ModGen: Theorem Proving by Model Generation*

Sun Kim Hantao Zhang
Department of Computer Science
The University of Iowa
Iowa City, IA 52242, U.S.A
{sunkim,hzhang}@cs.uiowa.edu

Abstract

ModGen (Model Generation) is a complete theorem prover for first order logic with finite Herbrand domains. ModGen takes first order formulas as input, and generates models of the input formulas. ModGen consists of two major modules: a module for transforming the input formulas into propositional clauses, and a module to find models of the propositional clauses. The first module can be used by other researchers so that the SAT problems can be easily represented, stored and communicated. An important issue in the design of ModGen is to ensure that transformed propositional clauses are satisfiable iff the original formulas are. The second module can be easily replaced by any advanced SAT problem solver. ModGen is easy to use and very efficient. Many problems which are hard for general resolution theorem provers are found easy for ModGen.

Introduction

Many theorem proving problems are difficult for today's theorem provers not because these problems are really hard but because the methods are not suitable to these problems. For example, one of test problems in Larry Wos' thought-provoking book (Wos 1988) (Test Problem 6) asks one to prove that any group of order 7 is commutative. Using OTTER (Mccune 1990), one of the best resolution-based theorem provers, this problem cannot be solved in hours. However, if we code this problem in the propositional logic, the problem can be solved in a couple of seconds.

The test problem mentioned above involves functions of finite domains. For this kind of problems, the constraint solving methods are better tools. The FINDER program (Stanley 1992) developed by John Slaney is a well-known program for model generation based on a constraint solving method.

The goal of our project is to create a subroutine of OTTER which has similar functionality as FINDER. However, instead of using any constraint solving methods, we prefer to use a decision procedure for the satisfiability (SAT) of propositional formulas:

- While the SAT problem is a special case of constraint satisfaction problems, many constraint satisfaction problems can be easily and efficiently converted into an instance of the SAT problem. The SAT problem is a core of a large family of computationally intractable NP-complete problems and has been identified as central to a variety of areas in computing theory and engineering.
- There has been great interet in designing efficient algorithms to solve the SAT problem. Various satisfiability testing methods are available, such as backtracking, resolution and its variations — the Davis-Putnam algorithm is one of the known methods. Some local search algorithms have been developed to solve large size instances of the SAT problem (Gu 1993).

Since we intended that the user of OTTER can easily use special methods to handle problems of finite domain, we of course must have a procedure which converts first order formulas into propositional clauses. We found that automatically converting first-order formulas into propositional formulas is not a trivial task. One of the *33 Research Problems* proposed in (Wos 1988) by Wos asks what criteria can be used effectively to choose between predicate and function notation for representing the problem under study. For instance, "the product of a and b equals c" can be written as $P(a,b,c)$ or $prod(a,b) = c$. However, no references are given in (Wos 1988) for the conditions which ensure that the initial formulas are satisfiable iff the converted formulas are. The existence of quantifiers in the formulas makes the problem more complicated.

For example, suppose one of the axioms for a group of order 7 is axiom $x * i(x) = e$, where x is a variable over a domain S of seven elements, say $S = \{1, 2, ..., 7\}$. We may obtain 7 instances of this axiom by replacing x by a value from 1 to 7. The resulting clauses are ground (i.e., variable-free) but they are not propositional clauses. We have to replace functions like $*$ and i by predicate symbols. We may introduce a predicate P_i such that $P_i(x, y)$ is true whenever $i(x) = y$. To get rid of $i(x)$ in that axiom, we may assume $i(x) = u$ and use $P_i(x, u) \Rightarrow (x * u = e)$ as a new axiom. Now

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

the question is: Since the two axioms are not logically equivalent, what additional information is needed to make them equivalent ? This questions must be answered if we want to correctly convert a set of formulas into a set of propositional clauses. We are also interested in the efficiency of the conversion, in the sense that the converted clauses have shorter and less duplicated clauses. These questions will be answered in this paper.

Traditional constraint solving methods do not need to convert general formulas into propositional clauses. For instance, FINDER (Stanley 1992) uses generate-and-test approach to test constraints represented by clauses. While the design philosophy of FINDER and ModGen are very different, FINDER has a great impact on the design of ModGen. Because of the different design philosophies, ModGen offers some advantages:

- ModGen accepts arbitrary formulas (including quantifiers) while FINDER accepts only clauses.
- ModGen can be used to generate propositional clauses for other programs and it is very easy to change SAT decision procedures in ModGen.
- Experimental results show that ModGen outperforms FINDER for almost all the examples tried.

This paper is organized as follows: At first, we give an overview of ModGen and show by examples how ModGen is used. Next, we describe how to correctly and efficiently convert general formulas into propositional clauses; we discuss how to handle function symbols and quantifiers. Finally, we present some experimental results of ModGen.

Overview of ModGen

ModGen mainly consists of two modules, *a propositional clause generator* and the program SATO (SAtisfiability Testing Optimized) (Zhang 1993, Zhang & Stickel 1994) which is an efficient implementation of the Davis-Putnam algorithm (Davis & Putnam 1960). The overall structure of ModGen is shown in Figure 1. ModGen takes first order formulas either in arbitrary form or in clausal form as input, and generates models of input formulas. All variables in input formulas should be of finite domain. Also, ranges of all functions in input formulas should be finite.

The propositional clause generator generates propositional clauses in clausal form from input formulas. If input formulas are not in clausal form, ModGen transforms them in clausal form, and then generates propositional clauses.

The generated propositional clauses are fed to SATO, which determines the satisfiability of the clauses. If the clauses are satisfiable, SATO generates their Herbrand models, that is, which propositional variables are set to true. Since there is a one-to-one mapping between a propositional variable and a function instance, from the Herbrand models generated by

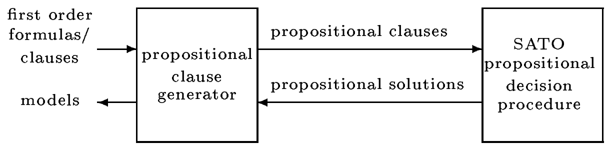
first order formulas/ clauses
models
propositional clause generator
propositional clauses
propositional solutions
SATO propositional decision procedure

Figure 1: Overview of ModGen

SATO, ModGen can print out solutions in terms of function values.

The input to ModGen consists of three parts: (a) sorts of finite elements, (b) functions (including predicates); and (c) (multisort) first order formulas. ModGen decides whether there exist any (or how many) models for the input formulas.

In the beginning of this project, we decided to use the syntax of OTTER (McCune 1990) for input formulas and clauses. This is because

- OTTER is the best known resolution-based theorem prover and is very popular in the community of automated theorem proving;
- we wish that ModGen become a complementary tool to OTTER for finite domain problems and many OTTER's input files can be directly input to ModGen without any modification.

For the above reasons, some code of OTTER has been used in implementing ModGen. Especially, the entire module for parsing input formulas has been used, so the syntax of ModGen is the same as that of OTTER.

We illustrate the use of ModGen by two simple examples.

The Queen Problem

The 8-queen problem is to find placements of 8 queens over an 8×8 chess board so that no two queens attack each other.

```
sort(board, 8).  % the number of queens is 8
func(p(board, board), bool).
% p(i, j) = true iff a queen is placed at (i, j).

list(usable). % a list of clauses
% (a) No two queens are on the same column.
-p(x, z) | -p(y, z) | (x = y).
% (b) No two queens are on the same row.
-p(z, x) | -p(z, y) | (x = y).
% (c) No two queens are on the same diagonal.
-p(x,y) | -p(u,v) | $ABS(x-u) = $ABS(y-v) | (x=u).
end_of_list.

formula_list(usable). % a list of formulas
% (d) Each column must have a queen.
(all x exists y p(x, y)).
end_of_list.
```

Note that the built-in function `$ABS(x)` returns the absolute value of x. It takes less than 0.3 second on an IBM RS6000 for ModGen to decide that there are 92 solutions to the above input. If the user likes to test the 15-queen problem, the only change to the input file is to replace 8 by 15. It takes 2.6 hours for ModGen to decide that there are 2,279,184 solutions for the case of 15. This result cannot compare to that of Sosic and Gu whose program can decide 3,000,000 queens in one minute (Sosic & Gu 1991). However, it is known that there exist solutions to any case but the exact numbers of solutions for large queen problems are still unknown. ModGen may be used to answer such unknown questions while Sosic and Gu's program cannot.

The queen problem can be also specified in terms of a function q such that $q(i) = j$ iff $p(i, j)$. Below is an input file to ModGen:

```
assign(MODEL, 1).      % search only one model
sort(row, 8).          % the number of queens is 8
func(q(row), row, bijective).
  % q being bijective implies that no two queens
  % in the same row or the same column and
  % there exists a queen for each column.
list(usable).
% No two queens are on the same diagonal.
-(q(u) = q(v) + x) | -($ABS(u - v) = x) | (x = 0).
end_of_list.
```

The second input file is much simpler than the first one — this shows the flexibility of ModGen for specifying problems. The following is the result of executing ModGen with the above input.

```
Model #1:
   row |  0 1 2 3 4 5 6 7
   ----+-----------------
    q  |  3 1 6 2 5 7 4 0
```

The Non-Obviousness Problem

The second example shows how easy to use ModGen, that is, ModGen can use some of OTTER's input file without any change. The example is called the "non-obviousness" problem and has appeared in many issues of *Newsletter of Association on Automated Reasoning* (Pelletier & Rudnicki 1986). The input file of OTTER (v3.0.0) is as follows:

```
set(auto).

list(usable).
   -p(a,b).
   -q(c,d).
   p(x,y) | q(x,y).
   q(x,y) | -q(y,x).
   p(x,z) | -p(x,y) | -p(y,z).
   q(x,z) | -q(x,y) | -q(y,z).
end_of_list.
```

The same file can be used by ModGen: The command `set(auto)`, which automatically turns on a set of inference rules of OTTER, is skipped by ModGen. ModGen assumes by default that `p` and `q` are binary predicates over $S = \{$`a, b, c, d`$\}$. While it takes 3.8 seconds for OTTER to show that the input clauses are unsatisfiable, it takes only 0.04 second on the same machine for ModGen; our result is the best among the results reported in the Newsletters of Automated Reasoning.

Propositional Clause Generation

In this section, we present a procedure which can correctly and efficiently convert general formulas into propositional clauses for functions of finite domains. The procedure consists of the following steps:

1. Transform general formulas into clauses.
2. Eliminate function symbols in each clause by introducing new predicates and variables.
3. Instantiate variables in each clause by values and evaluate the truth value of built-in functions and predicates.
4. Return each instantiated clause in an abstract form.

Methods for transforming general formulas into clauses can be found in many textbooks on logic programming. An abstract form of a clause is a list of integers such that the absolute value of each integer is the index of a propositional variable and the sign of the integer is the sign of the literal. In the following, we discuss only steps 2 and 3.

Eliminating function symbols

We will use functions with only one argument for notational convenience; functions with more than one argument can be treated similarly. As mentioned in the introduction, for each function f, we introduce a predicate P_f such that $f(x) = y$ iff $P_f(x, y)$. We can eliminate each term $f(t)$ in a clause formula by substituting a new variable u for $f(t)$, assuming $f(t) = u$. Thus, the transformation rule is as follows.

$$(\beta) \qquad \frac{L(f(t)) \mid M}{\neg P_f(t, x) \mid L(x) \mid M[f(t) \leftarrow x]}$$

where M is a disjunction of literals, $L(f(t))$ is a literal containing the term $f(t)$ and f is not a predicate.

The soundness and completeness of the above rule is ensured by the totalness of f:

Theorem 1 *Suppose $f(x) = y$ iff $P_f(x, y)$ and f is total. For any clause $L(f(t)) \mid M$ and any set S of clauses, $S_1 = S \cup \{L(f(t)) \mid M\}$ is satisfiable if and only if $S_2 = S \cup \{\neg P_f(t, x) \mid L(x) \mid M[f(t) \leftarrow x]\}$ is satisfiable.*

Proof: If S_1 is satisfiable, because $L(f(t)) \mid M$ implies $\neg P_f(t, x) \mid L(x) \mid M[f(t) \leftarrow x]$, S_2 must be satisfiable.

If S_1 is unsatisfiable, by Herbrand's theorem, there exists a unsatisfiable set G_1 of ground instances of S_1. For any Herbrand interpretation H on G_1, there much exist a ground clause in G_1 which is false in H. If this ground clause is not an instance of $L(f(t)) \mid M$, then this ground clause must be an instance of S_2, so H will falsify S_2.

If this ground clause is an instance of $L(f(t)) \mid M$, then it can be written as $\sigma L(f(t)) \mid \sigma M$ for some substitution σ. Because f is total, there must exist a value a such that $f(\sigma t) = a$. Consider the instance

$$(*) \qquad \neg P_f(\sigma t, a) \mid \sigma L(a) \mid \sigma M[f(t) \leftarrow a]$$

of $\neg P_f(t, x) \mid L(x) \mid M[f(t) \leftarrow x]$. Because $P_f(\sigma t, a)$ iff $f(\sigma t) = a$, $\neg P_f(\sigma t, a)$ is false under H. The rest literals of $(*)$ are also false under H because $\sigma L(f(t)) \mid \sigma M$ is false under H and $f(\sigma t) = a$. Hence H will falsify S_2, too. In other words, every interpretation will falsify S_2, so S_2 must be unsatisfiable. $\Box$.

While the totalness of functions is a sufficient condition for the above theorem, we were unable to weaken this condition further. In (Wos 1988), it is said that an equation like $prod(prod(x, y), z) = prod(x, prod(y, z))$ could be replaced by two clauses when using predicate notation:

$$-P(x, y, u) \mid -P(y, z, w) \mid -P(u, z, v) \mid P(x, w, v),$$
$$-P(x, y, u) \mid -P(y, z, w) \mid -P(x, w, v) \mid P(u, z, v).$$

By the above theorem, if *prod* is total, then only one clause is sufficient. However, when *prod* is partial, we do not know if the two clauses are sufficient to replace $prod(prod(x, y), z) = prod(x, prod(y, z))$.

To ensure that $f(x) = y$ iff $P_f(x, y)$, some formulas about P_f should be added:

1. **Totalness:** $\forall x \exists y \; . \; P_f(x, y)$.
2. **Image Uniqueness:**
   $\forall x \forall y_1 \forall y_2 \; . \; P_f(x, y_1) \wedge P_f(x, y_2) \Rightarrow (y_1 = y_2)$.

In ModGen, the transformation rule (β) is repeatedly applied to the general clauses until no functions are left, with the exception that when function symbols appear with equalities, the application of the rule becomes selective.

Dealing with equalities

When functions are used in equalities, we use special techniques to reduce the number of ground clauses generated from general clauses. There are two cases: (1) equalities between a function and a variable/constant, and (2) equalities between functions.

For the first case, say $f(x) = y$, we directly transform this equation into a literal, without introducing a new variable, that is, $P_f(x, y)$ instead of $u = y \mid \neg P_f(x, u)$; the latter is obtained by the transformation rule (β) and would result in too many ground clauses, because it introduces a new variable. The following lemmas ensures that the former transformation is sound.

Lemma 2 *If the clauses for the totalness and image-uniqueness properties of f are present, then $P_f(x, y)$ and $u = y \mid \neg P_f(x, u)$ are logically equivalent.*

Proof: Note that the instances of $u = y \mid \neg P_f(x, u)$ (after removing evaluable literals) are of form $\neg P_f(a_i, a_k)$ and the instances of $P_f(x, y)$ are of form $P_f(a_i, a_j)$. The ground clauses obtained from the totalness property of a function, that is, assuming the range of $f(x)$ is $\{a_1, ..., a_n\}$, are:

$$P_f(x, a_1) \mid ... \mid P_f(x, a_n) \qquad (1)$$

for *all* x. Using resolution, for any value a_i and a_j, we can deduce $P_f(a_i, a_j)$ from instances of $u = a_j \mid \neg P_f(a_i, u)$ and (1).

On the other hand, we can also deduce $\neg P_f(a_i, a_k)$ from $P_f(a_i, a_j)$ and the ground clause generated from the image-uniqueness property. Hence, $P_f(x, y)$ and $u = y \mid P_f(x, u)$ are logically equivalent under the presence of the totalness and image-uniqueness properties of f. $\Box$

For the second case, say $f(x) = g(y)$, we have two choices, to remove $f(x)$ first or to remove $g(y)$ first, and depending on the removal sequence, the resulting clause will be different. If we remove $f(x)$ first, then the resulting clause will be $\neg P_f(x, u) \mid P_g(y, u)$. On the other hand, if we remove $g(y)$ first, then the resulting clause will be $P_g(y, v) \mid \neg P_f(x, v)$. When a function is total, by the theorem in the previous subsection, it is sufficient to generate either of the two clauses.

Dealing with Skolem functions

ModGen takes first order formulas with quantifiers as input and then transforms these formulas into clauses; skolem functions may be introduced during this process. Skolem functions can be treated as ordinary functions which have the totalness and the image-uniqueness properties.

Skolem constants are also treated as ordinary functions, that is, 0-arity functions. The difference is that the image-uniqueness property can not be enforced.

Although treating skolem functions as ordinary functions is sufficient, for efficiency, we treat some skolem functions specially. If a skolem function has all the universally quantified variables as its arguments and does not appear in other clauses, then we can eliminate the skolem function as follows: Suppose $f(x)$ is a skolem function appearing in clause $C(f(x))$. We replace $f(x)$ by a new variable y and $C(f(x))$ is equivalent to $\exists y \; . \; C(y)$. If the domain of y is $\{a_1, ..., a_n\}$, then $\exists y \; . \; C(y)$ is equivalent to $C(a_1) \mid \cdots \mid C(a_n)$. If a skolem function appears in more than one clauses, then we can not eliminate it this way, because $\exists y(P(y) \wedge Q(y))$ is not equivalent to $(\exists y P(y)) \wedge (\exists Q(y))$ in general. For the same reason, we can eliminate skolem constants if a clause containing skolem constants is a ground clause and skolem constants do not appear in other clauses.

Example 3 Consider the formula $\forall y \exists x f(x) = y$. The clausal form of this formula is $f(S(y)) = y$, where S(y) is a skolem function. Assume that the sort of variables x and y contains n elements. Since the skolem

```
for each values of variables $v_1, ..., v_n$ in $C$ do
    for each evaluable literal $l$ in a clause $C$ do
        if $l$ is evaluated to false,
        then delete $l$ from $C$.
        else  exit // No propositional clause from $C$
    endfor
    // $C$ consists of literals with unevaluable predicates.
    generate a ground clause by instantiating each $v_i$
endfor
```

Figure 2: Evaluation of literals and propositional clause generation

function S has all universally quantified variables as its arguments, we can eliminate the skolem function as follows.

$$f(a_1) = y \mid ... \mid f(a_n) = y \qquad (2)$$

By instantiating (2), n ground clauses will be generated. On the contrary, if we did not eliminate the skolem function first, then we would instantiate $P_f(u, y) \mid \neg P_S(y, u)$ and n^2 ground clauses would be generated. □

Instantiating general clauses

The next step is to generate propositional clauses in case that all evaluable literals are evaluated to false while instantiating all variables in a clause. Evaluatable literals are those consisting of only variables and builtin functions/predicates like =(equality). If one of the evaluable literals in a clause is evaluated to true, then no propositional clause will be generated from the clause because the entire clause is eventually true. On the other hand, if one of evaluable literals in a clause is evaluated to false, then the literal will be deleted from the clause because this literal is known to be false, thereby having no effect on the evaluation of the clause. The procedure for generating propositional clauses from a clause, in which function instances are removed, is in Figure 2.

Example 4 The clause $f(x) < f(y) \mid x \geq y$ becomes, by substituting u for $f(x)$ and v for $f(y)$,

$$\neg P_f(x, u) \mid \neg P_f(y, v) \mid u < v \mid x \geq y \qquad (3)$$

where $P_f(x, u)$ is true iff $f(x) = u$ and $P_f(x, v)$ is true iff $f(x) = v$.
The next step is to instantiate all the variables in the formula 3. Assume for one instance that $u = 2, v = 1, x = 1$ and $y = 2$. Then a ground clause $(\neg P_f(1, 1) \mid \neg P_f(0, 2))$ will be generated because both $u < x$ and $x \geq y$ are evaluated to false. Assume another instance that $u = 1, v = 2, x = 1$ and $y = 2$. Then no ground clause will be generated because $u < x$ is evaluated to true. □

Queen of order	No. of models	FINDER (sec)	ModGen	
			No. of clauses	runtime(sec)
5	10	0.20	170	0.07
6	38	0.25	302	0.17
7	40	0.42	490	0.31
8	92	0.75	744	0.61
9	352	1.95	1074	1.34
10	724	6.15	1490	3.46
11	2680	26.10	2002	11.65
12	14200	131.73	2620	52.03
13	73712	725.78	3354	267.97
14	365596	4212.52	4214	1500.27
15	2279184	26604.08	5210	9323.61

Table 1: Experiment with Queen problems

QG5 of order	No. of models	NOELIMINATION		ELIMINATION	
		No. of clauses	runtime (sec)	No. of clauses	runtime (sec)
5	1	2211	0.14	1461	0.10
6	0	4552	0.31	3040	0.17
7	3	8401	0.47	5657	0.31
8	1	14301	0.81	9693	0.51
9	0	22879	1.29	15589	0.89
10	0	54846	4.09	43846	2.62
11	5	80279	10.01	64307	5.41
12	0	113683	25.37	91219	11.26
13	0	156573	562.84	125815	234.71

Table 2: Experiment with Quasigroup 5 problems

In general, it is possible that identical ground clauses are generated more than once. ModGen avoids duplication of ground clauses only for symmetric cases. When a clause to be instantiated is invariant to the exchange of two variables, this clause is said to be symmetric with respect to the two variables. The following example illustrates the elimination of redundant ground clauses by the symmetry checking.

Example 5 Consider $P_f(x, y) \mid P_f(x, z)$. Assume that $x = 1, y = 1$ and $z = 2$. Then $P_f(1, 1) \mid P_f(1, 2)$ will be generated. Assume also that $x = 1, y = 2$ and $z = 1$. Then $P_f(1, 2) \mid P_f(1, 1)$ will be generated. Clearly, the two ground clauses are identical.

If we exchange y for z in the above clause, the clause becomes $P_f(x, z) \mid P_f(x, y)$ which is identical to the original clause. Then, it is sufficient to generate ground clauses from $P_f(x, y) \mid P_f(x, z)$ only for $y \leq z$ since all ground clauses generated for $y > z$ are redundant. □

It is also possible to generate fewer ground clauses by applying some of the unit literal deletion and pure literal deletion. However, this kind of checkings can be done by the propositional decision procedure.

Experimental Results

We tested ModGen with the queen problem, quasigroup problems, and several puzzles. The experiment is done on a IBM RS6000/530. All times are taken as the best of three runs.

The result of the experiment with the queen problem is shown in Table 1, which also includes the result of FINDER for performance comparison. ModGen runs as more than twice faster than FINDER for queen problems of order > 10.

The experimental results with some quasigroup problems (Stanley, Fujita, & Stickel 1993) are listed

GQ6		INDIRECT		DIRECT	
of order	No. of models	No. of clauses	run time (sec)	No. of clauses	run time (sec)
5	0	48121	4.19	2711	0.13
6	0	959296	86.38	5632	0.25
7	0	23676843	2290.65	10459	0.47
8	2	-	-	17885	0.80
9	4	-	-	28711	1.37
10	0	-	-	43846	2.39
11	0	-	-	64307	5.88
12	0	-	-	91219	86.54
13	41760	-	-	125815	3287.47

Table 3: Experiment with Quasigroup 6 problem

problem	models	No. of clauses	runtime(sec)
Agatha	1	34	0.02
Nonobvious	0	162	0.04
Jobs	16	404	0.12
Steamroller	0	2250	0.21

Table 4: Experiment with puzzle problems

in Tables 2 and 3. The QG5 problem is to investigate the existence of Latin squares satisfying the identity $(((y * x) * y) * y) = x$ (viewing the square as a multiplication table). This problem includes two skolem functions which can be eliminated as explained in the previous section. The data in the column under ELIMINATION in Table 2 are the result of eliminating skolem functions and the data under NOELIMINATION are the result of treating the skolem function as ordinary functions. Eliminating skolem functions not only generates smaller number of ground clauses but also accelerates the search. The performance difference comes from the fact that ModGen generates one less order of ground clauses for formulas having skolem symbols.

The experimental results with another quasigroup problem, QG6 (Stanley, Fujita, & Stickel 1993), are is listed in Table 3. This problem is to investigate the existence of Latin squares satisfying the identity $((x * y) * y) = (x * (x * y))$, which is an equality between function instances ans is specially treated in ModGen. The data in the column under INDIRECT in Table 3 are the result of employing the transformation of $f(x) = y$ to $u = y \mid \neg P_f(x, u)$. The data under DIRECT in Table 3 are the result of employing the transformation of $f(x) = y$ to $P_f(x, y)$. As shown in Table 3, the performance difference is amazingly large. We could not experiment beyond order 7 for INDIRECT because of the excessive computing time. This performance difference is manifested by the observation that the INDIRECT strategy generates one more order of ground clauses for *each* literal having equality with function instances.

We also experimented with some puzzle problems such as Non-obviousness, Schubert's Steamroller (Stickel 1986), Jobs and Agatha; the results are listed in Table 4. Shubert's Steamroller has the conclusion negated so that it is unsatisfiable. This experiment shows that ModGen can solve puzzle problems very fast.

References

Bennett, F. 1989 Quasigroup Identities and Mendelsohn Designs, *Canadian Journal of Mathematics* 41: 341–368.

Gu, J. 1993 Local search for satisfiability (SAT) problem, *IEEE Trans. on Systems, Man, and Cybernetics* 23(4): 1108-1129.

Davis, M; Putnam, H 1960 A computing procedure for quantification theory, *J. of ACM* 7: 201-215.

McCune, W. W 1990 Otter 2.0 users' guide, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois.

Pelletier, F. J; Rudnicki, P 1986 Non-Obviousness, *Automated Reasoning Newsletter* 6: 4-5.

Selman, B.; Levesque, H.; Mitchell, D. 1992 A new method for solving hard satisfiability problems, In *Proceedings of AAAI'92*: 440-446.

Slaney, J. 1992 FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guide, Technical report TR-ARP-1/92, Automated Reasoning Project, Australian National University.

Slaney, J.; Fujita, M.; and Stickel, M. Automated reasoning and exhaustive search: Quasigroup existence problems To appear in *Computers and Mathematics with Applications.*

Sosic, R.; Gu, J. 1991 3,000,000 queens in less than one minute, *SIGART Bulletin*, 2(2): 22-24

Stickel, M. 1986 Shubert's steamroller problem: formulations and solutions, *J. of Automated reasoning* 2: 89-101.

Wos, L. 1988 *Automated reasoning: 33 Basic research problems*, Prentice Hall, New Jersey.

Zhang, H. 1993 SATO: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22: 1-3.

Zhang, H.; Stickel, M.: 1994 Implementing the Davis-Putnam Algorithm by Tries, Unpublished manuscript.