mace4 provides many more utility commands to assist the users. For example, the user can specify the command `assign(iterate_up_to,` n`)` so that Mace4 will iteratively increase the domain size one by one, from the current domain size to n, unless a model is found during the process; for each different domain size, Mace4 will restart the search. Mace4

also allows the user to use Prolog style variables in the clauses, or specify the precedence of operators for pretty printing. The user can also set the parameter values on the command line when invoking Mace4. For example, if the input file `group.in` contains the following line:

```
assign(iterate_up_to, 10).
```

then the command

```
% mace4 -n8 -m20 < group.in > group.out
```

tells Mace4 to search for up to 20 models of sizes 8, 9, and 10. There is also an option `-c`, which allows Mace4 to skip certain unrecognized commands. That way, Mace4 can use directly Otter's input files with `-n`, `-m`, and `-c`.

2.3 Preprocessing

The finite model generation problem can be regarded as a constraint satisfaction problem (CSP), where the constraints are a set of clauses with equality and functions. When finding an n-element model, a general practice is to instantiate the input clauses in all possible ways, substituting each of the n elements in the domain for each variable in the clauses. Then we obtain a set of ground clauses, denoted by GC. If a clause has v variables, then the clause has n^v ground instances. That is, the number of ground clauses grows exponentially with the number of variables in a clause. However, working with ground clauses greatly simplifies the needed inference rules. For instance, unification is not necessary and matching becomes identity checking. All the model generation programs, including Finder [41], Falcon [53], Mace4 and SEM, instantiate the input clauses before searching for the model.

Both Mace4 and SEM start by allocating a table for each function and predicate symbol. Constants are allocated a single cell, function symbols of arity 2 get an $n \times n$ table of cells, and so on. For sortless formulas, the range of values for the cells will be members of the domain $\{0, ..., n-1\}$, and values for predicate symbols will be 0 and 1, the boolean values. For the sorted formulas, the range of each cell is the sort of that cell.

3 Basic Search Algorithm in Mace4 and SEM

In principle, a finite model can be found by exhaustive search. We can enumerate all different assignments to the table cells. For each assignment, we check whether all the input clauses hold. If so, the assignment is a model. Obviously, such a procedure is not efficient. A better alternative is to use a backtracking search procedure.

3.1 A Backtracking Search Procedure

Backtracking search procedures are often used to solve finite domain CSPs, including the SAT problem. In this approach, one starts with an empty assignment

and repeatedly assigns a value to a new cell as long as the assignment does not violate the constraints; if a conflict occurs, it backtracks and assigns a different value to the variable. In this way, it explores the search space systematically and exhaustively. The performances of backtracking algorithms can be improved in a number of ways such as forward checking and lookahead [28].

The model generation process in Mace4 and SEM can be described by the recursive procedure in Fig. 1. The procedure uses the following parameters:

- $A \subset CE \times Dom$ is an assignment, i.e., a set of pairs of cells and values, where CE is the set of cells and Dom is the domain of values (elements). A is called a *partial model* and can be represented by a set of VA clauses (i.e., a set of equalities of form $(ce = e)$ where ce is a cell term and e is an element). It should be noted that A is functional, i.e., it may not contain two pairs $(ce = e_1)$ and $(ce = e_2)$ where e_1 and e_2 are different elements.
- $\mathcal{D} \subset CE \times \mathcal{P}(Dom)$, a set of unassigned cells and their possible values, where $\mathcal{P}(Dom)$ is the power set of Dom. In fact, each pair (ce, D) in $\mathcal{D}$ can be represented by a clause $ce = e_1 \vee \ldots \vee ce = e_i \vee \ldots \vee ce = e_k$, where $e_i \in D$. Such clauses are called *PV (possible values) clauses* in [60]. For each cell ce, ce appears uniquely either in A or $\mathcal{D}$, but not both.
- Ψ: constraints (i.e. the ground clauses).

Initially A is empty, Ψ is GC, and $\mathcal{D}$ contains $(ce, Dom(ce))$ for every cell ce. The following procedure will print all models of $(A, \mathcal{D}, \Psi)$ upon termination.

```
proc backtrack_search(A, D, Ψ)
{
   if Ψ = false then return;
   if D = ∅ then /* a model is found */
      { print(A); return; }
   choose and delete (ce_i, D_i) from D;
   for each e ∈ D_i do
      backtrack_search(propagate(A ∪ {(ce_i = e)}, D, Ψ));
}
```

Fig. 1. An abstract backtrack search procedure

The procedure `propagate`$(A, \mathcal{D}, \Psi)$ propagates assignment A in Ψ: It simplifies Ψ and may force some variables in $\mathcal{D}$ to be assigned. In fact, `propagate` is essentially a closure operation with respect to a set of sound inference rules. It repeatedly modifies $A, \mathcal{D}$, and Ψ until no further changes can be made. When it exits, it returns the modified triple $(A, \mathcal{D}, \Psi)$.

The procedure `propagate` repeatedly applies the following basic steps:

1. For each new assignment (ce, e) in A, replace the occurrence of ce in Ψ by e.

2. If there exists an empty clause in Ψ (i.e., each of its literals becomes `false` during the propagation), replace Ψ by `false`, and exit from the procedure. Otherwise, for every unit clause in Ψ (i.e. all but one of its literals become `false`), examine the remaining literal l.
   - l is a cell term ce, $(ce, D) \in \mathcal{D}$: If `true` $\in D$, then delete (ce, D) from $\mathcal{D}$ and add $(ce, \texttt{true})$ to A; otherwise, replace Ψ by `false`, and exit from the procedure.
   - l is the negation of a cell term, i.e. $\neg ce$, $(ce, D) \in \mathcal{D}$: If `false` $\in D$, delete (ce, D) from $\mathcal{D}$ and add $(ce, \texttt{false})$ to A; otherwise, exit with $\Psi =$ `false`.
   - l is of the form $\texttt{EQ}(ce, e)$ (or $\texttt{EQ}(e, ce)$), and $(ce, D) \in \mathcal{D}$: If $e \in D$, delete (ce, D) from $\mathcal{D}$ and add (ce, e) to A; otherwise, exit with $\Psi =$ `false`.
   - l is of the form $\neg\texttt{EQ}(ce, e)$ (or $\neg\texttt{EQ}(e, ce)$), and $(ce, D) \in \mathcal{D}$: If $e \in D$, then delete e from D.
3. For each pair $(ce, D) \in \mathcal{D}$, if $D = \emptyset$, then replace Ψ by `false`; if $D = \{e\}$ (i.e. $|D| = 1$), then delete the pair (ce, D) from $\mathcal{D}$, and add the pair $(ce = e)$ to A.

In the next section, we shall give a formal presentation of the above propagation rules as well as more sophisticated rules.

The execution of the search procedure `backtrack_search` can be represented as a search tree. Each node of the tree corresponds to a call of `backtrack_search` and also a partial model, and each edge corresponds to assigning a value to a cell.

3.2 Avoiding Search for Isomorphic Models

The efficiency of the search procedure depends on many factors. One factor is how we choose a cell and assign a value to it; another factor is how we can perform reasoning after we make a choice. Here we address the former issue first. The latter will be discussed in the next section.

During the search process, there are usually more than one unassigned cells, and we need to choose one of them to assign a value. The performance of the procedure `backtrack_search` is mostly affected by the search strategy. A common strategy is to choose the cell which has the fewest possible values (called the most constrained cell). This is usually quite effective for solving various constraint satisfaction problems, and it is also implemented in Mace4 and SEM.

For a typical finite model generation problem, it is very important to avoid the exploration of certain isomorphic subspaces. Two (partial) models are *isomorphic* if there is a permutation of the domain elements such that the models are the same under the permutation. In general, it is very expensive to eliminate all the isomorphic models during the search. In SEM as in Mace4, the least number heuristic (LNH) is used to explore the symmetry between domain values which produce isomorphic models. This technique turns out to be crucial in the success of solving many model generation problems. Other approaches have also been explored, see, e.g., [11,12].

LNH is based on the observation that in typical model generation problems, the domain elements are "equivalent" when the search begins, because each input formula (or axiom) usually specifies that all the elements have to satisfy some property, without naming any element. As we make choices during the search, some elements become "special", i.e., they are no longer equivalent to or symmetric with other elements as they were assigned to some cells. For the unisorted case, we can use a single integer, mdn, to denote which domain elements are symmetric. The domain is divided into two disjoint subsets, $[0..mdn]$ and $[(mdn + 1)..(n - 1)]$, where n is the size of the domain. The elements in the second subset have not gotten special status, and thus they are symmetric with each other. When we try to assign values from the second subset to a cell, we need only choose one value and assign it to the cell; the other values from the second subset are equivalent to the chosen value.

The value of mdn is (-1) if no domain element appears in the input formulas. It is increased as the search moves on. To keep as many symmetric elements as possible, we try to avoid giving special status to the domain elements as far as we can. This is usually achieved by trying the smallest elements as values for assignment to the cells. Hence the name *least number heuristic* (LNH).

For instance, suppose we have only one binary function symbol, f, and no domain element appears in the input clauses. Then typically the first level of the search tree has two branches:

1. Let $f(0,0) = 0$. We have $mdn = 0$. Then we check the cells $f(0,1)$, $f(1,0)$, $f(1,1)$ and pick up a cell having the smallest number of possible values.
2. Let $f(0,0) = 1$. We have $mdn = 1$. Then we check the cells $f(0,1)$, $f(1,0)$, $f(1,1)$, $f(0,2)$, $f(1,2)$, $f(2,2)$, $f(2,0)$, $f(2,1)$ and pick up a cell having the smallest number of possible values.

In other words, we search for a table of values for f, starting from the upper-left cell (i.e., the subsquare of size 1×1) and increasing the size of the subsquare gradually. There is no need to consider the assignment $f(0,0) = b$ where $b \neq 0, 1$, because if there exists a model in which $f(0,0) = b$, we may obtain a model in which $f(0,0) = 1$ by switching the names of 1 and b in the model (assuming 1 and b do not appear in the input formula). That is the basic idea of LNH.

Since SEM accepts many-sorted input, an array of integers, mdn_0, mdn_1, ..., is used and one integer of the array for each sort. Since both SEM and Mace4 use the idea of LNH, this will make the performances of SEM and Mace4 similar for many examples. The LNH technique has been used and extended by several researchers, including [3,1]. Instead of using as few individual elements as possible to preserve symmetries, Audemard and Henocque proposed the idea of XLNH [3] which heuristically selects and then fully generates the functions that appear in the problem, using a weighted directed graph of functional dependency. Audemard and Henocque experimented with XLNH in SEM and achieved very good results [3].

3.3 Selecting Cells

The *splitting rule* is implicitly given in the procedure `backtrack_search` of Fig. 1. It tries to extend a partial model by examining a selected unassigned cell. If assigning any one value to the cell leads to a complete model, the procedure terminates. The splitting rule "splits" on a possible values (PV) clause, and transforms the search problem into a number of easier subproblems.

The LNH technique forces a model generator to choose certain cells and values in the `backtrack_search` procedure. We first describe SEM's default rule for choosing cells. For cell selection, the rationale is to keep the value of *mdn* as small as possible. For simplicity, suppose there is only one binary function f and one unary function g in the input file. Then SEM will examine the following cells:

```
f(0,0), g(0),
f(0,1), f(1,0), f(1,1), g(1),
...
```

In each line, if there are several cells whose values are unknown, we select the cell whose domain of values is the smallest.

In Mace4, the cell selection is divided into two stages. The first stage, controlled by the parameter `selection_order`, determines a set of candidate cells. There are three values for `selection_order`: 0, 1, or 2 (default). If the value of `selection_order` is 0, then all unassigned cells are candidates. Let us say the *major* of a cell term *ce* is $max\{k_i \mid 1 \leq i \leq n\}$ if *ce* is $f(k_1, k_2, ..., k_n)$. If the value of `selection_order` is 1, and the major of the first unassigned cell is m, then all unassigned cells whose major is less than or equal to m are candidates; if the value is 2, then all unassigned cells whose major is less than or equal to the major of the current maximum constrained unassigned cell are candidates. The first method defeats the purpose of LNH and the last two methods use the idea of LNH and keep maximum constrained value as low as possible.

The second stage is controlled by the parameter `selection_measure`, which takes a value from 0 to 4 (default). If the value is 0, the first candidate is selected; if the value is 1, select the candidate with the greatest number of occurrences in the current set of ground clauses; if the value is 2, select the candidate that would cause the greatest amount of propagation; if the value is 3, select the candidate that would cause the greatest number of contradictions; if the value is 4, select the candidate with the smallest number of possible values, i.e., select (ce, D) with the minimal $|D|$. When the value of `selection_measure` is 2 or 3, the corresponding method is an expensive lookahead operation as for each cell being considered, all assignments and subsequent propagation are tried (and undone), and the statistics are then collected.

The impact of `selection_order` and `selection_measure` on the performance of Mace4 is huge for many examples. For example, to search for a noncommutative group of size 48, we may use the following input:

```
assign(domain_size, 48).
```

```
formulas(theory).
  e * x = x.
  x' * x = e.
  (x * y) * z = x * (y * z).
  A * B != B * A.
end_of_list.
```

Among 15 pairs of different selections for `selection_order` and `selection_measure`, only four pairs will allow Mace4 to find a model in less than one second. These four pairs are (`selection_order`, `selection_measure`) $= (1,0),(2,0),(1,4),(2,4)$; each of the other 11 pairs took Mace4 at least 10 minutes before we gave up trying.

As mentioned above, SEM's default selection strategy is to pick (ce, e) where $max(major(ce), e)$ is minimal. This strategy is different from any of Mace4's 15 strategies controlled by the parameters `selection_order` and `selection_measure`. As a result, SEM and Mace4 do not have exactly the same performance for most problems.

4 Constraint Propagation

It is important to obtain useful information (e.g., contradiction or new assignments) when a certain number of cells are assigned values and this process is called *constraint propagation*. We have to address the following issue: How can we implement constraint propagation efficiently?

This issue may be trivial for some constraint satisfaction algorithms because the constraints they accept are often assumed to be unary or binary (i.e., with one or two variables). 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 new domains, and hence an increase in problem size. This issue is particularly important to model generation because in this case, the constraints are represented by complex formulas. Thus we have to design special constraint propagation rules which should be very helpful to increase the efficiency of finite model searching. By experimenting with SEM, we identify a set of constraint propagation rules that are both efficient and easy to implement. In this section, we present the rules used in Mace4 and SEM.

As mentioned earlier, our goal is to derive a model (which can be represented by a set of VA clauses) such that all the clauses in Ψ are true. Essentially this can be achieved through a sequence of transformations defined by a set of rules [60], including the splitting rule. In addition, we need the following two types of rules:

- **Simplification Rules:** A current clause is simplified or removed from the clause set Ψ. Tautology deletion and subsumption are such rules.
- **Inference Rules:** A new clause is generated and added into the clause set Ψ without changing or removing old clauses. For instance, resolution and paramodulation are two well-known inference rules.

Roughly speaking, simplification rules are inference rules plus deletion (of original references). In the rest of this section, we describe these rules in detail. We assume that all involved clauses are ground.

4.1 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:

$$(\mathbf{ER1})\ \frac{C \cup \{t \neq t \vee M\}}{C \cup \{M\}} \qquad (\mathbf{ER2})\ \frac{C \cup \{e = e' \vee M\}}{C \cup \{M\}}\ (e \neq e')$$

Equality Subsumption:

$$(\mathbf{ES1})\ \frac{C \cup \{t = t \vee M\}}{C} \qquad (\mathbf{ES2})\ \frac{C \cup \{e \neq e' \vee M\}}{C}\ (e \neq e')$$

The above rules are sound, because we have, for all x, $x = x$; and $e \neq e'$ is assumed to be true for any two distinct elements e and e' 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:

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

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

Both **UR1** and **UR2** for ground clauses are more powerful than the Boolean constraint propagation (BCP for short) for propositional clauses[1], in the sense that more information can be derived during propagation.

Example 4.1 From $(f(1) = 0) \vee (f(1) \neq g(2))$ and $(f(1) = g(2))$, **UR1** can deduce $(f(1) = 0)$. However, $(f(1) = g(2))$ will not be represented by a unit propositional clause in the conventional conversion, which will convert $(f(1) = g(2))$ into a set of binary propositional clauses such as $(f(1) \neq a) \vee (g(2) = a)$ and $(f(1) = a) \vee (g(2) \neq a)$, where a is an element in the domain, and these binary clauses cannot be used in BCP. Thus, BCP cannot deduce new assignments from the propositional representation of the above two clauses. Because it is expensive

[1] BCP is the process of repeatedly removing false literals from propositional clauses and making literals in unit clauses to be true.

to perform full subsumption in first-order reasoning, only unit subsumption is considered in SEM.

Unit Subsumption:

$$(\mathbf{US1})\ \frac{C \cup \{t_1 = t_2 \vee M,\ \ t_1 = t_2\}}{C \cup \{t_1 = t_2\}}$$

$$(\mathbf{US2})\ \frac{C \cup \{t_1 \neq t_2 \vee M,\ \ t_1 \neq t_2\}}{C \cup \{t_1 \neq t_2\}}$$

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

Rewriting:

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

Using this rule, **UR1** becomes redundant if **ER1** is used. 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 term, i.e., $t_1 = t_2$ is a VA clause.

After a simplification rule is applied, the size of the clause set often becomes smaller. Some of the clauses may be reduced to VA or VE clauses, so that further simplification can be performed. If an empty clause is produced by the simplification process, that result is returned immediately.

4.2 Inference Rules

For the completeness of the algorithm `backtrack_search`, we do not need any new inference rules as the rule of splitting is implicitly given in `backtrack_search`. However, additional inference rules may help us to avoid some fruitless search space and speed up the search. The first inference rule we consider here involves the equality predicate and it is called *dismodulation*.

Dismodulation:

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

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 both equalities $f(t_1, ..., t_m) = f(s_1, ..., s_m)$ and $t_0 = s_0$ are true in the interpretation.

The above rule is quite similar to negative paramodulation [38][2] in that new information is derived from inequalities. The dismodulation rule in its general form has a 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 (of the form $ce \neq e$). In this case, M_1 and M_2 are empty, one of the two terms, $f(t_1, ..., t_m)$ and $f(s_1, ..., s_m)$, is a cell term, and the other term is like a cell term, except one subterm. Formally, the next two rules are an instance of the dismodulation rule.

VE Generation:

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

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

where e_i, $0 \leq i \leq m$, are elements and ce is a cell term.

Note that, to apply 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. For convenience, we will call the positive unit clause $f(e_1, .., ce, .., e_m) = e_0$ a *near VA clause*, and the negative unit clause $f(e_1, .., e_i, ..., e_m) \neq e_0$ a *near VE clause*.

The above rules of VE generation can be generalized slightly. Let us call a ground term *neat* if either it is a constant, or a cell term or all but one of the top-level function's arguments are elements in the domain and the only non-element argument is also neat. For example, $h(1, g(f(2, 4)), 3)$ is neat, while $h(1, f(2, 4), g(3))$ is not. We may replace the cell term ce in the above rules by a neat term t. The deduced inequality $t \neq e_i$ can participate in further 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 $f(2, 4) \neq 1$.

The VE generation rules are quite useful for some problems having complex terms. One example is the QG5 problem [21,42], which has the following clause: $f(f(f(y, x), y), y) = x$.

The soundness and completeness of the simplification and inference 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.

[2] Given two ground clauses $A \vee (a \neq b)$ and $P(a) \vee B$, where A and B are lists of literals, $P(a)$ is a literal containing the term a, negative paramodulation will infer the clause $A \vee \neg P(b) \vee B$.

Theorem 4.1 The set C of ground clauses has a model over D_n iff $C \Rightarrow^* C'$, where C' is satisfiable and contains a VA clause $ce = e$ for every cell ce.

The proof of this theorem is straightfoward following the completeness of the DPLL algorithm for propositional logic as every inference in the DPLL algorithm can be obtained by our inference rules. Note that the satisfiability of C' is very easy to check when it contains a VA clause $ce = e$ for every cell ce.

In Mace4, there are four flags that a user can enable or disable (the default is true for all four flags) to generate VE clauses. When a new VA or VE clause is generated, these flags are checked; if true, the corresponding rules are applied. That is, new VA or VE clauses serve as triggers of these VE generation rules. These four flags are:

- `neg_assign`: If true and a new VA clause is generated, VEG1 applies.
- `neg_elim`: If true and a new VE clause is generated, VEG2 applies.
- `neg_elim_near`: If true and a new near VE clause is generated, VEG1 applies.
- `neg_assign_near`: If true and a new near VA clause is generated, VEG2 applies.

Good data structures are crucial to the performance of many programs. To facilitate the reasoning and search as described in the previous section, relatively sophisticated data structures are used in both SEM and Mace4. These data structures support all the constraint propagation rules as well as backtracking.

When a cell term or a neat term is assigned a value, we need to find which terms in the ground clauses Ψ can be simplified. In SEM, we use some *occurrence lists* to keep track of those terms. For example, the cell $f(0,1)$ occurs in the following clauses:

$$g(f(0,1)) = 2. \quad f(f(0,1),2) = f(0,f(1,2)).$$

Such an occurrence list changes dynamically during the search. For example, suppose $f(0,1)$ is assigned the value 3. Then the cell $f(3,2)$ will occur in the second clause, although initially it does not. During backtracking, different kinds of information have to be restored, such as the values of cells and the occurrence lists.

In Mace4, in order to quickly locate neat assignments and neat eliminations as well as ordinary assignments and eliminations, Bill used a *complete discrimination tree* [32], that is a discrimination tree in which all possible branches are constructed at the start of the search, and only the leaves are updated for insertions and deletions. For Mace4, this complete discrimination tree is not very big, because the depth of neat terms is limited to 2 (that is, only near VA clauses or near VE clauses are considered). Insertions into the discrimination tree are recorded on the stack so that they are undone on backtracking.

5 Empirical Evaluation

As mentioned in the introduction, satisfiability is an important property of logical formulas. A formula is satisfiable or consistent if we can construct a model. Besides showing the consistency of formulas, a model generation tool can be used in various ways.

5.1 Algebra and Logic

In the introduction, we mentioned that a model can serve as a counterexample showing the non-theoremhood of a formula. Thus a model searcher may be used to identify a

false conjecture. In particular, such a tool may be used to check a set of axioms, showing that one axiom is independent of the other axioms. For example, in the mid-1980s, Allen and Hayes proposed axiomatizing temporal intervals by using a set of five axioms. Galton [22] later developed a three-element structure that satisfies the first three axioms but falsifies the fifth. With SEM, we can find a three-element counterexample [55]. The following is another example of such an application.

Example 5.1 A *Skew lattice* is a non-commutative generalization of a lattice which has two binary operators: $\vee$ and $\wedge$. In [30], Jonathan Leech asked if the following two middle distributive identities are independent:

$$x \wedge (y \vee z) \wedge x = (x \wedge y \wedge x) \vee (x \wedge z \wedge x) \tag{1}$$

$$x \vee (y \wedge z) \vee x = (x \vee y \vee x) \wedge (x \vee z \vee x) \tag{2}$$

Spinks showed that for the Skew lattice identities, neither (1) nor (2) implies the other [44]. This was proved by finding some 9-element models using SEM.

To show that (1) does not imply (2), let `f` denote $\wedge$ and `g` denote $\vee$, we give the following input to SEM:

```
9.
f(x,f(y,z)) = f(f(x,y),z).
g(x,g(y,z)) = g(g(x,y),z).
f(x,x) = x.
g(x,x) = x.
f(x,g(x,y)) = x.
g(x,f(x,y)) = x.
f(g(y,x),x) = x.
g(f(y,x),x) = x.
f(f(x,g(y,z)),x) = g(f(f(x,y),x),f(f(x,z),x)).
g(g(a,f(b,c)),a) != f(g(g(a,b),a),g(g(a,c),a)).
```

A model can be found instantly on a personal computer.

The generation of counter-examples is not always so easy.

Example 5.2 Tarski's *High School Problem*: Can the following set of identities form a basis for all the identities which hold for the natural numbers?

$$\begin{aligned}
x + y &= y + x \\
x + (y + z) &= (x + y) + z \\
x * 1 &= x \\
x * y &= y * x \\
x * (y * z) &= (x * y) * z \\
x * (y + z) &= (x * y) + (x * z) \\
1^x &= 1 \\
x^1 &= x \\
x^{y+z} &= x^y * x^z \\
(x * y)^z &= x^z * y^z \\
(x^y)^z &= x^{y*z}
\end{aligned}$$

Wilkie gave an identity which holds for the natural numbers, but cannot be derived from the above identities through equational reasoning:

$$(P^x + Q^x)^y * (R^y + S^y)^x = (P^y + Q^y)^x * (R^x + S^x)^y$$

where $P = 1+x$, $Q = 1+x+x*x$, $R = 1+x*x*x$, and $S = 1+x*x+x*x*x*x$. An interesting question arises: What is the smallest counter-example for Wilkie's identity? Using SEM, we obtained some empirical results, which show that the smallest counter-example has at least 11 elements [58]. But the problem is far from solved. It is still very challenging to find a 12-element counter-example, although it does exist [14].

A model searcher can be useful even when there is no conjecture. In fact, it can be used to formulate conjectures [56]. If a formula is satisfiable, we can learn something about the formula by examining its models. For instance, Kunen [29] obtained some structure theorems for conjugacy closed loops (CC-loops). During that study, he used SEM as well as the theorem prover Otter [34]. The following is quoted from [29]:

> SEM is used to construct finite examples. ... Once one has such an example, it is usually possible to describe it in a more conceptual way ... We originally tried to use SEM to construct a non-group CC-loop of order 15, but this failed, proving that there was no such loop. We then found the proof in this article ... Otter and SEM were very useful for quick experimentation and for checking out (often false) conjectures.

From the above, we see that a finite model searcher is complementary to a refutation-based theorem prover. These two classes of tools can be combined to perform more powerful reasoning than a single tool can. In particular, they may form a kind of "decision procedures" for certain logics like propositional modal logics [54].

5.2 Quasigroup Problems

In Example 2.1, we showed how to specify a Latin square. From the algebraic point of view, a Latin square indexed by S defines a cancellative groupoid $(S, *)$, where $*$ is a binary operation on S and the Latin square is the "multiplication table" of $*$. Such a groupoid is called *quasigroup*. For many problems in combinatorics, people are interested in quasigroups with additional constraints. For instance, the following constraints are presented in [21,42,49,50].

Code Name	Identity
QG3	$(x*y)*(y*x) = x$
QG4	$(y*x)*(x*y) = x$
QG5	$((y*x)*y)*y = x$

In [21,42], some previously open problems regarding the existence of quasigroups were solved for the first time, such as QG3.12 and QG4.12, a quasigroup of size 12 satisfying the constraints QG3 or QG4. Mace2 and SEM have been used to solve some previously open quasigroup problems. In particular, Mace2 found for the first time some incomplete Latin squares.

Example 5.3 In an incomplete Latin square, a subsquare is missing. The existence of these incomplete Latin squares is very useful for the construction of larger Latin squares. For instance, an incomplete Latin square with a hole of size two can be specified by the following SEM input file:

```
( elem [10] )
{ h : elem elem -> BOOL }
{ f : elem elem -> elem }
```

```
< x, y, z : elem >
[ h(0,0) ]
[ h(0,1) ]
[ h(1,0) ]
[ h(1,1) ]
[ -h(x,y) | EQ(x,0) | EQ(x,1) ]
[ -h(y,x) | EQ(x,0) | EQ(x,1) ]
[ h(x,z) | h(y,z) | -EQ(f(x,z),f(y,z)) | EQ(x,y) ]
[ h(x,y) | h(x,z) | -EQ(f(x,y),f(x,z)) | EQ(y,z) ]
[ EQ(f(x,x), x) ]
[ h(x,y) | EQ(f(f(y,x),f(x,y)), x) ]
```

where the meaning of $h(x, y)$ is that $h(x, y)$ is true if and only if the cell (x, y) is in the hole.

The problem for the above input is called IQG4.10.2, meaning a Latin square of size 10 with a hole of size two satisfying QG4. Mace2 was the first to find a model of IQG4.14.2. The only remaining open case for all IQG4.$n.k$ is when $(n, k) = (19, 2)$ [62].

5.3 Experimental Results

Now we compare the performance of Mace4 and SEM (all with the default parameters – unless specified otherwise). The results are given in Table 1. The problem name `ncg.12` stands for the non-commutative group problem of size 12. That is, the problem asks to find a non-commutative group of size 12.

Table 1. Performance Comparison

Problem	Sat	SEM	Mace4
skewl.9	Yes	0	0.00
ncg.12	Yes	0	0.02
ncg.13	No	1	1.11
ncg.48	Yes	?	0.49
QG3.9	No	?	1.88
QG3.10	No	?	297
QG4.9	Yes	0	0.03
QG4.10	No	128	259.20
IQG4.10.2	Yes	574	9.65
IQG4.11.2	Yes	> 3600	35.01
QG5.10	No	0	0.00
QG5.11	Yes	35	0.72
QG5.12	No	> 600	207.52

It can be seen that Mace4 is usually faster than SEM. But the performance of Mace4 also depends on the settings. For example, if we do not use the negative propagation rule (by specifying `clear(negprop)`), Mace4 will be much slower on the QG4 problem. For QG4.10, Mace4 does not terminate within 20 minutes.

There are two alternative ways of specifying a Latin square $(Q, *)$. One way is to use the single operator $*$ satisfying

$$((x * z) \neq (y * z)) \vee (x = y)$$
$$((x * y) \neq (x * z)) \vee (y = z)$$

The other way is to use three operators, $*$, $\backslash$, and $/$, satisfying

$$x * (x\backslash y) = y$$
$$x\backslash(x * y) = y$$
$$(x/y) * y = x$$
$$(x * y)/y = x$$

This is because for any Latin square $(Q, *)$, for any $a, b \in Q$, there exists a unique solution for the equation $a * x = b$. The value of $(a\backslash b)$ denotes this unique solution. Similarly, (a/b) denotes the unique solution for the equation $y * a = b$. Once the value of $*$ is decided, the values of $\backslash$ and $/$ can be decided as well.

The experiment shows that the later specification works better for model generators like Mace4 and SEM. For SAT solvers, the first specification is usually used to generate propositional clauses, as the second specification will use two times more propositional variables.

```
% quasigroup axioms (equational)
op(400, infix, [*,\,/]).

clauses(theory).
  x * (x \ y) = y.
  x \ (x * y) = y.
  (x / y) * y = x.
  (x * y) / y = x.
  (x * x) = x.
end_of_list.
```

For instance, to search for IQG4.10.2, Mace4 uses only one tenth of the time of the original formulation. For unsatisfiable problems, the computing times are roughly the same for both formulations. This example suggests that we may try to use different formulations to increase the chance of finding a model.

6 A Challenging Latin Square Problem

In this section, we report a success of solving a long-standing open Latin square problem using the finite model generators SEM and Mace4. Using these tools, we found for the first time a Latin square of size eleven with a missing hole of size three, which is orthogonal to its $(3, 2, 1)$-conjugate and both its main and back diagonals are distinct symbols. With the generation of this Latin square, we completely settled the existence problem for such Latin squares for all sizes.

6.1 Basic Concepts

A *transversal* in a Latin square is a set of positions, one per row and one per column, among which the symbols occur precisely once each. A *diagonal Latin square* is a Latin

square whose main and back diagonals are transversals. Two Latin squares $(Q, *)$ and $(Q, \otimes)$ are *orthogonal* if for any $x, y \in Q$, there exists unique $a, b \in Q$ such that $a * b = x$ and $a \otimes b = y$. That is, each symbol x in the first square meets each symbol y in the second square exactly once when the two Latin squares are superposed.

If $(Q, \otimes)$ is a Latin square, we may define on the set Q six binary operations $\otimes_{(1,2,3)}, \otimes_{(1,3,2)}, \otimes_{(2,1,3)}, \otimes_{(2,3,1)}, \otimes_{(3,1,2)}$, and $\otimes_{(3,2,1)}$ as follows: $x \otimes y = z$ if and only if

$$x \otimes_{(1,2,3)} y = z, \quad x \otimes_{(1,3,2)} z = y, \quad y \otimes_{(2,1,3)} x = z,$$
$$y \otimes_{(2,3,1)} z = x, \quad z \otimes_{(3,1,2)} x = y, \quad z \otimes_{(3,2,1)} y = x.$$

These six (not necessarily distinct) quasigroups $(Q, \otimes_{(i,j,k)})$ are called the *conjugates* of $(Q, \otimes)$ ([45,10]).

Example 6.1 Here are the six conjugates of a small Latin square:

(a)	(b)	(c)	(d)	(e)	(f)
1 4 2 3	1 2 4 3	1 3 2 4	1 2 4 3	1 3 4 2	1 3 2 4
2 3 1 4	4 3 1 2	2 4 1 3	3 4 2 1	3 1 2 4	3 1 4 2
4 1 3 2	2 1 3 4	4 2 3 1	2 1 3 4	2 4 3 1	4 2 3 1
3 2 4 1	3 4 2 1	3 1 4 2	4 3 1 2	4 2 1 3	2 4 1 3

(a) a Latin square; (b) its $(2, 1, 3)$-conjugate; (c) its $(3, 2, 1)$-conjugate; (d) its $(2, 3, 1)$-conjugate; (e) its $(1, 3, 2)$-conjugate; (f) its $(3, 1, 2)$-conjugate. Here the domain of elements is $\{1, 2, 3, 4\}$.

For more information on Latin squares and quasigroups, the interested reader may refer to the book of Dénes and Keedwell[18].

For convenience, we denote by (i, j, k)-CODLS(v) a Latin square A of order v, such that A is orthogonal to B, which is the (i, j, k)-conjugate of A and both A and B are diagonal, where $\{i, j, k\} = \{1, 2, 3\}$.

Let $H \subset Q$, $n = |H|$, $v = |Q|$, and $(Q, \otimes)$ be a Latin square with the symbols indexed by H missing from $(Q, \otimes)$ and the missing symbols consist of a missing subsquare right in the center of $(Q, \otimes)$. If we fill the missing square with an (i, j, k)-CODLS(n) over the subset H and the result is an (i, j, k)-CODLS(v) over Q, we say $(Q, \otimes)$ is an *incomplete (i, j, k)-conjugate orthogonal diagonal Latin square of order v with a missing subsquare of size n*, denoted by (i, j, k)-ICODLS(v, n). The subset H as well as the missing subsquare is called the *hole* of $(Q, \otimes)$.

Example 6.2 Here (a) is a $(3, 2, 1)$-ICODLS$(5, 1)$, where $Q = \{0, 1, 2, 3, 4\}$, $H = \{2\}$; if we fill 2 into the hole $\{2\}$, we obtain a $(3, 2, 1)$-CODLS(5). (b) is the $(3, 2, 1)$-conjugate of (a), which is also diagonal. (c) is a $(3, 2, 1)$-ICODLS$(8, 2)$; (d) is the $(3, 2, 1)$-conjugate of (c).

(a)

$\otimes$	0	1	2	3	4
0	4	2	0	3	1
1	0	3	1	4	2
2	1	4		0	3
3	2	0	3	1	4
4	3	1	4	2	0

(b)

$\otimes_{321}$	0	1	2	3	4
0	1	3	0	2	4
1	2	4	1	3	0
2	3	0		4	1
3	4	1	3	0	2
4	0	2	4	1	3

(c)

0	7	5	6	2	3	4	1
7	1	6	5	0	4	2	3
5	6	2	7	1	0	3	4
2	0	1			6	7	5
1	2	0			7	5	6
4	3	7	0	6	5	1	2
3	5	4	2	7	1	6	0
6	4	3	1	5	2	0	7

(d)

0	3	4	5	1	2	7	6
4	1	3	7	2	6	5	0
3	4	2	6	0	7	1	5
6	5	7			0	2	1
5	7	6			1	0	2
2	6	0	1	7	5	4	3
7	2	1	0	5	3	6	4
1	0	5	2	6	4	3	7

It is easy to see that the existence of an (i, j, k)-CODLS(v, n) requires that $v-n$ be even because the missing subsquare must be in the center. The following result is known.

Theorem 6.3 (*a*) *A* $(3, 2, 1)$-*CODLS*(v) *exists for all integers* $v \geq 1$, *except* $v \in \{2, 3, 6\}$ *and except possibly* $v = 10$.

(*b*) *For any integer* $1 \leq n \leq 6$, *a* $(3, 2, 1)$-*ICODLS*(v, n) *exists if and only if* $v \geq 3n+2$ *and* $v - n$ *is even, with the possible exception of* $(v, n) = (11, 3)$.

(*c*) *For any integer* $n \geq 1$, *a* $(3, 2, 1)$-*ICODLS*(v, n) *exists if* $v \geq 13n/4 + 93$ *and* $v - n$ *is even.*

We are able to remove the only possible exception of Theorem 6.3(*b*) by constructing a $(3, 2, 1)$-ICODLS$(11, 3)$ using finite model generators.

6.2 Specification of (3, 2, 1)-ICODLS(11, 3)

To specify that a Latin square is orthogonal to its $(3, 2, 1)$-conjugate, we may use the constraint $QG1$ as given in [49]:

$$QG1 : (x * y = z * w \wedge u * y = x \wedge u * w = z) \Rightarrow (x = z \wedge y = w)$$

For model generators like SEM and Mace4, the preprocessing process before the search is to obtain ground instances of the first-order formulas. Since the number of ground instances grows exponentially in the terms of the number of variables, it is better to use the equivalent constraint $QG1a$ where we have three variables instead of five:

$$QG1a : (u * y) * y \neq (u * w) * w \vee y = w$$

For SAT solvers, an alternative way is to introduce a new predication Φ and use the following two clauses containing four variables each [49]:

$$QG1b : x * y = s \wedge t * y = x \Rightarrow \Phi(x, s, t),$$
$$\Phi(x, s, t) \wedge \Phi(z, s, t) \Rightarrow x = z.$$

For the $(3, 2, 1)$-ICODLS$(11, 3)$ problem, this new specification will need 4394 propositional variables instead of 2197, but the number of propositional clauses is reduced from 659,994 to 56,004. It is shown in [49] that $QG1b$ is much better than $QG1$ for SAT solvers.

To specify both a Latin square and its $(3, 2, 1)$-conjugate are diagonal, that is, all the symbols on the main and back diagonals are distinct, we may use the following clauses:

$$(x * x) \neq (y * y) \vee x = y$$
$$(x * (n - 1 - x) \neq (y * (n - 1 - y) \vee x = y$$
$$z * x \neq x \vee z * y \neq y \vee x = y$$
$$z * (n - 1 - x) \neq x \vee z * (n - 1 - y) \neq y \vee x = y$$

where n is the size of the Latin square and the domain of x and y is assumed to be $\{0, 1, ..., n - 1\}$.

Finally, to specify the hole of a Latin square, we define a binary predicate $h(x, y)$ such that (x, y) is in the hole if and only if $h(x, y)$ is true. For the $(3, 2, 1)$-ICODLS$(11, 3)$ problem, the hole is the set $\{4, 5, 6\}$ and $h(x, y)$ can be specified by the following clauses:

$$\neg h(x,y) \vee h(y,x)$$
$$h(4,4)$$
$$h(4,5)$$
$$h(4,6)$$
$$h(5,5)$$
$$h(5,6)$$
$$h(6,6)$$
$$\neg h(x,y) \vee x = 4 \vee x = 5 \vee x = 6$$
$$\neg h(y,x) \vee x = 4 \vee x = 5 \vee x = 6$$

For every constraint P involving $x * y$, we replace it by $P \vee h(x,y)$. For instance, the diagonal constraint becomes

$$(x * x) \neq (y * y) \vee x = y \vee h(x,x) \vee h(y,y)$$
$$(x * (n-1-x) \neq (y * (n-1-y) \vee x = y \vee h(x,(n-1-x)) \vee h(y,(n-1-y))$$
$$z * x \neq x \vee z * y \neq y \vee x = y \vee h(z,x) \vee h(z,y)$$
$$z * (n-1-x) \neq x \vee z * (n-1-y) \neq y \vee x = y \vee h(x,(n-1-x)) \vee h(y,(n-1-y))$$

All the formulas in this section can be easily accepted by today's model generators like SEM and Mace4.

6.3 Construction of (3, 2, 1)-ICODLS(11, 3)

We used the `arithmetic` option of Mace4 so that the built-in functions for arithmetic operations are computed in the ground instances of the formulas. We add a new built-in function named `SUB`(x,y) for $x + (-y)$. Because $*$ and / are also built-in functions, we use @ to replace $*$, f for \ and g for /. The default value for `selection_order` is 0 when `arithmetic` is set; we overwrite it with the command `assign(selection_order, 2)` for better performance. The following is the input file for Mace4.

```
set(arithmetic).
assign(domain_size, 11).
assign(selection_order, 2).

op(400, infix, [@,g,f]).

formulas(theory).

% define hole
h(4,4).
h(4,5).
h(4,6).
h(5,5).
h(5,6).
h(6,6).
-h(y,x) | h(x,y).
-h(x,y) | x = 4 | x = 5 | x = 6.
-h(y,x) | x = 4 | x = 5 | x = 6.
x @ y != z | -h(x,z).
x @ y != z | -h(y,z).
```

```
% define main and back diagonals
x = y | (x @ x) != (y @ y) | h(x,x) | h(y,y).
x = y | (x @ SUB(10, x)) != (y @ SUB(10, y)) | h(x,SUB(10,x)) |
                              h(y, SUB(10, y)).
x = y | z @ x != x | z @ y != y | h(z,x) | h(z,y).
x = y | z @ SUB(10, x) != x | z @ SUB(10, y) != y | h(x,SUB(10,x)) |
                                                   h(y, SUB(10, y)).

% quasigroup axioms (equational)
 x @ (x g y) = y | h(x,y).
 x g (x @ y) = y | h(x,y).
(x f y) @ y = x | h(x,y).
(x @ y) f y = x | h(x,y).

% Orthogonal to its (3,2,1)-conjugate
(u @ y) @ y != (u @ w) @ w | y = w | h(u,y) | h(u,w) | h(w,y).
end_of_list.
```

For many model generation problems, once the specification is complete, our job is done and we let the model generator do the work. However, for challenging problems like $(3, 2, 1)$-ICODLS$(11, 3)$, running Mace4 continuously for over two weeks may not find a solution. One of the reasons that SEM and Mace4 are efficient because they can detect certain symmetry of models and reduce the search space. In SEM as well as in Mace4, the least number heuristic (LNH) is used to reduce the search space based on the exploration of symmetries between domain values which produce isomorphic models. This technique is crucial in the success of solving many model generation problems. Since the values $4, 5, 6$ and 10 appear in the specification, it makes the original LNH technique ineffective. However, the extended LNH technique as presented in [2,17] works perfectly by declaring that 0..3, 4..6, and 7..10 are the sets of equivalent values such that each set has its own least used number (for the set of 7..10, we keep track of the greatest used number).

With the improved built-in function and least number heuristic, Mace4 could find a model of $(3, 2, 1)$-ICODLS$(11, 3)$ after roughly 480 hours of running. On the other hand, we have run the SAT solvers Minisat [20] and Glucose [4] continuously for two weeks without success. The propositional clauses used all the known advanced techniques [17] (4,394 variables and 56,004 clauses).

Here is a part of the output of Mace4.

```
============================== DOMAIN SIZE 11 ========================

============================== MODEL =================================

interpretation( 11, [number=1, seconds=1154566], [

        function(@(_,_), [
               3, 9, 2, 5, 7, 8, 0,10, 6, 4, 1,
               0, 2, 9, 6, 8, 7, 1, 3, 4,10, 5,
               8, 4, 7, 3,10, 1, 2, 5, 9, 6, 0,
               4, 6, 5,10, 9, 0, 3, 2, 8, 1, 7,
               9, 0,10, 8, 0, 0, 0, 1, 7, 3, 2,
               1,10, 0, 2, 0, 0, 0, 9, 3, 7, 8,
```

```
              2, 7, 1, 9, 0, 0, 0, 0,10, 8, 3,
              5, 3, 4, 0, 1,10, 7, 8, 2, 9, 6,
              6, 5, 3, 7, 0, 9, 8, 4, 1, 2,10,
             10, 8, 6, 1, 2, 3, 9, 7, 5, 0, 4,
              7, 1, 8, 4, 3, 2,10, 6, 0, 5, 9 ]),

       function(f(_,_), [
              1, 4, 5, 7, 8, 3, 0, 6,10, 9, 2,
              5,10, 6, 9, 7, 2, 1, 4, 8, 3, 0,
              6, 1, 0, 5, 9,10, 2, 3, 7, 8, 4,
              0, 7, 8, 2,10, 9, 3, 1, 5, 4, 6,
              3, 2, 7,10, 0, 0, 0, 8, 1, 0, 9,
              7, 8, 3, 0, 0, 0, 0, 2, 9,10, 1,
              8, 3, 9, 1, 0, 0, 0,10, 0, 2, 7,
             10, 6, 2, 8, 0, 1, 7, 9, 4, 5, 3,
              2, 9,10, 4, 1, 0, 8, 7, 3, 6, 5,
              4, 0, 1, 6, 3, 8, 9, 5, 2, 7,10,
              9, 5, 4, 3, 2, 7,10, 0, 6, 1, 8 ]),

       function(g(_,_), [
              6,10, 2, 0, 9, 3, 8, 4, 5, 1, 7,
              0, 6, 1, 7, 8,10, 3, 5, 4, 2, 9,
             10, 5, 6, 3, 1, 7, 9, 2, 0, 8, 4,
              5, 9, 7, 6, 0, 2, 1,10, 8, 4, 3,
              1, 7,10, 9, 0, 0, 0, 8, 3, 0, 2,
              2, 0, 3, 8, 0, 0, 0, 9,10, 7, 1,
              7, 2, 0,10, 0, 0, 0, 1, 9, 3, 8,
              3, 4, 8, 1, 2, 0,10, 6, 7, 9, 5,
              4, 8, 9, 2, 7, 1, 0, 3, 6, 5,10,
              9, 3, 4, 5,10, 8, 2, 7, 1, 6, 0,
              8, 1, 5, 4, 3, 9, 7, 0, 2,10, 6 ]),

       relation(h(_,_), [
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
              0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
              0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ])
]).

============================== end of model ==========================
```

The above result was checked by Sato [48,51] to ensure its correctness.

Since we have found a $(3, 2, 1)$-ICODLS$(11, 3)$, we can now state an improvement of Theorem 6.3(b) as follows:

Theorem 6.4 *For any positive integer* $1 \leq n \leq 6$*, a* $(3,2,1)$*-ICODLS*(v,n) *exists if and only if* $v \geq 3n+2$ *and* $v-n$ *is even.*

There are sitll many open problems of Latin squares [50]. For instance, the existence of $(3,2,1)$-CODLS(10) in Theorem 6.3(a) is still unknown and we believe that it does not exist as we have searched for it for a long time. Other types of (i,j,k)-ICODLS(v,n) have been studied by several researchers. Du [19] and Bennett, Du and Zhang [9] investigated $(2,1,3)$-ICODLS(v,n). It is proved that a $(2,1,3)$-CODLS(v) exists if and only if $v \notin \{2,3,6\}$. For any positive integers v and n, a $(2,1,3)$-ICODLS(v,n) exists if and only if $v \geq 3n+2$ and $v-n$ is even, except possibly for $(v,n) \in \{(20,6),(26,8),(32,10)\}$. We believe that modern model generators can play an important role in solving these open problems. On the other hand, attacking these problems will motivate us to develop new techniques for model generation.

7 Conclusion and Future Works

Finite model generation is an important part of automated reasoning. Yet, the progress on new techniques and tools for finite model generation has been slow, as compared with those for solving the propositional satisfiability problem. This article gives an overview of the model generators Mace4 and SEM. In particular, we have compared the two generators on their common parts and differences, and provided some experimental results.

On one hand, we think that Mace4 and SEM can be regarded as successful, because they are quite efficient on some problems and are used by many researchers. On the other hand, their performance on many problems is not so desirable comparing to the SAT based model generators, as the latter can handle well propositional formulas while the former handles well larger equations.

In this section, we discuss several ways in which a finite model generator can be improved, and mention some related works.

- Congruence closure [39]. When designing Mace4 [35], Bill McCune had considered using the ground *congruence closure* algorithm to increase the propagation of constraints. We conducted an experimental study [61] and the results show that using the congruence closure algorithm can reduce the search space for some benchmark problems, as the congruence closure algorithm can deduce some useful information for finite model searching.
- Isomorphism elimination. Although LNH is already very effective in pruning the search space, there can still be some improvements. For example, it is possible to increase the effect of LNH by selecting cells and values appropriately, as proposed by McCune. An extended version of LNH, called XLNH, is suggested in [3]. It is very effective for solving problems having unary bijective functions. Another technique called DASH is described in [25].
- Cell/value selection heuristics. Gandalf [47] uses more sophisticated selection strategies. At each node of the search tree, it examines all the unassigned cells. For each cell, the tool tries every possible value and checks all the consequences. Then the tool either makes an assignment directly, or makes a choice based on a weighted sum of the numbers of consequences.
- Propagation rules. It is still necessary to design and implement more powerful propagation rules. For example, we need better rules for propagating the effects of negative unit clauses. Some work has been done in this area [7,2].

- Learning. Learning from previous failures during the search has been proven to be very useful in increasing the efficiency of SAT solvers. This technique has been investigated for finite model generators in [41] and [24].
- Data structures. It may be worthwhile to design other data structures for representing clauses and cells.

The above list is certainly not exhaustive. With the emergence of new applications, other issues may arise. We hope that more people will get interested in finite model searching, and more powerful tools will be developed.

Acknowledgments. The authors would like to thank the anonymous reviewers for their detailed comments.

References

1. Audemard, G., Benhamou, B.: Reasoning by Symmetry and Function Ordering in Finite Model Generation. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, p. 226. Springer, Heidelberg (2002)
2. Audemard, G., Benhamou, B., Henocque, L.: Predicting and Detecting Symmetries in FOL Finite Model Search. Journal of Automated Reasoning 36(3), 177–212 (2006)
3. Audemard, G., Henocque, L.: The eXtended Least Number Heuristic. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, p. 427. Springer, Heidelberg (2001)
4. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solver. In: Twenty-First International Joint Conference on Artificial Intelligence, IJCAI 2009 (2009)
5. Baumgartner, P., Fuchs, A., De Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. J. of Applied Logic 7(1), 58–74 (2009)
6. Baumgartner, P., Tinelli, C.: The Model Evolution Calculus. In: Baader, F. (ed.) CADE-19. LNCS (LNAI), vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
7. Benhamou, B., Henocque, L.: A new method for finite model search in equational theories: FMSET system. Fundamenta Informaticae 39(1,2), 21–38 (1999)
8. Bennett, F.E., Du, B., Zhang, H.: Existence of conjugate orthogonal diagonal Latin squares. J. Combin. Designs 5, 449–461 (1997)
9. Bennett, F.E., Du, B., Zhang, H.: Existence of self-orthogonal diagonal Latin squares with a missing subsquare. Discrete Math. 261, 69–86 (2003)
10. Bennett, F.E., Zhu, L.: Conjugate-orthogonal Latin squares and related structures. In: Dinitz, J., Stinson, D. (eds.) Contemporary Design Theory: A Collection of Surveys, pp. 41–96. Wiley, New York (1992)
11. Boy de la Tour, T.: Up-to-Isomorphism Enumeration of Finite Models - The Monadic Case. In: Bonacina, M.P., Furbach, U. (eds.) International Workshop First-Order Theorem Proving (FTP 1997). RISC-Linz Report Series No. 97-50, pp. 29–33. Schloss Hagenberg by Linz, Austria (1997)
12. Boy de la Tour, T.: Some Techniques of Isomorph-Free Search. In: Campbell, J., Roanes-Lozano, E. (eds.) AISC 2000. LNCS (LNAI), vol. 1930, pp. 240–252. Springer, Heidelberg (2001)
13. Bürckert, H.-J., Herold, A., Kapur, D., Siekmann, J.H., Stickel, M., Tepp, M., Zhang, H.: Opening the AC-unification race. J. of Automated Reasoning (4), 465–474 (1988)

14. Burris, S., Yeats, K.: The saga of the high school identities. Algebra Universalis 52, 325–342 (2004)
15. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building. Applied Logic Series, vol. 31. Kluwer Academic Publisher (2004)
16. Caferra, R., Zabel, N.: Extending Resolution for Model Construction. In: van Eijck, J. (ed.) JELIA 1990. LNCS, vol. 478, pp. 153–169. Springer, Heidelberg (1991)
17. Claessen, K., Sörensson, N.: New techniques that improve Mace-style finite model finding. In: Model Computation – Principles, Algorithms, Applications, CADE-19 Workshop W4, Miami, Florida, USA (2003)
18. Dénes, J., Keedwell, A.D.: Latin squares and their applications. Academic Press, New York (1974)
19. Du, B.: Self-orthogonal diagonal Latin square with missing subsquare. JCMCC 37, 193–203 (2001)
20. Een, N., Svrensson, N.: Minisat: A SAT solver with conflict-clause minimization. In: SAT 2005 (2005) (poster paper)
21. Fujita, M., Slaney, J., Bennett, F.: Automatic generation of some results in finite algebra. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI 1993), pp. 52–57 (1993)
22. Galton, A.: Note on a lemma of Ladkin. Journal of Logic and Computation 6(1), 1–4 (1996)
23. Hasegawa, R., Koshimura, M., Fujita, H.: MGTP: A Parallel Theorem Prover Based on Lazy Model Generation. In: Kapur, D. (ed.) CADE-11. LNCS, vol. 607, pp. 776–780. Springer, Heidelberg (1992)
24. Huang, Z., Zhang, H., Zhang, J.: Improving first-order model searching by propositional reasoning and lemma learning. In: The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, BC, Canada (May 2004)
25. Jia, X., Zhang, J.: A Powerful Technique to Eliminate Isomorphism in Finite Model Search. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 318–331. Springer, Heidelberg (2006)
26. Kapur, D., Zhang, H.: An Overview of RRL: Rewrite Rule Laboratory. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 513–529. Springer, Heidelberg (1989)
27. Kim, S., Zhang, H.: ModGen: Theorem proving by model generation. In: Proc. of National Conference of American Association on Artificial Intelligence (AAAI 1994), Seattle, WA, pp. 162–167. MIT Press (1994)
28. Kumar, V.: Algorithms for constraint satisfaction problems: A survey. AI Magazine 13(1), 32–44 (1992)
29. Kunen, K.: The structure of conjugacy closed loops. Transactions of the American Mathematical Society 352(6), 2889–2911 (2000)
30. Leech, J.: Skew lattices in rings. Algebra Universalis 26, 48–72 (1989)
31. Manthey, R., Bry, F.: SATCHMO: A Theorem Prover Implemented in Prolog. In: Lusk, E., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 415–434. Springer, Heidelberg (1988)
32. McCune, W.: Experiments with discrimination tree indexing and path indexing for term retrieval. J. of Automated Reasoning 9(2), 147–167 (1992)
33. McCune, W.: MACE 2.0 Reference Manual and Guide. Technical Memorandum No. 249, ANL/MCS-TM-249, Argonne National Lab, Argonne, IL, USA (1994), `http://www-unix.mcs.anl.gov/AR/mace2/`

34. McCune, W.: Otter 3.3 Reference Manual, Technical Memorandum No. 263, Argonne National Laboratory, Argonne, IL, USA (August 2003), http://www-unix.mcs.anl.gov/AR/otter/otter33.pdf
35. McCune, W.: Mace4 reference manual and guide, Technical Memorandum No. 264, Argonne National Laboratory, Argonne, IL, USA (August 2003), http://www.cs.unm.edu/~mccune/prover9/
36. McCune, W.: Library for Automated Deduction Research (2009), http://www.cs.unm.edu/~mccune/prover9/
37. McCune, W.: Solution of the Robbins problem. J. of Automated Reasoning 19(3), 263–276 (1997)
38. McCune, W., Henschen, L.J.: Experiments with semantic paramodulation. J. of Automated Reasoning 1(3), 231–261 (1985)
39. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM 27(2), 356–364 (1980)
40. Pichler, R.: Algorithms on Atomic Representations of Herbrand Models. In: Dix, J., Fariñas del Cerro, L., Furbach, U. (eds.) JELIA 1998. LNCS (LNAI), vol. 1489, pp. 199–215. Springer, Heidelberg (1998)
41. Slaney, J.: Finder: Finite Domain Enumerator. In: Bundy, A. (ed.) CADE-12. LNCS, vol. 814, pp. 798–801. Springer, Heidelberg (1994)
42. Slaney, J., Fujita, M., Stickel, M.: Automated reasoning and exhaustive search: Quasigroup existence problems. Computers & Math. with Appl. 29(2), 115–132 (1995)
43. Slaney, J., Lusk, E.L., McCune, W.: SCOTT: Semantically Constrained Otter (System Description). In: Bundy, A. (ed.) CADE-12. LNCS, vol. 814, pp. 764–768. Springer, Heidelberg (1994)
44. Spinks, M.: On middle distributivity for Skew lattices. Semigroup Forum 61(3), 341–345 (2000)
45. Stein, S.K.: On the foundations of quasigroups. Trans. Amer. Math. Soc. 85, 228–256 (1957)
46. Tammet, T.: Using Resolution for Deciding Solvable Classes and Building Finite Models. In: Barzdins, J., Bjorner, D. (eds.) Baltic Computer Science. LNCS, vol. 502, pp. 33–64. Springer, Heidelberg (1991)
47. Tammet, T.: Finite model building: improvements and comparisons. In: Model Computation – Principles, Algorithms, Applications, CADE-19 Workshop W4, Miami, Florida, USA (2003)
48. Zhang, H.: Sato: An Efficient Propositional Prover. In: McCune, W. (ed.) CADE-14. LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
49. Zhang, H.: Specifying Latin squares in propositional logic. In: Veroff, R. (ed.) Automated Reasoning and Its Applications, Essays in Honor of Larry Wos. MIT Press (1997)
50. Zhang, H.: Combinatorial designs by SAT solvers. In: Biere, A., Heule, M., Van Haaren, H., Walsh, T. (eds.) Handbook of Satisfiability, ch. 17. IOS Press (2009)
51. Zhang, H., Stickel, M.: Implementing the Davis-Putnam method. J. of Automated Reasoning 24, 277–296 (2000)
52. Zhang, J.: Problems on the Generation of Finite Models. In: Bundy, A. (ed.) CADE-12. LNCS, vol. 814, pp. 753–757. Springer, Heidelberg (1994)
53. Zhang, J.: Constructing finite algebras with Falcon. J. of Automated Reasoning 17(1), 1–22 (1996)
54. Zhang, J.: On the relational translation method for propositional modal logics. Technical Report ISCAS-LCS-96-12, Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (December 1996)

55. Zhang, J.: Showing the independence of an axiom for temporal intervals by model generation. Association for Automated Reasoning Newsletter, No. 40 (1998), http://www.aarinc.org/Newsletters/040-1998-06.html
56. Zhang, J.: System Description: MCS: Model-Based Conjecture Searching. In: Ganzinger, H. (ed.) CADE-16. LNCS (LNAI), vol. 1632, pp. 393–397. Springer, Heidelberg (1999)
57. Zhang, J.: Test problem and Perl scripts for finite model searching. Association for Automated Reasoning Newsletter, No. 47 (April 2000), http://www.aarinc.org/Newsletters/047-2000-04.html
58. Zhang, J.: Computer Search for Counterexamples to Wilkie's Identity. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 441–451. Springer, Heidelberg (2005)
59. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proc. 14th Int'l Joint Conf. on Artif. Intel. (IJCAI), pp. 298–303 (1995)
60. Zhang, J., Zhang, H.: Constraint Propagation in Model Generation. In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976, pp. 398–414. Springer, Heidelberg (1995)
61. Zhang, J., Zhang, H.: Extending Finite Model Searching with Congruence Closure Computation. In: Buchberger, B., Campbell, J. (eds.) AISC 2004. LNCS (LNAI), vol. 3249, pp. 94–102. Springer, Heidelberg (2004)
62. Zhang, X.: Incomplete perfect Mendelsohn designs with block size four. Discrete Mathematics 254, 565–597 (2002)