Model Generation Thoerem Prover : MGTP

Manual

Institute for New Generation Computer Technology

March 1995

Contents

Preface

This manual contains the guide to functions and operations of the Model Generation Thorem Prover MGTP developed by Institute for New Generation Computer Technology.

The MGTP represents a bottom-up thorem prover based on first order predicate logic. It eliminates redundancy in conjunctive matching to a possible extent, and operates in an extremely efficient fashion on parallel machines thanks to its elaborate implementation techniques to serve as a problem solver.

With its simplicity for problem description, the MGTP can be viewed as a knowledge representation language based on first order predicate logic. Among many features, it can process Non-Horn logical expression which otherwise cannot be followed with Prolog, and guarantees the completeness of search which the parallel logic programming language KL1 lacks.

The MGTP is available in three types depending on the nature of problems to be applied; ground version MGTP (MGTP/G), non-ground version MGTP (MGTP/N), and constraint MGTP (CMGTP) which is extended to solve constraint satisfaction problem, each being implemented with KL1, Prolog, and Klic.

In ground version MGTP, all candidate models to be generated are terms which contain no variable, that is, ground terms, and the disjunctions can be described in the consequent part of the rule. The non-ground version MGTP permits terms including variables, but no disjunctions can be described in the consequent part of the rule in the current implementation. CMGTP is a MGTP which is extended to solve constraint satisfaction problems, and can represent constraint propagation rules by means of negative atoms in order for efficient pruning. In terms of parallelismism, ground version MGTP and CMGTP have OR parallelism, and non-ground version MGTP has AND parallelism respectively implemented, thus achieving the number effect close to linearity for several benchmark problems.

The IFS makes all these software public. This manual provides functonal outline of these systems and describes how to operate for each version.

For comment on this manual or software-related bug report and comments, contact the following.

Institute for New Generation Computer Technology

Ryuzo Hasegawa	(hasegawa@icot.or.jp)
Mitusyuki Koshimura	(koshi@icot.or.jp)
Yasuyuki Shirai	(shirai@icot.or.jp)
Phone:	(03) 3456-4365
Fax:	(03) 3456-1618

Construction of the manual

The manual consists of the following chapters.

Chapter 1. MGTP Functions
This chapter describes the basic functions of MGTP, in particular, the framework of method for model generation and for eliminating redundancy. For functions specific to ground version, non-ground version and constraint MGTP, refer to the manual for functions for each relevant system.

Chapter 2. MGTP Input Language Manual
This chapter describes the syntax for MGTP input clauses with examples provided as required.

Chapter 3. Translator
For MGTP, the translator is required to translate the input clauses witten by the user into the language environment (KL1, Klic, Prolog) in which those clauses are to be executed. This chapter gives a brief outline of the translator function.

Chapter 4. MGTP/G Manual
This chapter describes the function of ground version MGTP, and the operation for each of KL1, Klic, and Prolog versions.

Chapter 5. MGTP/N Manual
This chapter describes the function of non-ground version MGTP, and the operation for each of KL1, Klic and Prolog .

Chapter 6. CMGTP Manual
This chapter describes the function of constraint MGTP (CMGTP) and the operation for each of KL1, Klic and Prolog versions.

Chapter 1

Description of MGTP function

This chapter relates to the basic function of the MGTP, and in particular, describes the framework of model generation, and how to eliminate redundancy.

1.1 Preface

The logic programming, which is based on the Horn Logic and plays a key role in the technology of the fifth generation computer project, has drawn the attention of many researchers as an innovative computing paradigm and has experienced a remarkable progress. In the field of theorem proving, Prolog is highlighted as a major achievement in the logic programming, and an attempt has been made to create a highly efficient theorem prover by means of efficient implementation technology applied to Prolog. [Sti88] [MB88]. These systems capitalize on the fact that the reasoning for Horn clauses can be implemented in an extremely efficient manner as the execution of Prolog programs.

Since the efficiency for a programming language is given a high priority in the logic type language including Prolog, part of the logical property required for treating first order predicate logic is sacrificed as such;

1) unification with occur check is eliminated for simplicity;

2) the clause with disjunctive head (Horn clause) cannot be processed;

3) NOT is a negation as failure, and classical logical NOT cannot be handled.

On the other hand, the parallel logic type language KL1 makes it easier to describe parallel programs which is based on the concept of inter-process communication with parallel processes while inheriting the advantage of Prolog as represented by logical variables. KL1 has an advantage of minimum chance of synchronization-related bugs, and is an excellent language in order to implement a parallel system in an efficient manner.

However, KL1 is a way back from Prolog in a logical viewpoint. In reality, KL1 clauses are no longer pure Horn clauses and contain non-logical elements such as commit operator. What's more, completeness is lost such that unification fails and backtracks are not permitted.

To cope with the above problems, generally acceptable solution is meta-programming technique with the penalty of appreciable deterioration of execution efficiency.

In order to solve the above problems and implement the theorem prover of first order predicate logic in an efficient manner with KL1, we adopted the model generation which provides the basis for SATCHNO as a prover. In the model generation method, if the target is a clause set which satisfies range-restrictedness, need for unification with occur-check is eliminated and efficient implementation is made possible.

The following paragraphs describe the principle of model generation and implementation of model generation type theorem prover with KL1 language briefly. It is followed by the description of two methods; RAMS and MERC; which eliminate redundancy in model generation.

1.2 Model generation

An any first order predicate formula can be transformed to clausal form (conjunctive normal form). One clause is a result of positive/negative literals disjunctively linked in an optional sequence. In this example, the following clause - $\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m$ - is expressed in the following implicational form;

$$A_1, \ldots, A_n \rightarrow B_1; \ldots; B_m$$

, where $A_i (1 \leq i \leq n), B_j (1 \leq j \leq m)$ is an atom, "," is a logical AND connector, ";" is a logical OR connector, "$\rightarrow$" is a implication. Atoms of left side of "$\rightarrow$" are called antecedent (literals), right side is called consequent (literals). If the antecedent is null, it is expressed in $A_0 = \mathit{true}$ and called positive clause, and if the consequent is null, it is expressed in $B_0 = \mathit{false}$ and called negative clause.

In the clause $A_1, \ldots, A_n \rightarrow B_1; \ldots; B_m$, if the following holds true,

$$\bigcup_{i=1}^{n} \mathit{var}\,(A_i) \supseteq \bigcup_{j=1}^{m} \mathit{var}\,(B_j)$$

this clause is said to satisfy the condition of range-restrictedness (*range-restrictedness*), where $\mathit{var}\,(F)$ is the set of all variables appearing in expression F.

If a clause set $\mathcal{S}$ contains the clauses which do not satisfy the range-restrictedness, it can be converted to a clause set which is logically equivalent to $\mathcal{S}$ where all clauses of the set satisfy the range-restrictedness. This conversion is performed as follows by introducing the reserved predicate called *dom* .

- Replace a clause which does not satisfy range-restrictedness:
  $A_1, \ldots, A_n \rightarrow B_1; \ldots; B$ with

$$\begin{array}{c} A_1, \ldots, A_n, \mathit{dom}\,(X_1), \ldots, \mathit{dom}\,(X_k) \rightarrow \\ B_1; \ldots; B_m \end{array}$$

  , Where the following holds true.

$$\{X_1, \ldots, X_k\} = \bigcup_{j=1}^{m} \mathit{var}\,(B_j) \setminus \bigcup_{i=1}^{n} \mathit{var}\,(A_i)$$

- If a set of constants appearing in a clause set$\mathcal{S}$ (if no constant appears, constant 'a') is $consts(\mathcal{S})$, and a set of function qualifiers is $functs(\mathcal{S})$, add to clause set $\mathcal{S}$ the following definition clause set. *dom*

$$\begin{array}{l} \{true \ \rightarrow dom\,(K) \mid K \in consts(\mathcal{S}\} \quad \cup \\ \{dom\,(X_1), \ldots, dom\,(X_n) \rightarrow \\ \quad \{dom\,(F_n(X_1, \ldots, X_n)) \mid F_n \in functs(\mathcal{S})\} \end{array}$$

Model generation is intended to obtain constructively a model for a given input clause (called MG clause) set $\mathcal{S}$ according to the following procedure. In the following description, the model in the course of generation is expressed by M and is called model candidates. The set of model candidates is expressed by $\mathcal{M}$.

1. Set the initial model candidates $M_0 = \emptyset$ as an element of M.
2. If a model extension rule or model rejection rule described later in this manual is applicable to the element M of $\mathcal{M}$, apply it and update $\mathcal{M}$.
3. Repeat the step in above 2 to the possible extent.
4. If either rule is no longer applicable to all M elements of $\mathcal{M}$, the procedure terminates.

If $\mathcal{M}$ is empty upon termination of the procedure, it can be determined that no model exists in $\mathcal{S}$ (that is, unsatisfiable). Otherwise, all $M \in \mathcal{M}$ elements are the models of $\mathcal{S}$.

Rules for model extension and discard are as follows.

- <u>Model extension rule</u> :

- <u>Model discard rule</u> :

The operation used to obtain σ in which antecedent$A_1, \ldots, A_n$ is satisfied by M is called conjunctive matching.

As an example, the proof tree for model generation related to the following clause set (called problem S) is shown in Figure 1.1.

$$\begin{array}{ll} C_1: & true \ \rightarrow p(a,a); q(b). \\ C_2: & r(X, f(X)) \rightarrow false\ . \\ C_3: & p(X,X), p(X,Y) \rightarrow r(X, f(Y)). \\ C_4: & q(X \rightarrow p(f(X), f(X)). \end{array}$$

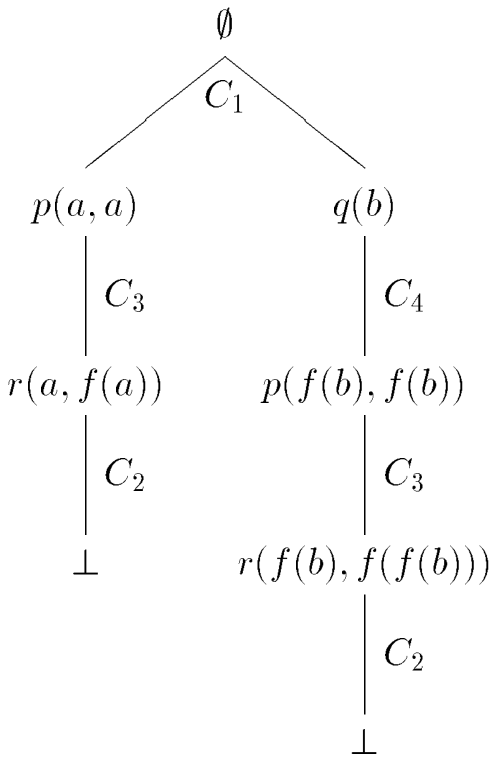

Figure 1.1: Figure of proof for problem S

1.3 Model generation theorem prover MGTP

1.3.1 Basic structure of MGTP

Only the atom as a model candidate element is generated dynamically in the course of reasoning in model generation. This atom normally becomes the ground atom if the target is the MG clause set which satisfies range-restrictedness. In the satisfiability checking (conjunctive matching) of antecedent, unification (matching) is performed for atoms which are elements of model candidates, and antecedent literals of MG clauses.

Head unification between KL1 goal and KL1 clause is intended to implement the synchronization feature for parallel operation in a concise manner, and is unified in one way where only the variables of KL1 clauses are permitted to be bound. As such, for implementation with KL1, it is considered simple and effective that MG clause be expressed in KL1 clause and the unified heads of KL1 clauses be used for conjunctive matching. This makes it possible to entrust the introduction of new variables associated with reproduction of MG clauses to KL1 operation.

So that the basic structure such as shown in Figure 1.2 is appropriate in order to implement model generation theorem prover MGTP in KL1. In other words, the prover main which controls model candidate M is separated from the part of KL1 clause in which MG

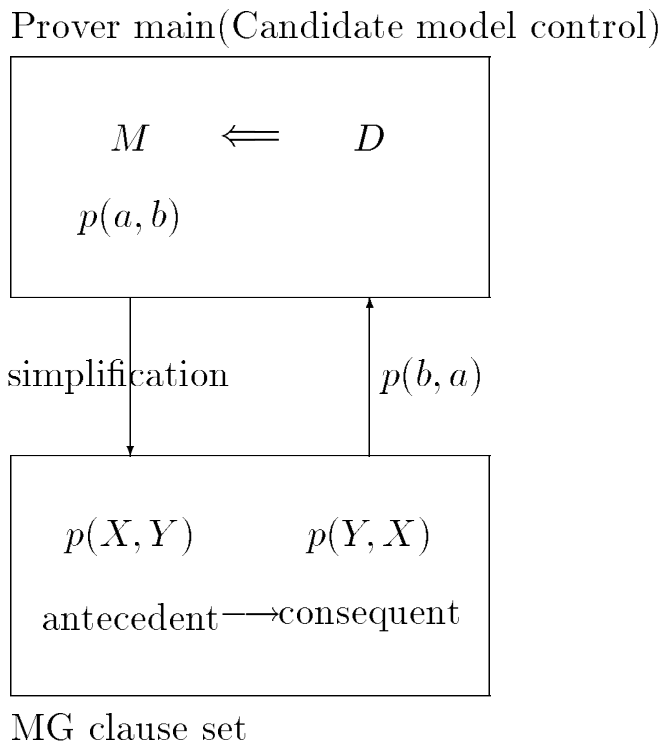

Figure 1.2: Implemented in prover KL1

clause set is expressed, and the MG clause $C_n : p(X,Y) \rightarrow p(Y,X)$ shown in Figure is expressed in the following KL1 clause. [1]

```
c(n,p(X,Y),R):-true|R=p(Y,X).
```

As a result, unification in satisfiability checking can be implemented in a fashion such that, the prover main chooses the atom $p(a,b)$ from M, calls KL1 clause of MG clause set part, and matches it to antecedent literals $p(X,Y)$ which is expressed in the head of the KL1 clause.

When unification of $p(X,Y)$ with $p(a,b)$ is successful, the consequent $p(b,a)$ is returned to the prover main. If $p(b,a)$ is not satisfied with M, it becomes a candidate of the atom to be subjected to extending of M, so that the prover main stores it in the model extending candidate set D. [2]

[1]As a matter of fact, 'otherwise' clause is placed at the end as a precaution for a chance of failure in unification.

[2]Practically, satisfiability checking may be conducted immediately before the atom is chosen as a candidate for model expansion.

```
c(1,true,[],R):-true|R=(p(a,a);q(b)).
c(2,r(X,f(X)),1:[X],R):-true|R=false.
c(3,p(X,X),[],R):-true|R=(1:[X]).
c(3,p(X,Y),1:[X],R):-true|R=r(X,f(Y)).
c(4,q(X),[],R):-true|R=p(f(X),f(X)).
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 1.3: KL1 clause representation of problem S

1.3.2 Representation of MG clause set in KL1

Assume that antecedent consists of two or more literals as usual, for example, such MG clause as

$$C_n : \quad p(X,X), q(X,Z) \rightarrow p(X$$

can be expressed in one KL1 clause as follows.

```
c(n,(p(X,X),q(X,Z)),R):-true|R=p(X,Z).
```

In this case, if you want to make an atom pair $\langle A_i, A_j\rangle$ from among the model candidates for each conjunctive matching process, the entire antecedents can be unified at a time. This process, however, may encounter redundancy. Although a pair of $\langle p(a,b), A_j\rangle$ has already proved unsuccessful in unification with $p(X,X)$ in the first literal, unnecessary conjunctive matching may have been conducted for all instances of A_j.

To remedy such redundancy, you may express a single MG clause in two or more KL1 clauses to enable conjunctive matching to be conducted for individual antecedent literals. The problem is how to keep the identity of the common variable consistent between antecedent literals. To solve this problem, you may express them in two KL1 clauses that follow.

```
c(n,p(X,X),[],R):-true|R=(1:[X]).
c(n,q(X,Z),1:[X],R):-true|R=p(X,Z).
```

In this example, X is the common variable shared among the first literal and the second literal as well as antecedent. And, by obtaining $1 : [X]$ by the third argument of `c/4` corresponding to the second literal, the identity of the common variable X is represented. Based on the way of above representation, a practical example of the foregoing problem MG clause set expressed in KL1 set is shown in Figure1.3.

In $c(N, A, V, R)$, N is the clause number, A is antecedent literals, V is the list of antecedent literals number and bounded shared variables, and R is the value to be returned to the prover main when unification is successful.

1.3.3 Implementation of the main routine

The main routine can be implemented in the form of KL1 program shown in Figure 1.4 and 1.5.

`mgtp/5` of the main loop receives model extending candidate set D, model candidate M, negative clause number list Cn, and non-negative clause number list Cg, and returns the result to Res.

If `D` is null, the main loop terminates as a result that MG set can be satisfied or satisfiable with `M` (model is `M`). If D is not null,
choose one of model extending candidate Δ by `pickup/3` , and in response to this, check the satisfiability with M by `subsTest/3`. If Δ is satisfied with M, main loop restarts for the remaining $D1$ of the model extending candidate set. If not, and Δ is 'disjunctive' clause, make selection of instances by `caseSplit/6`. If Δ is an atom, M is extended to include it. Where a model rejection rule is applicable to the extended model candidate $[\Delta|M]$, the main loop terminates as a result of MG set being unsatisfiable with M (M cannot be a model). Where no model rejection is applicable, apply conjunctive matching of antecedent to the extended model candidate $[\Delta|M]$ and make it as a new model extending candidate set New. It is added to the remaining $D1$ of the model extending candidate set by `addNew/3` to be New and the main loop is restarted.

Conjunctive matching `cjm/5` receives the list of clause number N and model candidate M, and performs conjunctive matching `cjm1/6` for all clauses in the list, and return the result in the differential list $Rh - Rt$. In the conjunctive matching procedure `cjm1/6` for one MG clause, conjunctive matching is made for all atoms in M against each antecedent literals. At the same time, KL1 representation `c/4` of MG set is called. For the said literal, if the result of matching is $fail$ on `c/4` with atom A in M, the result of matching on this literal-atom pair is null.

When the shared variable information $(\_ : \_)$ is returned, proceed with conjunctive matching on literal. In other cases, conjunctive matching on the entire antecedents is successful and consequent is returned, so that the result is filled in D-list of `Rh-Rm` and returned. This result is again filled in the differential list of `Rh-Rm` and returned.

1.4 Elimination of redundancy in conjunctive matching

1.4.1 Redundancy in conjunctive matching

For conjunctive matching of the basic MGTP, chances are that duplicate calculus is performed for the same combination of atoms between antecedent and model candidate M.

```
mgtp(D,M,Cn,Cg,Res):-true|
 empty(D,E),
 (E=yes->
 Res=sat;
 otherwise;true->
  pickup(D,Delta,D1),
  subsTest(Delta,M,S),
  (S=yes->mgtp(D1,M,Cn,Cg,Res);
  S=no,Delta=(_;_)->
   caseSplit(Delta,D1,M,Cn,Cg,Res);
  otherwise;true->
   cjm(Cn,Cn1,[Delta|M],F,[]),
   (F=[false|_]->
    Res=unsat;
   otherwise;true->
    cjm(Cg,Cg1,[Delta|M],New,[]),
    addNew(New,D1,NewD),
    mgtp(NewD,[Delta|M],Cn1,Cg1,Res)))).
caseSplit((A;B),D,M,Cn,Cg,Res):-true|
 caseSplit(A,D,M,Cn,Cg,R1),
 (R1=sat->Res=sat;
 otherwise;true->
  caseSplit(B,D,M,Cn,Cg,Res)).
otherwise.
caseSplit(A,D,M,Cn,Cg,Res):-true|
 addNew([A],D,NewD),  mgtp(NewD,M,Cn,Cg,Res).

subsTest((A;B),M,S):-true|
 subsTest(A,M,S1),
 (S1=yes->S=yes; otherwise;true->subsTest(B,M,S)).
otherwise.
subsTest(A,[A|_],S):-true|S=yes.
otherwise.
subsTest(A,[_|M],S):-true|subsTest(A,M,S).
subsTest(_,[],S):-true|S=no.
```

Figure 1.4: Basic MGTP prover

The following section deals with RAM method [FH91] and MERC method devised to eliminate redundant calculus in conjunctive matching.

1.4.2 RAMS method

RAMS method is intended to avoid duplicate calculus in conjunctive matching by providing the feature to store the record of matching for antecedents. This method is illustrated in Figure 1.6.

In general, for each literal of antecedent $A_1, \ldots, A_n$, one instance stack S_i is allocated to each literal A_i of antecedent $A_1, \ldots, A_n$ of MG clause.

In each S_i, the result of matching with the model candidate M of $A_1, \ldots, A_i$ is stored (the instance if successful). (Binding information is stored when successful.) S_1 contains the result of matching on A_1, and for the matching of $A_i (i > 1)$, matching between A_i itself

```
cjm([N|Cs],Cs1,M,Rh,Rt):-true|
 Cs1=[N|Cs2],
 cjm1(N,M,[],M,Rh,Rm), cjm(Cs,Cs2,M,Rm,Rt).
cjm([],Cs1,_,Rh,Rt):-true|Cs1=[],Rh=Rt.

cjm1(N,[A|M],V,Mh,Rh,Rt):-true|
 problem:c(N,A,V,R),
 (R=fail->Rh=Rm;
 R=(_:_)->cjm1(N,Mh,R,Mh,Rh,Rm);
 otherwise;true->Rh=[R|Rm]), cjm1(N,M,V,Mh,Rm,Rt).
cjm1(_,[],_,_,Rh,Rt):-true|Rh=Rt.
```

Figure 1.5: conjunctive matching of basic MGTP prover

and M is performed according to the result of each matching for $A_1, \ldots, A_{i-1}$ stored in S_{i-1}. This matching operation against $A_1, \ldots, A_{i-1}$ is hereafter represented as $S_{i-1} \circ M$.

Each instance stack S_i is divided into two parts; S_i^{k-1} stacked until antecedents appear and δS_i^k stacked in the current stage. Parameter k denotes the stage number of the main loop. The atom which extends the model candidate M^{k-1} until the previous stage is called model extending atom and expressed in Δ^k.

The task T_1^k in A_1 is the matching between A_1 and Δ^k, and the result is stacked in S_1. The task T_i^k in each literal $A_i (2 \leq i \leq n-1)$ updates the stack after conjunctive matching as follows.

$$\begin{aligned} \delta S_i^k &:= \delta S_{i-1}^k \circ (\Delta^k \cup M^{k-1}) \cup S_{i-1}^{k-1} \circ \Delta^k \\ S_i^k &:= S_i^{k-1} \cup \delta S_i^k \end{aligned}$$

Because T_n is the end of conjunctive matching, after the condition corresponding to the matching result is obtained, the result of matching itself on $A_1, \ldots A_n$ needs not be stored in stack.

Therefore, practically, stack S_n for the last literal A_n needs not be allocated.

By executing the sequence of tasks $T_1 \ldots, T_n$ in that order, conjunctive matching on Conjunctive matching of antecedent $A_1 \ldots, A_n$ can be implemented without redundancy. What's more, conjunctive matching on antecedent $A_1 \ldots, A_n$ can be made possible without redundancy. Figure 1.6 illustrates the circumstance where atom 1 is stacked on M in the second stage, and Δ^2 is atom 2. At the beginning, matching between A_1 and Δ^2 is successfully done, and the result (denoted as 2 in the figure for convenience) is stacked in S_1 as δS_1^2. Matching of A_2 and Δ^2 is successful under this δS_1^2, and the result (denoted as 22 in the in Figure) is stacked as one of δS_2^2.

RETE, another method intended to render the same effect as the above, is limited to Horn clauses. Because RAM method uses stacks, it works well with Non-Horn clauses.

Figure 1.8 shows the modification of conjunctive matching of the basic MGTP prover based on the RAM method.

1.4.3 MERC method

With MERC method, in general, conjunctive matching for M and Δ is performed in combination shown in Figure 1.9 against the antecedent $A_1, \ldots, A_n$ of MG clauses. The pattern shown in the first line in this figure shows that Δ is matched against A_1, an the atoms in M are matched against $A_2, \ldots, A_n$ respectively. Note that only the combination which allows at least one of $A_1, \ldots, A_n$ to be matching to Δ is chosen, and any combination which allows all of $A_1, \ldots, A_n$ to be matching to the atoms contained in M is excluded. As long as any combination permitting matching is excluded, redundancy between stages can be avoided.

Hereinafter, the literal matching to Δ is called entry literal, and the literal permitting the atoms in M collatable is called subsequent literal. In conjunctive matching process, at the beginning, Δ is matching against the entry literal, and if the result is successful, the process proceeds to collate the subsequent literals against the atom in M. The problem is that KL1 representation indicated in the previous section is fixed to only one literal sequence, and no optional literal can be an entry literal. To remedy this problem, MERC method should have one MG clause expressed in individual KL1 clauses to accommodate for different way of choosing entry literals.

For one pattern of entry literal which belongs to (I) in Figure 1.9, rearrange the order of $A_1, \ldots, A_n$, and consider one MG clause preceded by an entry literal and provide the KL1 clause corresponding to that MG clause. On the other hand, chances are that KL1 clause can be omitted for a pattern in which multiple antecedent literals become entry literals. For example, two A_j, A_k literals are matching against Δ at the same time only in case that they can be simplified (denoted as $A_j = A_k$), that is, factoring is possible. Among the patterns which belong to (II) in Figure, KL1 clause needs not be made available for a pattern that includes entry literal pair.

For multiple entry literals $A_{i1} \ldots A_{il}$ that can be simplified, compute the literal A_r ($A_{i1} = A_{i2} = \ldots = A_{il}$) resulting from factoring and provide the KL1 clause with this result taken as a new entry literal.

As an example, Figure 1.10 shows the representation of KL1 for problem S. When using MERC method, part of the conjunctive matching of the basic MGTP prover is modified as shown in Figure 1.11.

1.5 Comparison between RAMS and MERC methods

With the RAMS method, the result of matching of model candidate M against antecedent literals $A_1, \ldots, A_{i-1}$ is stored, and based on this memory, matching is performed between A_i and the latest atom Δ, while the MERC method has redundancy of duplicate calculus on matching for $A_1, \ldots, A_{i-1}$. This is referred as M-M redundancy.

On the other hand, the MERC method allows the matching between the entry literal A_i and Δ to take the precedence over the matching of the atom in M to the subsequent literal $A_1, \ldots, A_{i-1}$, while by the RAMS method, matching sequence is fixed according to the arrangement of antecedent literals of a given MG clause, so that matching to Δ is never be given precedence over the other. As a result, the MERC method dispenses with the matching of subsequent literals if matching of entry literals with Δ fails, whereas the RAMS method is likely to repeat redundant matching between entry literals and Δ for all possible combinations which promise a success in matching with subsequent literals. This is called Δ failure redundancy.

In terms of memory consumption, while the RAMS method uses additional memory to be allocated to the instance stacks in execution of the prover, the MERC method requires as many reproduced clauses as the number of entry literals.

As such, the impact of inter-stage redundancy, M-M redundancy, and Δ failure redundancy as well as memory consumption property on the overall performance of the prover is heavily dependent upon the problems, and the distinction between the RAMS and MERC for better performance is not defined clearly. Moreover, since the RAMS and MERC differ in their sequence of model extending candidate atoms to be combined in conjunctive matching, and the sequence of Δ to be chosen is usually different from each other, the result of proving process will differ accordingly unless model extending candidate atoms are rearranged.

1.6 Conclusion

The model generation theorem prover MGTP was implemented in a parallel logic language KL1. Using the model generation scheme eliminates the need for unification with occur-check for the clause set which satisfies the range-restrictedness, and renders the matching process fully useful for intended purpose, thus making it possible to implement a highly efficient theorem prover with the feature of KL1 fully incorporated into the system. The essential points for using KL1 processing system include the following.

- Logical variables in the input clauses are directly expressed in KL1 variables.
- Unification of input clauses is enabled in the form of head unification in KL1.
- New variables required for reproduction of input clauses are automatically obtained from KL1 clause call mechanism.

What's more, two methods - RAMS and MERC - were developed for efficient conjunctive matching in model generation. These two methods have shown no major difference in their capability of eliminating inter-stage redundancy in conjunctive matching, and can significantly improve the running efficiency of the MGTP.

For topics of parallel MGTP, refer to the Ground version MGTP Manual, the Non-Ground version MGTP Manual, and appropriate part of the CMGTP Manual, as well as other reference literature ([FHKF92] など).

Ground version MGTP and CMGTP have OR parallelism, and non-ground version MGTP has AND parallelism, respectively implemented in KL1 version, and they have successfully achieved a close-to-linear yield by processor number for several benchmark problems.

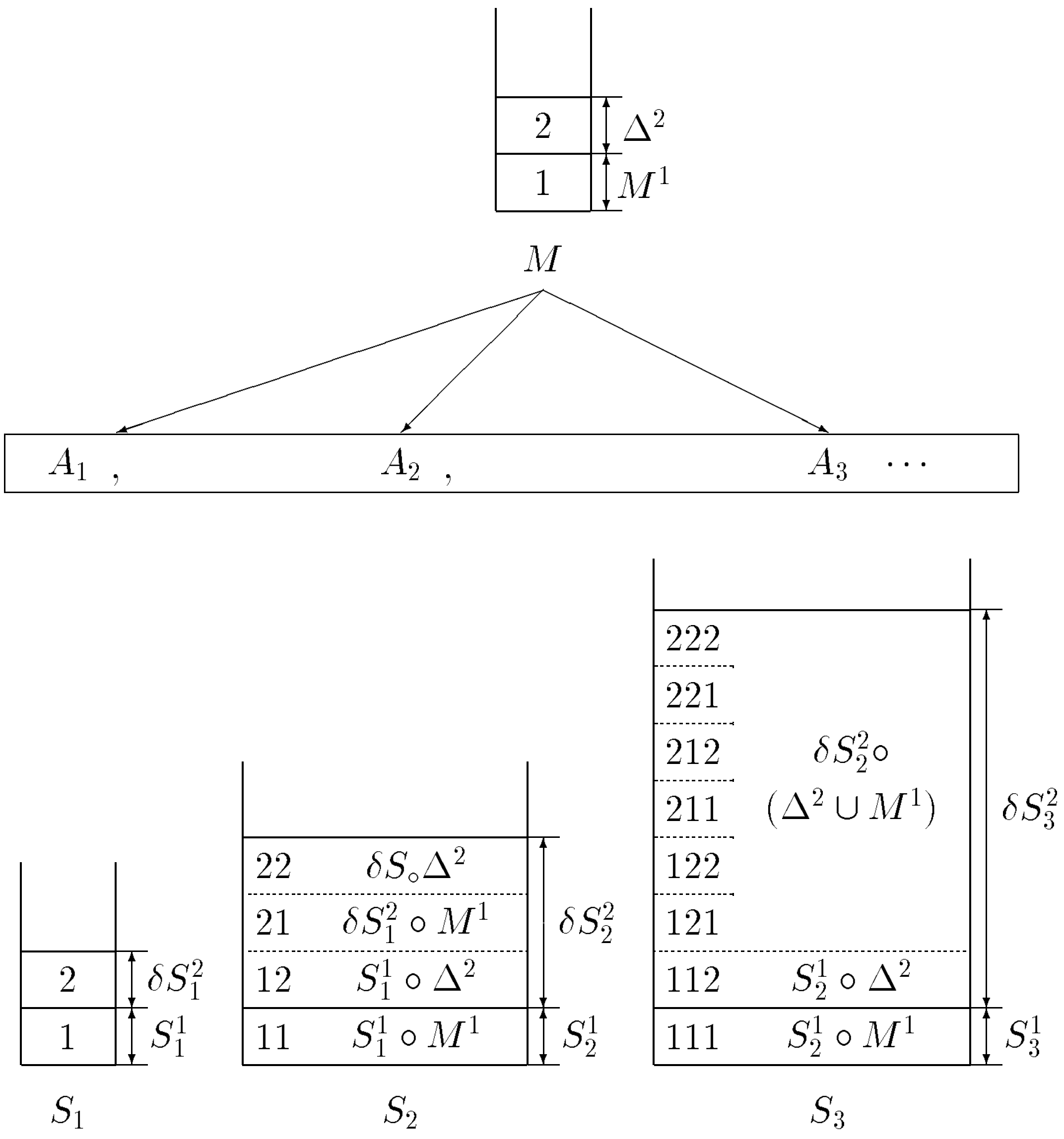

Figure 1.6: Instance stacker

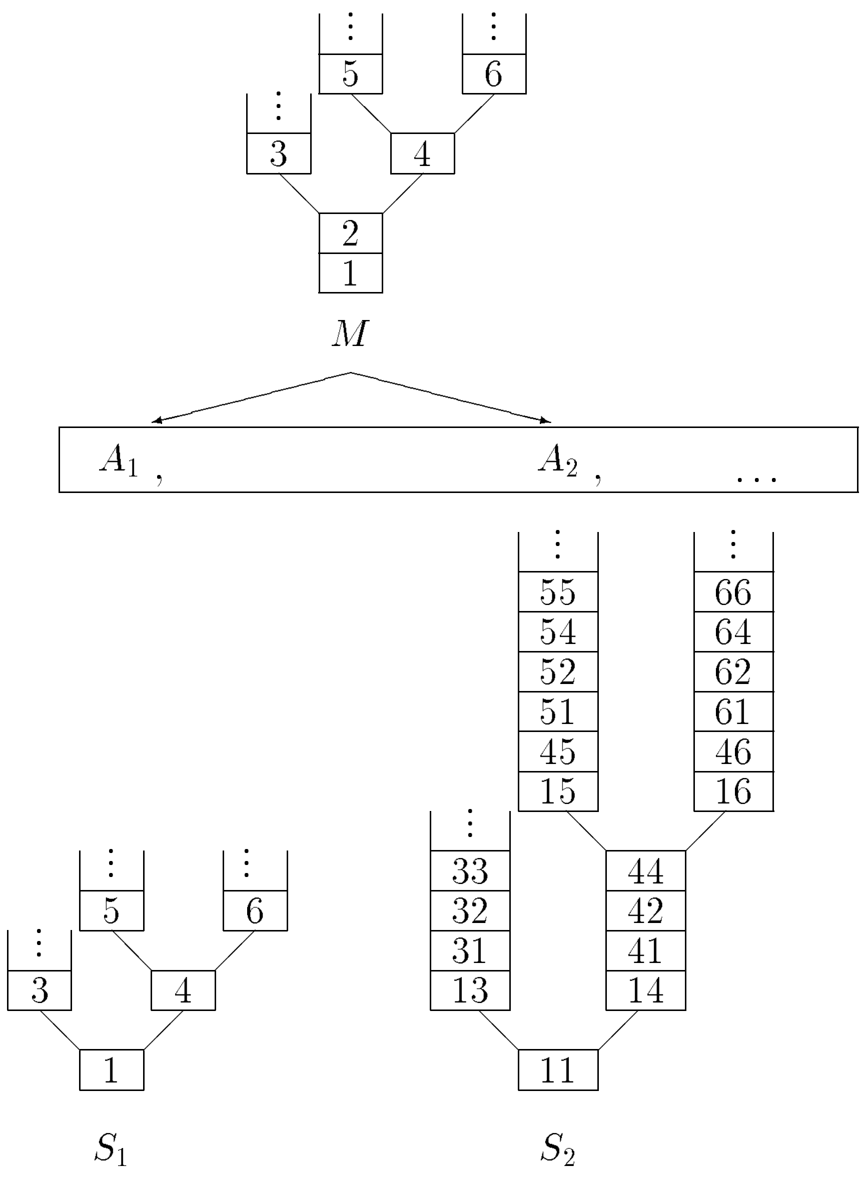

Figure 1.7: Branchstack

```
cjm([N:L|Cs],Cs1,Delta,M,Rh,Rt):-true|
 Cs1=[N:L1|Cs2],
 cjm1(N,L,L1,Delta,M,[[]],[],Rh,R1),
 cjm(Cs,Cs2,Delta,M,R1,Rt).
cjm([],Cs1,_,_,Rh,Rt):-true|Cs1=[],Rh=Rt.

cjm1(N,[Sn|Ls],Ls1,Delta,M,Sp,Dp,Rh,Rt):-true|
 Ls1=[NewSn|Ls2],
 cjm2(N,[Delta|M],Dp,NewSn,S1,Dn,D1),
 cjm2(N,[Delta],Sp,S1,Sn,D1,[]),
 cjm1(N,Ls,Ls2,Delta,M,Sn,Dn,Rh,Rt).
cjm1(N,[],Ls1,Delta,M,Sp,Dp,Rh,Rt):-true|
 Ls1=[],
 cjm2(N,[Delta|M],Dp,Rh,R1,_,_),
 cjm2(N,[Delta],Sp,R1,Rt,_,_).

cjm2(N,As,[V|Vs],Sh,St,Dh,Dt):-true|
 cjm3(N,As,V,Sh,S1,Dh,D1),
 cjm2(N,As,Vs,S1,St,D1,Dt).
cjm2(_,_,[],Sh,St,Dh,Dt):-true|
 Sh=St,Dh=Dt.

cjm3(N,[A|As],V,Sh,St,Dh,Dt):-true|
 problem:c(N,A,V,R),
 (R=fail->Sh=S1,Dh=D1;
  R=(_:_)->Sh=[R|S1],Dh=[R|D1];
  otherwise;true->Sh=[R|S1]),
 cjm3(N,As,V,S1,St,D1,Dt).
cjm3(_,[],_,Sh,St,Dh,Dt):-true|
 Sh=St,Dh=Dt.
```

Figure 1.8: RAM version of 'and' clause collator

	A_1	A_2	A_3	$\ldots$	A_n
(I)	Δ	M	M	$\ldots$	M
	M	Δ	M	$\ldots$	M
	M	M	Δ	$\ldots$	M
			$\ldots$		
	M	M	M	$\ldots$	Δ
(II)	Δ	Δ	M	$\ldots$	M
	M	Δ	Δ	$\ldots$	M
			$\ldots$		
	Δ	Δ	Δ	$\ldots$	Δ

Figure 1.9: conjunctive matching by MERC method

```
c(c1,true,      [],R):-true|R=(p(a,a);q(b)).
c(c2,r(X,f(X)),[],R):-true|R=false.
c(c3_1,p(X,X),    [],R):-true|R=(1:[X]).
c(c3_1,p(X,Y),1:[X],R):-true|R=r(X,f(Y)).
c(c3_2,p(X,Y),    [],R):-true|R=(1:[X,Y]).
c(c3_2,p(X,X),1:[X,Y],R):-true|R=r(X,f(Y)).
c(c3_1_2,p(X,X), [],R):-true|R=r(X,f(X)).
c(c4,q(X),      [],R):-true|R=p(f(X),f(X)).
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 1.10: MERC version KL1 representation of problem S

```
cjm([N|Cs],Cs1,[Delta|M],Rh,Rt):-true|
 Cs1=[N|Cs2],
 problem:c(N,Delta,[],R),
 (R=fail->Rh=Rm;
  R=(_:_)->cjm1(N,M,R,M,Rh,Rm);
  otherwise;true->Rh=[R|Rm]),
 cjm(Cs,Cs2,[Delta|M],Rm,Rt).
cjm([],Cs1,_,Rh,Rt):-true|Cs1=[],Rh=Rt.
```

Figure 1.11: MERC version of 'and' clause collator

Bibliography

[HF95] Ryuzo Hasegawa, Hiroshi Fujita, MGTP : Model Generation Theorem Prover in Parallel Logic Language KL1: 並列論理型言語 KL1 によるモデル生成型定理証明系 in Japanese, ICOT TR 885, 1994.

[CFS94] Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1, *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming* (1994).

[FH91] Fujita, H. and Hasegawa, R.: A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. *Proc. 8th ICLP,* pp. 535–548 (1991).

[MB88] Manthey, R. and Bry, F.:SATCHMO: A Theorem Prover Implemented in Prolog, *Proc. 9th CADE,* pp. 415–434 (1988).

[Sti88] Stickel, M. E.: A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, *Journal of Automated Reasoning,* 4, pp. 353–380 (1988).

[SSY93] Suttner, C., Sutcliffe, G. and Yemenis, T.: The TPTP Problem Library, *Proc. 12th CADE,* pp. 252–266 (1993).

[UC90] Ueda, K. and Chikayama, T.: Design of the Kernel language for the Parallel Inference Machine. *The Computer Journal,* Vol. 33, No. 6, pp. 494–500 (1990).

[FHKF92] M. Fujita, R. Hasegawa, M.Koshimura and H. Fujita, *Model Generation Theorem Provers on a Parallel Inference Machine*, In Proc. of FGCS92, 1992.

Chapter 2

MGTP Input Language Specification

This chapter relates to the standard syntax for the input language of MGTP followed by examples.

2.1 MGTP input clause syntax

As described in Chapter 1 for function, the input clause of MGTP comes in three types; positive clause, negative clause, and neutral clause, but all clauses basically consist of (implication) `-->`. The left side of the implication is called Antecedent, and the right side is called Consequent.

The antecedent of a positive clause is expressed as `true`, and the consequent of a negative clause is expressed as `false`. For a clause which does not satisfy the condition of the range restrictedness, it is possible to transform it to the clause to satisfy the condition by using a predicate such as `dom`.

The antecedent and consequent are allowed to directly call the target language thus making it possible to perform compare operation between each integer.

The following example shows the syntax of MGTP input clauses expressed in BNF.

MGTP-Clause ::=	Positive-Clause | Negative-Clause | Neutral-Clause
Positive-Clause ::=	' `true`' ' `-->`' Consequent '.'
Negative-Clause ::=	Antecedent ' `-->`' ' `false`' '.'
Neutral-Clause ::=	Antecedent ' `-->`' Consequent '.'
Antecedent ::=	Antecedent-Atom ',' Antecedent-Atom *
Antecedent-Atom ::=	MGTP-Atom | ' `{{`' [Guards] '`}}`'
Consequent ::=	['`{{`' [Procedures] '`}}`' ' `->`'] Consequent-Group
Consequent-Group ::=	Consequent-Unit ',' Consequent-Unit *
Consequent-Unit ::=	Atom-Group | Disjunctive-Atom-Group
Disjunctive-Atom-Group ::=	Atom-Group ' ;' Atom-Group *
Atom-Group ::=	MGTP-Atom ',' MGTP-Atom *
MGTP-Atom ::=	Term
Guards ::=	Call ',' Call *
Procedures ::=	Call ',' Call *

```
%%  Problems/MSC/MSC006-1.mg  Ground-MGTP/pl input-clauses
%%  infinite_domain(no).
%%  max_number_of_doms_in_a_clause(2).
%%  total_number_of_doms(2).
%%  number_of_NonHorn_clauses(1).
%%  max_number_of_consequents_in_a_clause(2).
%%  number_of_negated_literals(0).

true-->dom(a),dom(b),dom(c),dom(d).
dom(A),dom(B)-->p(A,B);q(A,B).
p(A,B),p(B,C)-->p(A,C).
q(A,B),q(B,C)-->q(A,C).
q(A,B)-->q(B,A).
p(a,b)-->false.
q(c,d)-->false.
%%  Problems/MSC/MSC006-1.mg  EOF
```

Figure 2.1: Example of MGTP input clauses

In the above example, 'Term' indicates the term of Prolog or KL1. In other words, it corresponds to predicate. Representation of variables also complies with the same for Prolog and KL1. 'Call' signifies the call statement of the relevant program, and such expression as `{{X>Y}}` or `{{X1:=X+1,X1>Y}}` in Prolog is possible.

For details of different syntax, see the manual for each version.

2.2 Example problem

Figure2.1 shows an example of MGTP input clauses. This problem is the result of MSC006-1 (Figure 2.2) in TPTP problem library[1] transformed into MGTP input clauses.

Figure 2.3 shows an example of eight queen problems expressed in MGTP. The predicate `p(M,N)` indicates that the queen is placed in row `M` and column `M`. At the beginning, the candidate for the range of each row is written in disjunction form as a positive clause. In this example, the program call enclosed in `{{ }}` is used as the basis to determine that the queen is placed in the same row or column, or placed in the diagonal line, thus rejecting other model candidates.

```
%--------------------------------------------------------------------------
% File     : MSC006=NonObv-1 : Released v0.0.0, Updated v0.11.5.
% Domain   : Miscellaneous
% Problem  : A ``non-obvious'' problem
% English  : Suppose there are two relations, P and Q. P is
%            transitive, and Q is both transitive and symmetric.
%            Suppose further the ``squareness'' of P and Q: any two
%            things are related either in the P manner or the Q
%            manner. Prove that either P is total or Q is total.
% Refs     : Pelletier F.J., and Rudnicki P. (1986), Non-Obviousness,
%            In Wos L. (Ed.), Association for Automated Reasoning
%            Newsletter (6), Association for Automated Reasoning,
%            Argonne, Il, 4-5.% Source   : [Pelletier & Rudnicki, 1986]
% Names    : nonob.lop [SETHEO]
% Syntax   : Number of clauses          :    6 (   1 non-Horn)(   2 units)
%            Number of literals         :   12 (   0 equality)
%            Number of predicate symbols :    2 (   0 propositions)
%            Number of function symbols  :    4 (   4 constants)
%            Number of variables        :   10 (   0 singletons)
%            Maximal clause size        :    3
%            Maximal term depth         :    1
%--------------------------------------------------------------------------
input_clause(p_transitivity,hypothesis,
    [--p(X,Y),
     --p(Y,Z),
     ++p(X,Z)]).
input_clause(q_transitivity,hypothesis,
    [--q(X,Y),
     --q(Y,Z),
     ++q(X,Z)]).
input_clause(q_symmetry,hypothesis,
    [--q(X,Y),
     ++q(Y,X)]).
input_clause(all_related,hypothesis,
    [++p(X,Y),
     ++q(X,Y)]).
input_clause(p_is_not_total,hypothesis,
    [--p(a,b)]).
input_clause(prove_q_is_total,theorem,
    [--q(c,d)]).
%----------------------------------------------------------------------
```

Figure 2.2: MSC006-1(TPTP problem library)

```
true --> p(1,1);p(1,2);p(1,3);p(1,4);p(1,5);p(1,6);p(1,7);p(1,8).
true --> p(2,1);p(2,2);p(2,3);p(2,4);p(2,5);p(2,6);p(2,7);p(2,8).
true --> p(3,1);p(3,2);p(3,3);p(3,4);p(3,5);p(3,6);p(3,7);p(3,8).
true --> p(4,1);p(4,2);p(4,3);p(4,4);p(4,5);p(4,6);p(4,7);p(4,8).
true --> p(5,1);p(5,2);p(5,3);p(5,4);p(5,5);p(5,6);p(5,7);p(5,8).
true --> p(6,1);p(6,2);p(6,3);p(6,4);p(6,5);p(6,6);p(6,7);p(6,8).
true --> p(7,1);p(7,2);p(7,3);p(7,4);p(7,5);p(7,6);p(7,7);p(7,8).
true --> p(8,1);p(8,2);p(8,3);p(8,4);p(8,5);p(8,6);p(8,8);p(8,8).
p(N,M),p(N1,M),{{N \= N1}} --> false.
p(N,M),p(N,M1),{{M \= M1}} --> false.
p(N,M),p(N1,M1),{{N \= N1,M \= M1,A:=N-N1,B:=M-M1,A=B}} --> false.
p(N,M),p(N1,M1),{{N \= N1,M \= M1,A:=N-N1,B:=M1-M,A=B}} --> false.
```

Figure 2.3: Example of 8 queens expressed in MGTP

Bibliography

[1] Suttner, C., Sutcliffe, G. and Yemenis, T.: The TPTP Problem Library, *Proc. 12th CADE,* pp. 252-266 (1993).

Chapter 3

Translator

æ

Chapter 4

MGTP/G Manual

This chapter outlines the functions of MGTP/G and describes the related operation.

MGTP/G represents a model generation theorem prover intended for clause set which satisfies the range restrictedness. A clause is called range-restricted if all variables in the consequent appear in the antecedent.

MGTP/G is available in Prolog version operating on SICStus Prolog, KLIC version operating on KLIC, and KL1 version operating on PIM.

4.1 Functions of MGTP/G

MGTP/G does not need full unification during conjunctine matching because of range-restrictedness. As a result, an efficient implementation is made possible by utilizing the KL1's head unification; a variable in the input clause can be directly expressed in KL1 variable, and the unification of input clauses can be implemented in the form of head unification in KL1 clauses. Prolog's built-in unfication is also used for efficient implementation.

In any of those versions, the problem provided by MGTP clauses is transformed into Prolog clause set or KL1 clause set in the pre-processing program. The result is linked with the MGTP/G main program and is subjected to the prover.

4.1.1 Prolog version

Prolog version is implemented by the RAMS method which eliminates redundant computation in conjunction matching which provides the basis for the model generation theorem prover, and provides three strategies.

Strategy I Extension applies to all model extension candidates. Model rejection rule applies to the model candidates obtained.

Strategy II An element is selected from among the model extension candidates and is used for extension. The remaining model extension candidates are placed in the stack [1] and

[1]each model candidate is assigned one stack. This stack is thrown away when the corresponding model candidate is rejected.

used for next model extension. The model rejection rule applies to the extended model candidates.

Strategy III Extensions are applies using all model extension candidates which contain no disjunction. The model extension candidates containing disjunction are placed in the stack. The model rejection rule applies to the model candidate (no more than one). If only the model extension candidates which contain disjunction are present, one of them is selected and used for extension. The remaining model extension candidates are placed in the stack. The model rejection rule applies to the (two or more) model candidates obtained.

In any version, Prolog predicates can be called from MGTP clauses.

4.1.2 KLIC version

KLIC version is available in three types; NAIVE, which is the base of the model generation theorem prover, RAMs, which incorporates the RAMS method and MERC which eliminates redundant computation in conjunction matching. The basic type of those methods is NAIVE.

The RAMS method is designed to avoid redundant calculus in conjunction matching by having a stack to store the record of antecedent matching. The problem is that it tends to increased memory required to hold the record of matching. The MERC methods are the remedy for the disadvantage of the RAMS method. The MERC is designed to replace the conjunctive matching with duplicate combination of patterns for matching so that conjunction matching applies in accordance with these patterns.

Functions other than those mentioned above include the following.

- Proof unchanging mechanism is available for fair comparison of each type.
- KL1 predicates can be called from the problem clause.
- Proof trace and proof statistical information can be displayed by choosing appropriate options.

4.1.3 KL1 version

The function of this version is exactly the same as Prolog version (See 4.1.1) with the exception that KL1 call can be used instead of Prolog call.

KL1 guard predicate for antecedent and KL1 body predicate for consequent can be called respectively.

4.2 MGTP/G operation guide

4.2.1 Prolog version

Prolog version is implemented on SICStus-Prolog. Reasoning starts after translating MG clauses into Prolog clauses and compliing obtained Prolog clauses. Clause translation, compilation and execution are entirely performed in the Prolog environment.

Figure ?? shows the outline of above operating procedure. (1) Translate MG clauses into Prolog clauses by using the translator`mgtp2pl`. (2) Compile Prolog clauses by the Prolog compiler. (3) Start reasoning with already compiled engine.

4.2.1.1 Installation

The system consists of MGTP/G engine and the translator. It needs no installation procedure such as compilation and registration, and only needs to put source files of the system in a directory.

Figure 4.1 shows the files required for installation. The file is organized with the program which translates MGTP/G engine `top.pl` and MG clauses into Prolog clauses, and a group of its utilities`mgtp2uty.pl`.

Table 4.1: File organization

File name	File contents	Used language
`topI.pl`	MGTP/G engine(Strategy I)	Prolog
`topII.pl`	MGTP/G engine(Strategy II)	Prolog
`topIII.pl`	MGTP/G engine(Strategy III)	Prolog
`mgtp2pl.pl`	MG clause–Prolog clause translator	Prolog
`mgtp2uty.pl`	`mgtp2pl.pl` utility program	Prolog

4.2.1.2 Operation

Because Prolog version is implemented on Prolog systems, all operations are performed on Prolog systems. This section describes the method for clause translation, compiling Prolog clauses by the compiler, and compiling the engine, then shows the process of solving example problems in practical manner. The examples and notation shown below apply only to SICStus-Prolog processing systems and may differ from those of other Prolog processing systems.

4.2.1.2.1 Setup

`top.pl` `mgtp2pl.pl` are Prolog programs. It is necessary to compile them first thing.[2]

```
compile(['mgtp2pl.pl','top.pl']).
```

`mgtp2uty.pl` is called from `mgtp2pl.pl` and automatically compiled if it resides in the same directory.

4.2.1.2.2 MG clause– Prolog clause translation

The translator `mgtp2pl.pl`, translates MG clauses into Prolog clauses which are compiled by Prolog compiler and liked with MGTP/G engine.

```
mgtp2pl(<MG file >, <PROLOG file >).
```

[2]Registration by `consult/1` is also acceptable. Refer to the attached manual for using `compile/1,consult/1` command which depends on Prolog systems.

Specify the file name of the problem described in MG clause to `<MG file>`, and specify the file name for output of Prolog clause to `<PROLOG file>`. Extension (`.mgtp, .pl`) for `<MG file>, <PROLOG file>` may be omitted. In addition, it is possible to omit `<PROLOG file>` itself. If `<PROLOG file>` is omitted, `<MG file >.pl` is used as `<PROLOG file >`. If `user` is specified for `<PROLOG file >`, the translated Prolog clauses are displayed on the currently running terminal.

After MG cause - Prolog clause translation by `mgtp2pl`, run the Prolog compiler mentioned later in this section. `mgtp2pl.pl` provides a function to execute these two processes with a single instruction.

```
mgtpcomp(<MG file >, <PROLOG file >).
```

`mgtpcomp` performs clause translation in its internal processing called by `mgtp2pl`, and runs the Prolog compiler to compile `<PROLOG file >`. As similar in case to `mgtp2pl`, `<MG file >, <PROLOG file >` of `mgtpcomp` is allowed to omit extension and `<PROLOG file >`.

4.2.1.2.3 Compile

Run the Prolog compiler to compile and register Prolog clauses.

```
compile(<PROLOG file >).
```

Specify the file name of the Prolog clauses obtained by the translator ?? to `<PROLOG file >`. `<PROLOG file >` is linked with MGTP/G engine `top.pl`.

4.2.1.2.4 Execution

When the environment is ready for execution after the process of previous clause translation and compilation, reasoning starts by executing `do(S).` .

```
do(S).
```

On completion of reasoning, either refutation fail (sat) or refutation success (unsat) is reported to variable `S` as the result of execution. The contents of the report is shown below. If the refutation fails, detected model group `model([`*AtomM,...,AtomN*`])` is output to variable `S`, and if succeeds, the rejected model candidates `rejected([`*AtomM,...,AtomN*`])` are output.

```
  sat   :  S = [model([Atom1,...,AtomL]),...,model([AtomM,...,AtomN])].
unsat   :  S = [rejected([Atom1,...,AtomL]),...,rejected([AtomM,...,AtomN])].
```

4.2.1.2.5 Example of operation

The following example shows a series of operations from setup to clause translation, compilation, and execution. This is an example of solution to the sample problem `example.mgtp` shown in Figure 4.1 . (In the example below, the directory locations of home and source files are denoted as `$HOME`, `$HOME/mgtpg` for convenient.)

```
p(A,B),p(B,C)-->p(A,C).
q(A,B),q(B,C)-->q(A,C).
q(A,B)-->q(B,A).
dom(A),dom(B)-->p(A,B);q(A,B).
p(a,b)-->false.
q(c,d)-->false.
true-->dom(a),dom(b),dom(c),dom(d).
```

Figure 4.1: Exercise (example.mgtp)

1. **Setup**

```
prompt[1]% cd $HOME/mgtpg
prompt[2]% ls -F
 example.mgtp      mgtp2pl.pl        mgtp2uty.pl       top.pl
prompt[3]% sicstus
SICStus 2.1 #8:  Tue May 11 21:04:52 JST 1993
{consulting $HOME/.sicstusrc...  }
{$HOME/.sicstusrc consulted, 0 msec 80 bytes}
| ?- compile(['mgtp2pl.pl','top.pl']).
{compiling $HOME/mgtpg/mgtp2pl.pl...}
{compiling $HOME/mgtpg/mgtp2uty.pl...}
{$HOME/mgtpg/mgtp2uty.pl compiled, 1750 msec 51216 bytes}
{$HOME/mgtpg/mgtp2pl.pl compiled, 3540 msec 99312 bytes}
{compiling $HOME/mgtpg/mgtp2x/top.pl...}
{Warning:  [FLIS,LIS] - singleton variables in do/7 in lines 17-18}
{Warning:  [False] - singleton variables in do1Decide/9 in lines 27-27}
{Warning:  [DM,ID,M] - singleton variables in satisfyClause/9 in lines
41-42}
{$HOME/mgtpg/top.pl compiled, 630 msec 11216 bytes}
yes
| ?-
```

2. **MG clause–KL1 clause translation**

```
| ?- mgtp2pl('example.mgtp','example.pl').
[ program saved in "example.pl".  ]
yes
| ?-
```

3. **compile**

```
| ?- compile('example.pl').
{compiling $HOME/mgtpg/example.pl...}
{$HOME/mgtpg/example.pl compiled, 530 msec 22288 bytes}
yes
| ?-
```

4. **run**

```
| ?- do(S).
S = [rejected([p(a,b),...,dom(a)]),rejected([p(a,b),...,dom(a)]),
···,rejected([q(c,d),...,dom(a)]),rejected([q(c,d),...,dom(a)])] ?
yes
| ?-
```

The above is the example of series of operations from setup to clause translation and execution. According to the result of the exercise, rejected model candidates `[rejected([p(a,b), ..., dom(a)]), ...]` are output to variable `S`, thus indicating that exercise `example.mgtp` is `unsat`. If , otherwise, refutation fails, the model should have been output to variable `S`.

4.2.2 KLIC version

Operation of KLIC version is, as similar in case to already introduced Prolog version, based on KL1 compilation technology which translates a given input clause (MG clause) into KL1 clause. The MG clause translated into KL1 clause is compiled and linked together with the engine, making it possible to obtain the executable object.
The above operational procedure is outlined in Figure ?? . (1) Run the translator `mg2kl1` to translate MG clause into KL1 clause. (2) Compile and link KL1 clauses and engine (either of NAIVE, RAMS, or MERC) with KLIC to obtain the executable file (a.out).

4.2.2.1 Installation

NAIVE, RAMS, and MERC source files is put on a directory and translator `mg2kl1` is registered.

4.2.2.1.1 File organization

Table 4.2 shows the files required for installation. Program `mg2kl1.pl` which translates MG clauses into KL1 clauses, and the main program of NAIVE, RAMS,MERC are provided. (`merc_alls_st.kl1` is the MERC program for exhaustive search which tries to find all models of MG clauses.)

Each main source file may be put on recognizable directory upon compiling and linking.

4.2.2.1.2 Registration of `mg2kl1`

`mg2kl1.pl` is MG clause - KL1 clause translator described in SICStus-Prolog. It is compiled with SICStus-Prolog (or SICStus equivalent Prolog) and registered as an executable file `mg2kl1`.
Procedure for registration is shown below. (In the example, each directory through which the home, source file and path is passing is respectively shown as `$HOME`, `$HOME/mgtpg`, and

Table 4.2: File organization

File name	File contents	Used language
mg2kl1.pl	MG clause–KL1 clause translator	Prolog
naive_a_st.kl1	NAIVE body	KL1
rams_a2_st.kl1	RAMS body	KL1
merc_a_st.kl1	MERC body	KL1
merc_alls_st.kl1	exhaustive solution search type MERC body	KL1

$HOME/bin.)

Example of mg2kl1 registration

```
prompt[1]% cd $HOME/mgtpg
prompt[2]% sicstus
SICStus 2.1 #8:  Tue May 11 21:04:52 JST 1993
{ consulting $HOME/.sicstusrc...  }
{$HOME/.sicstusrc consulted, 0 msec 80 bytes}
| ?- compile(mg2kl1).
{compiling $HOME/mgtpg/mg2kl1.pl ... }
{$HOME/mgtpg/mg2kl1.pl compiled, 3450 msec 92016 bytes}
yes
| ?- save.
{SICStus state saved in $HOME/mgtpg/mg2kl1}
prompt[3]% ls -F
 merc_alls_st.kl1   mg2kl1*            naive_a_st.kl1
 merc_a_st.kl1      mg2kl1.pl          rams_a2_st.kl1
prompt[4]% mv mg2kl1 $HOME/bin
prompt[5]%
```

4.2.2.2 Operation

As mentioned earlier, KLIC version obtains the executable object after a series of processes from MG clause - KL1 clause translation, compiling the engine and linking by mg2kl1. This section gives a brief guide to translation, compilation and linking followed by a practical example.

4.2.2.2.1 MG clause–KL1 clause translation

Translate MG clauses into KL1 clauses by using MG clause–KL1 clause translator mg2kl1. The KL1 clauses become an relocatable object by compilation.

```
mg2kl1 <problem name> <type >
```

Specify either of the file name of the problem written in the MG clause (no extension) or the type (naive, ramsor merc). mg2kl1 reads the MG clauses in the file <problem name

`>.mg` and translates then into the KL1 clauses, which are output to a file `<problem name >_< type >.kl1`. Note that the form of translated KL1 clauses depends on the type of MGTP/G engine. So, different type requires a different form of KL1 clause.

4.2.2.2.2 Compile, linking

Compile and link the KL1 clause and the engine by using KLIC.

```
klic -O2 <problem name >_<executable >.kl1 <executable
>_a_st.kl1
```

Specify the file name (with no extension) of the problem to `<problem name >`, specify any of `naive, rams, merc` to `<executable >`. With `-O2` specified as an option upon compiling, it is possible to shorten execution time or to reduce file size. Detailed information on KLIC option can be obtained by specifying `klic -h`. For other information on KLIC, see details in the attached manual.

4.2.2.2.3 Execution

Execute the executable object **a.out** (or the object name if specified) created in the previous compilation and linking.

```
a.out -h1m
```

When execution is interrupted due to memory shortage, option `-h1m` may make execution succeed. (Use care that execution time may be slightly shortened as a side effect of attaching `-h1m`.)

4.2.2.2.4 Example of operation

Here is an example of a series of above operations from translation, compilation, linking, and execution. The following is finding a solution to the exercise `example.mg` shown in Figure 4.2 by using `merc_a_st.kl1` in the engine.

```
p(A,B),p(B,C)-->p(A,C).
q(A,B),q(B,C)-->q(A,C).
q(A,B)-->q(B,A).
dom(A),dom(B)-->p(A,B);q(A,B).
p(a,b)-->false.
q(c,d)-->false.
true-->dom(a),dom(b),dom(c),dom(d).
```

Figure 4.2: Example (`example.mg`)

1. **MG clause–KL1 clause translation**

```
prompt[1]% cd $HOME/mgtpg
prompt[2]% mg2kl1 example merc
prompt[3]% ls -F
 example.mg         merc_alls_st.kl1   mg2kl1*            naive_a_st.kl1
 example_merc.kl1   merc_a_st.kl1      mg2kl1.pl          rams_a2_st.kl1
prompt[4]%
```

2. **compile, link**

```
prompt[4]% cd klic
prompt[5]% klic -O2 merc_a_st.kl1 example_merc.kl1
prompt[6]% ls -F
 a.out*             example_merc.kl1   merc_a_st.kl1      naive_a_st.kl1
 atom.c             funct.c            merc_alls_st.kl1   predicates.c
 atom.h             funct.h            mg2kl1*            predicates.o
 atom.o             funct.o            mg2kl1.pl          rams_a2_st.kl1
 example.mg         klic.db
prompt[7]%
```

3. **run**

```
prompt[7]% a.out -h1m
unsat

No.  of branches = 384
average length of a branch = 26
        minimum length of a branch = 6
        maximum length of a branch = 30
total cons counts for M = 1902
No.  of CJMs     = 106406
cjm calls        = 25878
cjm1 calls       = 88518
total cjm* calls = 114396

prompt[8]%
```

The above example shows a series of steps from translation to execution. The result is `unsat`, indicating a success in refutation. If refutation fails, the result is `sat`. Table tab:MGTPGKstats shows the meaning of statistical information obtained at the same time.

4.2.3 KL1 version

Operation is almost the same as in Prolog version (see 4.2.1). **4.2.3.1 Installation**

Table 4.4 lists the required files.

Put the necessary files into one directory and specify `Install` from listener with `take` command.

Table 4.3: Meaning of statistical information

Display contents	mn1cDisplay contents
No. of branches	No. of branches of contradict proof tree
average length of a branch	average value of branch length (size of rejected model candidate)
minimum length of a branch	minimum of the same
maximum length of a branch	maximum of the same
total cons counts for M	total atoms of model extend
No. of CJMs	total calls for KL1 clause c/4 in conjunction matching
cjm calls	total calls for KL1 clause cjm/5 in conjunction matching
cjm1 calls	total calls for KL1 clause cjm1/6 in conjunction matching
total cjm* calls	total calls for cjm, cjm1

Table 4.4: File organization

File name	File contents	Used language
`trans.kl1`	MG clause–KL1 clause translation program	KL1
`Pretrans.kl1`	Preprocessing of MG clause–KL1	
clause translation program	KL1	
`Pretrans.kl1`	Preprocessing of MG clause–KL1	
clause translation program	KL1	
`mgtpI.kl1`	Interpreter body(Strategy I)	KL1
`mgtpII.kl1`	Interpreter body(Strategy II)	KL1
`mgtpIII.kl1`	Interpreter body(Strategy I)	KL1
`s1.kl1`	(dummy's)problem clause	KL1
`mgtpMac.mac`	macro set	KL1 macro
`termMemoryG.kl1`	distinction tree program	KL1
`plib.sav`	library	KL1(compiled)
`listlib.sav`	library	KL1(compiled)
`Startup`	system startup file	listener take file
`Install`	instal file	listener take file

```
[17] take("Install").

[18] load["plib","listlib"] .
Load File : icpsi520::>sys>user>miyuki>KL1>MGTPG>plib.sav.1
Load File icpsi520::>sys>user>miyuki>KL1>MGTPG>listlib.sav.1
"miyuki::$plib$error" Updated
"miyuki::$plib$window" Updated
"miyuki::$plib$timer" Updated
"miyuki:: listlib" Updated
"miyuki:: plib" Updated
"miyuki::$plib$file" Updated
"miyuki::$plib$string_io" Updated
Load Succeeded

[19] compile["termMemoryG"] .
** KL1 Compiler **
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>termMemoryG.kl1.1
!WARNING! no with_macro declaraion. assumed ‘pimos’.
[termMemoryG.kl1@0] Compile Module : tmG
=>  :- module tmG .
Compile Succeeded : tmG

"miyuki:: tmG" Updated
Total Number of Warning : 1
Compilation Time = 17558 [MSEC]

[20] inter([prompt("Input ENGINE Name>> "),getl(L)])’;’
    compile([L,"s1"]) .Input ENGINE Name>> mgtpI
** KL1 Compiler **
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>mgtpI.kl1.1
!WARNING! no with_macro declaraion. assumed ‘pimos’.
[top.kl1@0] Compile Module : mgtp
=>  :- module mgtp .
Compile Succeeded : mgtp

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>s1.kl1.1
!WARNING! no with_macro declaraion. assumed ‘pimos’.
[s1.kl1@0] Compile Module : mgtp_p
=>  :- module mgtp_p .
Compile Succeeded : mgtp_p

"miyuki:: mgtp" Updated
"miyuki:: mgtp_p" Updated
Total Number of Warning : 2
```

```
Compilation Time = 18914 [MSEC]

[22] compile["trans","preTrans"],compile["mgtpMac"] .
** KL1 Compiler **
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>mgtpMac.mac.1
[mgtpMac.mac@0] Compile Module : mgtpMac
=>  :- macro_module(mgtpMac) .
Compile Succeeded : mgtpMac

"miyuki:: mgtpMac" Updated
Compilation(s) Succeeded
Compilation Time = 22233 [MSEC]
** KL1 Compiler **
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>trans.kl1.3
[trans1.kl1@0] Compile Module : mgtp_trans
=>  :- module mgtp_trans .
Compile Succeeded : mgtp_trans

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>preTrans.kl1.1
[preTrans.kl1@0] Compile Module : mgtp_pre_trans
=>  :- module mgtp_pre_trans .
Compile Succeeded : mgtp_pre_trans

"miyuki:: mgtp_pre_trans" Updated
"miyuki:: mgtp_trans" Updated
Compilation(s) Succeeded
Compilation Time = 57689 [MSEC]

[23] unload([mgtp_p,mgtp,mgtp_trans,mgtp_pre_trans,mgtpMac,tmG],L) .
Unload File : icpsi520::>sys>user>miyuki>KL1>MGTPG>mgtpI.sav.4
miyuki::mgtp_p unloaded
miyuki::mgtp unloaded
miyuki::mgtp_trans unloaded
miyuki::mgtp_pre_trans unloaded
miyuki::mgtpMac unloaded
miyuki::tmG unloaded
Unload Succeeded

[24]
```

4.2.3.2 Operation

Upon completion of installation, and if `take Startup` from the listener, the necessary files are compiled or loaded, then a new window (MGTP window) opens.

In the following example, all files have already been stored in the directory named

MGTPG.

4.2.3.2.1 Startup

```
[5] cd("MGTPG").

[6] take("Startup").        : Initiate Startup.

[7] load["plib","listlib"] . : Library load
Load File : icpsi520::>sys>user>miyuki>KL1>MGTPG>plib.sav.1
Load File : icpsi520::>sys>user>miyuki>KL1>MGTPG>listlib.sav.1
"miyuki::$plib$error" Loaded
"miyuki::$plib$window" Loaded
"miyuki::$plib$timer" Loaded
"miyuki:: listlib" Loaded
"miyuki:: plib" Loaded
"miyuki::$plib$file" Loaded
"miyuki::$plib$string_io" Loaded
Load Succeeded

[8] inter([prompt("Input ENGINE Name>> "),getl(L)])';' load[L] .
Input ENGINE Name>> mgtpI
Load File : icpsi520::>sys>user>miyuki>KL1>MGTPG>mgtpI.sav.4
"miyuki:: tmG" Loaded
"miyuki:: mgtp" Loaded
"miyuki:: mgtp_p" Loaded
"miyuki:: mgtpMac" Loaded
"miyuki:: mgtp_pre_trans" Loaded
"miyuki:: mgtp_trans" Loaded
Load Succeeded
L ="mgtpI"

[9] pimos:: listener: go(at(109,199),char(70,30),"font:test_11") .
            : Start MGTP window
```

4.2.3.2.2 Compiling MGTP clause

```
[6] compile("s2.mgtp").      : Compile S2problem
** KL1 Compiler **
Parameter File : "icpsi520::>sys>user>miyuki>KL1>MGTPG>compile.param.2"

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPG>s2.mgtp.1
!WARNING! no with_macro declaraion. assumed 'pimos'.
[s2.mgtp] Compile Module : mgtp_p
Compile Succeeded : mgtp_p
```

```
"miyuki:: mgtp_p" Updated
Total Number of Warning : 1
"miyuki:: mgtp" Updated
Relink Succeeded
Compilation Time = 14038 [MSEC]
```

4.2.3.2.3 Execution

```
[7] mgtp:do(U,S).   : Start proving(if unsatisfied,  U bounded to unsat,
 S = S               : if satisfied, S bounded to sat.
U = unsat

[8]
```

Chapter 5

MGTP/N Manual

This chapter outlines the algorithm of MGTP/N and gives a guide to its usage. The MGTP/N is a model generation theorem prover for Horn problems. The Horn problem is a clause set consisting of only Horn clauses [1]. The range restrictedness applied in MGTP/G is not particularly imposed.

The MGTP/N is available in Prolog version running on SICStus Prolog and KL1 version running on PIM. KL1 version has model sharing type and model distribution type depending on difference in internal memory managements.

5.1 Function of MGTP/N

5.1.1 Prolog version

With a compact system core (1 page in A4 size), Prolog version promises the user to view the whole picture of the system with ease of understanding. Since the storage of model candidate or model extending candidate is entirely is implemented by assert and retract functions of Prolog, the subsequent improvement in the speed of reference and update to Prolog's internal database (e.g., improved index algorithm) will further promise faster inference.

It is also possible to specify forward and backward subsumption, sorting and deletion strategies, and conjunctive matching order.

5.1.2 KL1 version

KL1 version is aiming at making fully use of the power of PIM. It is available in two types, model sharing and model distribution types, depending on the difference in how to hold model candidate and model extending candidate. The model sharing type allows each processor to hold an entire model candidate and model extending candidates. In model distribution type KL1 version, on the other hand, model candidate and model extending candidate are distributed across all processors, each processor merely retaining part of the candidate.

In the model sharing version, communication cost in subsumption check can be reduced to a minimum, thus making it possible to achieve paralleling effect in tune. In the model

[1] number of consequent atoms is at most 1

```
(0)  Init: $MD[i] := C_i$ for $i = 1, \ldots, k$ where $\{C_1, C_2, \ldots, C_k\} = \{C \mid (true \rightarrow C) \in S\}$
(1)  Init: $g := 1; s := k$;
(2)  **while** $g \leq s$ **do begin**
(3)   **foreach** $A_1, \ldots, A_n \rightarrow C$
(4)    **foreach** $(i_1, \ldots, i_n)$ s.t. $\forall j(1 \leq i_j \leq g)$ and $\exists j(i_j = g)$
(5)     **if** $\exists\sigma$(substitution)$\forall j(MD[i_j]\sigma = A_j\sigma$ and $C\sigma$is new$)$ **then begin**
(6)      $s := s + 1; MD[s] := C\sigma$; **end**; /* model extension */
(7)      **foreach** $A_1, \ldots, A_m \rightarrow false$ **do**
(8)       **foreach** $(i_1, \ldots, i_m)$ s.t. $\forall j(1 \leq i_j \leq s)$ and $\exists j(i_j = s)$ **do**
(9)        **if** $\exists\sigma'(substitution)\forall j(MD[i_j]\sigma' = A_j\sigma')$ **then return** (unsat);
            /* model reject */
(10)    **end**;
(11)  $g := g + 1$;
(12) **end** ;
(13) **return** (sat);
```

Figure 5.1: MGTP/N sequential alogrithm

distribution type, on the other hand, communication cost in subsumption check becomes higher whereas memory scalability parallelizing effet in spaces can be obtained.

Either version has a built-in forward subsumption check and can incorporate the deletion strategy.

5.2 MGTP/N sequential algorithm

This section is a guide to sequential alogrithm, model sharing parallel algorithm, and model distribution parallel algorithm. See [?] for details of model sharing parallel algorithm.

5.2.1 Sequential algorithm

Figure 5.1 shows the sequential alogrithm. Branching is not considered here because this section deals only with Horn clauses. Any version mentioned in this section is based on this algorithm.

MD is an array used to hold (cosequent) atoms created with model extension applied where atoms from $MD[1]$ to $MD[g-1]$ indicate the model candidates, and $MD[g]$ to $MD[s]$ indicate the model extension candidates (a set of generation atoms to be added to the model candidates). The model candidates are initialized in a null set, and the model extension candidates are initialized in the set of consequent agoms of positive clauses respectively. ((0), (1)).

(3) through (6) correspond to model extension. The antecedent of each nested clause is tested to determine if it is satisfied with the model candidate (the first half of the condition

of (4) and (5), and if it is found satisfiable, the consequent is tested for subsumption (the last half of the condition of (5)), and if it is not subsumed, it is added to MD ((6)).

$C\sigma$ is new in the condition of (5) indicates that the generation atom $C\sigma$ is not subsumed in any element of MD ($\not\exists\sigma'$(substitution)$C\sigma = MD[i]\sigma'$ for $\forall i(1 \leq i \leq s)$). For subsumption test, this thesis deals only with the forward subsumption, and does not deal with the backward subsumption test which is to determine whether $C\sigma$ subsumes the element of MD ($\exists i(1 \leq i \leq s)\exists\sigma'$(substitute)$C\sigma\sigma' = MD[i]$) . [2].

Subsequent models (7) through (9) is subject to model rejection. The antecedent of each negative clause is tested to determine if it is satisfied with the model extension candidate, and if found satisfiable, the proving procedure exits.

5.2.2 Model sharing parallel algorithm

5.2.2.1 Algorithm using lock and unlock

The simplest way for parallelization of the serial algorighm in Figure 5.1 is to execute each one of **while** loop in parralel. Within the loop, conjunction matching of nested clause (the first half of (4) and (5)), subsumption test (the last half of (5)), and updating of MD (6) are performed in the first place, then the test proceeds with conjunction matching of negative clauses (from (7) to (9)).

We assume it as generate-and-test calculus which links two processes; one is the generation process (G process) performing conjunction matching of nested clauses, and the other is the rejection test process (T process) performing conjunction matching of negative clauses; and devised the parallel algorithm within the loop.

The problem here is the subsumption test conducted in parallel and updating of MD. Because the subsumption test and updating of MD are conducted in a competitive manner, a certain type of locking feature is required.

A solution was to provide a buffer used to store the atoms newly generated by the G process. Figure 5.2 shows the parallel algorithm which uses this buffer (expressed in Buf). The atoms which a group of G processes generates via Buf are submitted to the T process. The G process repeats the process from (G0) to (G6), and the T process repeats the process from (T0) to (T8) respectively. Generation atoms are stored in Buf by the G process ((G5)), and these atoms are taken by the T process from Buf ((T0)).

Exclusive control of multiple G processes is conducted with lock on g ((G0)). The procedure from (G1) to (G3) is similar in case to serial version. The processing from (T4) to (T6) is also precisely the same as in serial processing.

Subsumption test is conducted in the T process ((T2)). Since MD is updated every moment, it should be locked in order to conduct a complete subsumption test. After locking MD ((T1)) , conduct the subsumption test on the atom δ taken from Buf ((T2)), and if it is not subsumed, register it to MD, and release the lock ((T3)).

5.2.2.2 Algorithm using master process

[2]Because $MD[i]$ subsumed in $C\sigma$ should be deleted, it is necessary to keep the data consistent, so that parallelization is more difficult than in the forward subsumption test. In KL1 version where destructive substitution is not permitted, copy is needed on deletion.

```
(0)(1)  Figure 5.1same as (0)(1)
(1)       (2)Init: $Buf := \emptyset; b := 1;$

process G
(G0) lock $g; g' := g; g := g + 1;$ unlock $g;$
(G1) foreach $A_1, \ldots, A_n \rightarrow C$ do
(G2)  foreach $(i_1, \ldots, i_n)$ s.t. $\forall j(1 \leq i_j \leq g') \wedge \exists j(i_j = g')$ do
(G3)   if $\exists\sigma\forall j(MD[i_j]\sigma = A_j\sigma)$
(G4)    then begin lock $Buf;$
(G5)    $Buf[b] := C\sigma; b := b + 1;$
(G6)    unlock $Buf;$ end;

process T
(T0) lock $Buf; \delta := Buf[b]; b := b - 1;$ unlock $Buf;$
(T1) lock $MD;$
(T2) if $\delta$ is new then begin
(T3)  $s := s + 1; MD[s] := \delta;$ s' := s; unlock $MD;$
(T4)  foreach $A_1, \ldots, A_m \rightarrow false$ do
(T5)   foreach $(i_1, \ldots, i_m)$ s.t. $\forall j(1 \leq i_j \leq s') \wedge \exists j(i_j = s')$ do
(T6)    if $\exists\sigma'\forall j(MD[i_j]\sigma' = A_j\sigma')$ then
(T7)     return (unsat);
(T8) end else unlock $MD$
```

Figure 5.2: MGTP/N parallel algorithm (lock&unlock)

Figure 5.3 shows the various results of review reflected on this algorithm. This algorithm comes in three types, G process and T process as well as Master (M) process. Relation of logical link of those processes is shown in Figure 5.4. The upper G process group and the lower T process are linked via the M process placed in the middle of the figure.

In Figure 5.3, Occam-like channel notation is used to represent inter-rocess communication. It indicates that the value of variable X is output to channel *channel* by $channel!X$, and the value of variable X is input to G process from channel *channel* by $channel?X$. MGC is the channel used for communication from M process to G process, MTC is the channel used for communication from M process to T process, TMC is the same from T process to M process. By means of these communication channels, the operation equivalent of lock function shown in Figure 5.2 can be implemented. [3]

After initialization in the M process, `(M1)` and `(M2)`$\sim$`(M4)` are executed in parallel. In `(M1)`, the responsible range (indicated by g) of conjunction matching is assigned to each G process, and at the same time, buffer New_g used to store the generated atoms is provided and assigned. Because New_g is made available for each G process, no lock is required when updated(`(G4)`).

Subsumption test is conducted separately in the G and T processes (condition part of `(G3)` and `(T1)`). Of the subsumption test, what is conducted in the G process is called local subsumption test (LS), and conducted in the T process is called global subsumption test (GS). Here, the number of MD elements held by the PE upon conducting LS is assumed as m to indicate the range of the subsumption test. For example, $\forall i(1 \leq i \leq m)\ \nexists\sigma'(C\sigma = MD[i]\sigma')$ is expressed by $C\sigma$ is new to $MD[1,\ldots,m]$. The atoms, generated and pass through LS, are stored in New_g, and at the same time, m which indicates the scope of LS is also stored (`(G4)`).

The T process first receives the combination of the atom δ generated by the G process and GS range $(m,\ldots,s')$ from the M process (`(T0)`). Then, if δ is not subsumed, the process assigns δ to $MD[s'+1]$ and sends $s'+1$ to the M process. If δ is subsumed, the process sends s' to the M process. The value (s' or $s'+1$) sent to the M process allocates the range of GS requiired for the atom next to δ.

5.2.2.3 Several improvements for reduced sequentiality

In principle, this parallelization scheme has two sequentialities that pose a detetrent factor to introducing parallel performance. One is the sequentiality in subsumption test and the other is what is intended to guarantee the invariance of proving.

1. Subsumption test

   For a complete forward subsumption test conducted with newly generatd atoms, sequentiality takes place because it is necessary to keep the order of generation and to conduct the subsumption test on the previously generated atom set. In order to reduce subh sequentiality, the subsumption test was separated into LS and GS.

   Thanks to the model sharing scheme introduced, LS for each generated atom can be conducted separately in each PE with no impact imposed on other PEs. However, the

[3] A practical implementaion is performed by means of sending and receiving of undefined variables of KL1.

```
(0)  Figure 5.1 same as (0)
(1)  Init: $s := k$;

process M
(M0) Init: $g := 1; g' := 1; b' := 1$
(M1) $MGC$ ! $\{g, New_g\}$; $g := g + 1$; goto (M1);
(M2) while $New_{g'}[b'] = close$ then begin
(M3)     $g' := g' + 1; b' := 1$ end
(M4) $MTC$ ! $\{New_{g'}[b'], s\}$; $TMC$ ? s; goto(M2);

process G
(G0) $MGC$ ? $\{g, New_g\}$; $b := 1$;
(G1) foreach $A_1, \ldots, A_n \rightarrow C$ do
(G2)     foreach $(i_1, \ldots, i_n)$ s.t. $\forall j(i_j \leq g) \wedge \exists j(i_j = g)$ do
(G3)         if $\exists \sigma \forall j(MD[i_j]\sigma = A_j\sigma \wedge$ $C\sigma$ is new to $MD[1, \ldots, m])$ then begin
(G4)             $New_g[b] := \{C\sigma, m\}; b := b + 1$ end;
(G5)     $New_g[b] := close$;

process T
(T0) $MTC$ ? $\{\{\delta, m\}, s'\}$;
(T1) if $\delta$ is new to $MD[m + 1, \ldots, s']$ then begin
(T2)     $MD[s' + 1] := \delta$; $TMC$ ! $s' + 1$;
(T3)     foreach $A_1, \ldots, A_m \rightarrow false$ do
(T4)         foreach $(i_1, \ldots, i_m)$ s.t. $\forall j(i_j \leq s') \wedge \exists j(i_j = s')$ do
(T5)             if $\exists \sigma' \forall j(MD[i_j]\sigma' = A_j\sigma')$ then
(T6)                 return (unsat);
(T7) end else $TMC$ ! $s'$;
```

Figure 5.3: Parallel algorithm of model sharing MGTP/N

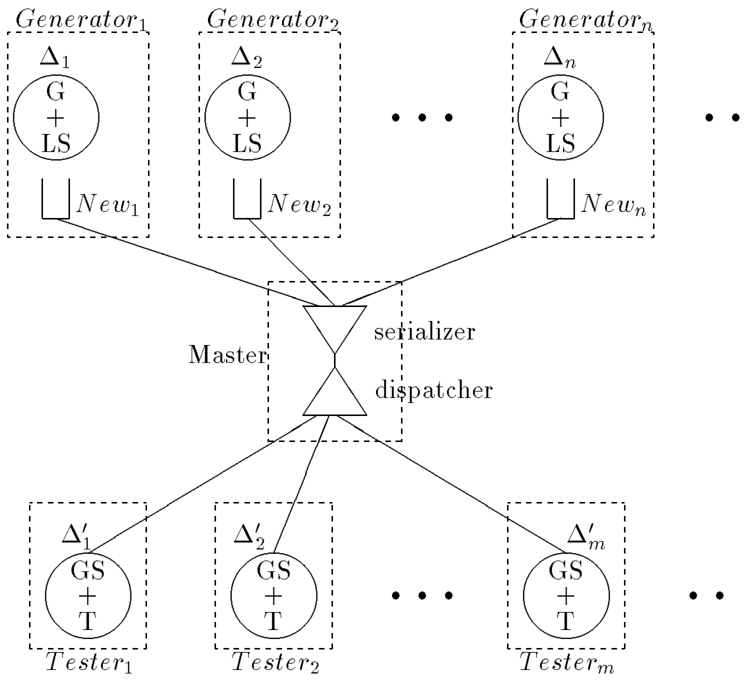

Figure 5.4: Figure of p rocess linking of model sharing MGTP/N

larger the scope of LS (in other words, the larger the m contained in `(G3)` as shown in Figure 5.3, the higher the possibility of atoms to be subsumed. The result is turned out to be favorable to reduce the load of communication and GS. This is indicative of the need for such an implementation to be propagated over all other PEs.

For the LS independently executable for each PE, GS still has remaining sequentiality. This stems from the reason that the subsumption test for the next atom is never completed until the test for the preceding atom is finished. According to Figure 5.3, for example, the judgement on "δ is new to $MD[m+1,\ldots,s']$" in `(T1)` requires that all values of $MD[m+1],\ldots,MD[s']$ have been determined, or in other words, the associated subsumption tests have been entirely completed.

In order to reduce the sequentiality of this GS, the global array TMD (called temporary stack list) used to retain the atoms already undergoing LS test is provided. This array is, in practice, exactly similar to Buf used in Figure 5.2. Of the elements of TMD, the atoms not subsumed by GS become the element of MD, so that MD becomes partial set of TMD.

If T TMD is used, the procedure in Figure 5.3 may be changed to the following. It is assumed that TMD undergoes the initialization similar in case to MD, and is initialized to $ts := k+1$.

(1) Change `(M4)` as follows. : $MTC\ !\ \{New_{g'}[b'], s, ts\}; ts := ts+1; TMC\ ?\ s;$

(2) Change `(T0)` as follows. : $MTC\ ?\ \{\{\delta, m\}, s', ts'\};$

(3) Insert the following between `(T0)` and `(T1)`. : $TMD[ts'] := \delta$

(4) Change the condition part of (`T1`) as follows. : δ is new to $TMD[(m+1)^\flat, \ldots, ts'-1]$
Where $()^\flat$ is the function from the index of MD to the index of TMD, and satisfies $MD[m] = TMD[(m)^\flat]$.

According to (3), the atoms are stacked in the temporary stack list (TMD) before GS starts, so that the GS of other atoms needs not wait for the termination of the foregoing GS. Possibility is that some atoms expected to be subsumed may be placed in the temporary stack list, and this may result in redundancy in a way such that the subsumption test for those atoms is not conducted in the absence of the temporary stack list. Reduction of sequentiality by means of temporary stack list is obtained at the cost of such redundancy.

2. Invariance of proving

   To render the proving invariant, the fetching order of generated atoms must be fixed. The fixed fetching order is nothing other than the occurrence of sequentiality. When two G processes G_1 and G_2 are operating in parallel, for example, generation atom set New_1 of G_1 , and generation atom set New_2 of G_2 are generated in parallel, but it is not possible to fetch the preceding atom first. Unless they are fetched in the predetermined order, the proof can vary for each execution. Assume that the atom is predetermined to be fetched from New_1. If the generation of New_1 is delayed for some reason, the next atom cannot be fetched even if New_2 has already been generated, so that the T process stalls.Letting the G process run in parallel on G_2, $G_3 \ldots$ and so on while execution of G_1 is stagnant can eliminate apparently idle PEs, but chances are that wasteful atoms are generated, thus running counter to the concept of delayed generation.

   In order to reduce the above sequentiality, it was determined to refine the grain size of the G process. Practically, in terms of the work unit (in Figure 5.3, processing of the loop of (`G1`) against an atom $MD[g]$) entrusted to the G process, is subdivided and equally allocated to $1/k$ and is treated as the work unit. The result is smaller grain size in the G process with no appreciable disparity observed.

5.2.3 Model distibuted parallel algorithm

The model distributed parallel algorithm is implemented by master slave configuration. It parallelized the `while` loop of the sequential algorithm (Figure **??**). The loop consists of three phases; the generation (the first half of the condition part of *(4)* through *(5)* subsumption test (the last half of the condition part of *(5)* and rejection test *(8)* and *(9)*. Each phase is parallized. And the model candiate and model extending candidate are distributed into each PE.

Figure **??** shows the processing flow in the unit step.

1. Selection of model extension candidate

The model extending candidate d is selected from a processor j,[4] and is distributed to all other processors.

2. Conjunctive matching Each node takes model extending candidate d as input,

   (a) Conjunctive matching (unification) of mixed clauses against model candidate set M in each PE and d, and combination of d are performed. [5]

   (b) If the conjunctive matching sucessed, generate atom set CS in the consequent part of the mixed clauses, and give ID to each atom.

3. Edit the generated atoms.

   The master process gathers generated atoms and attached a switch to each of them in order to broadcast information on the result of subsumption testing in each PE. These atoms are referd as D. [6]

4. Subsumption testing of the element in D against the model candidate + model extending candidate MD and D are performed, and subsumed atoms are discarded.

5. Subsumption test

   [7]

6. Conjunctive matching of negative clause.

   Conjunctive matching (unification) of the negative clause against the model candidate (+ model extending candidate) MD and unsubsumed D is performed. If the matching succeed, the processing exits.

7. Updating the internal data

   Each unsubsumed element of D is distributed in accordance with master's specification and stored as an element of model extending candidate. The internal status Fd as the feedback information for the next step is also output. [8]

5.3 Operation

MGTP input clause is translated into Prolog clause when in Prolog version, whereas it is translated into KL1 clause in case of KL1 version. Currently, neither version provides the applicable translator, but a file is attached as an exercise for the problems listed in [?].

[4]registered as a new model candidate to the process j of after conjunction matching.

[5]All of conjunctive matchings against combination of ds are repeated at all nodes.

[6]It becomes the input data to the last half of the processor.

[7]backward subsumption test is not performed subsumption testing against D is duplicated at all nodes.

[8]This information provides the basis to determine the node j by which the model extension candidate should be selected in the next step.

5.3.1 Operation of Prolog version

5.3.1.1 Necessary file

File name	File contents
`mgtp.pl`	MGTP/N engine
`mgtpS.pl`	MGTP/N engine (with sort function)
`unify.pl`	unification with occur-check program
`pro*.pl`	(translated) positive clause and mixed clause
`pro*t.pl`	(translated) negative clause

5.3.1.2 Compiling, execution example

1. Compilation :

```
koshi@ss130[3]% sicstus    : SISCtus Prolog startup
SICStus 2.1 #8: Tue May 11 21:04:52 JST 1993
| ?- compile([mgtp,unify]).     : compiling NGTP/N engine and
                                  unification program
{compiling /home2/koshi/prolog/MGTPN/mgtp.pl...}
{Warning: [X] - singleton variables in do1/1 in lines 7-8}
{Warning: [F] - singleton variables in weight/3 in lines 35-36}
{Warning: [T] - singleton variables in weightArg/4 in lines 36-38}
{Warning: [X] - singleton variables in init/0 in lines 70-72}
{Warning: [X] - singleton variables in init/0 in lines 72-73}
{Warning: [X] - singleton variables in init/0 in lines 73-74}
{Warning: [F,N,Ref] - singleton variables in countData/2 in lines
          74-76}
{Warning: [N,X] - singleton variables in countData/2 in lines 76-80}

{Warning: [F] - singleton variables in countData/2 in lines 80-81}
          {compiled /home2/koshi/prolog/MGTPN/mgtp.pl in module mgtp,
           340 msec 42976 bytes}
{compiling /home2/koshi/prolog/MGTPN/unify.pl...}
{Warning: [X,Y] - singleton variables in unifyArg/3 in lines 6-8}
{Warning: [X] - singleton variables in notOccursIn/2 in lines 17-18}
{Warning: [F] - singleton variables in notOccursIn/2 in lines 18-19}
{Warning: [X,Y] - singleton variables in notOccursInArg/3 in lines
          19-21}
          {compiled /home2/koshi/prolog/MGTPN/unify.pl in module
           unify, 160 msec 9248 bytes}

yes
| ?-
```

2. Execution example

```
| ?- compile([pro2,pro2t]).    : Compiling problem clause
{compiling /home2/koshi/prolog/MGTPN/pro2.pl...}
{compiling /home2/koshi/prolog/MGTPN/pro1.pl...}
{Warning: [Y] - singleton variables in delta/1 in lines 6-8}
{Warning: [X1,Y] - singleton variables in '$p/1'/3 in lines 25-26}
{Warning: [F] - singleton variables in useless1/1 in lines 50-51}
{Warning: [A,X] - singleton variables in useless1Arg/3 in lines 54-56}
{Warning: [Fa] - singleton variables in useless1Arg/3 in lines 56-57}
{compiled /home2/koshi/prolog/MGTPN/pro1.pl in module P, 170 msec
          15232 bytes}
The procedure delta/1 is being redefined.
    Old file: /home2/koshi/prolog/MGTPN/pro1.pl
    New file: /home2/koshi/prolog/MGTPN/pro2.pl
Do you really want to redefine it? (y, n, p, or ?) p
{Warning: [Y] - singleton variables in delta/1 in lines 4-6}
{Warning: [Y] - singleton variables in delta/1 in lines 6-7}
{/home2/koshi/prolog/MGTPN/pro2.pl compiled, 220 msec 15408 bytes}
{compiling /home2/koshi/prolog/MGTPN/pro2t.pl...}
{compiling /home2/koshi/prolog/MGTPN/pro1t.pl...}
{compiled /home2/koshi/prolog/MGTPN/pro1t.pl in module Pt,
          90 msec 2256 bytes}
The procedure p/1 is being redefined.
    Old file: /home2/koshi/prolog/MGTPN/pro1t.pl
    New file: /home2/koshi/prolog/MGTPN/pro2t.pl
Do you really want to redefine it? (y, n, p, or ?) p
{/home2/koshi/prolog/MGTPN/pro2t.pl compiled, 130 msec 2624 bytes}
yes
| ?- do.     : Start proving
Succeed      : Proving succeeded
yes
| ?-
```

【Note】 If problem solution continues : It is necessary to compile the problem clause again after initializing the internal database with `init` instruction.

```
| ?- init.  : Initializing internal database

yes
| ?- compile([pro3,pro3t]).       : Compiling new problem clause
{compiling /home2/koshi/prolog/MGTPN/pro3.pl...}
```

```
{compiling /home2/koshi/prolog/MGTPN/pro2.pl...}
{compiling /home2/koshi/prolog/MGTPN/pro1.pl...}
{Warning: [Y] - singleton variables in delta/1 in lines 6-8}
{Warning: [X1,Y] - singleton variables in '$p/1'/3 in lines 25-26}
{Warning: [F] - singleton variables in useless1/1 in lines 50-51}
{Warning: [A,X] - singleton variables in useless1Arg/3 in lines 54-56}
{Warning: [Fa] - singleton variables in useless1Arg/3 in lines 56-57}
{compiled /home2/koshi/prolog/MGTPN/pro1.pl in module
          P, 210 msec 3440 bytes}
The procedure delta/1 is being redefined.
    Old file: /home2/koshi/prolog/MGTPN/pro1.pl
    New file: /home2/koshi/prolog/MGTPN/pro2.pl
Do you really want to redefine it? (y, n, p, or ?) p
{Warning: [Y] - singleton variables in delta/1 in lines 4-6}
{Warning: [Y] - singleton variables in delta/1 in lines 6-7}
{/home2/koshi/prolog/MGTPN/pro2.pl compiled, 270 msec 3504 bytes}
{/home2/koshi/prolog/MGTPN/pro3.pl compiled, 290 msec 3584 bytes}
{compiling /home2/koshi/prolog/MGTPN/pro3t.pl...}
{compiling /home2/koshi/prolog/MGTPN/pro1t.pl...}
{compiled /home2/koshi/prolog/MGTPN/pro1t.pl in module Pt,
          90 msec 1744 bytes}
The procedure p/1 is being redefined.
    Old file: /home2/koshi/prolog/MGTPN/pro1t.pl
    New file: /home2/koshi/prolog/MGTPN/pro3t.pl
Do you really want to redefine it? (y, n, p, or ?) p
{/home2/koshi/prolog/MGTPN/pro3t.pl compiled, 119 msec 2112 bytes}

yes
| ?-
```

5.3.1.3 The form of translated clauses –example–

1. MGTP input clause (problem 1)

   a) Positive clause

$$\begin{array}{l} true \rightarrow p(i(X, i(Y, X))). \\ true \rightarrow p(i(i(X, i(Y, Z)), i(i(X, Y), i(X, Z)))). \\ true \rightarrow p(i(n(n(X)), X)). \\ true \rightarrow p(i(X, n(n(X)))). \\ true \rightarrow p(i(i(X, Y), i(n(Y), n(X)))). \end{array}$$

   b) Negative clause

$$p(i(i(a, i(b, c)), i(b, i(a, c)))) \rightarrow false.$$

c) Mixed clause
$p(X), p(i(X,Y)) - > p(Y).$

2. Prolog clause after translation

a) Positive clause(Module is 'P')

```
delta(p(i(X,i(Y,X)))).
delta(p(i(i(X,i(Y,Z)),i(i(X,Y),i(X,Z))))).
delta(p(i(n(n(X)),X))).delta(p(i(X,n(n(X))))).
delta(p(i(i(X,Y),i(n(Y),n(X))))).
```

b) Negative clause(Module is 'Pt')

```
p(i(i(a,i(b,c)),i(b,i(a,c))))mgtp:proofEnd.
```

c) Mixed clause(Module is 'P')

```
p(X) :- copy_term(X,Xc), '$p/1'(3, X,Xc).
p(i(X1,Y)) :- '$p/1'(2, X, v(X,X1,Y)).
p(X) :- '$p/1'(1, i(X1,Y), v(X,X1,Y)).

'$p/1'(1, A2, Vars) :- model(p(A2)), '$p/2'(1,Vars).
'$p/1'(2, A2, Vars) :- model(p(A2)), '$p/2'(2,Vars).
'$p/1'(3, X, Xc) :- '$p/2'(3, Xc, v(X,X1,Y)).

'$p/2'(1, v(X,X1,Y)) :- unify:unify(X,X1), mgtp:consq(p(Y)).
'$p/2'(2, v(X,X1,Y)) :- unify:unify(X,X1), mgtp:consq(p(Y)).
'$p/2'(3, i(X1,Y), v(X,X1,Y)) :- unify:unify(X,X1), mgtp:consq(p(Y)).
```

【Note】 When MGTP negative clause is $p(X), p(i(X, i(i(a, i(b, c)), i(b, i(a, c))))) \rightarrow false.$, it is translated as follows;

```
p(X) :-'$p/1'(1,i(X1,i(i(a,i(b,c)),i(b,i(a,c)))),v(X,X1)).
p(i(X1,i(i(a,i(b,c)),i(b,i(a,c))))):-'$p/1'(2, X,v(X,X1)).
%p(i(i(a,i(b,c)),i(b,i(a,c)))) :- mgtp:proofEnd.

'$p/1'(1, A2, Vars):- 'P':model(p(A2)),  '$p/2'(1,Vars).
'$p/1'(1, A2, Vars):- 'P':current(p(A2)),'$p/2'(1,Vars).'
'$p/1'(1, A2, Vars):- 'P':delta(p(A2)),  '$p/2'(1,Vars).
'$p/1'(2, A2, Vars):- 'P':model(p(A2)),  '$p/2'(2,Vars).
'$p/1'(2, A2, Vars):- 'P':current(p(A2)),'$p/2'(2,Vars).
'$p/1'(2, A2, Vars):- 'P':delta(p(A2)),  '$p/2'(2,Vars).

'$p/2'(1, v(X,X1)) :- unify:unify(X,X1), mgtp:proofEnd.
'$p/2'(2, v(X,X1)) :- unify:unify(X,X1), mgtp:proofEnd.

.
```

5.3.2 KL1 version(model sharing)operation

5.3.2.1 Necessary files

File name	File contents
`mgtpN.kl1`	NGTP/N engine
`print.kl1`	output routine
`meta.kl1`	unification, matching procedure
`tm.kl1`	discrimination tree program
`tmLib.kl1`	discrimination tree program interface
`pro*.kl1`	(translated)problem clause

5.3.2.2 Compiling, execution, operation

1. Compiling :

```
[1] compile["mgtpN","print","meta","tm","tmLib","pro3"].
       : Compiling necessary files

** KL1 Compiler **
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>print.kl1.1
[dbg.kl1@0] Compile Module : mgtp_d
Compile Succeeded : mgtp_d

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>meta.kl1.1
[meta0m1.kl1@0] Compile Module : meta
Compile Succeeded : meta
Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>tmLib.kl1.1
!WARNING! no with_macro declaraion. assumed ‘pimos’.
[mgtpLibne.kl1@1] Compile Module : mgtp_lib
Compile Succeeded : mgtp_lib

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>tm.kl1.1
[tm_ne.kl1@0] Compile Module : tm
Compile Succeeded : tm

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>pro3.kl1.1
[pro30m2.kl1@1] Compile Module : mgtp_p
Compile Succeeded : mgtp_p

Compile File : icpsi520::>sys>user>miyuki>KL1>MGTPN>lmgtpN.kl1.23
[lmgtpNp_mlvns1Debug2_6.kl1@1] Compile Module : mgtp
Compile Succeeded : mgtp
```

```
!WARNING! register shortage in the guard 2 time(s) : master/ 19
!WARNING![lmgtpNp_mlvns1Debug2_6.kl1@343] unused predicate found
         : min/ 3
!WARNING![lmgtpNp_mlvns1Debug2_6.kl1@345] unused predicate found
         : max/ 3
"mgtp:: meta" Loaded
"mgtp:: mgtp_lib" Loaded
"mgtp:: mgtp_d" Loaded
"mgtp:: mgtp_p" Loaded
"mgtp:: tm" Loaded
"mgtp:: mgtp" Loaded
Compilation Time = 121325 [MSEC]
 6664355 reductions
 122714 msec

[2]
```

2. Execution example : (For 1 PE configuration)

```
[2] mgtp:do([{0,0}],[{0,0}],1000,10000,50,10,10,10,1,{1,1},4096,
  {3960,3980,2600},{4000,3900,4010,2500},S,0)@node(0).
distribute(500,buf:7,store:473,tested:473,eqlast:0,ls_IML:486,
  ls_ML:461,distrG:89)/0.
store(500,21,[{0,{47,47,16}},{1,{47,47,16}}]) /0.
reach(500) / 0.
distribute(1000,buf:6,store:973,tested:973,eqlast:0,ls_IML:982,
  ls_ML:956,distrG:128)/0.store(1000,20,[{0,{51,51,18}},
  {1,{67,67,13}}]) / 0.reach(1000) / 0.distribute
  (1500,buf:26,store:1464,tested:1464,eqlast:0,ls_IML:1457,
  ls _ML:1423,distrG:165)/0.
store(1500,20,[{1,{71,71,17}},{0,{84,84,10}}]) / 0.
reach(1500) / 0.
goal([{1,{138,138,19}},{0,{1797,1758,13}}],i(i(a,i(b,c)),
  i(i(a,b),i(a,c)))).
 S =[otterOrder,'unify+unifyOc',rejected,input: 5,model: 1758,
  kept: 1753,tested: 1757,keptG: 61,distr: 1797,distrG:
  183,eqlast: 0]
 65247574 reductions
 729807 msec

[3]
```

【Note】

- For PIM/m 64 PE configuration

```
mgtp:do([{0,62}],[{0,62}],1000,40,50,10,100,100,1,
  {2,2},4096,{3960,3980,2600},{4000,3900,4010,2500},
  S,63)@node(63).
```

- For PIM/m 256 PE configuration

```
mgtp:do([{1,252}],[{1,252}],1000,80,50,10,400,400,1,
  {2,2},4096,{3960,3980,2600},{4000,3900,4010,2500},
  S,253)@node(254).
```

- For PIM/p 32 PE(4 cluster) configuration

```
mgtp:do([{0,3}],[{0,3}],1000,160,50,10,50,50,8,
  {16,16},4096,{3960,3980,2600},{4000,3900,4010,2500},
  S,3)@node(3).
```

- For PIM/p 64 PE(8 cluster) configuration

```
mgtp:do([{0,7}],[{0,7}],1000,200,50,10,100,100,8,
  {16,16},4096,{3960,3980,2600},{4000,3900,4010,2500},
  S,7)@node(7).
```

- For PIM/p 128 PE(16 cluster) configuration

```
mgtp:do([{0,15}],[{0,15}],1000,200,50,10,200,200,8,
  {16,16},4096,{3960,3980,2600},{4000,3900,4010,2500},
  S,15)@node(15).
```

【Note】 To cotinue to prove other problem., it is necessary to compile the program file, then relink.

```
[16] compile(["pro280m2"]).       : Compiling next problem
** KL1 Compiler **
Compile File :
      icpsi520::>sys>user>miyuki>KL1>MGTPN>pro280m2.kl1.1
[pro280m2.kl1@1] Compile Module : mgtp_p
=>  :- module mgtp_p .
Compile Succeeded : mgtp_p

"mgtp:: mgtp_p" Updated
Compilation(s) Succeeded
Compilation Time = 16192 [MSEC]
 244554 reductions
 17024 msec

[17] relink(mgtp,[mgtp_p]) .      : Relink
"mgtp:: mgtp" Updated
Relink Succeeded
 22115 reductions
 2159 msec

[18]
```

5.3.3 KL1version (model distributed) operation

5.3.3.1 Necessary files

File name	File contents
`mgtpv0.kl1`	interpreter body
`util.kl1`	utility

5.3.3.2 Example of compiling, execution, operation

This section describes how to start and operate the model distributed MGTP/N on PIM/m.

1. Startup:

```
[1] compile(["mgtpv0","util"]).
      :Compile the source file under the directory.

[2] st.   :Set the timer.

[3] pl.   :Length of print
```

Chapter 6

CMGTP Manual

This manual describes the function and operation of the constraint MGTP (hereinafter referred to as CMGTP) which is an extended version of the model generation theorem prover MGTP, allowing it to deal with the constraint satisfaction problems in finite domains.

The CMGTP permits the representation in which negative atoms are used in the input clause. This system is also furnished with negative atom-based pruning function that provides an advantage of directly describing the constraint propagation rule by using negative atoms.

Quasigroup problems in finite algebra are one of the effective applications of CMGTP and have been a target of challenge by ICOT with the support of CMGTP.

CMGTP permits the constraint propagation, which otherwise cannot be followed with the original MGTP or constraint logic languages, making it possible to significantly reduce the search space.

On parallel machines, CMGTP can be used on PIM/m and SparcCenter, and in particular, a close-to-linear speedups was achiebed on PIM/m thanks to a deliberate parallelization scheme.

CMGTP can not only deal with quasigroup problems but can describe general constraint satisfaction problems, thus permiting experiments on constraint propagation.

For any comment or problems with this software, please contact the following.

March 3, 1995

New Generation Computer
Technology Development Organization

Y. Shirai	(shirai@icot.or.jp)
R. Hasegawa	(hasegawa@icot.or.jp)

6.1 Functions of CMGTP

6.1.1 Outline

MGTP is a bottom-up theorem prover based on the first order predicate logic. It represents a problem solver which efficiently operates on parallel machines, with the redundancy in conjunctive matching reduced to a minimum as well as a deliberate workmanship in implementation. For ease of problem description, MGTP can be viewed as a knowledge representation language based on the first order predicate logic. It can deal with Non-Horn logic expression which otherwise cannot be followed with Prolog while it guarantees the completeness which is absent in parallel logic language KL1.

MGTP can be easily described for constraint satisfaction problems in a finite domain, and has offered such an achievement to give solution to several outstanding problems with the presence of quasigroup in finite algebra through parallel.

Such achievement with MGTP has triggered the research world-wide with various approaches addressing to quasigroup problems. As a result, what has been disclosed is that while problems can be easily described with MGTP, the information important for pruning cannot be propagated and, instead, a large amount of redundant branches has to be searched for. Based on these findings, we developed the constraint MGTP (CMGTP), an improved version of MGTP in order to solve constraint satisfaction problems.

The CMGTP permits MGTP of representation with negative atoms, making it possible to describe the negative propagation rules whereby candidates can be deleted in advance under the unit simplification rule. The result is that the CMGTP can reduce the efforts in search for redundant branches to a minimum in quasigroup problems and achieve a favorable result by parallelization in terms of efficiency.

6.1.2 Quasigroup problems

For the importance of quasigroup problems which are one of the most useful applications of the CMGTP, and for convenience of explaining the constraint propagation function of the CMGTP, this section briefly describes quasigroup problems[B89]. For details of quasigroup problems and approaches undertaken, see [SH95].

6.2 Definition of Quasigroup(QG) Problems

Quasigroup

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

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

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

$\circ$	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 6.1: Latin square (order 5)

Quasigroup Problems

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 the original quasigroup constraints. QG problems are existence problems in Latin squares which satisfy some specific constraints. The additional constraints for each QG problem are shown below:

QG1 : $\forall abcd \in Q.\ (ab = cd \land a \circ_{321} b = c \circ_{321} d) \Rightarrow (a = b \land c = d)$
QG2 : $\forall abcd \in Q.\ (ab = cd \land a \circ_{312} b = c \circ_{312} d) \Rightarrow (a = b \land c = d)$
QG3 : $(\forall abcd \in Q.\ (ab)(ba) = a) \land$
$(\forall abcd \in Q.\ (ab = cd \land a \circ_{213} b = c \circ_{213} d) \Rightarrow (a = b \land c = d))$
QG4 : $\forall ab \in Q.\ (ab)(ba) = b$
QG5 : $\forall ab \in Q.\ ((ba)b)b = a$
QG6 : $\forall ab \in Q.\ (ab)b = a(ab)$
QG7 : $\forall ab \in Q.\ a(ba) = (ba)b$

Since the Latin square shown in Fig.?? satisfies the condition of QG5, QG5 has at least one idempotent solution for an order of 5.

There are still many open QG problems; for example, QG5 of order n ($n \geq 17$) has never been solved.

6.2.1 Constraint MGTP

6.2.2 Key Features of CMGTP

MGTP is a full first-order theorem prover [1] based on the model generation method[MB88]. One merit of solving QG problems by MGTP is that they can be described in very short first-order forms. This enables concise problem descriptions. In the case of QG5, MGTP only requires seven input clauses. However, MGTP has the demerit that it cannot propagate negative constraints since it is based on forward reasoning and only uses positive atoms.

[1]Although MGTP imposes a condition called *range-restrictedness* to a clause set, any first order predicate can be made range-restricted by introducing the *dom* predicate.

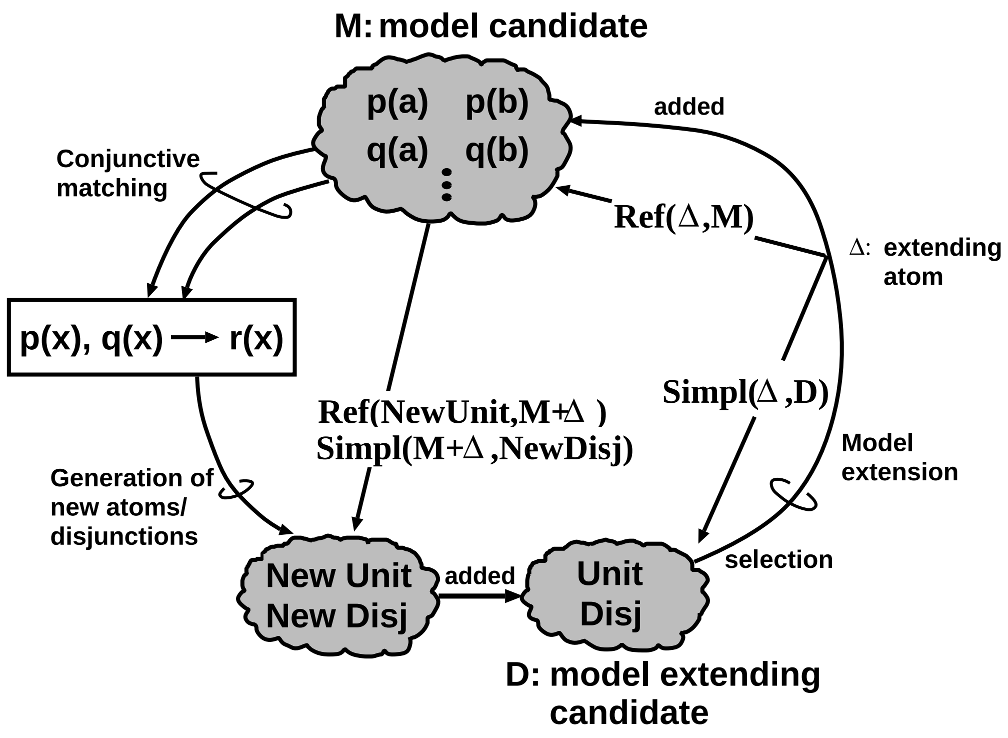

Figure 6.2: CMGTP model generation processes

To overcome this, we developed CMGTP (Constraint MGTP) in SICStus Prolog with a slight modification to the original MGTP. CMGTP introduces the following key features:

- negative atoms can be used to represent negative constraints propagation,
- the integrity constraint (unit refutation), $A, \neg A \rightarrow false$ is introduced,
- unit simplification is performed between atoms in the model candidate set and disjunctions in the model-extending candidate set.

6.2.3 The Algorithm of CMGTP

Fig.6.2 shows the CMGTP model generation processes, whose structure is basically the same as MGTP. We first note the algorithm of MGTP briefly.

In Fig.6.2, MGTP input clauses are represented in an implicational form:

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

where A_i $(1 \leq i \leq n)$ and C_j $(1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, \ldots, A_n$; and the consequent is a disjunction of $C_1, \ldots, C_m$.

Model Extension Rule: If there is a clause $\mathcal{A} \rightarrow \mathcal{C}$ and a substitution σ such that $\mathcal{A}\sigma$ is satisfied in the *model candidate set* M, then $\mathcal{C}\sigma$ (which corresponds to NewUnit if $\mathcal{C}\sigma$ is an atom, or to NewDisj if $\mathcal{C}\sigma$ is a disjunction in Fig.6.2) is added to the *model extending candidate set* D . We call the matching process between the antecedent and elements in M, *conjunctive matching.* M retains atoms that have been used in model extension. D is a set of model extending candidates which are generated as a result of application of the model extension rule.

Δ is called an *extending candidate*, which is selected from D according to some criterion (basically we use the unit preference strategy). If an atom is selected as Δ, that atom is added to M unless it is subsumed in M. Conjunctive matching is performed using Δ and M for the next model extension. If a disjunction $C_1; \ldots; C_m$ is selected as Δ, MGTP performs case-splitting unless some C_i is subsumed in M, creating m new branches : $\langle M, D \cup \{C_1\}\rangle, \ldots, \langle M, D \cup \{C_m\}\rangle$.

If $false$ is derived for a branch $\langle M, D\rangle$, that branch is terminated with *unsat*. If D becomes empty, that branch is terminated with *sat* and M is returned as a model. The detailed algorithm of the original MGTP is described in [FH91].

The differences between CMGTP and MGTP lie in the unit refutation processes and the unit simplification processes. If there exist A and $\neg A$ in M then $false$ is derived by the unit refutation mechanism as follows:

$$\frac{\begin{array}{cc} (\mathrm{M}) & (\mathrm{M}) \\ \vdots & \vdots \\ \neg A & A \end{array}}{false}$$

If for an atom $\neg A \in M (A \in M)$, there exists a disjunction which includes $A(\neg A)$, then $A(\neg A)$ is removed from that disjunction by the unit simplification mechanism as follows:

$$\frac{\begin{array}{cc} (\mathrm{M}) & (\mathrm{D}) \\ \vdots & \vdots \\ \neg A & \mathcal{D}_1 \vee A \vee \mathcal{D}_2 \end{array}}{\mathcal{D}_1 \vee \mathcal{D}_2} \qquad \frac{\begin{array}{cc} (\mathrm{M}) & (\mathrm{D}) \\ \vdots & \vdots \\ A & \mathcal{D}_1 \vee \neg A \vee \mathcal{D}_2 \end{array}}{\mathcal{D}_1 \vee \mathcal{D}_2}$$

If a disjunction becomes empty as a result of unit simplification, then $false$ is derived.

In Fig.6.2, $Ref(U_1, U_2)$ and $Simpl(U, D)$ are the functions described above, where U_1, U_2, U, are sets of atoms and D is a set of disjunctions. $Ref(U_1, U_2)$ returns $false$ if there exist $A \in U_1, B \in U_2$, s.t., A and B are complementary. In this case, the branch is terminated with *unsat*. $Simpl(U, D)$ returns the simplified set of disjunctions by a set of atoms U. If $false$ is derived as a result of simplification, then *Simpl* returns $false$, and the branch is terminated with *unsat*.

There are 2 refutation processes and 2 simplification processes added to the original MGTP:

- $Ref(\{\Delta\}, M)$
- $Ref(NewUnit, M \cup \{\Delta\})$
- $Simpl(\{\Delta\}, D)$

- $Simpl(M \cup \{\Delta\}, NewDisj)$

where Δ is an atom. As a result, these functions guarantee that for any atom $A \in M$, A and $\neg A$ are not both in the current M, and disjunctions in the current D have already been simplified by all atoms in M.

CMGTP Rules for QG5.5

Fig.6.3 shows the CMGTP input clauses for QG5.5. In this figure, (M1) defines the domain of variables, (M2) corresponds to the idempotence condition, (M3)-(M5) are for generating disjunctions, (M6)-(M8) represent the distinctness property, (M9) is a heuristic rule in order to remove isomorphic proofs, and (M10)-(M18) are the constraint propagation rules for QG5. As shown here, constraint propagation rules in Fig.?? can be written directly as CMGTP input clauses.

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

$$\begin{array}{l} p(Y,X,A), p(A,Y,B) \rightarrow p(B,Y,X). \\ p(Y,X,A), \neg p(B,Y,X) \rightarrow \neg p(A,Y,B). \\ \neg p(B,Y,X), p(A,Y,B) \rightarrow \neg p(Y,X,A). \end{array}$$

where negative information is propagated by using the last 2 rules.

In the representation of Fig.6.3, we can write the integrity constraint $P, \neg P \rightarrow false$ as an input clause instead of using the built-in Ref function. In this case, we do not need the function Ref.

6.2.4 Parallelization scheme

MGTP has a structure ideally suited to parallelization, and has been achieving a desirable quantity effect in both AND and OR parallelizations [FHKF92]. The CMGTP is basically in accordance with MGTP algorithm and is expected to have a favorable effect on parallelization. Since quasigroup problems relate to Non-Horn problems where many chances of case-splitting are expected, the CMGTP will promise a strong impact on more effective parallelization. The following paragraphs explain the method for parallelization in the CMGTP.

As shown in Figure 6.4, the proof tree of the CMGTP constitutes a tree structure by OR branches. However, because the length of a branch is unforeseen and the number of case-splitting is undefined in quasigroup problems, their tree structure is not a balanced tree such as in pigeon hole problems, but assumes an inbalanced tree.

When case-splitting takes place at a node in the tree in parallel execution, one branch is processed by its own processor while other branches are submitted to other processors. At this time, assuming the branches inherited from the parent node are single job, it results in

$$\begin{array}{ll}
true \rightarrow dom(1), dom(2), dom(3), dom(4), dom(5). & (M1) \\
true \rightarrow p(1,1,1), p(2,2,2), p(3,3,3), p(4,4,4), p(5,5,5). & (M2) \\
dom(M), dom(N), \{M \backslash= N\} \rightarrow & \\
\quad p(M,N,1); p(M,N,2); p(M,N,3); p(M,N,4); p(M,N,5). & (M3) \\
dom(M), dom(N), \{M \backslash= N\} \rightarrow & \\
\quad p(M,1,N); p(M,2,N); p(M,3,N); p(M,4,N); p(M,5,N). & (M4) \\
dom(M), dom(N), \{M \backslash= N\} \rightarrow & \\
\quad p(1,M,N); p(2,M,N); p(3,M,N); p(4,M,N); p(5,M,N). & (M5) \\
p(M,N,X), dom(M1), \{M1 \backslash= M\} \rightarrow \neg p(M1,N,X). & (M6) \\
p(M,N,X), dom(N1), \{N1 \backslash= N\} \rightarrow \neg p(M,N1,X). & (M7) \\
p(M,N,X), dom(X1), \{X1 \backslash= X\} \rightarrow \neg p(M,N,X1). & (M8) \\
dom(X), dom(Y), \{X1\ is\ X-1, Y < X1\} \rightarrow \neg p(X,5,Y). & (M9) \\
p(Y,X,A), p(A,Y,B) \rightarrow p(B,Y,X). & (M10) \\
p(Y,X,A), p(B,Y,X) \rightarrow p(A,Y,B). & (M11) \\
p(A,Y,B), p(B,Y,X) \rightarrow p(Y,X,A). & (M12) \\
p(Y,X,A), \neg p(B,Y,X) \rightarrow \neg p(A,Y,B). & (M13) \\
p(Y,X,A), \neg p(A,Y,B) \rightarrow \neg p(B,Y,X). & (M14) \\
\neg p(Y,X,A), p(B,Y,X) \rightarrow \neg p(A,Y,B). & (M15) \\
\neg p(B,Y,X), p(A,Y,B) \rightarrow \neg p(Y,X,A). & (M16) \\
\neg p(A,Y,B), p(B,Y,X) \rightarrow \neg p(Y,X,A). & (M17) \\
p(A,Y,B), \neg p(Y,X,A) \rightarrow \neg p(B,Y,X). & (M18)
\end{array}$$

Figure 6.3: CMGTP rules for QG5.5

the presence of a total of 12 jobs in the example shown in Figure 6.4. In general, jobs are diverse in length. For example, `Job1` is a branch generated at an extremely shallow level and the job is large in length, whereas, `Job8` or `Job12` is a branch generated at a level deeper than the foregoing, and is shorter than `Job1`.

The scheme for parallelization may be transposed to the question as to how to allocate such a job to the processor. We launched an experiment conducted in three different methods that follow.

- Cyclic allocation scheme
- Probabilistic allocation scheme
- Limited branch scheme

In either scheme, however, a branch generated at a shallow level has a strong likelihood of getting longer in its own length and has a high probability of producing many branches. To cope with these constraints, the branches generated at shallower levels are given the high priority over the branches generated at deeper levels for processing in each processor. The following paragraphs illustrate how each allocation scheme works.

Cyclic allocation scheme

With this scheme, all of the newly generated jobs are allocated in a recyclic manner. In other words, when branching takes place in a processor n, one branch takes care of itself and other branches, namely new branches, are allocated in the order beginning with processor $n+1$.

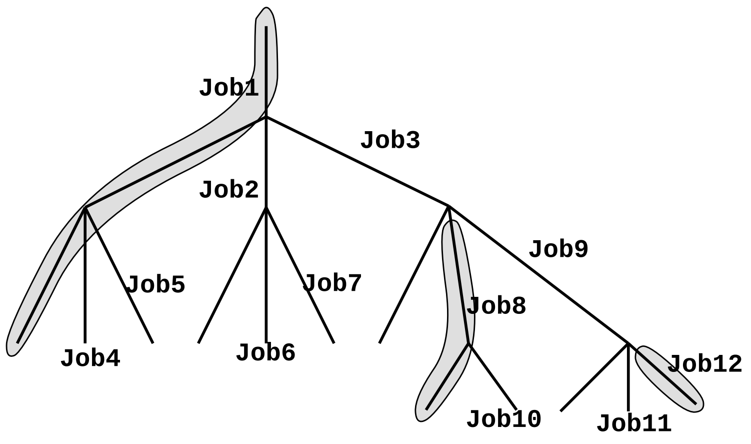

Figure 6.4: Proof tree of CMGTP

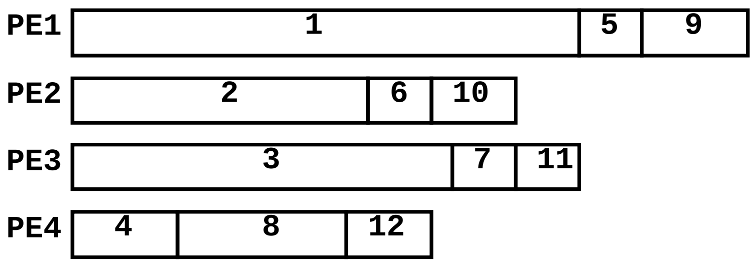

Figure 6.5: Cyclic allocation scheme

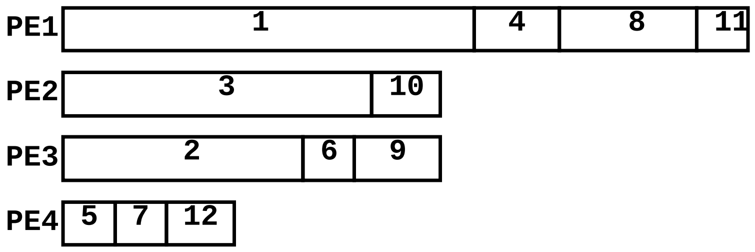

Figure 6.6: Probabilistic allocation scheme

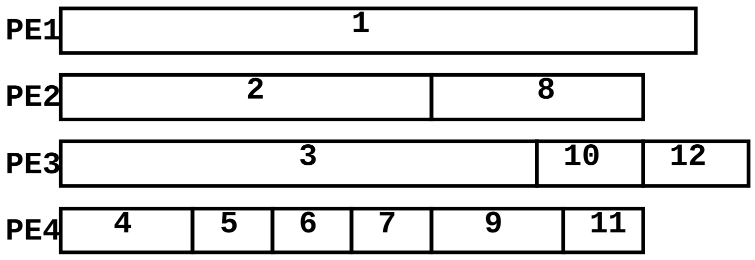

Figure 6.7: Limited branch scheme ($Limit = 1$)

The outcome is that the number of jobs entrusted to each processor is equal to each other upon completion of execution. Figure 6.5 shows an example of allocating processors with the cyclic scheme of the proof tree shown in Figure 6.4. Assume that there are four processors (`PE1,PE2,PE3,PE4`), and `PE1` is responsible for `Job1`, then `Job2, Job3, ...` are allocated to `PE2, PE3, ...` in that order.

In the proof tree in Figure 6.4, a total of 12 jobs are generated, and in the long run, each of four processors is supposed to process three branches. With an eye to each moment during execution, however, the number of branches entrusted to each processor is undefined. At the moment `Job10` ends up with `PE2`, `PE2` becomes jobless and stalls in idle status while `PE1` is still filled with three branches. In this example, because `Job1` branch is large in length, eventually, the execution runtime spent in `PE1` determines the overall execution runtime.

Probabilistic allocation scheme

With this scheme, the processors to which newly generated jobs are allocated are determined on a probability basis. Figure 6.6 shows the allocation of processors with the probabilistic scheme of the proof tree shown in Figure 6.4. Practically, this scheme poses the problem similar to what is experienced in the above cyclic allocation. That is, if a processor is assigned a long branch, the processor requires execution time longer than required by other processor, thus increasing the overall execution time. This problem stems from failure in monitoring the load imposed on each processor in execution. With the limited branch scheme mentioned below, the number of jobs allocated to each processor at each moment is monitored by the master processor and new jobs are allocated to the most idle processor.

Limited branch scheme

With this scheme, when a new job is generated, a new processor is selected so that the number of jobs allocated to each job is made equal to each other. In this way, the master processor looks about the number of jobs allocated to each processor, and allocates new jobs to the processor whose allocated number of jobs is the least of the rest of the processors. If there is a processor with a longer branch, the number of branches in process decreases in timid steps, and the processor is no longer assigned additional jobs.

The limited branch scheme provides a parameter named *Limit* whose value can be changed in order to conduct an experiment. This parameter indicates the number of branches each processor can hold at a time. If $Limit = 1$, each processor cannot process two or more jobs in parallel, so that when a new branch is generated, it must always be allocated to a processor currently in idle status. If all processors are busy (that is, in a condition filled with the jobs equivalent of the value of *Limit*), the new job is suspended until an idle processor appears.

Figure 6.7 shows an example of allocation of processors in the case of $Limit = 1$. In general, setting *Limit* to a small value results in the reduced number of jobs allocated to each processor at the same time, thus contributing to memory saving as well as making it possible to allocate optimum jobs.

Conversely, should *Limit* value be smaller and the granularity of jobs relatively small with frequent branching, a large number of suspended jobs should occur and the master processor can no longer accommodate for job distribution (Mater Bottleneck status) so that a number of processors waiting for job is expected to appear.

6.2.5 Discussion on parallel execution

This section briefly illustrates the effect of parallelization in the CMGTP by means of an experiment conducted at ICOT.

Of the three allocation schemes mentioned above, the limited branch scheme exhibits the best achievement with a high quantity effect. Figures 6.8, 6.9 and 6.10 show the run-time monitor upon termination of execution for each scheme. In the cyclic allocation or probabilistic allocation, particularly in the latter half of the allocation, an extremity of difference exists in the load distribution among the processors, leading to longer execution time. As mentioned earlier, jobs are allocated without allowance for the load of jobs imposed on each processor. As a result, the processors responsible for longer jobs are supposed to determine an overall execution time. Conversely, the limited branch scheme allows the load to be distributed almost uniformly for each processor and they can exit execution at almost the same time.

Figure 6.11 shows a graph illustrating the quantity effect with each scheme. The ratio of speedups is shown on a run time basis spent by 62 machines. It is apparent that with the limited branch scheme, an execution with 256 machines, accounting for four times the number of processors executing a 62-PEs run, exhibits 3.97 times the execution time (if $Limit = 1$) of the foregoing, achieving a close-to-linear speedup.

On the other hand, the cyclic or probabilistic allocation scheme is not expected to achieve a speedup effect equivalent of the number of processions/PEs even if increased.

6.2.6 Conclusion

It was revealed that the bottom-up theorem prover MGTP furnished with problem representation capability in the first order predicate logic and a high-speed inference function is also applicable to constraint propagation problems in a finite domain such as quasigroup problems, with the penalty of insufficient pruning because negative constraint propagation cannot be carried out. We thought such disadvantage of the MGTP revealed when applied to normal constraint satisfaction problems stems from its inability to handle negative atoms. and developed the constraint MGTP (CMGTP) system on SICStus Prolog that can handle negative atoms, and is also equipped with unit simplification mechanism and unit refutation mechanism.

The CMGTP has pruning performance equivalent of DDPP or the CP we developed based on constraint logic languages, and represents a system inheriting the convenience of problem description in the first order predicate logic form inherent in MGTP.

With the CMGTP, the intuitive constraint propagation rules can be directly described as the input clauses of the CMGTP. This poses a fair contrast to DDPP where about 1,000,000 clauses in QG5 order 10 is required (as well as requiring a separate translator used to generate those clauses).

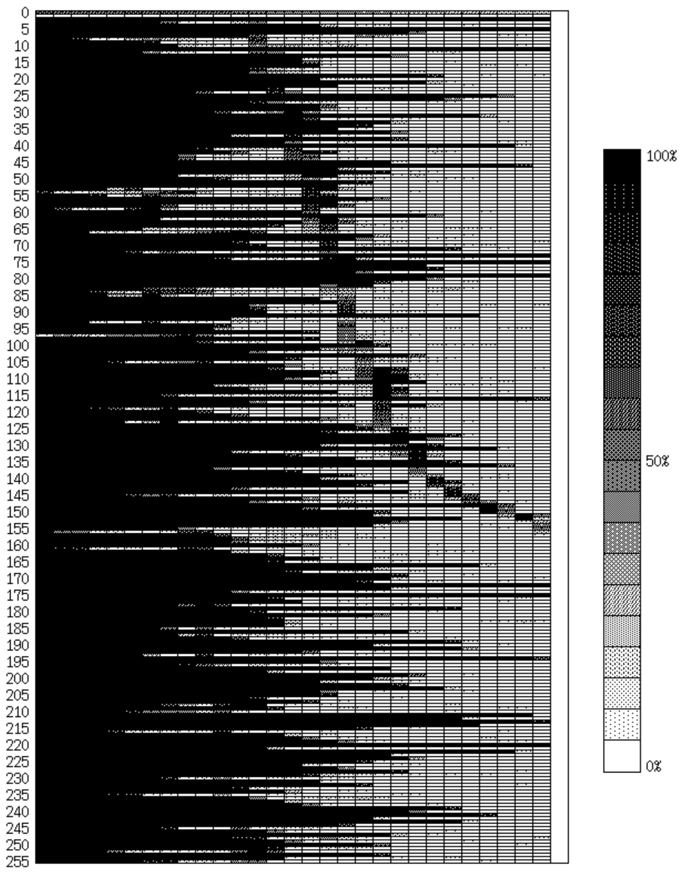

Figure 6.8: Execution monitor in cyclic allocation (upon termination)

For parallelization, a series of experiments was conducted for each of the cyclic allocation, probabilistic allocation and limited branch schemes. For quasigroup problems, we confirmed that the limited branch scheme is the best of all. Quantity effect also exhibited a close-to-linear achievement.

The CMGTP is blessed with excellent description performance as to constraint propagation, and exhibits an enhanced throughput thanks to the ongoing efforts in parallelization. Such achievements may promise ever wide-spread applications of the CMGTP to serve as a solver of general finite domain constraint satisfaction problems in addition to quasigroup problems.

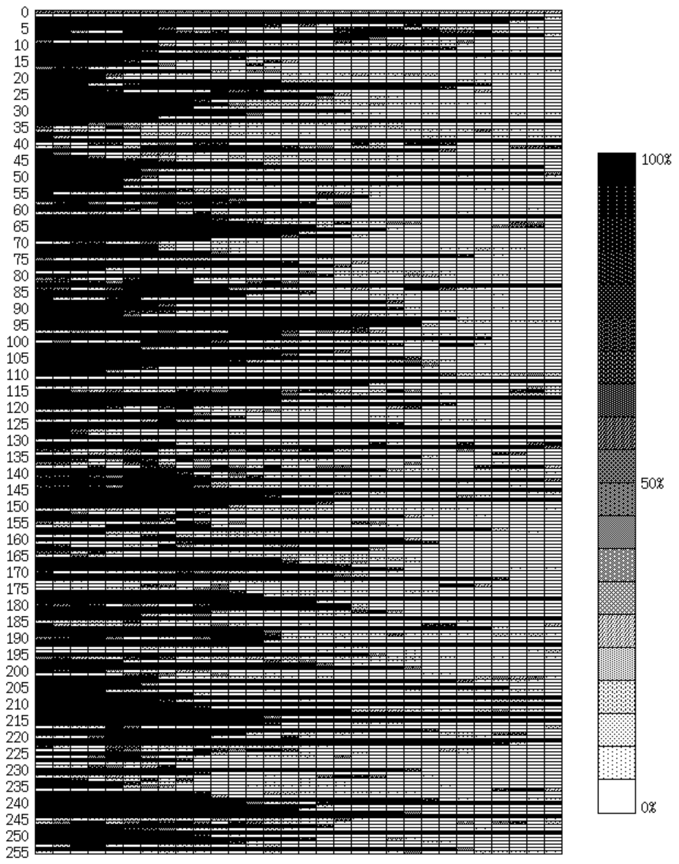

Figure 6.9: Execution monitor in probabilistic allocation(upon termination)

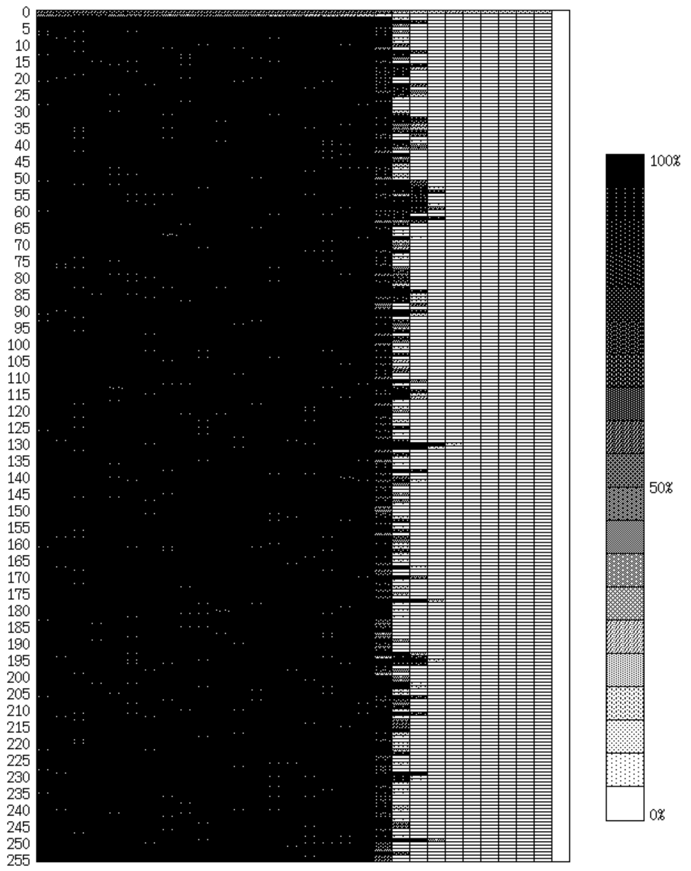

Figure 6.10: Execution monitor with limited branch scheme ($Limit = 1$) (upon termination)

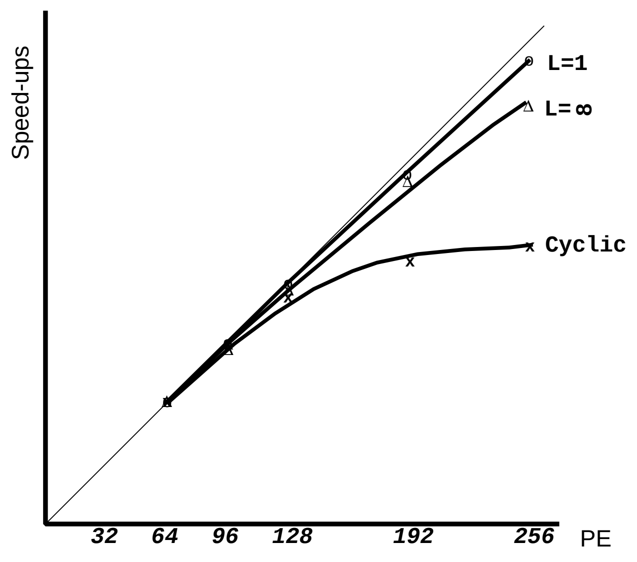

Figure 6.11: Quantity effect

Bibliography

[FH91] H. Fujita and R. Hasegawa, *A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm*, In Proc. of the Eighth International Conference on Logic Programming, The MIT Press, 1991.

[FHKF92] M. Fujita, R. Hasegawa, M.Koshimura and H. Fujita, *Model Generation Theorem Provers on a Parallel Inference Machine*, In Proc. of FGCS92, 1992.

[B89] F. Bennett, *Quasigroup Identities and Mendelsohn Designs*, In Canadian Journal of Mathematics 41, pp.341-368, 1989.

[CS89] P. V. Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press,1989.

[S92] J. Slaney, *FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guides*, Technical report TR-ARP-1/92, Automated Reasoning Project, Australian National University, 1992.

[S94] J. Slaney, *FINDER, Finite Domain Enumerator System Description*, In Pro. of CADE-12, Nancy, France, 1994.

[MB88] R. Manthey and F. Bry, *SATCHMO: a theorem prover implemented in Prolog*, In Proc. of CADE-9, Argonne, Illinois, 1988.

[SFS93] J.Slaney, M.Fujita, and M.Stickel, *Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems*, To appear in Computers and Mathematics with Applications.

[FSB93] M. Fujita, J. Slaney, and F. Bennett, *Automatic Generation of Some Results in Finite Algebra*, In Proc. of International Joint Conference on Artificial Intelligence, 1993.

[H94] R. Hähnle, *MGTP with an extended constraint language*, private communication, 1994.

[FK94] M. Fujita, F. Kumeko, (Masayuki Fujita, Fumihiro Kumeno,) Discovery of new findings on finite algebra with MGTP, Journal of Information Processing Institute Vol 35, No 7 (pp1282-1292), 1994 (in Japanese).

[HS94] R. Hasegawa and Y. Shirai, *Constraint Propagation of CP and CMGTP: Experiments on Quasigroup Problems*, Workshop 1C (Automated Reasoning in Algebra), CADE-12, 1994.

[SH95] Y. Shirai and R. Hasegawa, *Two Approaches for Finite-Domain Constraint Satisfaction Problems*, In Proc. of 12th International Conference on Logic Programming, 1995. (to appear)

6.3 Prolog version operation manual

6.3.1 Installation

The tar file in CMGTP Prolog version contains the following files.

`mgtp4.pl`	CMGTP engine
`mgpl4.pl`	Translator
`QG5-7.mg4`	Problem sample file

`mgtp4.pl` is a CMGTP engine, or a program which reads the given input clauses and repeats the model generation process. `mgpl4.pl` is a translator used to translate user-written input clauses into Prolog program. For details of translator, see the function's manual of the applicable translator. As similar to a translator in other versions, the translator of CMGTP Prolog version is designed to generate a program of term memory. `QG5-7.mg4` is a typical sample problem and shows an example of QG5 order 7 on quasigroup proglems. See details in Chapter 1, "CMGTP Functions" for quasigroup proglems. This program was developed on SICStus Prolog and operates on SICStus Prolog systems.

6.3.2 Compiling translator

Because the translator is written in SICStus Prolog, it is necessary to start SICStus Prolog first in order to create the execution object.

```
$ sicstus
SICStus 2.1 #8: Tue May 11 21:04:52 JST 1993
| ?-
```

Then compile and save `mgpl4`.

```
| ?- compile(mgpl4).
| ?- save_mgpl(mgpl4).
```

As a result, the compiled executable object `mgpl4` is created.

6.3.3 Translation of problems to KL1 program

To translate the problem file into a Prolog program which permits reference by the CMGTP engine, execute the following.

```
$ mgpl4 <problem file>
```

The problem must have the file named `<problem file>.mg`. To specify the above `<problem file>`, no extension is required. Translating the file into a Prolog program by means of the translator creates a Prolog file named
`<problem file>_mg4.pl`.

6.3.4 Creation of CMGTP executable object

Then create the CMGTP executable object. The executable object can be obtained by compiling the program `mgtp4.pl` on SICStus Prolog and saving.

```
$ sicstus
SICStus 2.1 #8: Tue May 11 21:04:52 JST 1993
| ?- compile(mgtp4).
| ?- save_mgtp(mgtp4).
```

As a result, the executable object `mgtp4` for CMGTP is created.

6.3.5 Execution

Execution proceeds as follows.

```
$ mgtp4 <problem file>
```

The following example shows how to translate QG5-7.mg and execute.

```
mgpl4 QG5-7
mgtp4 QG5-7
```

6.3.6 Result of executable

During executable, you get comments when case-splitting arises or a model is found, or when a branch fails. Upon completion of executable, total number of models and branches, and executable time are displayed. The result of executing CMGTP for QG5-7.mg is shown below.

```
$ mgtp4 QG5-7

{compiling QG5-7_mg4.pl...}
{QG5-7_mg4.pl compiled, 1730 msec 45904 bytes}
:(1,dom(1))
:(2,dom(2))
:(3,dom(3))
:(4,dom(4))
:(5,dom(5))
:(6,dom(6))
:(7,dom(7))
:(8,p(1,1,1))
:(9,p(2,2,2))
:(10,p(3,3,3))
:(11,p(4,4,4))
  ......
```

```
:(89,not(p(7,4,4)))
:(215,not(p(6,4,7)))
:(216,not(p(4,6,7)))
--> p(1,7,6)
:(217,p(1,7,6))
:(218,not(p(4,7,6)))
  ......

:(266,not(p(6,4,2)))
--> p(1,6,7)
:(267,p(1,6,7))
  ......

:(317,p(2,4,3))
:(318,not(p(2,5,3)))
Failed Branch := 1
--> p(3,6,7)
:(267,p(3,6,7))
  ......

:(303,not(p(3,2,6)))
Failed Branch := 2
--> p(4,7,6)
:(217,p(4,7,6))
  ......

:(260,not(p(6,2,1)))
--> p(1,6,7)
:(261,p(1,6,7))
  ......

:(350,p(1,2,4))
model case:2.1:size(350).
--> p(2,6,7)
:(261,p(2,6,7))
  ......

:(350,p(2,3,4))
model case:2.2:size(350).
--> p(3,6,7)
:(261,p(3,6,7))
  ......

:(350,p(3,1,4))
model case:2.3:size(350).
====================================
satisfiable.
```

```
- - - - - - - - - - - - - - - - - - -
Number of Models   : 3
Failed Branches    : 2
Execution Runtime : 9.541 seconds.
- - - - - - - - - - - - - - - - - - -
$
```

On the released versions, all models generated are displayed and, in particular, the generation of many negative models are apparent in CMGTP. If you don't need the information w.r.t. negative models, the user can comment out the portion in which models are displayed in the program. `-->` indicates case-splitting takes place, and `Failed Branch` is displayed when a branch fails.

When a model is found, information such as `model case:2.1:size(350)` is displayed. If you need to see a model, it is easier for the user to write the model output routine to display a model i na table form in the program. Upon termination of the program, the number of generated models, the number of failed models and the executable runtime are displayed.

6.4 KLIC version operation manual

6.4.1 Installation

tar file of CMGTP includes the following files.

`cmgtp.kl1`	CMGTP engine
`mg2kl1.pl`	translator
`QG5-8.mg`	problem sample file

`cmgtp.kl1` is the CMGTP engine and represents a program which reads the given input clauses and repeats the model generation process.

`mg2kl1.pl` is a translator used to translate user written input clauses into KL1 program. For details of the translator, see the Function Manual for the translator. As similar to the translator in other versions, the translator in CMGTP KLIC version is designed to generate the program of the term memory.

As earlier mentioned in the section for KL1 version, no translator is currently available to KL1 version, so that the translator used for this KLIC version is used.

`QG5-8.mg` is a typical sample problem and shows an example of QG5 order 8 on quasigroup problems. See details in Chapter 1, "Functins of CMGTP" for quasigroup problems.

6.4.2 Compiling the translator

Since the translator is written in SICStus Prolog, it is necessary to activate SICStus Prolog first to generate the executable object.

```
$ sicstus
SICStus 2.1 #8: Tue May 11 21:04:52 JST 1993
| ?-
```

Then compile and save `mg2klic`.

```
| ?- compile(mg2kl1).
| ?- save.
```

As a result, the compiled executable object `mg2kl1` is generated.

6.4.3 Translation of problems to KL1 program

To translate the problem file into a KL1 program which permits reference by CMGTP engine, execute the following.

```
$ mg2kl1 <problem file> merc
```

Where `merc` specifies the version name which should always be specified for the currently released version. The problem must have the file name of `<problem file>.mg`. When specifying the above `<problem file>`, no extension is required.

A KL1 file named
`<problem file>_merc.kl1` is created by translation to the KL1 program with the translator.

6.4.4 Compiling

Then compile and link the CMGTP engine and the problem by KLIC.

```
$ klic cmgtp.kl1 <problem file>_merc.kl1
```

Compile option may be specified as necessary. For details, see the KLIC manual. As a result of compiling, the executable object `a.out` is generated.

6.4.5 Execution

Execution proceeds with the following.

```
$ a.out
```

For options to be specified for execution, see the KLIC manual.

6.4.6 Result of execution

During execution, you can get comments when case-splitting arises or a model is found, or when branch fails. Upon completion of execution, total number of models and branches, and execution time are displayed. The result of executing CMGTP for QG5-8.mg is shown below.

Of the failed branches, `failed_branch_by_Fsimpl` indicates a failure because a certain disjunction becomes to be empty due to unit simplification, while `failed_branch_by_Fref` indicates a failure due to unit refutation.

```
[p(1,8,3),p(4,8,3)]
--> p(1,8,3)
[p(4,1,8),p(5,1,8),p(6,1,8),p(7,1,8)]
--> p(4,1,8)
failed_branch_by_Fref
--> p(5,1,8)
failed_branch_by_Fref
--> p(6,1,8)
failed_branch_by_Fref
--> p(7,1,8)
failed_branch_by_Fref
--> p(4,8,3)
[p(1,8,4),p(5,8,4)]
--> p(1,8,4)
[p(2,1,8),p(5,1,8),p(6,1,8),p(7,1,8)]
--> p(2,1,8)
failed_branch_by_Fsimpl
--> p(5,1,8)
failed_branch_by_Fref
--> p(6,1,8)
```

```
failed_branch_by_Fref
--> p(7,1,8)
failed_branch_by_Fref
--> p(5,8,4)
[p(1,8,5),p(6,8,5)]
--> p(1,8,5)
failed_branch_by_Fsimpl
--> p(6,8,5)
[p(2,1,8),p(4,1,8)]
--> p(2,1,8)
failed_branch_by_Fsimpl
--> p(4,1,8)
model found

heap size = 1048576 words
43950 ms total; 43570 user; 380 system
  0 swaps; 34 minor page faults; 0 major page faults
  0 block inputs; 0 block outputs
  1226 context switches (0 voluntary)
  122 GC
  251 suspensions; 250 resumptions
sat
************************************
No. of Failed Branches  = 10
No. of Models           = 1
************************************
```

6.5 KL1 version operation manual

6.5.1 Feature of KL1 version

KL1 version assumes parallel execution on PIM/m as a premise to release only the program used for quasigroup problems with parallel vervion.

The translator is the same as used in KLIC, but KL1 translator is not available to CMGTP. As a workaround, the difference of the built-in functin between KLIC and KL1 for generated programs should be corrected with manual input.

Because availability of CMGTP is expected to shift from PIM to parallel UNIX machines, future support for KL1 version is no longer available.

Figure 6.12 lists the files included in the current release.

Problem files include QG1 order 6 through 9, QG2 order 7 through 9, QG5 order 7 through 16, and QG7 order 11 through 11 respectively.

CMGTP engine is available in cyclic allocation version and probabilistic allocation version depending on the diffrence of parallelization scheme. And limited branch allocation version is provided with *Limit* $= 1, 2, 3, 5, 10, \infty$. The setting location of parameter *Limit* in the limited branch allocation version is easier to locate, so that preferable parameters can be executed by replacing the target part with an optional number you like.

6.5.2 Execution on PIM/m

This section gives a practical way to solve quasigroup problems with CMGTP on PIM.

The user may open the linster and set a preferable environment such as for monitor display and output of statistics information. In the following example, the user displays xrmonitor on the screen and specifies the mode to output the statistics information.

```
Shell> listener

Listener> setenv pmeter:display_node = ss145 .
Listener> xrmonitor & .
Listener> st.
Listener>
```

Then, load the GC program for MGTP and specify 70 in level. This means that when memory consumption of a processor exceeds 70 %, GC is globally invoked.

```
Listener> load("gc.sav").
Listener> mgtp_gc:allocate(70) & .
Listener>
```

In compiling the program, the problem and CMGTP engine should be compiled at the same time. Or each of compiled objects should be linked.

Problem file :

`QG1-6_index.kl1`	QG1 order 6
`QG1-7_index.kl1`	QG1 order 7
`QG1-8_index.kl1`	QG1 order 8
`QG1-9_index.kl1`	QG1 order 9
`QG2-7_index.kl1`	QG2 order 7
`QG2-8_index.kl1`	QG2 order 8
`QG2-9_index.kl1`	QG2 order 9
`QG5-7_index.kl1`	QG5 order 7
`QG5-8_index.kl1`	QG5 order 8
`QG5-9_index.kl1`	QG5 order 9
`QG5-10_index.kl1`	QG5 order 10
`QG5-11_index.kl1`	QG5 order 11
`QG5-12_index.kl1`	QG5 order 12
`QG5-13_index.kl1`	QG5 order 13
`QG5-14_index.kl1`	QG5 order 14
`QG5-15_index.kl1`	QG5 order 15
`QG5-16_index.kl1`	QG5 order 16
`QG7-11_index.kl1`	QG7 order 11
`QG7-12_index.kl1`	QG7 order 12
`QG7-13_index.kl1`	QG7 order 13
`QG7-14_index.kl1`	QG7 order 14

CMGTP engine :

`cmgtpCycle.kl1`	cyclic allocation scheme
`cmgtpProb.kl1`	probabilistic allocation scheme
`cmgtpLimit1.kl1`	limit branch allocation scheme ($Limit = 1$)
`cmgtpLimit2.kl1`	limit branch allocation scheme ($Limit = 2$)
`cmgtpLimit3.kl1`	limit branch allocation scheme ($Limit = 3$)
`cmgtpLimit5.kl1`	limit branch allocation scheme ($Limit = 5$)
`cmgtpLimit10.kl1`	limit branch allocation scheme ($Limit = 10$)
`cmgtpLimitInf.kl1`	limit branch allocation scheme ($Limit = \infty$)

Utility :

`gc.sav`	GC utility for MGTP

Figure 6.12: Files included in current release

```
Listener> compile(['QG513_index.kl1','cmgtpLimit5.kl1']).
Listener>
```

Execution goes with module name `mgtp`, predicate name `do`, and the size of Latin square and number of used processors are specified in the argument. The following attempt is to solve QG5 order 13 by 256 processor according to the above example.

```
Listener> mgtp:do(13,256).
Listener>
```

Upon completion of the execution, number of branches, number of models and execution time are displayed.

```
Listener> mgtp: do(13,256) .

Number of Failed Branches = 15173
Number of Models = 0
 5498685340 reductions
 229369 msec
```

The results of this execution indicates that the number of models is 0, number of failed branches is 15,173, and execution time is about 229 seconds.