Parallel Theorem-Proving System : MGTP

Ryuzo Hasegawa

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
hasegawa@icot.or.jp

Abstract

The parallel theorem-proving system MGTP has been developed at ICOT.

MGTP exploits OR-parallelism for non-Horn problems and AND-parallelism for Horn problems. MGTP achieves a more than 200-fold speedup on a parallel inference machine PIM/m with 256 processing elements. Using MGTP, we succeeded in proving difficult mathematical problems including open problems that cannot be proven on sequential systems.

We added two new MGTP features: Non-Horn Magic Set (NHM) and Constraint MGTP (CMGTP). NHM is a technique to enhance the search capacity by suppressing generation of irrelevant model candidates, thereby making MGTP a practical prover. CMGTP is an extension of MGTP to deal with constraint satisfaction problems, enabling constraint propagations with negative atoms. CMGTP can prune search spaces required for the original MGTP by orders of magnitude.

We studied several techniques necessary for the development of applications, such as negation as failure, abductive reasoning and modal logic systems, on MGTP. These techniques share a basic idea, which is to use MGTP as a meta-programming system for each application.

1 Introduction

Theorem proving is an important basic technology that gave rise to the logic programming being pursued as a key technology in the implementation of the Fifth Generation Computer Systems (FGCS).

We started research on parallel theorem provers in 1989 in the FGCS project with the aim of integrating logic programming and theorem proving technologies.

The immediate goal of this research was to develop a fast theorem-proving system on the PIM, by effectively utilizing KL1 languages[34] and logic programming techniques. We intended this system to be an "advanced general-purpose inference engine" that can facilitate the development of intelligent knowledge programming software in KL1. Theorem proving is typical of symbolic processing that demands large-scale computation and huge amounts of memory. Thus, it is also a good application for KL1 and PIM machines[28].

We have developed a parallel model-generation based theorem prover MGTP in KL1 on PIM machines[12, 8]. We adopted the model generation method of SATCHMO[24] as the proof procedure. Our reasons are the following. This method has a merit of not needing full unification; that is, it needs matching only. In addition, it is easy to incorporate mechanisms to prune search spaces, such as lemmatization, subsumption tests, and other deletion strategies.

Two versions of MGTP are being built: one is a ground MGTP (MGTP/G), for dealing with ground models, and the other is a non-ground MGTP (MGTP/N), for dealing with non-ground models.

MGTP/G exploits OR parallelism from non-Horn problems by independently exploring each branch of a proof tree caused by case splitting. On the other hand, MGTP/N exploits AND parallelism from Horn problems that do not cause case splitting.

At present, both systems achieves almost linear speedup on a PIM/m 256 PE system. With MGTP/G, we succeeded in solving some open quasigroup problems in finite algebra[9, 17]. MGTP/N on a single PIM/m PE attains a speed comparable to OTTER[25] on a SPARC II. With MGTP/N on the 256 PE system, we also succeeded in solving several hard (condensed detachment) problems[26] that could not be solved by OTTER with any strategy[13, 20]. Our success shows the effectiveness of a large-scale parallel theorem prover.

Besides mathematical problems, MGTP is applicable to other AI-oriented applications, such as constraint satisfaction problems, design and planning, and hardware/software verification. In particular, MGTP/G has actually been used as a rule-based engine for the legal reasoning system HELIC-II[29] developed at ICOT.

Research on MGTP can be divided into three aspects: (1) implementation, (2) extension of the MGTP features, and (3) application. In the FGCS project, we placed emphasis on the first aspect, then became convinced that MGTP could be put into practical use. In the Follow-on project, we thus shifted our research efforts from the

pursuit of efficiency to the enhancement of the functions.

(1) Implementation

During the development of MGTP systems, we established the following techniques to improve the efficiency of MGTP.

- avoiding redundancy in conjunctive matching (RAMS/MERC)[7].
- reducing time and space complexity by suppressing over-generation of atoms (Lazy Model Generation)[15, 14].
- compiling problem clauses and main proof procedures into KL1 clauses.
- parallelizing MGTP on a multi-processor system with a distributed memory architecture[13, 20].

(2) Extension of the MGTP features

Since MGTP is a bottom-up theorem prover, it may derive unnecessary facts and cause redundant case splitting. To avoid this, we developed a method called Non-Horn Magic Set (NHM)[16]. This method is a natural extension of the magic set developed in the deductive database field, and can be applied to non-Horn problems.

Through research on proving quasigroup problems with MGTP, we found that MGTP lacks negative constraint propagation ability. Then, we developed CMGTP (Constraint MGTP)[17] that can handle constraint propagations with negative atoms.

(3) Application

MGTP can be viewed as a meta-programming system. We can build various reasoning systems on MGTP by writing the inference rule used for each system as MGTP input clauses. Along this idea, we developed several techniques and reasoning systems necessary for AI applications. These include:

- a method to incorporate negation as failure into MGTP[18].
- abductive reasoning systems[19], in which a method similar to the above method is used.
- modal logic systems[22, 1].
- mode analysis of FGHC programs[33].

The following sections describe the main results of research on the parallel theorem prover MGTP in the FGCS and the Follow-on projects, focusing on parallelization of MGTP, and the extension and application aspects.

2 Outline of MGTP

2.1 Model Generation Method

An MGTP clause is represented by an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; the consequent is a disjunction of $C_1, C_2, \ldots, C_m$. A clause is said to be *positive* if its antecedent is $true (n = 0)$, *negative* if its consequent is $false (m = 0)$, and otherwise *mixed* $(m \neq 0, n \neq 0)$.

The following two rules act on the model generation method.

- Model extension rule: If there is a clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model candidate M and $C\sigma$ is not satisfied in M, extend M by adding $C\sigma$ to M.
- Model rejection rule: If there is a negative clause whose antecedent $A\sigma$ is satisfied in a model candidate M, reject M.

We call the process obtaining $A\sigma$, a *conjunctive matching* of the antecedent literals against the element in a model candidate. Note that the antecedent *true* of a positive clause is satisfied by any model.

The task of model generation is to try to construct a model for a given set of clauses, starting with an empty set as a model candidate. If the clause set is satisfiable, a model should be found. This method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to ensure that no model exists for the clause set.

2.2 Sequential MGTP Algorithm

Figure 1 shows a sequential model generation algorithm of MGTP. This algorithm searches one branch of a proof tree and does not contain a procedure for case-splitting. It returns *sat* if a set S of clauses is satisfiable, otherwise it returns *unsat*.

MD is an array to retain atoms generated by model extension. In this algorithm, k is the number of positive clauses in S. $MD[1], \ldots, MD[g-1]$ represent elements of the model candidate M, $MD[g], \ldots, MD[s]$ represent elements of the model-extending candidate D (a set of model-extending atoms which are generated as a result of the application of the model extension rule and that are going to be added to M). $g-1$ is the number of atoms that have been used for model extensions and g specifies the range of the next model extension. s is the total number of derived atoms retained in $M \cup D$ and specifies the range of model rejection testing.

```
(0)  Init: MD[i] := C_i for i = 1,...,k where {C_1,C_2,...,C_k} = {C | (true → C) ∈ S}
(1)  Init: g := 1; s := k;
(2)  while g ≤ s do begin
(3)      foreach A_1,...,A_n → C
(4)          foreach (i_1,...,i_n) s.t. ∀j(i_j ≤ g) and ∃j(i_j = g)
(5)              if ∃σ∀j(MD[i_j]σ = A_jσ and Cσ is new) then begin
(6)                  s := s + 1; MD[s] := Cσ; /* model extension */
(7)                  foreach A_1,...,A_m → false
(8)                      foreach (i_1,...,i_m) s.t. ∀j(i_j ≤ s) and ∃j(i_j = s)
(9)                          if ∃σ'∀j(MD[i_j]σ' = A_jσ') then
(10)                             return (unsat); /* model rejection */
(11)             end;
(12)     g := g + 1;
(13) end ;
(14) return (sat);
```

Figure 1: A sequential algorithm of MGTP

Initially, M and D are set to an empty set and a set of atoms appearing in the consequent parts of positive clauses, respectively (`(0)` and `(1)`). "$C\sigma$ is new" in `(5)` means that a generated atom $C\sigma$ is not subsumed by any atom in $M \cup D$ ($\neg\exists i(1 \leq i \leq s)\exists\sigma'(C\sigma = MD[i]\sigma')$).

`(3)` to `(6)` are procedures for model extension. We perform conjunctive matchings of each mixed clause against M (`(4)` and the former part of the conditional in `(5)`). If the conjunctive matching succeeds, we make subsumption tests for $C\sigma$ (the latter part of the conditional in `(5)`), and store unsubsumed $C\sigma$ in D(`(6)`).

`(7)` to `(10)` are procedures for model rejection tests. We perform conjunctive matchings of each negative clause against $M \cup D$, and return *unsat* if the conjunctive matching succeeds.

The computational mechanism for MGTP is essentially based on the "generate-and-test" scheme. This, therefore, would cause over-generation of atoms, which is incurred by generation processes, leading to a waste of time and memory space.

To solve this problem, we developed the *lazy model generation* method[14]. In this method, a generator process to perform model extension generates a specified number of atoms only when required by a tester process to perform model rejection testing. The lazy mechanism can be used to control the difference in speed between the generator and the tester processes, thereby avoiding unnecessary computations and reducing the time and space complexity.

3 Parallelization

There are several ways to parallelize the proving process in the MGTP prover.

These are to exploit parallelism in:

- conjunctive matching in the antecedent part,
- subsumption tests, and
- case splitting

For ground non-Horn cases, it is sufficient to exploit OR parallelism induced by case splitting. Here we use OR parallelism to seek a multiple model, which produces multiple solutions in parallel.

For Horn clause cases, we have to exploit AND parallelism. The main source of AND parallelism is conjunctive matching. Performing subsumption tests in parallel is also very effective for Horn clause cases.

In the current MGTP, we have not yet considered non-ground and non-Horn cases.

3.1 OR Parallelization

Though OR parallelism can easily be exploited by exploring branches in different PEs, inter-PE communication increases because the number of branching combinatorially explodes in general and it is necessary to copy a computational environment (model candidates and model extending candidates) when branching is required.

To suppress the amount of communication, we first introduced *bounded-OR* parallelization. In this scheme, we allocate a task (a branch to explore) to a different PE every time case splitting occurs until all PEs have enough tasks to keep them busy. Later, each PE explores branches assigned without further distributing tasks to any other PEs. Bounded-OR parallelization works well when a proof tree is well-balanced.

However, for ill-balanced proof trees, dynamic load balancing must be devised to equalize all PE's loads. Figure 2 shows a task allocation method based on dynamic load balancing. In this method, one of the branches is solved in the same PE and the remained branches are allocated to other PEs. This is a mixture of depth-first and breadth-first searches. We experimented with several PE allocation methods: (1) probabilistic allocation,

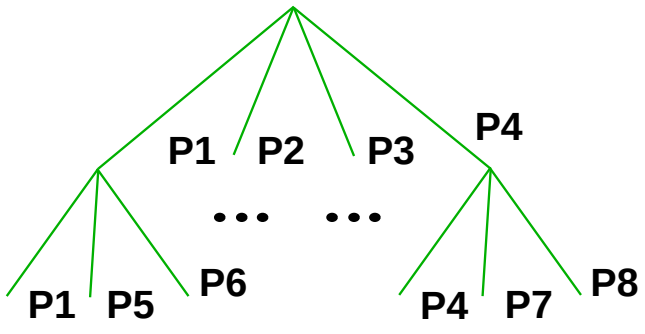

Figure 2: Task Allocation for OR Parallelization

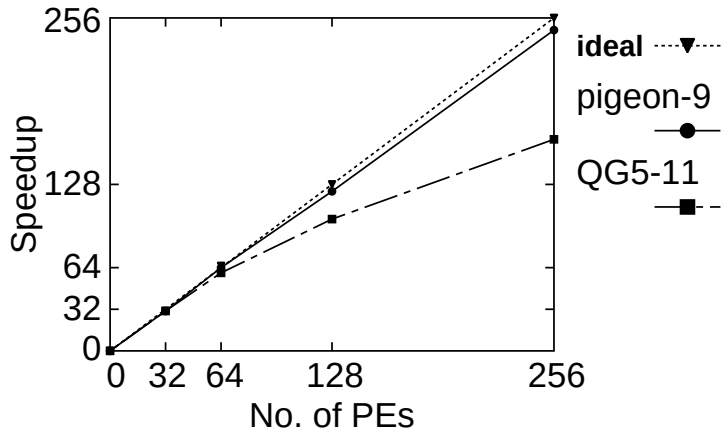

Figure 3: Speedup Ratio (OR Parallelization)

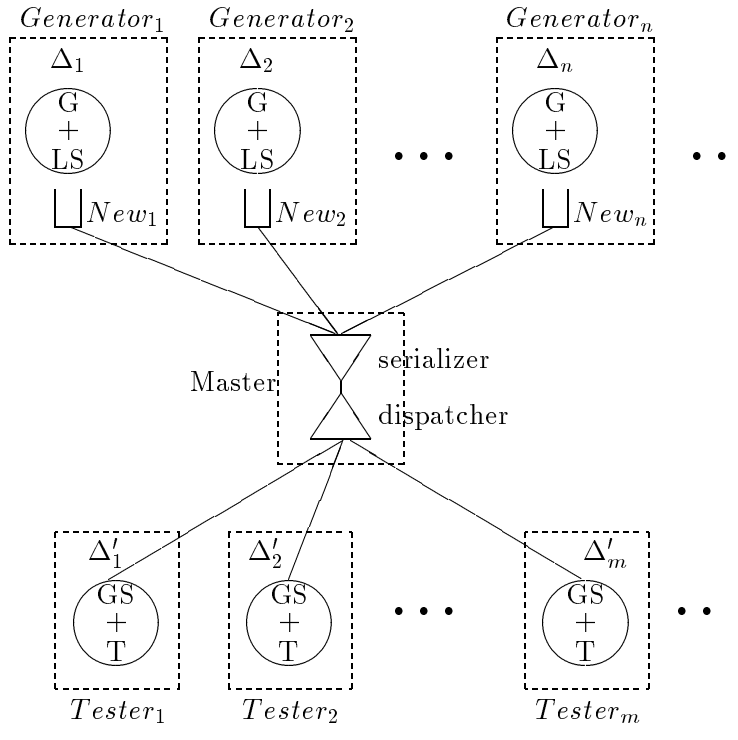

Figure 4: AND parallelization of MGTP

(2) circular allocation by modulo, (3) allocating tasks to a PE whose load is light and (4) allocating tasks to an idle PE. For solving quasigroup existence problems, (2) and (3) showed good parallel performance.

3.1.1 OR Parallel Performance

Figure 3 shows the OR parallel performance obtained for pigeon hole and quasigroup (QG) existence problems on PIM/m. The former is a typical benchmark in automated theorem proving and the latter is called QG5. Solving these problems causes combinatorially intensive case-splitting.

While the proof tree for the pigeon hole problem is well-balanced, that for the QG problem is ill-balanced. Thus, it is harder to equalize the load of each PE for the QG problem. We obtained a 170-fold speedup with 256 PEs for QG5-11. The speedup ratio would be improved by devising a load balancing scheme.

3.2 AND Parallelization

During the design, we studied the following alternatives:

1. proof changing vs. proof unchanging
   While a proof unchanging prover does not change the proof length according to the number of PEs used, a proof changing prover may change the proof length. The proof changing prover may get super-linear speedup and can obtain good parallel performance. However, it would be difficult to distinguish between parallelization effects and strategy effects.
2. model copying vs. model distribution
   For model copying, each PE has a copy of the model and model-extending candidates. For model distribution, they are distributed to each PE. Although the model distribution method can obtain memory scalability, it has the problem that the communication cost increases since generated atoms need to flow to all PEs for subsumption tests and conjunctive matchings.

We established the following policy: (1) Distinction between the speedup effect caused by parallelization and the search-pruning effect caused by strategies, and (2) Pursuit of both execution efficiency and ease of implementation. Accordingly, we adopted proof unchanging and model copying, then implemented an AND parallel MGTP based on the lazy model generation method[14].

Figure 4 shows a process network of the AND parallel MGTP. There are three types of processes: generator (G), tester (T), and master (M) processes. G/T processes perform conjunctive matchings for mixed/negative clauses. The M process mediates between G and T processes. In our implementation, several G and T processes are allocated to each PE.

A group of G processes interconnects with a group of T processes via the M process. The M process gives G and T processes individual tasks according to their requests. When a G/T process works out a given task, it requests a next task from the M process. This is repeated until a proof is obtained.

The M process replies to the G and T processes simultaneously. On receipt of a request from a G process, the M process sends a buffer New_i to the G process and orders it to perform conjunctive matchings and to store newly-generated atoms into New_i where New_i is shared by the M process and the G process. On receipt of a request from a T process, the M process sends an atom in $New_{i'}$ to the T process to make it perform rejection testings on the newly-generated atoms.

Subsumption tests are performed in G and T processes. We call a subsumption test in a G process a *local subsumption test* (LS) and that in a T process a *global subsumption test* (GS). An atom that passes LS is stored into New_i. We use discrimination trees[31] to implement LS for quick retrieval of atoms.

The M process equalizes the loads of G and T processes by monitoring these processes. This is the most important task of the M process.

Each G/T process requests a task to the M process after completing the previous task. The M process replies to their requests in order of arrival and assigns them exclusive tasks. The M process prevents G processes from generating too many atoms by monitoring the number of atoms stored in New buffers and by keeping that number in a moderate range. This number indicates the difference between the number of atoms generated by G processes and the number of atoms tested by T processes.

When the above number exceeds the specified range, the M process suppresses replying to demands from G processes. On the other hand, when the number drops below the range, the M process prefers G processes to T processes. By simply controlling G and T processes with the buffering mechanism mentioned above, the idea of lazy model generation can be implemented. This also enables us to balance the computational load of G and T processes, thus keeping a high running rate.

3.2.1 AND Parallel Performance

Figure 5 shows AND parallel performance for solving condensed detachment problems[26] on PIM/m with 256 PEs.

These problems consist of a detachment rule (modus ponens), several axioms (positive clauses), and a negated conjecture that we want to prove (negative clauses). They are given as a set of Horn clauses only, as follows.

 # 22 (i: implication/ n: negation)
$p(X),\ p(i(X,Y)) \rightarrow p(Y).$
$true \rightarrow p(i(i(X,Y),i(i(Y,Z),i(X,Z)))).$
$true \rightarrow p(i(i(n(X),X),X)).$
$true \rightarrow p(i(X,i(n(X),Y))).$
$p(i(i(n(a),c),i(i(b,c),i(i(a,b),c)))) \rightarrow false.$

Proving time (sec) obtained with 32 PEs for each problem is as follows: #49:18600, #44:9700, #22:8600, #79:2500, and #82:1900. The numbers of atoms that have been kept in $M \cup D$ when the proof is obtained are #49:20600, #44:15100, #22:36500, #79:14200, and #82:15100. More than a 230-fold speedup was attained for #49 and #44 and a 170∼180-fold speedup for #22, #79 and #82.

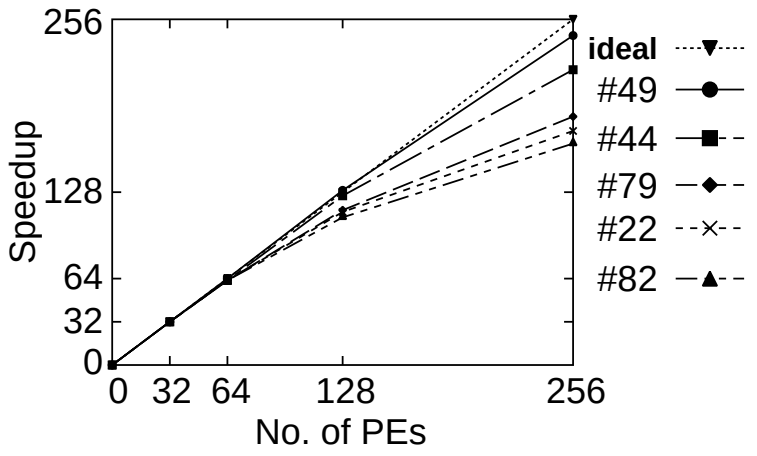

Figure 5: Speedup Ratio (AND Parallelization)

Problems #49 and #44 take longer than the others and have the characteristics that $|M \cup D|$ slows up, since almost all generated atoms are discarded by local subsumption testing. The longer the proving time is and the slower $|M \cup D|$ grows, the better parallel performance is.

To verify the effectiveness of an AND parallel MGTP, we challenged 12 ALL-FAIL condensed detachment problems. These problems could not be solved by OTTER with any strategy proposed in [26].

7 of 12 problems were solved within an hour except for problem #23, in which the maximum number of atoms being stored in M and D ($|M + D|$) was 85100. The problems we failed to solve were such that this size exceeds 100000 and more than 5 hours are required to solve them.

It would be possible to solve these problems by spending much more time. However, this is a limitation of the model copying method at present since at this size of $|M + D|$, the memory consumption rate is over 80 percent thereby causing frequent garbage collection.

3.2.2 Model Distribution

In order to solve the memory limitation problem mentioned above, we are now developing a model-distribution based AND parallel MGTP:

1. Model candidate M and model-extending candidate D are divided into n parts $M_1, \cdots, M_n$ and $D_1, \cdots, D_n$, respectively. Each pair of M_i and D_i is allocated to a processor PE_i.

2. At PE_i, model extension is performed using an atom Δ and M_i, and newly-generated atoms are stored in the buffer New_i. Rejection testing is performed using an atom Δ' and $M_i \cup D_i$.

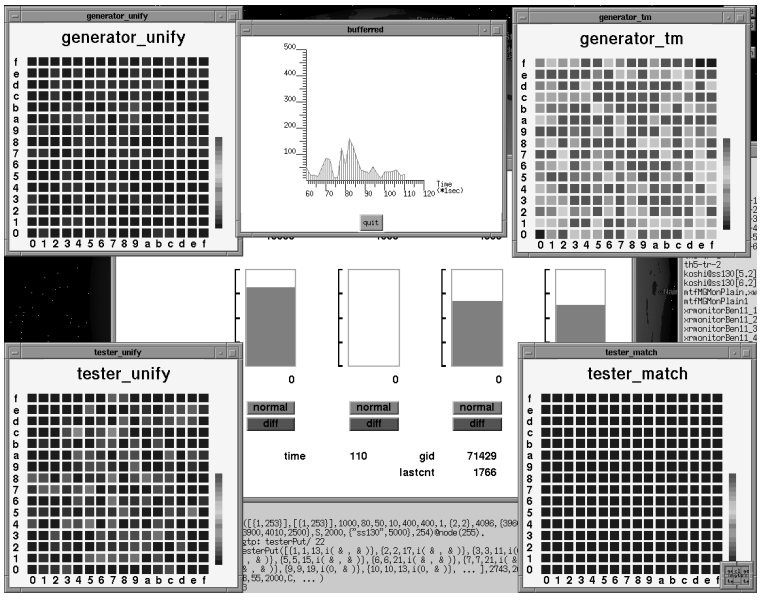

Figure 6: MGTP Runtime Monitor

3. Each atom in New_i is flown to all PEs for subsumption testing.

4. Unsubsumed atoms are to be distributed so that each PE has even size of M_i and D_i

In the recent experiments, a good parallel performance is obtained up to about 100 PEs.

3.3 MGTP Runtime Monitor

In the parallel performance debugging, it is essential to observe the behavior of processes allocated to PEs, as well as the PE's utilization. We developed an MGTP runtime monitor to visualize the load of a process and the proving process of AND parallel MGTP.

Figure 6 shows the graphic display of the runtime monitor. There are 4 windows at the corners: the upper 2 windows display the load of the G processes at an interval and the lower 2 windows display that of the T processes. Each window has $16 \times 16 = 256$ subwindows which display the process load at each PE. The color of the subwindow represents the load of the corresponding process.

The central 4 bar graphs display how many atoms are generated, buffered in New, stored in MD (unsubsumed), and tested at an interval or in total. We can verify whether the buffering mechanism works well or not by watching these windows. When the buffer becomes close to empty, we can see that the load of G processes become heavy in the upper 2 windows. On the other hand, when the number of the buffered atoms exceeds a limit, we can see that T processes become busy in the lower 2 windows.

Experiments with various theorems have indicated that graphic displays are helpful in distributing work loads evenly and determining which process creates a bottleneck.

4 Extensions of MGTP Features

4.1 Non-Horn Magic Set

The basic behaviors of model generation theorem provers, such as SATCHMO and MGTP, are to detect a violated clause under some interpretation, called a model candidate, and to extend the model candidate so that the clause is satisfied. However, since SATCHMO selects any possible violated clause for model extension without any selection criterion. Therefore, if the given program includes non-Horn clauses irrelevant to prove the given goal, it may cause combinatorial explosion of the number of generated model-candidates.

Loveland *et al.* [36, 23] have proposed a method called relevancy testing to avoid redundant model-candidate extensions with irrelevant non-Horn clauses. Let HC be a set of Horn clauses, and I be a current model candidate. A relevant literal is defined as a goal called in a failed search to prove $\bot$ from $HC \cup I$ or a goal called in a failed search to prove the antecedent of a non-Horn clause by Prolog execution.

The use of relevancy testing can restrict the selecting of violated non-Horn clauses for model extension to only those all of whose consequent literals are ground instances of some relevant literals. In relevancy testing, however, there are some overheads because the Prolog procedure is performed whenever violated non-Horn clauses are detected.

On the other hand, compared to top-down provers, a model generation prover like SATCHMO or MGTP can avoid solving duplicate subgoals because it is based on bottom-up evaluation. However, it also has the disadvantage of generating irrelevant atoms to prove the given goal.

Thus it is necessary to combine bottom-up with top-down so as to use goal information contained in negative clauses, and to avoid generating useless model candidates. For this purpose, several methods such as magic sets, Alexander templates, and bottom-up meta-interpretation[4] have been proposed in the field of deductive databases.

All of these transform the given Horn intentional databases to efficient Horn intentional databases, which generate only ground atoms relevant to the given goal in extensional databases. However, these were restricted to Horn programs.

We developed a transformation method applicable to non-Horn clauses. We call it the non-Horn magic set (NHM). NHM is a natural extension of the magic set yet works within the framework of the model generation method. Stickel[32] proposes another extension for non-Horn clauses, simulating top-down execution based on the model elimination procedure within a forward chaining paradigm.

By the NHM method, a clause $A_1, \cdots, A_n \rightarrow B_1; \cdots; B_m$ in the given program is transformed into the following clauses:

$$\begin{aligned} T1: \quad & goal(B_1), \cdots, goal(B_m) \\ & \rightarrow goal(A_1), \cdots, goal(A_n). \\ T2: \quad & goal(B_1), \cdots, goal(B_m), A_1, \cdots, A_n \\ & \rightarrow B_1; \cdots; B_m. \end{aligned}$$

Here, the meta-predicate $goal(A)$ means that an atom A is a goal. The transformation $T1$ simulates top-down evaluation. If all consequent literals $B_1, \cdots, B_m$ are goals, then all antecedent literals $A_1, \cdots, A_n$ of the clause are solved in parallel. On the other hand, the transformation $T2$ simulates relevancy testing. If all consequent literals $B_1, \cdots, B_m$ of the clause are goals, then the original clause may be used for extending the model candidates. That is the original clause is not used for extending the model candidates if there exists any consequent literal B_j such that B_j is not a goal. This transformation method is called a breadth-first NHM. We have another transformation method called the depth-first NHM, in which all antecedent literals $A_1, \cdots, A_n$ of the clause are solved in sequence with the left-to-right strategy.

The NHM method has the same power as relevancy testing. Therefore, the NHM method can also avoid combinatorial explosion of the number of model candidates generated with irrelevant violated non-Horn clauses to the given goal. Since the NHM method statically transforms the given program, there are no overheads as in [23] which performs relevancy testing dynamically.

4.2 Constraint MGTP

In this section, we present an extension of the MGTP system called CMGTP (Constraint MGTP), which can solve finite-domain constraint satisfaction problems such as quasigroup (QG) existence problems [3] in finite algebra.

In 1992, M. Fujita and J. Slaney[9] first succeeded in solving several open QG problems by using FINDER and MGTP. Later, it was shown that other systems such as DDPP or CHIP could solve QG problems more efficiently. Such research has revealed that the original MGTP lacks negative constraint propagation ability. This motivated us to develop an experimental system CP based on the CLP (constraint logic programming) scheme.

We found that the constraint propagation mechanism used in CP can be realized by the slightly modified MGTP system, called CMGTP[17].

4.2.1 Quasigroup Problems

A Quasigroup is a pair $\langle Q, \circ \rangle$ where Q is a finite set, $\circ$ a binary operation on Q and for any $a, b, c \in Q$,

$\circ$	1	2	3	4	5
1	1	3	2	5	4
2	5	2	4	3	1
3	4	5	3	1	2
4	2	1	5	4	3
5	3	4	1	2	5

Figure 7: Latin square (order 5)

$$a \circ b = a \circ c \Rightarrow b = c$$
$$a \circ c = b \circ c \Rightarrow a = b.$$

The multiplication table of this binary operation $\circ$ forms a latin square (shown in Fig.7).

In a quasigroup, we can define the following inverse operations $\circ_{ijk}$ called *(ijk)-conjugate*:

$$\begin{aligned} x \circ_{123} y = z \quad & \Longleftrightarrow \quad x \circ y = z \\ x \circ_{231} y = z \quad & \Longleftrightarrow \quad y \circ z = x \\ x \circ_{312} y = z \quad & \Longleftrightarrow \quad z \circ x = y \end{aligned}$$

Multiplication tables of the inverse operations defined above also form latin squares.

We have been trying to solve 7 categories of QG problems (called QG1, QG2,..., QG7), each of which is defined by adding some constraints to original quasigroup constraints. For example, QG5 constraint is defined as $\forall ab \in Q.\ ((ba)b)b = a$.

4.2.2 CP

While in CLP languages, domain variables are used to represent constraints, in CP, domain element variables are introduced as well as domain variables. Fig.8 shows the variables in a third-order latin square used in CP where domain variables V_{ij} range over $\{1, 2, 3\}$ $(1 \leq i, j \leq 3)$ and domain element variables $A_k, B_k, \ldots, I_k$ range over $\{yes, no\}$ $(1 \leq k \leq 3)$.

Let V be a domain variable whose domain is $\{1, 2, \ldots, n\}$, and $(A_1, A_2, \ldots, A_n)$ be a vector of domain element variables w.r.t. V. The value of A_i determines whether $V = i (A_i = yes)$ or $V \neq i (A_i = no)$.

For QG problems, we maintain three squares according to (1,2,3)-, (2,3,1)- and (3,1,2)-conjugates. Domain element variables in these 3 squares can be linked (unified) with each other. (shown in Fig.8 and Fig.9).

Using shared (unified) variables facilitates constraint propagation like :

$$\forall abc.\ (a \circ_{123} b = c \Leftrightarrow b \circ_{231} c = a \Leftrightarrow c \circ_{312} a = b) \quad (1)$$
$$\forall abc.\ (a \circ_{123} b \neq c \Leftrightarrow b \circ_{231} c \neq a \Leftrightarrow c \circ_{312} a \neq b) \quad (2).$$

Ordinary CLP does allow constraint propagation like (1), but (2) is not possible in general because domain element variables cannot be handled directly. By this unification, CP can propagate negative information which can be overlooked in ordinary CLP systems.

S–square : (123)-conjugate

∘	1	2	3
1	V_{11} $(A_1\ A_2\ A_3)$	V_{12} $(B_1\ B_2\ B_3)$	V_{13} $(C_1\ C_2\ C_3)$
2	V_{21} $(D_1\ D_2\ D_3)$	V_{22} $(E_1\ E_2\ E_3)$	V_{23} $(F_1\ F_2\ F_3)$
3	V_{31} $(G_1\ G_2\ G_3)$	V_{32} $(H_1\ H_2\ H_3)$	V_{33} $(I_1\ I_2\ I_3)$

Figure 8: The variables in a third-order latin squares

I–square : (231)-conjugate

∘	1	2	3
1	W_{11} $(A_1\ D_1\ G_1)$	W_{12} $(A_2\ D_2\ G_2)$	W_{13} $(A_3\ D_3\ G_3)$
2	W_{21} $(B_1\ E_1\ H_1)$	W_{22} $(B_2\ E_2\ H_2)$	W_{23} $(B_3\ E_3\ H_3)$
3	W_{31} $(C_1\ F_1\ I_1)$	W_{32} $(C_2\ F_2\ I_2)$	W_{33} $(C_3\ F_3\ I_3)$

R–square : (312)-conjugate

∘	1	2	3
1	U_{11} $(A_1\ B_1\ C_1)$	U_{12} $(D_1\ E_1\ F_1)$	U_{13} $(G_1\ H_1\ I_1)$
2	U_{21} $(A_2\ B_2\ C_2)$	U_{22} $(D_2\ E_2\ F_2)$	U_{23} $(G_2\ H_2\ I_2)$
3	U_{31} $(A_3\ B_3\ C_3)$	U_{32} $(D_3\ E_3\ F_3)$	U_{33} $(G_3\ H_3\ I_3)$

Figure 9: The variables in (231)-, (312)-conjugate latin squares

$\mathbf{true} \rightarrow \mathbf{dom(1)}, \mathbf{dom(2)}, \mathbf{dom(3)}, \mathbf{dom(4)}, \mathbf{dom(5)}.$

$\mathbf{dom(M)}, \mathbf{dom(N)} \rightarrow$
$\quad \mathbf{p(M,N,1)}; \mathbf{p(M,N,2)}; \mathbf{p(M,N,3)};$
$\quad \mathbf{p(M,N,4)}; \mathbf{p(M,N,5)}.$

$\mathbf{p(Y,X,V1)}, \mathbf{p(V1,Y,V2)}, \mathbf{p(V2,Y,V)}, \{\mathbf{V} \backslash = \mathbf{X}\} \rightarrow \mathbf{false}.$

$\mathbf{p(X,X,V)}, \{\mathbf{V} \backslash = \mathbf{X}\} \rightarrow \mathbf{false}.$

$\mathbf{p(X,Y1,V)}, \mathbf{p(X,Y2,V)}, \{\mathbf{Y1} \backslash = \mathbf{Y2}\} \rightarrow \mathbf{false}.$

$\mathbf{p(X1,Y,V)}, \mathbf{p(X2,Y,V)}, \{\mathbf{X1} \backslash = \mathbf{X2}\} \rightarrow \mathbf{false}.$

$\mathbf{p(X,5,Y)}, \{\mathbf{X1\ is\ X-1}, \mathbf{Y < X1}\} \rightarrow \mathbf{false}.$

Figure 10: MGTP rules for QG5.5

Table 1: Comparison of experimental results (QG5)

	Failed Branches				
Order	DDPP*	FINDER*	MGTP*	CP	CMGTP
9	15	40	239	15	15
10	50	356	7026	38	38
11	136	1845	51904	117	117
12	443	13527	2749676	372	372
13				13927	13927
14				64541	64541
15				130425	130425
16				19382469	

4.2.3 CMGTP

The structure of CMGTP model generation processes is basically the same as MGTP. The differences between CMGTP and MGTP lie in the unit refutation processes and the unit simplification processes with negative atoms. We can use negative atoms explicitly in CMGTP to represent constraints. If there exist P and $\neg P$ in M then $false$ is derived by the unit refutation mechanism. If for a unit clause $\neg P_i \in M (P_i \in M)$, there exists a disjunction which includes $P_i (\neg P_i)$, then $P_i (\neg P_i)$ is removed from that disjunction by the unit simplification mechanism.

The refutation and simplification processes added to MGTP guarantee that for any atom $P \in M$, P and $\neg P$ are not both in the current M, and disjunctions in the current D have already been simplified by all unit clauses in M.

Fig.10 shows the original MGTP rules for QG5.5. These rules can be rewritten into CMGTP rules in order to propagate negative information using negative atoms. For example, the original MGTP rule for QG5,

$$p(Y,X,A), p(A,Y,B), p(B,Y,C), X \neq C \rightarrow false$$

can be rewritten in CMGTP rules as follows:

$$p(Y,X,A), p(A,Y,B) \rightarrow p(B,Y,X).$$
$$p(Y,X,A), \neg p(B,Y,X) \rightarrow \neg p(A,Y,B).$$
$$\neg p(B,Y,X), p(A,Y,B) \rightarrow \neg p(Y,X,A).$$

In the above rules, negative information is propagated by using the last 2 rules.

In this sense, CMGTP can be considered as a meta language for representing constraint propagation.

4.2.4 Experimental Results

Table 1 compares the experimental results for QG problems on CP, CMGTP and other systems. The numbers of failed branches generated by CP and CMGTP are almost equal to DDPP and less than those from FINDER and MGTP. In fact, we confirmed that CP and CMGTP have the same pruning ability as DDPP by comparing the proof trees generated by these systems. The slight differences in the number of failed branches were caused by the different selection functions used.

For general performance, CP was superior to the other systems in almost every case. In particular we found that no model exists for QG5.16 by running CP on a Sparc-10 for 21 days in October 1993. It was the first new result we obtained. On the other hand, CMGTP is about 10 times slower than CP. The reason of this difference is caused mainly by the manipulation of term memory. We are now trying to make term memory efficient in CMGTP, and also to parallelize CMGTP processes on PIM/m and parallel UNIX machines.

5 Applications

5.1 Embedding Negation as Failure into MGTP

Negation as failure is one of the most important techniques developed in the logic programming field, and logic programming supporting this feature can be a powerful knowledge representation tool. Recently, declarative semantics such as the *answer set* semantics[11] have been given to extensions of logic programs containing both negation as failure (*not*) and classical negation ($\neg$), where the negation as failure operator is considered to be a nonmonotonic operator.

However, for such extended classes of logic programs, the top-down approach cannot be used for computing the answer set semantics because there is no local property in evaluating programs. Thus, we need bottom-up computation for correct evaluation of negation as failure formulas. For this purpose, we use the framework of MGTP, which can find the answer sets as the fixpoint of model candidates.

Here, we introduce a method[18] to transform any logic program (with negation as failure) into a *positive disjunction program* (without negation as failure) [27] for which MGTP can compute the minimal models.

5.1.1 Translation into MGTP rules

(1) Positive Disjunction Programs

A *positive disjunctive program* is a set of rules of the form:

$$A_1 \mid \ldots \mid A_l \leftarrow A_{l+1}, \ldots, A_m\,, \qquad (1)$$

where $m \geq l \geq 0$ and each A_i is an atom.

The meaning of a positive disjunctive program Σ can be given by the *minimal models* of Σ [27].

The minimal models of positive disjunctive programs can be computed using MGTP. We express each rule of the form (1) in a positive disjunctive program as the following MGTP input clauses:

$$A_{l+1}, \ldots, A_m \rightarrow A_1 \mid \ldots \mid A_l\,. \qquad (2)$$

(2) General and Extended Logic Programs

MGTP can also compute the stable models of a general logic program[10] and the answer sets of an extended disjunctive program[11] by translation into positive disjunctive programs.

An *extended logic program* is a set of rules of the form:

$$L_1 \mid \ldots \mid L_l \leftarrow L_{l+1}, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n \quad (3)$$

where $n \geq m \geq l \geq 0$ and each L_i is a literal. This logic program is called a general logic program if $l \leq 1$, and each L_i is an atom.

While a general logic program contains negation-as-failure but does not contain classical negation, an extended disjunctive program contains both of them.

In evaluating $\mathit{not}\, L$ in a bottom-up manner, it is necessary to interpret $\mathit{not}\, L$ with respect to a fixpoint of computation because, even if L is not currently proved, L might be proved in subsequent inferences. When we have to evaluate $\mathit{not}\, L$ in a current model candidate we split the model candidate in two: (1) the model candidate where L is assumed not to hold, and (2) the model candidate where it is necessary that L holds. Each negation-as-failure formula $\mathit{not}\, L$ is thus translated into negative and positive literals with a modality expressing belief, i.e., "disbelieve L" (written as $\neg \mathsf{K}\, L$) and "believe L" (written as $\mathsf{K} L$).

Based on the above discussion, we translate each rule of the form (3) to the following MGTP rule:

$$L_{l+1}, \ldots, L_m \rightarrow H_1 \mid \ldots \mid H_l \mid \mathsf{K} L_{m+1} \mid \ldots \mid \mathsf{K} L_n \quad (4)$$

where $H_i \equiv \neg \mathsf{K} L_{m+1} \wedge \ldots \wedge \neg \mathsf{K} L_n \wedge L_i$ $(i = 1, \ldots, l)$

For any MGTP rule of the form (4), if a model candidate S' satisfies $L_{l+1}, \ldots, L_m$, then S' is split into $n-m+l$ $(n \geq m \geq 0,\ 0 \leq l \leq 1)$ model candidates.

In order to reject model candidates when their guesses turn out to be wrong, the following two schemata (integrity constraints) are introduced:

$$\neg \mathsf{K} L, L \rightarrow \qquad \text{for every literal } L \in \mathcal{L}\,. \qquad (5)$$

$$\neg \mathsf{K} L, \mathsf{K} L \rightarrow \qquad \text{for every literal } L \in \mathcal{L}\,. \qquad (6)$$

Added to the schemata above, we need the following 3 schemata to deal with classical negation. Below, $\overline{L}$ is the literal complement to a literal L.

$$L, \overline{L} \rightarrow \qquad \text{for every literal } L \in \mathcal{L}\,. \qquad (7)$$

$$\mathsf{K} L, \overline{L} \rightarrow \qquad \text{for every literal } L \in \mathcal{L}\,. \qquad (8)$$

$$\mathsf{K} L, \mathsf{K}\overline{L} \rightarrow \qquad \text{for every literal } L \in \mathcal{L}\,. \qquad (9)$$

Next is the condition to guarantee stability at fixpoint that all of the guesses made so far in a model candidate S are correct.

For every ground literal L, \quad if $\mathsf{K} L \in S$, then $L \in S$.

The above computation by the MGTP is sound and complete with respect to the answer set semantics.

5.1.2 Remarks

The technique presented here is simply based on a bottom-up model generation method together with integrity constraints over K-literals expressed by object-level schemata on the MGTP.

Compared with other approaches, the proposed method has several computational advantages: in a word, it can find *all* minimal models for *every* class of groundable logic program or disjunctive database, *incrementally*, *without backtracking*, and *in parallel*.

This method has been applied to a legal reasoning system [29]. We can see some advantages of the proposed method from the viewpoint of this application.

5.2 Bottom-up Abduction by MGTP

Abduction has recently been recognized as a very important form of reasoning for various AI problems. An *abductive framework* is a pair (Σ, Γ), where Σ is a set of formulas and Γ is a set of literals. Given a closed formula G, a set E of ground instances of Γ is an *explanation of* G *from* (Σ, Γ) if

1. $\Sigma \cup E \models G$, and
2. $\Sigma \cup E$ is consistent.

The computation of explanations of G from (Σ, Γ) can be seen as an extension of proof-finding by introducing a set of hypotheses from Γ that, if they could be proved by preserving the consistency of the augmented theories, would complete the proof of G.

5.2.1 Abduction by Model Generation

Here, we introduce a method[19], which we call the Skip method, to implement abductive reasoning systems built on the MGTP. We consider the first-order abductive framework (Σ, Γ), where Σ is a set of *range-restricted Horn clauses* and Γ is a set of *atoms* (*abducibles*).

The simplest way to implement reasoning with hypotheses is as follows. For each hypothesis H in Γ, we supply a clause of the form:

$$\rightarrow H \mid \neg \mathsf{K} H\,, \tag{10}$$

where $\neg \mathsf{K} H$ means that "H is not assumed to be true in the model". Namely, each hypothesis is assumed either to hold or not to hold. For dealing with $\neg \mathsf{K} H$, we need the axiom schema as an integrity constraint :

$$\neg \mathsf{K} H,\, H \rightarrow \qquad \text{for every hypothesis } H\,. \tag{11}$$

The above technique, however, may generate $2^{|\Gamma|}$ model candidates. To reduce them as much as possible, we can use a method similar to the implementation of negation as failure in Section 5.1, that delays case-splitting for each hypothesis. That is, we do not supply any clause of the form (10) for any hypothesis of Γ, but, instead, introduce hypotheses when they are necessary. When abducibles $H_1, \ldots, H_n$ $(n \geq 0)$ from Γ appear in the antecedent of a Horn clause in Σ as:

$$A_1 \wedge \ldots \wedge A_l \wedge \underbrace{H_1 \wedge \ldots \wedge H_n}_{abducibles} \rightarrow C\,,$$

we transform this clause into a non-Horn clause:

$$A_1, \ldots, A_l \rightarrow H_1, \ldots, H_n, C \mid \neg \mathsf{K} H_1 \mid \ldots \mid \neg \mathsf{K} H_n\,. \tag{12}$$

In this transformation, each hypothesis H_j in the antecedent is shifted to the right-hand side of the clause in the form of $\neg \mathsf{K} H_j$. Moreover, each H_j is *skipped* instead of being resolved, and is added to consequent C of the rule since C becomes true whenever all A_i's and H_j's are true.

5.2.2 Remarks

The Skip method has been applied to a logic-circuit design system[19]. Although we need to further investigate how to avoid possible combinatorial explosion in constructing model candidates for the Skip method, we conjecture that the Skip method will be promising from the viewpoint of OR-parallelism.

5.3 Modal Logic in MGTP

In this section, we describe a technique to implement efficient modal theorem provers that transforms modal formulae into input clauses for MGTP. The technique, called the *modal clause transformation method*, is based on partial evaluation of the rewriting rules for the modal tableau method.

5.3.1 Motivations and Backgrounds

Modal logics have been gaining popularity in various domains of Computer Science. For such applications, modal logics require fast and efficient theorem provers. Recently, the translation proof method [30] which translates modal formulae into classical formulae has been proven to be useful because it can be applied to various modal systems. The translation approach has another merit in that it can employ many control strategies developed for theorem proving in classical logic. Unfortunately, the previously proposed methods do not address the issue of controlling inference to reduce the search space.

To take advantage of the above, we proposed a version of the translation method, called the *modal clause transformation method*[1]. This method is based on the following meta-programming method[21, 22].

5.3.2 Meta-programming Method

The meta-programming method implements the rewriting rules for the tableau method as schemata encoded as MGTP input clauses, and simulates the modal tableau method[5] on MGTP. The following are the MGTP input clauses representing the tableaux expansion rules and the close condition:

α-rule :
$f(F \vee G, W) \;\rightarrow\; f(F, W), f(G, W).$
$t(F \wedge G, W) \;\rightarrow\; t(F, W), t(G, W).$

β-rule :
$t(F \vee G, W) \;\rightarrow\; t(F, W); t(G, W).$
$f(F \wedge G, W) \;\rightarrow\; f(F, W); f(G, W)$

ν-rule:
$t(\Box F, W), path(W, V) \;\rightarrow\; t(F, V).$

π-rule:
$f(\Box F, W) \;\rightarrow\; \{new_world(V)\}, path(W, V), f(F, V).$

close condition:
$t(F, W), f(F, W) \;\rightarrow\; false.$

where F, G are modal formulae; $t(F,W)/f(F,W)$ represents that F is true/false in the world W; $path(W,V)$ represents that V is accessible from W; $\{\cdots\}$ is a sequence of KL1 predicates and $new_world(V)$ creates a new world V.

5.3.3 Modal Clause Transformation Method

Since the meta-programming method simulates the tableau method on MGTP, the prover tends to create too many branches. We therefore apply a partial evaluation technique to the meta-programming method, thereby suppressing the generation of branches which can easily be checked to be closed. We call this translation method the *basic modal clause transformation method*.

We represent modal formulae as sets of *modal clauses*. A modal clause is a disjunction of *modal atoms* and their negations, where a modal atom is a propositional atom or a modal formula with a modal operator $\Box$ followed by a modal clause. Any modal formula can be represented as a set of modal clauses. This can be proved by induction on the structure of modal formulae.

Given a set of modal clauses, the translator first tries to apply the α rules or β rules of the modal tableau methods to the modal clause set. The result consists of *signed modal atoms*, that is, MGTP atoms in the form of either $t(\varphi,w)$ or $f(\varphi,w)$ where φ is a modal atom. A signed modal atom $t(\varphi,w)$ indicates that a modal atom φ is true in world w, and $f(\varphi,w)$ indicates that φ is false in w.

If there is a signed modal atom to which the π rule (or the ν rule) can be applied, an MGTP clause that is a specialization of the π rule (or the ν rule) is generated. For example, the following MGTP clause is generated for the signed modal atom $t(\Box(p \wedge \Box q \supset \Box r \vee s), W)$.

$$t(\Box(p \wedge \Box q \supset \Box r \vee s), W), path(W,V), t(p,V), t(\Box q,V) \\ \rightarrow t(\Box r,V) \mid t(s,V).$$

Where $path(W,V)$ denotes that world V is accessible from world W.

By translating modal formulae so that close condition testing is replaced with pattern matching in the antecedents of translated clauses, instead of generating such branches, we can suppress the generation of redundant branches that can easily be checked to be closed. We have proved that this transformation preserves the satisfiability.

5.3.4 Incorporating NHM

In practical applications, formulae usually contain many subformulae irrelevant to the proof. To avoid the generation of irrelevant branches, we adapted the NHM method that transforms input clauses so as to simulate top-down reasoning. We analyze input modal formulae to incorporate control information specific to a given input modal formula into its translated formula. The integrated translation method is called the *NHM modal clause transformation method*.

We apply the NHM method only to positive information (i.e., $t(\varphi,W)$). For example, the following MGTP clauses are generated for the signed modal atom $t(\Box(p \wedge \Box q \supset \Box r \vee s), W)$.

$$goal(\Box r,V), goal(s,V), path(W,V) \rightarrow goal(p,V), \\ goal(\Box q,V), goal(\Box(p \wedge \Box q \supset \Box r \vee s),W).$$

$$goal(\Box r,V), goal(s,V), t(\Box(p \wedge \Box q \supset \Box r \vee s),W), \\ path(W,V), t(p,V), t(\Box q,V) \rightarrow t(\Box r,V) \mid t(s,V).$$

5.3.5 Evaluation

The cost of the modal clause transformation has been shown to be linear to the length of the input modal clauses.

We have tested the above mentioned methods on several modal formulae. The basic modal transformation method is generally superior to the meta-programming method. The reason is that the meta-programming method generates branches for every atom in the antecedents, while the basic transformation method processes conjunctive matching of the antecedents instead of generating branches. For formulae which contain irrelevant subformulae, the NHM transformation method is much better than the other two methods since it does not generate irrelevant branches. This merit becomes more important as the program size increases.

5.3.6 Related Work

So far, there have been few reports on efficient strategies for modal theorem proving. Auffray et al. [2] proposed a modal version of resolution strategies such as input and linear resolution. These strategies, however, impose restrictions on modal formulae, the so-called *modal Horn clauses*, while our method can be applied to any modal formula.

From the viewpoint of the modal clause transformation method, all previously proposed translation methods compute the π rule and the ν rule completely. Compared with previously proposed translation methods, the modal clause transformation method offers the following advantages.

1. Translated clauses are range-restricted. Hence, efficient theorem proving is possible, as matching is sufficient instead of full unification.

2. It is possible to restrict the invocation of the π rule and the ν rule. We can therefore avoid generating branches irrelevant to the proof.

5.4 Mode Analysis of FGHC Programs

This section describes a method of mode analysis[33] for FGHC (Flat GHC) programs[34] with MGTP. Mode analysis is a kind of fixpoint computation that corresponds to generating a model in MGTP. The generated model includes mode information, which lets us know a variable's mode and mode consistency in the program. Mode information is useful not only for compiler optimization but also for static bug detection.

5.4.1 Mode Constrains of FGHC programs

In our method, formulas representing mode constraints [35] are translated into a set of clauses for MGTP. Mode analysis of the entire program is reduced to computing a model of the set of clauses.

We outline mode constraints using the simple FGHC program listed below.

```
s([],          D    ) :- true | t(D).
s([push(X)|S],D    ) :- true | s(S,[X|D]).
s([pop(X)|S], [Y|D]) :- true | X=Y, s(S,D).
```

The mode of the first argument is *in* because predicate s waits for its first argument being instantiated with `[]` or a cons cell. This is represented by $m(\langle s,1\rangle) = in$. $\langle s,1\rangle$ indicates the first argument of the predicate s. The car part of the first argument of s is represented by a concatenation like $\langle s,1\rangle\langle .,1\rangle$ ($\langle .,1\rangle$ represents the car part, $\langle .,2\rangle$ represents the cdr part.). We can point out any position of terms and atoms using this notation, called *path*. $m(Path)$ represents the mode pointed out by $Path$. Similarly, we get $m(\langle s,1\rangle\langle .,1\rangle) = in$ and $m(\langle s,2\rangle) = in$.

The variable D of the first clause appears in the head and body parts. From this, we can say that the mode of the second argument of s equals to the mode of the first argument of t. This is represented by $m/\langle s,2\rangle = m/\langle t,1\rangle$. m/p means $\forall q((m/p)(q) = m(pq))$ where p and q are paths and pq is a concatenation of p and q.

Similarly, when looking at variables D, S and X of the second clause, we get $m/\langle s,2\rangle = m/\langle s,2\rangle\langle .,2\rangle$, $m/\langle s,1\rangle\langle .,2\rangle = m/\langle s,1\rangle$ and $m/\langle s,1\rangle\langle .,1\rangle\langle u,1\rangle = m/\langle s,2\rangle\langle .,1\rangle$($u$ is an abbreviation for *push*).

The mode of the path indicated by X is the inverse of the mode of Y, because both X and Y in the third clause appear in the head part and are unified with each other in the body part. This is represented by $m/\langle s,1\rangle\underline{\langle .,1\rangle}\langle o,1\rangle = \overline{m/\langle s,2\rangle\langle .,1\rangle}$($o$ is the abbreviation for *pop*). $\overline{m()}$ means mode inversion.

5.4.2 Mode Analysis in MGTP

The mode constraints have two kinds of forms: $m() = in$ (or $m() = out$) and $m/p = m/q$ (or $m/p = \overline{m/q}$). Roughly speaking, $m() = in$ is translated to a positive clause $true \rightarrow m() = in$, and $m/p = m/q$ is translated to two mixed clauses $m/p \rightarrow m/q$ and $m/q \rightarrow m/p$.

Table 2 shows the transformation of the mode constraints. $m(Path, Mode)$ is a relation that represents that the mode of path $Path$ is $Mode$. And, a path is represented by a list notation. For example, $\langle s,1\rangle\langle .,1\rangle$ is translated to $[s/1,./1]$.

Mode consistency is also checked by a negative clause: $m(P,M), m(P,\overline{M}) \rightarrow false$ which means that the mode is inconsistent when a path P has two modes: *in* and *out*.

Example For the above program, we have 3 positive (one literal) clauses and 8 mixed clauses. We start with an empty model candidate $M_0 = \emptyset$. M_0 is first expanded to $M_1 = \{m([s/1],in), m([s/2],in), m([s/1,./1],in)\}$, by applying the model extension rule to the 3 positive clauses.

One direction of rule (4) $m([s/2|X],M) \rightarrow m([t/1|X],M)$ is applicable to M_1 because $m([s/2],in)(\in M_1)$ can be unified with the antecedent $m([s/2|X],M)$ by substitution $\{[]/X, in/M\}$. So we get the consequent $m_1 = m([t/1],in)$. This means that M_1 is extended by (4) to $M_2 = M_1 \cup \{m_1\}$. Repeating a similar production of mode information, we get the final result: $M_1 \cup \{m([t/1],in), m([s/2,./2],in), m([t/1,./2],in)\}$.

6 Conclusion

We have overviewed research and development of the parallel theorem proving system MGTP in both the FGCS and the Follow-on Projects.

To show the effectiveness of a large-scale parallel theorem prover, we first aimed to achieve a linear speedup effect on the PIM and to solve "hard problems" which are difficult for sequential systems to prove because of the limits of time and space.

We mainly challenged mathematical theorems such as the condensed detachment and quasigroup existence problems. We obtained better results than expected. We succeeded in proving difficult problems including open quasigroup existence problems that sequential provers could not prove.

However, some parallelization issues became clear through the development of parallel MGTP.

Although the current AND parallel implementation shows good parallel performance, it still has the sequentiality of global subsumption testing and lacks memory scalability. The sequential problem would become more serious when we introduce a sorting strategy for the selection of model-extending atoms.

In the current parallel implementation, we have to properly use an AND parallel MGTP for Horn problems and an OR parallel MGTP for non-Horn problems separately. Thus, it is necessary to build a parallel version of MGTP which can combine AND- and OR-parallelization for proving a set of general clauses.

Table 2: Mode constrains → MGTP clause transformation

Mode constrains	MGTP clause	
$m(\langle s,1\rangle) = in$	$true \rightarrow m([s/1], in)$	(1)
$m(\langle s,2\rangle) = in$	$true \rightarrow m([s/2], in)$	(2)
$m(\langle s,1\rangle\langle .,1\rangle) = in$	$true \rightarrow m([s/1,./1], in)$	(3)
$m/\langle s,2\rangle = m/\langle t,1\rangle$	$m([s/2\|X], M) \leftrightarrow m([t/1\|X], M)$	(4)
$m/\langle s,2\rangle = m/\langle s,2\rangle\langle .,2\rangle$	$m([s/2\|X], M) \leftrightarrow m([s/2,./2\|X], M)$	(5)
$m/\langle s,1\rangle\langle .,1\rangle\langle u,1\rangle = \underline{m/\langle s,2\rangle\langle .,1\rangle}$	$m([s/1,./1,u/1\|X], M) \leftrightarrow m([s/2,./1\|X], M)$	(6)
$m/\langle s,1\rangle\langle .,1\rangle\langle o,1\rangle = \overline{m/\langle s,2\rangle\langle .,1\rangle}$	$m([s/1,./1,o/1\|X], M) \leftrightarrow m([s/2,./1\|X], \overline{M})$	(7)

† $m(P,M) \leftrightarrow m(P',M')$ represents two MGTP clauses:
$m(P,M) \rightarrow m(P',M')$ and $m(P',M') \rightarrow m(P,M)$.

‡ $\overline{M}$ is the inverse mode of M i.e., $\overline{in} = out$ and $\overline{out} = in$.

To enhance the MGTP's pruning ability, we extended the MGTP features in two directions: non-Horn Magic set (NHM) and Constraint MGTP (CMGTP). NHM is a key technology for making MGTP a practical prover and applicable to several applications such as disjunctive databases, abductive reasoning and modal logic systems. The essence of the NHM method is to simulate a top-down evaluation in a framework of bottom-up computation by statical clause transformation to propagate goal (negative) information, thereby pruning search spaces. This propagation is closely related to the technique developed in CMGTP to manipulate (negative) constraints. Thus, further research is needed to clarify whether the NHM method can be incorporated to CMGTP.

Lastly, we have presented several techniques to develop applications on top of MGTP. We have shown that it is easy to implement on MGTP negation as failure, abductive reasoning and modal logic systems that are necessary for knowledge representation and advanced problem-solving systems. Mode analysis of FGHC programs presented here is useful for both the optimization of KL1 compilers and NHM transformation.

The basic idea of these techniques is to translate formulas with special properties, such as non-monotonicity and modality, into first order clauses (MGTP clauses) on which MGTP works as meta-interpreter. The manipulation of these properties is thus reduced to generate-and-test problems for model candidates. These can then be handled by the MGTP very efficiently through case-splitting of disjunctive consequences and rejection of inconsistent model candidates.

The final goal of the MGTP research is to integrate automated reasoning technology with logic programming technology. MGTP embodies an exhaustive searching property which KL1 lacks and has the potential capacity to give a new logic programming paradigm based on bottom-up computation.

In the future, we will develop a new version of MGTP with the KLIC system[6] running on a UNIX machine for portability and available to other people outside ICOT. We will also make further extensions to MGTP in order to develop MGTP as a programming language system, and we will explore application areas suitable for MGTP.

Acknowledgments

The research on the parallel theorem proving system in the Follow-on Project was carried out by the MGTP group in the First Research Department at ICOT. The author would like to thank all who have engaged in the development of MGTP.

Thanks are also due to Prof. Kazuhiro Fuchi of the University of Tokyo, Prof. Koichi Furukawa of Keio University, Prof. Fumio Mizoguchi of the Science University of Tokyo, Prof. Makoto Amamiya of Kyushu University, and all the PTP-TG members for their fruitful discussions and helpful comments.

Finally, we would like to express our gratitude to Dr. Shunichi Uchida, the director of the ICOT Research Center, and Dr. Takashi Chikayama, the manager of the First Research Department, who have given encouragement and support in this work.

References

[1] J. Akahani, K. Inoue, and R. Hasegawa. Bottom-Up Modal Theorem Provers based on Modal Clause Transformation. Technical Report 874, ICOT, 1994.

[2] Y. Auffray, P. Enjalbert, and J-J. Hebrard. Strategies for Modal Resolution: Results and Problems. *J. Automated Reasoning*, 6:1–38, 1990.

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

[4] François Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.

[5] M. Fitting. First-Order Modal Tableaux. In *J. Automated Reasoning*, volume 4, pages 191–213, 1988.

[6] T. Fujise, T. Chikayama, K. Rokusawa, and A. Nakase. KLIC: A Portable Implementation of KL1. In *Proc. Int. Symp. on Fifth Generation Computer Systems*, December 1994.

[7] H. Fujita and R. Hasegawa. A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm. In *Proc. 8th Int. Conf. on Logic Programming*. The MIT Press, 1991.

[8] M. Fujita, R. Hasegawa, M. Koshimura, and H. Fujita. Model Generation Theorem Provers on a Parallel Inference Machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 357–375, Tokyo, 1992.

[9] M. Fujita, J. Slaney, and F. Bennett. Automatic Generation of Some Results in Finite Algebra. In *Proc. IJCAI-93*, 1993.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth Int. Conf. and Symp. of Logic Programming*, pages 1070–1080, Seattle, WA, 1988.

[11] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[12] R. Hasegawa and M. Fujita. Parallel Theorem Provers and Their Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 132–154, Tokyo, 1992.

[13] R. Hasegawa and M. Koshimura. An AND Parallelization Method for MGTP and Its Evaluation. In *Proc. First Int. Symp. on Parallel Symbolic Computation*, pages 194–203, Linz, 1994.

[14] R. Hasegawa, M. Koshimura, and H. Fujita. Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers. In *Proc. Int. Workshop on Automated Reasoning*, pages 221–238, Beijing, 1992. also in ICOT TR-751 1992.

[15] R. Hasegawa, M. Koshimura, and H. Fujita. MGTP: A Parallel Theorem Prover Based on Lazy Model Generation. In *Proc. 11th Int. Conf. on Automated Deduction*, pages 776–780, 1992. (System Abstract).

[16] R. Hasegawa, Y. Ohta, and K. Inoue. Non-Horn Magic Sets and Their Relation to Relevancy Testing. Technical Report 834, ICOT, 1993. Dagstuhl Seminor on Deduction in Germany, 1993 Workshop on Finite Domain Theorem Proving, 1994.

[17] R. Hasegawa and Y. Shirai. Constraint Propagation of CP and CMGTP: Experiments on Quasigroup Problems. In *Proc. Workshop 1C (Automated Reasoning in Algebra), CADE-12*, Nancy, France, 1994.

[18] K. Inoue, M. Koshimura, and R. Hasegawa. Embedding Negation as Failure into a Model Generation Theorem Prover. In *Proc. 11th Int. Conf. on Automated Deduction*, pages 400–415. Springer-Verlag, 1992. LNAI 607.

[19] K. Inoue, Y. Ohta, R. Hasegawa, and M. Nakashima. Bottom-Up Abduction by Model Generation. In *Proc. IJCAI-93*, 1993. ICOT TR-816.

[20] M. Koshimura and R. Hasegawa. A Method for AND Parallelizing Model Generation Theorem Provers. To appear in Trans. IEICE D-1 (in Japanese).

[21] M. Koshimura and R. Hasegawa. Modal Propositional Tableaux on a Model Generation Theorem Prover. In *Proc. Logic Programming Conf. '91*, pages 43–52, Tokyo, 1991. ICOT. also in ICOT TR-665 (in Japanese).

[22] M. Koshimura and R. Hasegawa. Modal Propositional Tableaux in a Model Generation Theorem Prover. In *Proc. Third Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 145–151, UK, 1994. also in ICOT TR-860.

[23] D. W. Loveland, D. W. Reed, and D. S. Wilson. SATCHMORE: SATCHMO with RElevancy. Technical report, Department of Computer Science, Duke University, Durham, North Carolina, 1993. CS-1993-06.

[24] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. In *Proc. 9th Int. Conf. on Automated Deduction*, Argonne, Illinois, 1988.

[25] W. W. McCune. *OTTER 2.0 Users Guide*. Argonne National Laboratory, 1990.

[26] W. W. McCune and L. Wos. Experiments in Automated Deduction with Condensed Detachment. In *Proc. 11th Int. Conf. on Automated Deduction*, pages 209–223, Saratoga Springs, NY, 1992.

[27] J. Minker. On indefinite databases and the closed world assumption. In *Proc. Sixth Int. Conf. on Automated Deduction*, pages 292–308. Springer-Verlag, 1982. Lecture Notes in Computer Science **138**.

[28] H. Nakashima, K. Nakajima, S. Kondoh, Y. Takeda, Y. Inamura, S. Onishi, and K. Masuda. Architecture and Implementation of PIM/m. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 425–435, Tokyo, 1992.

[29] K. Nitta, Y. Ohtake, S. Maeda, M. Ono, H. Ohsaki, and K. Sakane. HELIC-II: A Legal Reasoning System on the Parallel Inference Machine. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 1115–1124, Tokyo, June 1992.

[30] H. J. Ohlbach. A resolution calculus for modal logics. In *Proc. 9th Int. Conf. on Automated Deduction*, pages 500–516, 1988.

[31] M. Stickel. The path-indexing method for indexing terms. Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1989.

[32] Mark E. Stickel. Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction. Technical Report 664, ICOT, July 1991. A revised version is to appear in *J. Automated Reasoning*.

[33] E. Tick and M. Koshimura. Static Mode Analyses of Concurrent Logic Languages. Technical Report 875, ICOT, 1994. To appear in J. Programming Languages Design and Implementation.

[34] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer J.*, 33:494–555, December 1990.

[35] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. Revised version of the paper in *Proc. 7th Int. Conf. on Logic Programming*, 1990.

[36] D. S. Wilson and D. W. Loveland. Incorporating Relevancy Testing in SATCHMO. Technical report, Department of Computer Science, Duke University, Durham, North Carolina, 1989. CS-1989-24.