PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Theorem Provers and Their Applications

Ryuzo Hasegawa and Masayuki Fujita

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

Abstract

This paper describes the results of the research and development of automated reasoning systems(ARS) being conducted by the Fifth Research Laboratory at ICOT. The major result was the development of a parallel theorem proving system MGTP (Model Generation Theorem Prover) in KL1 on a parallel inference machine, PIM. Currently, we have two versions of MGTP. One is MGTP/G, which is used for dealing with ground models. The other is MGTP/N, used for dealing with non-ground models. With MGTP/N, we have achieved a more than one-hundred-fold speedup for condensed detachment problems on a PIM/m consisting of 128 PEs. Nonmonotonic reasoning and program synthesis are taken as promising and concrete application area for MGTP provers. MGTP/G is actually used to develop legal reasoning systems in ICOT's Seventh Research Laboratory. Advanced inference and learning systems are studied for expanding both reasoning power and application areas. Parallel logic programming techniques and utility programs such as 'meta-programming' are being developed using KL1. The technologies developed are widely used to develop applications on PIM.

1 Introduction

The final goal of the Fifth Generation Computer Systems (FGCS) project was to realize a knowledge information processing system with intelligent user interfaces and knowledge base systems on parallel inference machines. A high performance and highly parallel inference mechanism is one of the most important technologies to come out of our pursuit of this goal.

The major goal of the Fifth Research Laboratory, which is conducted as a subgoal of the problem-solving programming module of FGCS, is to build very efficient and highly parallel automated reasoning systems (ARS) as advanced inference systems on parallel inference machines (PIM), taking advantage of the KL1 language and PIMOS operating system. On ARS we intend to develop application systems such as natural language processing, intelligent knowledgebases, mathematical theorem proving systems, and automated programming systems. Furthermore, we intend to give good feedback to the language and operating systems from KL1 implementations and experiments on parallel inference hardware in the process of developing ARS.

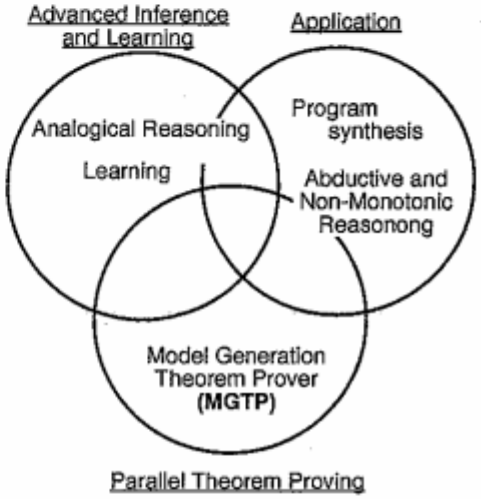

Figure 1: Goals of Automated Reasoning System Research at ICOT

We divided ARS research and development into the following three goals (Figure 1):

(1) Developing Parallel Theorem Proving Technologies on PIM
Developing very efficient parallel theorem provers on PIM by squeezing the most out of the KL1 language is the major part of this task. We have concentrated on the model generation method, whose inference mechanism is based on hyper-resolution. We decided to develop two types of model generation theorem provers to cover ground non-Horn problems and non-ground Horn problems. To achieve maximum performance on PIM, we have focused on the

technological issues below:

(a) Elimination of redundant computation
Eliminating redundant computation in the process of model generation with the least overhead is an important issue. Potential redundancy lies in conjunctive matching at hyper-resolution steps or in the case splitting of ground non-Horn problems.

(b) Saving time and space by eliminating the over generation of models
For the model generation method, which is based on hyper-resolution as a bottom-up process, over generation of models is an essential problem of time and space consumption. We regard the model generation method as generation and test procedures and have introduced a controlling mechanism called *Lazy Model Generation*.

(c) Finding PIM-fitting and scalable parallel architecture
PIM is a low communication cost MIMD machine. Our target is to find a parallel architecture for model generation provers, which draws the maximum power from PIM. We focused on OR parallel architecture to exploit parallelism in the case splitting of a ground non-Horn prover, MGTP/G, and on AND parallel architecture to exploit parallelism in conjunctive matching and subsumption tests of a non-ground Horn prover, MGTP/N.

One of the most important aims of developing theorem provers in KL1 is to draw the maximum advantage of parallel logic programming paradigms from KL1. Programming techniques developed in building theorem provers help to, or are commonly used to, develop various applications, such as natural language processing systems and knowledge base systems, on the PIM machines based on logic programming and its extension. We focused on developing meta-programming technology in KL1 as a concrete base for this aim. We think it is very useful to develop broader problem solving applications on PIM and to extend KL1 to support them.

(2) Application
A model generation theorem prover has a general reasoning power in various AI areas because it can simulate the widely applied tableaux method effectively. Building an efficient analytic tableaux prover for modal propositional logic on model generation theorem provers was the basic goal of this extension. This approach could naturally be applied to abductive reasoning in AI systems and logic programming with negation as failure linked with broader practical AI applications such as diagnosis.

We focused on automated programming as one of the major application areas for theorem provers in the non-Horn logic systems, in spite of difficulty. There has been a long history of program synthesis from specifications in formal logic. We aim to make a first-order theorem prover that will act as a strong support tool in this approach. We have set up three different ways of program construction: realizability interpretation in the constructive mathematics to generate functional programs, temporal propositional logic for protocol generation, and the Knuth-Bendix completion technique for interface design of concurrent processes in Petri Net. We stressed the experimental approach in order to make practical evaluation.

- Advanced Inference and Learning
Theorem proving technologies themselves are rather saturated in their basic mechanisms. In this subgoal, extension of the basic mechanism from deductive approach to analogical, inductive, and transformational approaches is the main research target. Machine learning technologies on logic programs and meta-usage of logic are the major technologies that we decided to apply to this task.

  By using analogical reasoning, we intended to formally simulate the intelligent guesswork that humans naturally do, so that results could be obtained even when deductive systems had no means to deduce to obtain a solution because of incomplete information or very long deductive steps.

  Taking the computational complexity of inductive reasoning into account, we elaborated the learning theories of logic programs by means of predicate invention and least-general generalization, both of which are of central interest in machine learning.

  In transformational approach, we used fold/unfold transformation operations to generate new efficient predicates in logic programming.

The following sections describe these three tasks of research on automated reasoning in ICOT's Fifth Research Laboratory for the three years of the final stage of ICOT.

2 Parallel Theorem Proving Technologies on PIM

In this section, we describe the MGTP provers which run on Multi-PSI and PIM. We present the technical essence of KL1 programming techniques and algorithms that we developed to improve the efficiency of MGTP.

2.1 Parallel Model Generation Theorem Prover MGTP

The research on parallel theorem proving systems aims at realizing highly parallel advanced inference mechanisms that are indispensable in building intelligent knowledge information systems. We started this research project on parallel theorem provers about two and a half years ago. The immediate goal of the project is to develop a parallel automated reasoning system on the parallel inference machine, PIM, based on KL1 and PIMOS technology. We aim at applying this system to various fields such as intelligent database systems, natural language processing, and automated programming.

At the beginning, we set the following as the main subjects.

- To develop very fast first-order parallel theorem provers
  As a first step for developing KL1-technology theorem provers, we adopted the model generation method on which SATCHMO is based as a main proof mechanism. Then we implemented a model-generation based theorem prover called MGTP. Our reason was that the model generation method is particularly suited to KL1 programming as explained later. Based on experiences with the development of MGTP, we constructed a "TP development support system " which provided us with useful facilities such as a proof tracer and a visualizer to see the dynamic behavior of the prover.

- To develop applications
  Although a theorem prover for first-order logic has the potential to cover most areas of AI, it has not been so widely used as Prolog. One reason for this is the inefficiency of the proof procedure and the other is lack of useful applications. However, through research on program synthesis from formal specification[Hasegawa *et al.*, 1990], circuit verification, and legal reasoning[Nitta *et al.*, 1992], we became convinced that first-order theorem provers can be effectively used in various areas. We are now developing an automated program synthesis system, a specification description system for exchange systems, and abductive and non-monotonic reasoning systems on MGTP.

- To develop KL1 programming techniques
  Accumulating KL1 programming techniques through the development of theorem provers is an important issue. We first developed KL1 compiling techniques to translate given clauses to corresponding KL1 clauses, thereby achieving good performance for ground clause problems. We also developed methods to parallelize MGTP by making full use of logical variables and the stream data type of KL1.

- To develop KL1 meta-programming technology
  This is also an important issue in developing theorem provers. This issue is discussed in Section 2.1.2 Meta-Programming in KL1. We have implemented basic meta-programming tools called *Meta-Library* in KL1. The meta-library is a collection of KL1 programs which offers routines such as full unification, matching, and variable managements.

2.1.1 Theorem Prover in KL1 Language

Recent developments in logic programming have made it possible to implement first-order theorem provers efficiently. Typical examples are PTTP by Stickel [Stickel 1988], and SATCHMO by Manthey and Bry [Manthey and Bry 1988].

PTTP is a backward-reasoning prover based on the model elimination method. It can deal with any first-order formula in Horn clause form without loss of completeness and soundness.

SATCHMO is a forward-reasoning prover based on the model generation method. It is essentially a hyper-resolution prover, and imposes a condition called range-restricted on a clause so that we can derive only ground atoms from ground facts. SATCHMO is basically a forward-reasoning prover but also allows backward-reasoning by employing Prolog over the Horn clauses.

The major advantage of these systems is because the input clauses are represented with Prolog clauses and most parts of deductions can be performed through normal Prolog execution.

In addition to this method we considered the following two alternative implementations of object-level variables in KL1:

(1) representing object-level variables with KL1 ground terms

(2) representing object-level variables with KL1 variables

The first approach might be the right path in meta-programming where object- and meta-levels are separated strictly, thereby giving it clear semantics. However, it forces us to write routines for unification, substitution, renaming, and all the other intricate operations on variables and environments. These routines would become considerably larger and more complex than the main program, and introduce overhead to orders of magnitude.

In the second approach, however, most of operations on variables and environments can be performed beside the underlying system instead of running routines on top of it. Hence, it enables a meta-programmer to save writing tedious routines as well as gaining high efficiency. Furthermore, one can also use Prolog var predicates to write routines such as occurrence checks in order to make built-in unification sound, if necessary. Strictly speaking, this approach may not be chosen since it makes the

distinction between object- and meta-level very ambiguous. However, from a viewpoint of program complexity and efficiency, the actual profit gained by the approach is considerably large.

In KL1, however, the second approach is not always possible, as in the Prolog case. This is because the semantics of KL1 never allows us to use a predicate like Prolog var. In addition, KL1 built-in unification is not the same as Prolog's counterpart, in that unification in the guard part of a KL1 clause is limited to one way and a unification failure in the body part is regarded as a semantic error or exception rather than as a failure which merely causes backtrack in Prolog. Nevertheless, we can take the second approach to implement a theorem prover where ground models are dealt with, by utilizing the features of KL1 as much as possible.

Taking the above discussions into consideration, we decided to develop both the MGTP/G and MGTP/N provers so that we can use them effectively according to the problem domain being dealt with.

The ground version, MGTP/G, aims to support finite problem domains, which include most problems in a variety of fields, such as database processing and natural language processing.

For ground model cases, the model generation method makes it possible to use just matching, rather than full unification, if the problem clauses satisfy the *range-restrictedness* condition [1].

This suggests that it is sufficient to use KL1's head unification. Thus we can take the KL1 variable approach for representing object-level variables, thereby achieving good performance.

The key points of KL1 programming techniques developed for MGTP/G are as follows: (Details are described in the next section.)

- First, we translate a given set of clauses into a corresponding set of KL1 clauses. This translation is quite simple.
- Second, we perform conjunctive matching of a literal in a clause against a model element by using KL1 head unification.
- Third, at the head unification, we can automatically obtain fresh variables for a different instance of the literal used.

The non-ground version, MGTP/N, supports infinite problem domains. Typical examples are mathematical theorems, such as group theory and implicational calculus.

For non-ground model cases, where full unification with occurrence check is required, we are forced to follow the KL1 ground terms approach. However, we do not necessarily have to maintain variable-binding pairs as processes in KL1. We can maintain them by using the vector facility supported by KL1, as is often used in ordinary language processing systems. Experimental results show that vector implementation is several hundred times faster than process implementation.

In this case, however, we cannot use the programming techniques developed for MGTP/G. Instead, we have to use a conventional technique, that is, interpreting a given set of clauses instead of compiling it into KL1 clauses.

[1] A clause is said to be range-restricted if every variable in the clause has at least one occurrence in its antecedent.

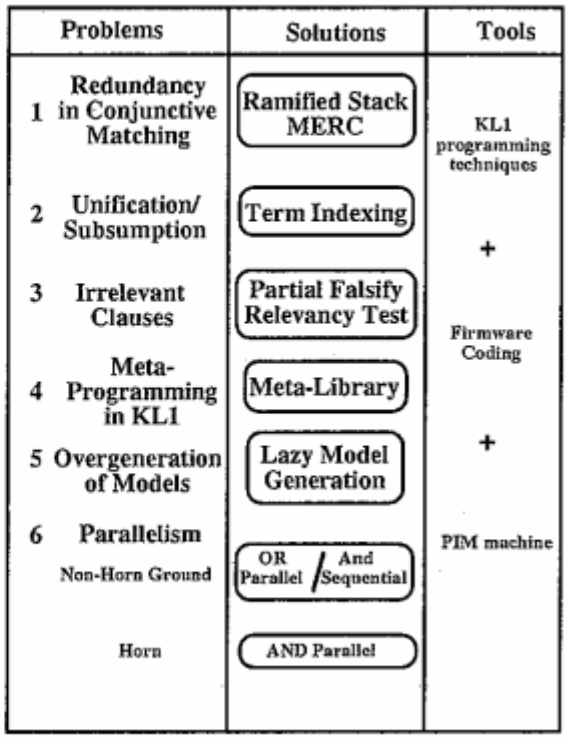

Figure 2: Major Problems and Technical Solutions

2.1.2 Key Technologies to Improve Efficiency

We developed several programming techniques in the process of seeking ways to improve the efficiency of model generation theorem provers. Figure 2 shows a list of the problems which prevented good performance and the solutions we obtained. In the following sections we outline the problems and their solutions.

Redundancy in Conjunctive Matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching. Let us consider a clause having two antecedent literals, and suppose we have a model candidate M at some stage i in the

proof process. To perform conjunctive matching of an antecedent literal in the clause against a model element, we need to pick all possible pairs of atoms from M. Imagine that we are to extend M with a model-extending atom Δ, which is in the consequent of the clause, but not in M. Then in the next stage, we need to pick pairs of atoms from $M \cup \Delta$. The number of pairs amounts to:

$$(M \cup \Delta)^2 = M \times M \cup M \times \Delta \cup \Delta \times M \cup \Delta \times \Delta.$$

However, doing this in a naive manner would introduce redundancy. This is because $M \times M$ pairs were already considered in the previous stage. Thus we must only choose pairs which contain at least one Δ.

(1) RAMS Method

The key point of the RAMS (Ramified Stack) method is to retain in a literal instance stack the intermediate results obtained in conjunctive matching. They are instances which are a result of matching a literal against a model element. This algorithm exactly computes a repeated combination of Δ and an atom picked from M without duplication([Fujita and Hasegawa 1990]).

For non-Horn clause cases, the literal instance stack expands a branch every time case splitting occurs, and grows like a tree. This is how the RAMS name was derived. Each branch of the tree represents a different model candidate.

The ramified-stack method not only avoids redundancy in conjunctive matching but also enables us to share a common model. However, it has one drawback: it tends to require a lot of memory to retain intermediate literal instances.

(2) MERC Method

The MERC (Multi-Entry Repeated Combination) method([Hasegawa 1991]) tries to solve the above problem in the RAMS method. This method does not need a memory to retain intermediate results obtained in the conjunctive matching. Instead, it needs to prepare $2^n - 1$ clauses for the given clause having n literals as its antecedent.

The outline of the MERC method is shown in Figure 3. For a clause having three antecedent literals, $A_1, A_2, A_3 \rightarrow C$, we prepare seven clauses, each of which corresponds to a repeated combination of Δ and M, and perform the conjunctive matching using the combination pattern. For example, a clause corresponding to a combination pattern $[M, \Delta, M]$ first matches literal A_2 against Δ. If the match succeeds, the remaining literals, A_1 and A_3, are matched against an element picked out of M. Note that each combination pattern includes at least one Δ, and that the $[M, M, M]$ pattern is excluded.

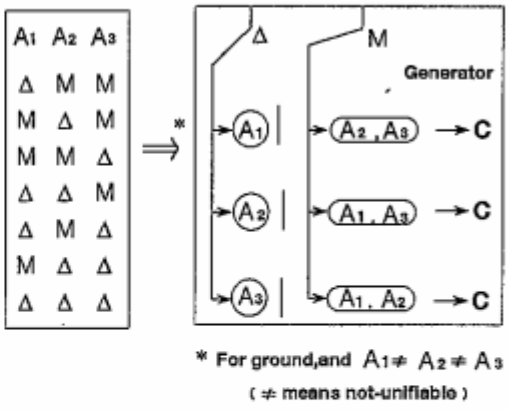

Figure 3: Multiple-Entry Repeated Combination (MERC) Method

There are some trade-off points between the RAMS method and the MERC method. In the RAMS method, every successful result of matching a literal A_i against model elements is memorized so as not to rematch the same literal against the same model element. On the other hand, the MERC method does not need such a memory to store the information of partial matching. However, it still contains a redundant computation. For instance, in the computation for $[M, \Delta, \Delta]$ and $[M, \Delta, M]$ patterns, the common subpattern, $[M, \Delta]$, will be recomputed. The RAMS method can remove this sort of redundancy.

Speeding up Unification/Subsumption

Almost all computation time is used in the unification and subsumption tests in the MGTP. Term indexing is a classical way and the only way to improve this process to one-to-many unification/subsumption. We used the discrimination tree as the indexing structure.

Figure 4 shows the effect of Term Memory on a typical problem on MGTP/G.

Optimal use of Disjunctive Clauses

Loveland et. al. [Wilson and Loveland 1989] indicated that irrelevant use of disjunctive clauses in the ground model generation prover rises useless case splitting, thereby leads to serious redundant searches. Arbitrary selected disjunctive clauses in MGTP may lead to a combinatorial explosion of redundant models. An artificial yet suggestive example is shown in Figure 5.

In MGTP/G, we developed two methods to deal with this problem. One method is to introduce upside-down

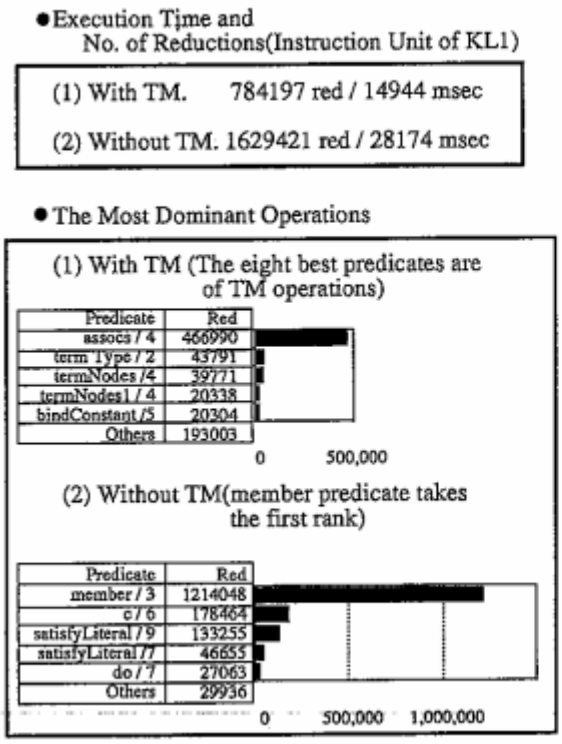

Figure 4: Speed up by Term Memory

```
false :−p(c, X, Y)                    (1)
false :−q(X, c, Y)                    (2)
false :−r(X, Y, c)                    (3)
      s(a)                            (4)
      s(b)                            (5)
      s(c)                            (6)
s(X), s(Y), s(Z) →
p(X, Y, Z); q(X, Y, Z); r(X, Y, Z)    (7)
```

Figure 5: Example Problem to Relevancy Testing

meta-interpretation(UDMI)[Stickel 1991] into MGTP/G. By using upside-down meta-interpretation, the above problem was compiled into the bottom-up rules in Figure 6.

Note that this is against the range restricted rule but is safe with Prolog-unification.

The other method is to keep the positive disjunctive clauses obtained by the process of reasoning. False checks are made independently on each literal in the disjunctive model elements with unit models and if the check succeeds then that literal is eliminated. The disjunctive models can be sorted by their length. This method showed considerable speed-up for n-queens problems and enumeration of finite algebra.

```
      true → gp(c, X, Y).             (1 − 1)
gp(c, X, Y), p(c, X, Y) → false.      (1 − 2)
      true → gq(X, c, Y).             (2 − 1)
gq(X, c, Y), q(X, c, Y) → false.      (2 − 2)
      true → gr(X, Y, c).             (3 − 1)
gr(X, Y, c), r(X, Y, c) → false.      (3 − 2)
         true → s(a)                  (4)
         true → s(b)                  (5)
         true → s(c)                  (6)
      s(X), s(Y), s(Z),
gp(X, Y, Z), gq(X, Y, Z), gr(X, Y, Z) →
   p(X, Y, Z); q(X, Y, Z); r(X, Y, Z) (7)
```

Figure 6: Compiled code in UDMI

Meta-Programming in KL1

Developing fast meta-programs such as unification and matching programs is very significant in making a prover efficient. Most parts of proving processes are the executions of such programs. The efficiency of a prover depends on how efficient meta-programs are made.

In Prolog-Technology Theorem Provers such as PTTP and SATCHMO, object-level variables[2] are directly represented by Prolog variables. With this representation, most operations on variables and environments can be performed beside the underlying system Prolog. This means that we can gain high efficiency by using the functions supported by Prolog. Also, a programmer can use the Prolog var predicate to write routines such as occurrence checks in order to make built-in unification sound, if such routines are necessary.

Unfortunately in KL1, we cannot use this kind of technique. This is because:

1) the semantics of KL1 never allow us to use a predicate like var,

2) KL1 built-in unification is not the same as its Prolog counterpart in that unification in the guard part of a KL1 clause can only be one-way, and

3) a unification failure in the body part is regarded as a program error or exception that cannot be backtracked.

We should, therefore, treat an object-level variable as constant data (ground term) rather than as a KL1 variable. It forces us to write routines for unification, substitution, renaming, and all the other intricate operations of variables and environments. These routines can become extremely large and complex compared to the main program, and may make the overhead bigger.

To ease the programmer's burden, we developed *Meta-Library*. This is a collection of KL1 programs to support meta-programming in KL1 [Koshimura *et al.*, 1990].

[2]variables appearing in a set of given clauses

The meta-library includes facilities such as full unification with occurrence check, and variable management routines. The performance of each program in the meta-library is relatively good. For example, unification program runs at $0.25 \sim 1.25$ times the speed of built-in unification.

The major functions in meta-library are as follows.

```
unify(X,Y, Env,^NewEnv)
unify_oc(X,Y, Env,^NewEnv)
match(Pattern,Terget, Env,^NewEnv)
oneway_unify(Pattern,Terget, Env,^NewEnv)
copy_term(X,^NewX, Env,^NewEnv)
shallow(X,Env, ^NewNnv)
freeze(X,^FrozenX, Env)
melt(X,^MeltedX, Env)
create_env(^Env, Size)
fresh_var(Env, ^VarAndNewEnv)
equal(X,Y, Env, ^YN)
is_type(X,Env, ^Type)
unbound(X,Env, ^YN)
database(RequestStream)
get_object(KL1Term, ^Object)
get_kl1_term(Object, ^KL1Term)
```

Over-Generation of Models

A more important issue with regard to the efficiency of model generation based provers is reducing the total amount of computation and memory space required in proof processes.

Model-generation based provers must perform the following three operations.

- create new model elements by applying the model extension rule to the given clauses using a set of model-extending atoms Δ and a model candidate set M (model extension).
- make a subsumption test for a created atom to check if it is subsumed by the set of atoms already being created, usually by the current model candidate.
- make a false check to see if the unsubsumed model element derives false by applying the model rejection rule to the tester clauses (rejection test).

The problem with the model generation method is the huge growth in the number of generated atoms and in the computational cost in time and space, which is incurred by the generation processes. This problem becomes more critical when dealing with harder problems which require deeper inferences (longer proofs), such as Lukasiewicz problems.

To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes, and that generation should be performed only when required by the test.

Table 1: Comparison of complexities (for unit tester clause)

Algorithm	T	S	G	M
Basic	ρm^2	$\mu\rho^2 m^{4\alpha}$	$\rho^3 m^4$	$\rho^3 m^4$
Full-test/Lazy	ρm^2	$\mu m^{2\alpha}$	m^2	ρm^2
Lazy & Lookahead	m^2	$(\mu/\rho)m^\alpha$	m/ρ	m

† m is the number of elements in a model candidate when *false* is detected in the basic algorithm.

‡ ρ is the survival rate of a generated atom, μ is the rate of successful conjunctive matchings ($\rho \leq \mu$), and $\alpha (1 \leq \alpha \leq 2)$ is the efficiency factor of a subsumption test.

For this we proposed a lazy model generation algorithm[Hasegawa *et al.*, 1992] that can reduce the amount of computation and space necessary for obtaining proofs.

Table 1 compares the complexities of the model generation algorithms[3], where T(S/G) represents the number of rejection tests (subsumption tests/model extensions), and M represents the number of atoms stored.

From a simple analysis, it is estimated that the time complexity of the model extension and subsumption test decreases from $O(m^4)$ in the algorithms without lazy control to $O(m)$ in the algorithms with lazy control. For details, refer to [Hasegawa *et al.*, 1992].

Parallelizing MGTP

There are three major sources when parallelizing the proving processes in the MGTP prover: multiple model candidates in a proof, multiple clauses to which model generation rules are applied, and multiple literals in conjunctive matching.

Let us assume that the prime objective of using the model generation method is to find a model as a solution. There may be alternative solutions or models for a given problem. We take it as OR-parallelism to seek these multiple solutions at the same time.

According to our assumption, multiple model candidates and multiple clauses are taken as sources for exploiting OR-parallelism. On the other hand, multiple literals are the source of AND-parallelism since all the literals in a clause relate to a single solution, where shared variables in the clause should have compatible values.

For ground non-Horn cases, it is sufficient to exploit OR parallelism induced by case splitting. For Horn clause cases, we have to exploit AND parallelism. The

[3]The basic algorithm taken by OTTER[McCune 1990] generates a bunch of new atoms before completing rejection tests for previously generated atoms. The full-test algorithm completes the tests before the next generation cycle, but still generates a bunch of atoms each time. Lookahead is an optimization method for testing wider spaces than in Full-test/Lazy.

main source of AND parallelism is conjunctive matching. Performing subsumption tests in parallel is also very effective for Horn clause cases.

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

(1) Parallelization of MGTP/G

With the current version of the MGTP/G, we have only attempted to exploit OR parallelism on the Multi-PSI machine.

(a) Processor allocation

The processor allocation methods that we adopted achieve 'bounded-OR' parallelism in the sense that OR-parallel forking in the proving process is suppressed so as to meet restricted resource circumstances.

One way of doing this, called *simple allocation*, is sketched as follows. We expand model candidates starting with an empty model using a single master processor until the number of candidates exceeds the number of available processors, then distribute the remaining tasks to slave processors. Each slave processor explores the branches assigned without further distributing tasks to any other processors. This simple allocation scheme for task distribution works fairly well since communication costs can be minimized.

(b) Speed-up on Multi-PSI

One of the examples we used is the N-queens problem given below.

$$
\begin{aligned}
C_1: &\quad true \rightarrow p(1,1); p(1,2); \ldots; p(1,n).\\
&\quad \ldots\\
C_n: &\quad true \rightarrow p(n,1); p(n,2); \ldots; p(n,n).\\
C_{n+1}: &\quad p(X_1,Y_1), p(X_2,Y_2),\\
&\qquad unsafe(X_1,Y_1,X_2,Y_2)\\
&\qquad\qquad \rightarrow false.
\end{aligned}
$$

The first N clauses express every possible placing of queens on an N by N chess board. The last clause expresses the constraint that a pair of queens must satisfy. So, the problem would be solved when either one model (one solution) or all the models (all solutions) are obtained for the clause set. The performance has been measured on an MGTP/G prover running on a Multi-PSI using the simple allocation method stated above.

The speedup obtained using up to 16 processors are shown in Figure 7. For the 10-queens problem, almost linear speedup is obtained as the number of processors increases. The speedup

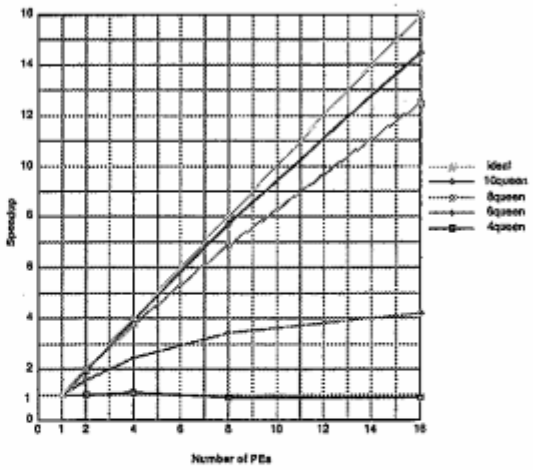

Figure 7: Speedup of MGTP/G on Multi-PSI (N-queens)

rate is rather small for the 4-queens problem only. This is probably because in such a small problem, the constant amount of interpretation overhead would dominate the proper tasks for the proving process.

(2) Parallelization of MGTP/N

For MGTP/N, we have attempted to exploit AND parallelism for Horn problems.

We have several choices when parallelizing model-generation based theorem provers:

1) proofs which change or remain unchanged according to the number of PEs used
2) model sharing (copying in a distributed memory architecture) or model distribution, and
3) master-slave or master-less.

A proof changing prover may achieve super-linear speedup while a proof unchanging prover can achieve, at most, linear speedup.

The merit of model sharing is that time consuming subsumption testing and conjunctive matching can be performed at each PE independently with minimal inter-PE communication. On the other hand, the benefit of model distribution is that we can obtain memory scalability. The communication cost, however, increases as the number of PEs increases, since generated atoms need to flow to all PEs for subsumption testing.

The master-slave configuration makes it easy to build a parallel system by simply connecting a sequential version of MGTP/N on a slave PE to the master PE. However, it needs to be designed with devices so as to minimize the load on the master

Table 2: Performance of MGTP/N (Th 5 and Th 7)

Problem		16 PEs	64 PEs
Th5	Time (sec)	41725.98	11056.12
	Reductions	38070940	40759689
	KRPS/PE	57.03	57.60
	Speedup	1.00	3.77
Th7	Time (sec)	48629.93	13514.47
	Reductions	31281211	37407531
	KRPS/PE	40.20	43.25
	Speedup	1.00	3.60

process. On the other hand, a master-less configuration, such as a ring connection, allows us to achieve pipeline effects with better load balancing, whereas it becomes harder to implement suitable control to manage collaborative work among PEs.

Our policy in developing parallel theorem provers is that we should distinguish between the speedup effect caused by parallelization and the search-pruning effect caused by strategies. In the proof changing parallelization, changing the number of PEs is merely betting, and may cause the strategy to be changed badly even though it results in the finding of a shorter proof.

Given the above, we implemented a proof unchanging version of MGTP/N in a master-slave configuration based on lazy model generation. In this system, generator and subsumption processes run in a demand-driven mode, while tester processes run in a data-driven mode. The main features of this system are as follows:

1) Proof unchanging allows us to obtain greater speedup as the number of PEs increases;
2) By utilizing the synchronization mechanism supported by KL1, sequentiality in subsumption testing is minimized;
3) Since slave processes spontaneously obtain tasks from the master and the size of each task is well equalized, good load balancing is achieved;
4) By utilizing the stream data type of KL1, demand driven control is easily and efficiently implemented.

By using the demand driven control, we can not only suppress unnecessary model extensions and subsumption tests but also maintain a high running rate that is the key to achieving linear speedup.

Figure 8 displays the speedup ratio for condensed detachment problems #3, #58, and #77, taken from[McCune and Wos 1991], by running the

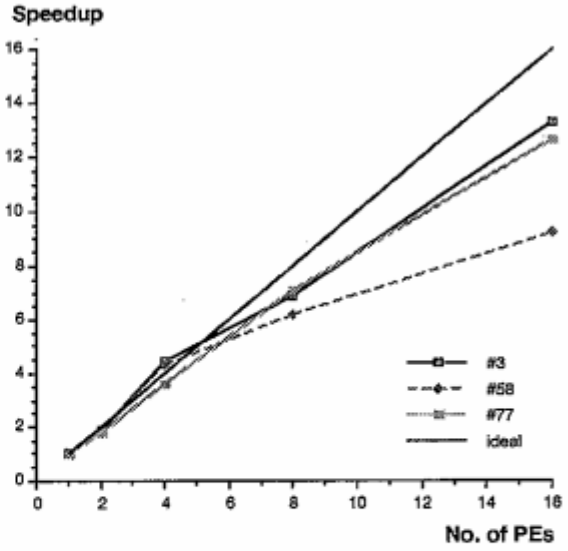

Figure 8: Speedup ratio

MGTP/N prover on Multi-PSI using 16PEs. The execution times taken to solve these problems are 218, 12, and 37 seconds. As shown in the figure, there is no saturation in performance up to 16 PEs and greater speedup is obtained for the problems which consume more time.

Table 2 shows the performance obtained by running MGTP/N for Theorems 5 and 7 [Overbeek 1990], which are also condensed detachment problems, on Multi-PSI with 64 PEs. We did not use heuristics such as sorting, but merely limited term size and eliminated tautologies. Full unification is written in KL1, which is thirty to one hundred times slower than that written in C on SUN/3s and SPARCs. Note that the average running rate per PE for 64 PEs is actually a little higher than that for 16 PEs. With this and other results, we were able to obtain almost linear speedup.

Recently we obtained a proof of Theorem 5 on PIM/m with 127 PEs in 2870.62 sec and nearly 44 billion reductions (thus 120 KRPS/PE). Taking into account the fact that the PIM/m CPU is about twice as fast as that of Multi-PSI, we found that almost linear speedup can be achieved, at least up to 128 PEs.

2.2 Reflection and Parallel Meta-Programming System

Reflection is the capability to feel the current state of the computation system or to dynamically modify it. The form of *reflection* we are interested in is the *computational reflection* proposed by [Smith 1984]. We try to

incorporate meta-level computation and computational reflection in logic programming language in a number of directions.

As a foundation, a reflective sequential logic language *R-Prolog** has been proposed [Sugano 1990]. This language allows us to deal with syntactic and semantic objects of the language itself legally by means of several coding operators. The notion of computational reflection is also incorporated, which allows computational systems to recognize and modify their own computational states. As a result, some of the extra-logical predicates in Prolog can be redefined in a consistent framework. We have also proposed a reflective parallel logic programming language RGHC (Reflective Guarded Horn Clauses)[Tanaka and Matono 1992]. In RGHC, a *reflective tower* can be constructed and collapsed in a dynamic manner, using *reflective predicates*. A prototype implementation of RGHC has also been performed. It seems that RGHC is unique in the simplicity of its implementation of *reflection*. The meta-level computation can be executed at the same speed as its object-level computation. we also try to formalize distributed reflection, which allows concurrent execution of both object level and meta level computations [Sugano 1991]. The scope of reflection is specified by grouping goals that share local environments. This also models the eventual publication of constraints.

We have also built up several application systems based on meta-programming and reflection. These are the experimental programming system ExReps [Tanaka 1991], the process oriented GHC debugger [Maeda *et al.*, 1990, Maeda 1992] and the strategy management shell [Kohda and Maeda 91a, Kohda and Maeda 1991b].

ExReps is an experimental programming environment for parallel logic languages, where one can input programs and execute goals. It consists of an abstract machine layer and an execution system layer. Both layers are constructed using meta-programming techniques. Various *reflective operations* are implemented in these layers.

The process oriented GHC debugger provides high-level facilities, such as displaying processes and streams in tree views. It can control a the behavior of a process by interactively blocking or editing its input stream data. This makes it possible to trace and check program execution from a programmer's point of view.

A strategy management shell takes charge of a database of load-balancing strategies. When a user job is input, the current leading strategy and several experimental alternative strategies for the job are searched for in the database. Then the leading task and several experimental tasks of the job are started. The shell can evaluate the relative merits between the strategies, and decides on the leading strategy for the next stage when the tasks have terminated.

3 Applications of Automated Reasoning

ARS has a wider application area if connected with logic programming and a formal approach to programming. We extended MGTP to cover modal logic. This extension has lead to abductive reasoning in AI systems and logic programming with negation as failure linked with broader practical applications such as fault diagnostics and legal reasoning. We also focused on programming, particularly parallel programs, as one of the major application area of formal logic systems in spite of difficulties. There has been a long history of program synthesis from specifications in formal logic. We are aiming to make ARS, the foundational strength of this approach.

3.1 Propositional Modal Tableaux in MGTP

MGTP's proof method and the tableaux proof procedure[Smullyan 1968] are very close in computationally. Each rule of tableaux is represented by an input clause for MGTP in a direct manner. In other words, we can regard the input clauses for MGTP as a tableaux implementation language, as Horn clauses are a programming language for Prolog.

MGTP tries to generate a model for a set of clauses in a bottom-up manner. When MGTP successfully generates a model, it is found to be satisfiable. Otherwise, it is found to be unsatisfiable.

$$apply(MGTP, ASetOfClauses) = \begin{cases} satisfiable \\ unsatisfiable \end{cases}$$

Since we regard MGTP as an inference system, a propositional modal tableaux[Fitting 1983, Fitting 1988] has been implemented in MGTP.

$$apply(MGTP, TableauxProver(Formula)) = \begin{cases} satisfiable \\ unsatisfiable \end{cases}$$

In tableaux, a close condition is represented by a negative clause, an input formula by a positive clause and a decomposition rule by a mixed clause for MGTP in a direct manner[Koshimura and Hasegawa 1991].

There are two levels in this prover. One is the MGTP implementation level, the other is the tableaux implementation level. The MGTP level is the inference system level at which we mainly examine speedup of inference such as redundancy elimination and parallelization. At the tableaux level, inference rules, which indicate the property of a proof domain, are described. It follows that we mainly examine the property of the proof domain at the tableaux level. It is useful and helpful to have these two levels, as we can separate the description for the property of the domain from the description for the inference system.

3.2 Abductive and Nonmonotonic Reasoning

Modeling sophisticated agents capable of *reasoning with incomplete information* has been a major theme in AI. This kind of reasoning is not only an advanced mechanism for intelligent agents to cope with some particular situations but an intrinsically necessary condition to deal with commonsense reasoning. It has been agreed that neither human beings nor computers can have all the information relevant to mundane or everyday situations. To function without complete information, intelligent agents should draw some unsound conclusions, or augment theorems, by applying such methods as closed-world assumptions and *default reasoning*. This kind of reasoning is *nonmonotonic*: it does not hold that the more information we have, the more consequences we will be aware of. Therefore, this inference has to anticipate the possibility of later revisions of beliefs.

We treat reasoning with incomplete information as a reasoning system with hypotheses, or *hypothetical reasoning* [Inoue 1988], in which a set of conclusions may be expanded by incorporating other hypotheses, unless they are contradictory. In hypothetical reasoning, inference to reach the best explanations, that is, computing hypotheses that can explain an observation, is called *abduction*. The notion of explanation has been a fundamental concept for various AI problems such as diagnoses, synthesis, design, and natural language understanding. We have investigated methodologies of hypothetical reasoning from various angles and have developed a number of abductive and nonmonotonic reasoning systems.

Here, we shall present hypothetical reasoning systems built upon the MGTP [Fujita and Hasegawa 1991]. The basic idea of these systems is to translate formulas with special properties, such as nonmonotonic provability (*negation as failure*) and *consistency* of abductive explanations, into some formulas with a kind of modality so that the MGTP can deal with them using classical logic. The extra requirements for these special properties are thus reduced to generate-and-test problems for model candidates. These, can, then, be handled by the MGTP very efficiently through case-splitting of non-unit consequences and rejection of inconsistent model candidates. In the following, we show how the MGTP can be used for logic programs containing negation as failure, and for abduction.

3.2.1 Logic Programs and Disjunctive Databases with Negation as Failure

In recent theories of logic programming and deductive databases, declarative semantics have been given to the extensions of logic programs, where the negation-as-failure operator is considered to be a *nonmonotonic* modal operator. In particular, logic programs or deductive databases containing both negation as failure (*not*) and classical negation ($\neg$) can be used as a powerful knowledge representation tool, whose applications contain reasoning with incomplete knowledge [Gelfond and Lifschitz 1991], default reasoning, and abduction [Inoue 1991a]. However, for these extended classes of logic programs, the top-down approach cannot be used for computation because there is no local property in evaluating programs. For example, there has been *no* top-down proof procedure which is sound with respect to the *stable model semantics* for general logic programs. We thus need bottom-up computation for correct evaluation of negation-as-failure formulas.

In [Inoue *et al.*, 1992a], a bottom-up computation of *answer sets* for any class of function-free logic programs is provided. These classes include the *extended disjunctive databases* [Gelfond and Lifschitz 1991], the proof procedure of which has not been found. In evaluating $not\, P$ in a bottom-up manner, it is necessary to interpret $not\, P$ with respect to a fixpoint of computation because, even if P is not currently proved, P might be proved in subsequent inferences. We thus came up with a completely different way of thinking for *not*. When we have to evaluate $not\, P$ in a current model candidate we split the model candidate in two: (1) the model candidate where P is assumed not to hold, and (2) the model candidate where it is necessary that P holds. Each negation-as-failure formula $not\, P$ is thus translated into negative and positive literals with a modality expressing belief, i.e., "disbelieve P" (written as $\neg \mathrm{K}P$) and "believe P" (written as $\mathrm{K}P$).

Based on the above discussion, we translate any logic program (with negation as failure) into a *positive disjunctive program* (without negation as failure) of which the MGTP can compute the minimal models. The following is an example of the translation of general logic programs. Let Π be a general logic program consisting of rules of the form:

$$A_l \leftarrow A_{l+1}, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n, \quad (1)$$

where, $n \geq m \geq l \geq 0$, $1 \geq l \geq 0$, and each A_i is an atom. Rules without heads are called *integrity constraints* and are expressed by $l = 0$ for the form (1). Each rule in Π of the form (1) is translated into the following MGTP rule:

$$A_{l+1}, \ldots, A_m \rightarrow \neg \mathrm{K}A_{m+1}, \ldots, \neg \mathrm{K}A_n, A_l \mid \mathrm{K}A_{m+1} \mid \ldots \mid \mathrm{K}A_n . \quad (2)$$

For any MGTP rule of the form (2), if a model candidate S' satisfies $A_{l+1}, \ldots, A_m$, then S' is split into $n - m + l$ ($n \geq m \geq 0$, $0 \leq l \leq 1$) model candidates. Pruning rules with respect to "believed" or "disbelieved" literals are expressed as the following integrity constraints. These are dealt with by using object-level schemata on the MGTP.

$$\neg \mathrm{K}A, A \rightarrow \quad \text{for every atom } A \quad (3)$$

$$\neg \mathrm{K}A, \mathrm{K}A \rightarrow \quad \text{for every atom } A \quad (4)$$

Given a general logic program Π, we denote the set of rules consisting of the two schemata (3) and (4) by $tr(\Pi)$,

and the MGTP rules obtained by replacing each rule (1) of Π by a rule (2). The MGTP then computes the fixpoint of model candidates, denoted by $\mathcal{M}(tr(\Pi))$, which is closed under the operations of the MGTP. Although each model candidate in $\mathcal{M}(tr(\Pi))$ contains "believed" atoms, we should confirm that every such atom is actually derived from the program. This checking can be done very easily by using the following constraint. Let $S' \in \mathcal{M}(tr(\Pi))$.

For every ground atom A, if $\mathsf{K}A \in S'$, then $A \in S'$. (5)

Computation by using MGTP is sound and complete with respect to the stable model semantics in the sense that: S is an answer set (or *stable model*) of Π if and only if S is one of the atoms obtained by removing every literal with the operator K from a model candidate S' in $\mathcal{M}(tr(\Pi))$ such that S' satisfies condition (5).

Example: Suppose that the general logic program Π consists of the four rules:

$$R \leftarrow not\, R,$$
$$R \leftarrow Q,$$
$$P \leftarrow not\, Q,$$
$$Q \leftarrow not\, P.$$

These rules are translated to the following MGTP rules:

$$\rightarrow \neg\mathsf{K}R, R \mid \mathsf{K}R,$$
$$Q \rightarrow R,$$
$$\rightarrow \neg\mathsf{K}Q, P \mid \mathsf{K}Q,$$
$$\rightarrow \neg\mathsf{K}P, Q \mid \mathsf{K}P.$$

In this example, the first MGTP rule can be further reduced to

$$\rightarrow \mathsf{K}R.$$

if we prune the first disjunct by the schema (3). Therefore, the rule has computationally the same effect as the integrity constraint:

$$\leftarrow not\, R.$$

This integrity constraint says that every answer set has to contain R: namely, R *should be derived*. Now, it is easy to see that $\mathcal{M}(tr(\Pi)) = \{S_1, S_2, S_3\}$, where $S_1 = \{\mathsf{K}R, \neg\mathsf{K}Q, P, \mathsf{K}P\}$, $S_2 = \{\mathsf{K}R, \mathsf{K}Q, \neg\mathsf{K}P, Q, R\}$, and $S_3 = \{\mathsf{K}R, \mathsf{K}Q, \mathsf{K}P\}$. The only model candidate that satisfies the condition (5) is S_2, showing that $\{Q, R\}$ is the unique stable model of Π. Note that $\{P\}$ is not a stable model because S_1 contains $\mathsf{K}R$ but does not contain R.

In [Inoue *et al.*, 1992a], a similar translation was also given to extended disjunctive databases which contain classical negation, negation as failure and disjunctions. Our translation method not only provides a simple fixpoint characterization of answer sets, but also is very helpful for understanding under what conditions each model is stable or unstable. The MGTP can find all answer sets incrementally, without backtracking, and in parallel. The proposed method is surprisingly simple and does not increase the computational complexity of the problem more than computation of the minimal models of positive disjunctive programs. The procedure has been implemented on top of the MGTP on a parallel inference machine, and has been applied to a legal reasoning system.

3.2.2 Abduction

There are many proposals for a logical account of *abduction*, whose purpose is to generate query explanations. The definition we consider here is similar to that proposed in [Poole *et al.*, 1987]. Let Σ be a set of formulas, Γ a set of literals and G a closed formula. A set E of ground instances of Γ is an *explanation of G from* (Σ, Γ) if

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

The computation of explanations of G from (Σ, Γ) can be seen as an extension of proof-finding by introducing a set of hypotheses from Γ that, if they could be proved by preserving the consistency of the augmented theories, would complete the proofs of G. Alternatively, abduction can be characterized by a consequence-finding problem [Inoue 1991b], in which some literals are allowed to be hypothesized (or *skipped*) instead of proved, so that new theorems consisting of only those skipped literals are derived at the end of deductions instead of just deriving the empty clause. In this sense, abduction can be implemented by an extension of deduction, in particular of a top-down, backward-chaining theorem-proving procedure. For example, Theorist [Poole *et al.*, 1987] and SOL-resolution [Inoue 1991b] are extensions of the Model Elimination procedure [Loveland 1978].

However, there is nothing to prevent us from using a bottom-up, forward-reasoning procedure to implement abduction. In fact, we developed the abductive reasoning system APRICOT/0 [Ohta and Inoue 1990], which consists of a forward-chaining inference engine and the ATMS [de Kleer 1986]. The ATMS is used to keep track of the results of inference in order to avoid both repeated proofs of subgoals and duplicate proofs among different hypotheses deriving the same subgoals.

These two reasoning styles for abduction have both merits and demerits, which are complementary to each other. Top-down reasoning is directed to the given goal but may result in redundant proofs. Bottom-up reasoning eliminates redundancy but may prove subgoals unrelated to the proof of the given goal. These facts suggest that it is promising to simulate top-down reasoning using

a bottom-up reasoner, or to utilize cashed results in top-down reasoning. This upside-down meta-interpretation [Bry 1990] approach has been attempted for abduction in [Stickel 1991], and has been extended by incorporating consistency checks in [Ohta and Inoue 1992].

We have already developed several parallel abductive systems [Inoue *et al.*, 1992b] using the the bottom-up theorem prover MGTP. We outline four of them below.

1. **MGTP+ATMS** (Figure 9).

   This is a parallel implementation of APRICOT/0 [Ohta and Inoue 1990] which utilizes the ATMS for checking consistency. The MGTP is used as a forward-chaining inference engine, and the ATMS keeps a current set of beliefs M, in which each ground atom is associated with some hypotheses. For this architecture, we have developed an upside-down meta-interpretation method to incorporate the top-down information [Ohta and Inoue 1992].

   Parallelism is exploited by executing the parallel ATMS. However, because there is only one channel between the MGTP and the ATMS, the MGTP often has to wait for the results of the ATMS. Thus, the effect of parallel implementation is limited.

2. **MGTP+MGTP** (Figure 10).

   This is a parallel version of the method described in [Stickel 1991]. In addition, consistency is checked by calling another MGTP (*MGTP_2*). In this system, each hypothesis H in Γ is represented by $fact(H, \{H\})$, and each Horn clause in Σ of the form:

   $$A_1 \wedge \ldots \wedge A_n \supset C\,,$$

   is translated into an MGTP rule of the form:

   $$fact(A_1, E_1), \ldots, fact(A_n, E_n) \rightarrow$$

   $$fact(C, \mathrm{cc}(\bigcup_{i=1}^{n} E_i))\,,$$

   where E_i is a set of hypotheses from Γ on which A_i depends, and the function cc is defined as:

   $$\mathrm{cc}(E) = \begin{cases} E & \text{if } \Sigma \cup E \text{ is consistent} \\ \mathrm{nil} & \text{if } \Sigma \cup E \text{ is not consistent} \end{cases}$$

   A current set of beliefs M is kept in the form of $fact(A, E)$ representing a meta-statement that $\Sigma \cup E \models A$, but is stored in the inference engine (*MGTP_1*) itself. Each time *MGTP_1* derives a new ground atom, the consistency of the combined hypotheses is checked by *MGTP_2*.

   The parallelism comes from calling multiple *MGTP_2*'s at one time. This system achieves more speed-up than the MGTP+ATMS method. However, since *MGTP_1* is not parallelized, the effect of parallelization depends heavily on how much consistency checking is being performed in parallel at one time.

3. **All Model Generation Method.**

   No matter how good the MGTP+MGTP method might be, the system still consists of two different components. The possibilities for parallelization therefore remain limited. In contrast, model generation methods do not separate the inference engine and consistency checking, but realize both functions in a single MGTP. In such a method, the MGTP is used not only as an inference engine but also as a generate-and-test mechanism so that consistency checks are automatically performed. For this purpose, we can utilize the extension and rejection of model candidates supplied by the MGTP. Therefore, multiple model candidates can be kept in distributed memories instead of keeping one global belief set M, as done in the above two methods, thus great amounts of parallelism can be obtained.

   The all model generation method is the most direct way to implement reasoning with hypotheses. For each hypothesis H in Γ, we supply a rule of the form:

   $$\rightarrow H \mid \neg \mathsf{K} H\,, \qquad (6)$$

   where $\neg \mathsf{K} H$ means that H is not assumed to be true in the model. Namely, each hypothesis is assumed either to hold or not to hold. Since this system may generate $2^{|\Gamma|}$ model candidates, the method is often too explosive for several practical applications.

4. **Skip Method.**

   To limit the number of generated model candidates as much as possible, we can use a method to delay the case-splitting of hypotheses. This approach is similar to the processing of negation as failure with the MGTP [Inoue *et al.*, 1992a], introduced in the previous subsection. That is, we do not supply any rule of the form (6) for any hypothesis of Γ, but instead, we introduce hypotheses when they are necessary. When a clause in Σ contains negative occurrences of abducible predicates $H_1, \ldots, H_m$ ($H_i \in \Gamma$, $m \geq 0$) and is in the form:

   $$A_1 \wedge \ldots \wedge A_l \wedge \underbrace{H_1 \wedge \ldots \wedge H_m}_{abducibles} \supset C\,,$$

   we translate it into the following MGTP rule:

   $$A_1, \ldots, A_l \rightarrow$$

   $$H_1, \ldots, H_m, C \mid \neg \mathsf{K} H_1 \mid \ldots \mid \neg \mathsf{K} H_m\,. \qquad (7)$$

   In this translation, each hypothesis in the premise part is *skipped* instead of being resolved, and is moved to the right-hand side. This operation is

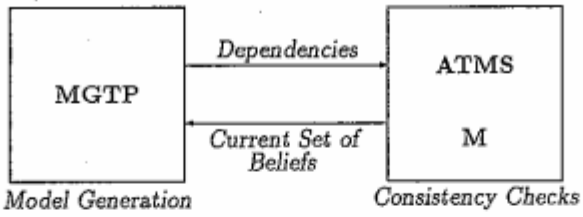

Figure 9: MGTP+ATMS

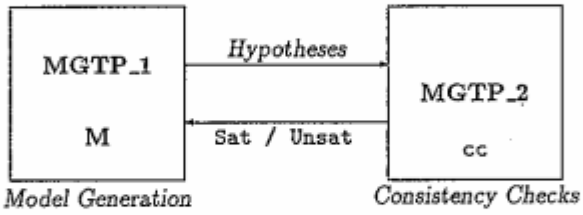

Figure 10: MGTP+MGTP

a counterpart to the **Skip** rule in the top-down approach defined in [Inoue 1991b]. Just as in schema (3) for negation as failure, a model candidate containing both H and $\neg KH$ is rejected by the schema:

$$\neg KH, H \rightarrow \qquad \text{for every hypothesis } H\,.$$

Some results of evaluation of these abductive systems as applied to planning and design problems are described in [Inoue *et al.*, 1992b]. We are now improving their performance for better parallelism. Although we need to investigate further how to avoid possible combinatorial explosion in model candidate construction for the skip method, we conjecture that the skip method (or some variant thereof) will be the most promising from the viewpoint of parallelism. Also, the skip method may be easily combined with negation as failure so that knowledge bases can contain both abducible predicates and negation-as-failure formulas as in the approach of [Inoue 1991a].

3.3 Program Synthesis by Realizability Interpretation

3.3.1 Program Synthesis by MGTP

We used *Realizability Interpretation* (an extension of *Curry-Howard Isomorphism*) in the area of constructive mathematics [Howard 1980], [Martin 1982] in order to give an executable meaning to proofs obtained by efficient theorem provers.

Our approach for combining prover technologies and *Realizability Interpretation* has the following advantages:

- This approach is prover independent and all provers are possibly usable.
- *Realizability Interpretation* has a strong theoretical background.
- *Realizability Interpretation* is general enough to cover concurrent programs.

Two systems MGTP and PAPYRUS, developed in ICOT, are used for the experiments on sorting algorithms in order to get practical insights into our approach(Figure 11).

A model generation theorem prover (MGTP) implemented in KL1 runs on a parallel machine:Multi-PSI. It searches for proofs of specification expressed as logical formulae. MGTP is a hyper-resolution based bottom up (infers from premises to goal) prover. Thanks to KL1 programming technology, MGTP is simple but works very efficiently if problems satisfy the *range-restrictedness* condition. The inference mechanism of MGTP is similar to SATCHMO[Manthey and Bry 1988], in principle. Hyper-resolution has an advantage for program synthesis in that the inference system is constructive. This means that no further restriction is needed to avoid useless searching.

PAPYRUS (PArallel Program sYnthesis by Reasoning Upon formal Systems) is a cooperative workbench for formal logic. This system handles the proof trees of user defined logic in *Edinburgh Logical Framework(LF)*[Harper *et al.*, 1987]. A typed lambda term in LF represents a proof and a program can be extracted from this term by lambda computation. This system treats programs (functions) as the models of a logical formula by user defined *Realizability Interpretation*. PAPYRUS is an integrated workbench for logic and provides similar functions to PX[Hayashi and Nakano 1988], Nuprl[Constable *et al.*, 1986], and Elf[Pfenning 1988].

We faced two major problems during research process:

- Program extraction from a proof in clausal form, and

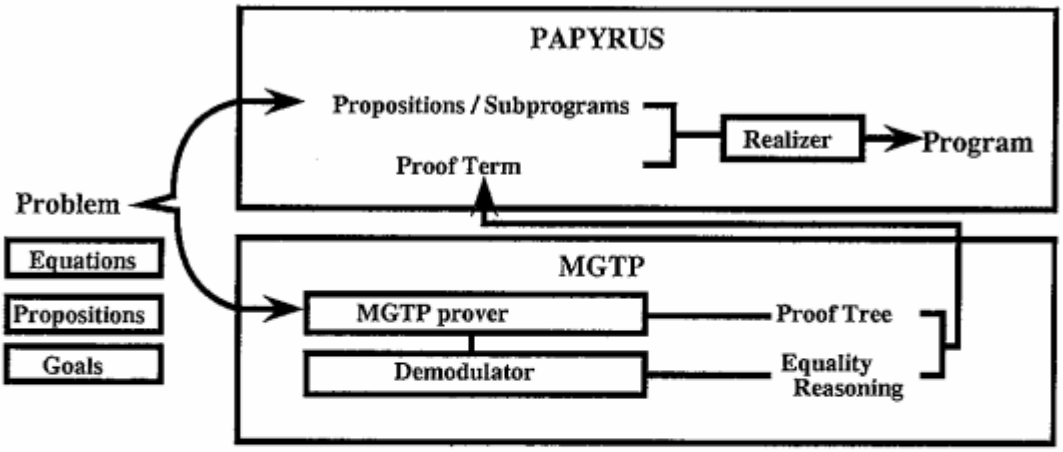

Figure 11: Program Synthesis by MGTP

- Incorporation of induction and equality.

The first problem relates to the fact that programs cannot be extracted from proofs obtained by using the excluded middle, as done in classical logic. The rules for transforming formulae into clausal form contains such a prohibited process. This problem can be solved if the program specification is given in clausal form because a proof can be obtained from the clause set without using the excluded middle. The second problem is that all induction schemes are expressed as second-order propositions. In order to handle this, second-order unification will be needed, which still is impractical. However, it is possible to transform a second-order proposition to a first-order proposition if the program domain is fixed.

Proof steps of equality have nothing to do with computation, provers can use efficient algorithms for equality as an attached procedure.

3.3.2 A Logic System for Extracting Interactive Processes

There has been some research [Howard 1980, Martin 1982, Sato 1986] and [Hayashi and Nakano 1988] into program synthesis from constructive proofs. In this method, an interpretation of formulas is defined, and the consistent proof of the formula can be translated into a program that satisfies the interpretation. Therefore we can identify the formula as the specification of the program, proof as programming, and proof checking as program checking. Though this method has many useful points, the definition of a program in this method is only "λ Term (function)". Thus it is difficult to synthesize a program as a parallel process by which computers can communicate with the outside world.

We proposed a new logic μ, that is, a constructive logic extended by introducing new operators μ and $\natural$. The operator μ is a fixpoint operator on formulae. We can express the non-bounded repetition of inputs and outputs with operators μ and $\natural$. Further, we show a method to synthesize a program as a parallel process like CCS[Milner 1989] from proofs of logic μ. We also show the proof of consistency of Logic μ and the validity of the method to synthesize a program.

3.4 Application of Theorem Proving to Specification of a Switching System

We apply a theorem proving technique to the software design of a switching system, whose specifications are modeled by finite state diagrams.

The main points of this project are the following:

1) Specification description language Ack, based on a transition system.

2) Graphical representation in Ack.

3) Ack interpreter by MGTP.

We introduce the protocol specification description language, Ack. It is not necessary to describe all state transitions concretely using Ack, because several state transitions are deduced from one expression by means of theorem proving. Therefore, we can get a complete specification from an ambiguous one.

Ack is based on a transition system (S, s_0, A, T), where S is a set of state, s_0 ($\in$ S) is an initial state, A is a set of actions, and $T(T \subseteq S \times A \times S)$ is a set of transition relations.

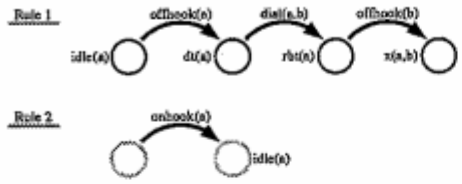

Figure 12: An example of Ack specification

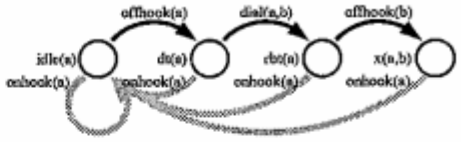

Figure 13: An interpretation result of Ack specification

Graphical representation in Ack consists of labeled circles and arrows. A circle means a state and an arrow means an action. Both have two colors: black and gray. This means that when a gray colored state transition exists a black colored state transition exists.

Textual phrase representation in Ack can be represented by a first order predicate logic by the following.

$$\forall \mathbf{X} \exists \mathbf{Y}(\mathbf{A}[\mathbf{X}] \rightarrow \mathbf{B}[\mathbf{X},\mathbf{Y}]).$$

where $\mathbf{A}[\mathbf{X}]$ and $\mathbf{B}[\mathbf{X},\mathbf{Y}]$ are conjunctions of the following atomic formulas.

$\mathbf{state(S)}$ - S means a state.

$\mathbf{trans(A,S_0,S_1)}$ - An action A means a state S_0 to a state S_1.

$\mathbf{A}[\mathbf{X}]$ corresponds to grayed color state transitions and $\mathbf{B}[\mathbf{X},\mathbf{Y}]$ corresponds to black color state transitions.

The Ack interpreter is described by MGTP. This type of formula is translated into an MGTP formula. A set of models deduced from Ack specification formulae form a complete state transition diagram.

Figure 12 shows an example of Ack specification.

Rule 1 of Figure 1 means the existence of an action sequence from an initial state $idle(a)$ such that $offhook(a) \rightarrow dial(a,b) \rightarrow offhook(b)$. This is represented by the following formula.

$$\begin{aligned} \rightarrow\ & \mathbf{trans(offhook(a),idle(a),dt(a))}, \\ & \mathbf{trans(dial(a,b),dt(a),rbt(a))}, \\ & \mathbf{trans(offhook(b),rbt(a),x(a,b))}. \end{aligned}$$

Rule 2 of Figure 1 means that the action $offhook(a)$ changes *any state* to $idle(a)$. It is represented by the following formula.

$$\begin{aligned} & \forall \mathbf{S}(\mathbf{state(S)} \wedge \mathbf{state(idle(a))} \\ & \rightarrow \mathbf{trans(onhook(a),S,idle(a))}) \end{aligned}$$

Figure 13 shows an interpretation of the result of Figure 12.

In this example, the following four transitions are automatically generated.

- action $onhook(a)$ from $idle(a)$ to $idle(a)$.
- action $onhook(a)$ from $dt(a)$ to $idle(a)$.
- action $onhook(a)$ from $rbt(a)$ to $idle(a)$.
- action $onhook(a)$ from $x(a,b)$ to $idle(a)$.

3.5 MENDELS ZONE: A Parallel Program Development System

MENDELS ZONE is a software development system for parallel programs. The target parallel programming language is MENDEL, which is a textual form of Petri Nets, MENDEL is then translated into the concurrent logic programming language KL1 and executed on the Multi-PSI. MENDEL is regarded as a more user-friendly version of the language. MENDEL is convenient for the programmer to use to design cooperating discrete event systems.

MENDELS ZONE provides the following functions:

1) Data-flow diagram visualizer
   [Honiden *et al.*, 1991]

2) Term rewriting system :
   Metis[Ohsuga *et al.*, 90][Ohsuga *et al.*, 91]

3) Petri Nets and temporal logic based programming environment
   [Uchihira *et al.*, 90a][Uchihira *et al.*, 90b]

For 1), we define the decomposition rule for data-flow diagram and extract the MENDEL component from decomposed data-flow diagrams. A detailed specification process from abstract specification is also defined by a combination of data-flow diagrams and equational formulas.

For 2), Metis is a system to supply experimental environment for studying practical techniques for equational reasoning. The policy of developing Metis is enabling us to implement, test, and evaluate the latest techniques for inference as rapidly and freely as possible. The kernel function of Metis is Knuth-Bendix (KB) completion procedure. We adopt Metis as a tool for verifying the MENDEL component. The MENDEL component can be translated into a component of Petri Nets.

For 3), following sub-functions are provided:

1. Graphic editor

   The designer constructs each component of Petri Nets using the graphic editor, which provides creation, deletion, and replacement. This editor also supports expansion and reduction of Petri Nets.

2. Method editor

   The method editor provides several functions specific to Petri Nets. Using the method editor, the designer describes methods (their conditions and actions) in detail using KL1.

3. Component library

   Reusable component are stored in the component library. The library tool supports browsing and searching for reusable components.

4. Verification and synthesis tool

   Only the skeletons of Petri Nets structures are automatically retracted (slots and KL1 codes of methods are ignored) since our verification and synthesis are applicable to bounded net. The verification tools verifies whether Petri Nets satisfy given temporal logic constraints.

5. Program execution on Multi-PSI

   The verified Petri Nets are translated into their textual form (MENDEL programs). The MENDEL programs are compiled into KL1 programs, which can be executed on Multi-PSI. During execution, firing methods are displayed on the graphic editor, and values of tokens are displayed on the message window. The designer can check visually that program behaves to satisfy his expectation.

4 Advanced Inference and Learning

It is expected that we will, before long, face a *software crisis* in which the necessary quantity of computer software cannot be provided even if we were all to engage in software production. In order to avoid this crisis, it is necessary for a computer system itself to produce software or new information adaptively in problem-solving. The aim of the study on advanced inference and learning is to explore the underlying mechanism for such a system.

In the current stage in which we have no absolute approach to the goal, we have had to do *exhaustive* searches. We have taken three different but co-operative approaches: logical, computational and empirical. In the logical approach, *analogical reasoning* has been analyzed formally and mechanisms for analogical reasoning have been explored. In the computational approach, we have studied *inventing new predicates*, which are one of the most serious problems in learning logic programs. We have also investigated the application of *minimally multiple generalization* for constructive logic pograms learning. In the empirical approach, we have studied automated programming, especially, the *logic program transformation and synthesis* method based on unfold/fold transformation which is a well-known technique for deriving correct and efficient programs.

The following subsections briefly describe these studies and their results.

4.1 Analogical Reasoning

Analogical reasoning is often said to be at the very core of human problem-solving and has long been studied in the field of artificial intelligence. We treat a general type of analogy, described as follows: when two objects, B (called the *base*) and T (called the *target*), share a property S (called the *similarity*), it is conjectured that T satisfies another property P (called the *projected property*) which B satisfies as well.

In the study of analogy, the following have been central problems:

1) Selection of an object as a base w.r.t a target.

2) Selection of pertinent properties for drawing analogies.

3) Selection of a property for projection w.r.t. a certain similarity.

Unfortunately, most previous works were only partially successful in answering these questions, by proposing solutions a priori.

Our objective is to clarify, as formally aspossible,the general relationship between those analogical factors T, B, S, and P under a given theory $\mathcal{A}$. To find the relationship bvetween the analogical factors would answer these problems once and for all. In [Arima 1992, Arima 1991], we clarify such a relation and show a general solution.

When analyzing analogical reasoning formally based on classical logic, the following are shown to be reasonable:

- Analogical reasoning is possible only if a certain form of rule, called the *analogy prime rule* (APR), is a deductive theorem of a given theory. If we let $S(x) = \Sigma(x, S)$ and $P(x) = \Pi(x, P)$, then the rule has the following form:

  $$\forall x, s, p.\ J_{att}(s,p) \wedge J_{obj}(x,s) \wedge \Sigma(x,s) \supset \Pi(x,p),$$

  where each of $J_{att}(s,p)$, $J_{obj}(x,s)$, $\Sigma(x,s)$ and $\Pi(x,p)$ are formulae in which no variable other than its argument occurs freely.

- An analogical conclusion is derived from the APR, together with two particular conjectures: one conjecture is $J_{att}(S,P)$ where, from the information about

the base case, $\Sigma(B,S)$ $(= S(B))$ and $\Pi(B,P)$ $(= P(B))$. The other is $J_{obj}(T,S)$ where, from the information about the target case, $\Sigma(T,S)(= S(T))$.

Also, a candidate based on abduction + deduction is shown for a non-deductive inference system which can yield both conjectures.

4.2 Machine Learning of Logic Programs

Machine Learning is one of the most important themes in the area of artificial intelligence. A learning ability is necessary not only for processing and maintaining a large amount of knowledge information but also for realizing a user-friendly interface. We have studied the invention of new predicates is one of the most serious problems in learning logic programs. We have also investigated the application of minimally multiple generalization to the constructive learning of logic programs.

4.2.1 Predicate Invention

Shapiro's model inference gives a very important strategy for learning programs - an incremental hypothesis search using contradiction backtracing. However, his theory assumes that an initial hypothesis language with enough predicates to describe a target model is given to the learner. Furthermore, it is assumed that the teacher knows the intended model of all the predicates. Since this assumption is rather severe and restrictive, for the practical applications of learning logic programs, it should be removed. To construct a learning system without such assumptions, we have to consider the problem of predicates invention.

Recently, several approaches to this challenging and difficult problem have been presented [Muggleton and Buntine 1988], and [Ling 1989]. However, most of them do not give sufficient analysis on the computational complexity of the learning process, which is where the hypothesis language is growing. We discussed the problem as nonterminal invention in grammatical inference. As is well known, any context-free grammar can be expressed as a special form of the DCG (definite clause grammar) logic program. Thus, nonterminal invention in grammatical inference corresponds to predicate invention.

We have proposed a polynomial time learning algorithm for the class of simple deterministic languages based on nonterminal invention and contradiction backtracking[Ishizaka 1990]. Since the class of simple deterministic languages strictly includes regular languages, the result is a natural extension of our previous work[Ishizaka 1989].

4.2.2 Minimally Multiple Generalization

Another important problem in learning logic programs is to develop a constructive algorithm for learning. Most learning by induction algorithms, such as Shapiro's model inference system, are based on a search or enumerative method. While search and enumerative methods are often very powerful, they are very expensive. A constructive method is usually more efficient than a search method.

In the constructive learning of logic programs, the notion of least generalization [Plotkin 1970] plays a central role. Recently, Arimura proposed a notion of minimally multiple generalization (*mmg*) [Arimura 1991], a natural extension of least generalization. For example, the pair of heads in a clause in a normal append program is one head in the *mmg* for the Herbrand model of the program. Thus, *mmg* can be applied to infer the heads of the target program. Arimura has also given a polynomial time algorithm to compute *mmg*.

We are now investigating an efficient constructive learning method using *mmg*.

4.3 Logic Program Transformation / Synthesis

Automated programming is one important advanced inference problem. In researching automatic program transformation and synthesis, the unfold/fold transformation [Burstall and Darlington 1977, Tamaki and Sato 1984] is a well-known program technique to derive correct and efficient programs.

Though unfold/fold rules provide a very powerful methodology for program development, the application of those rules needs to be guided by strategies to obtain efficient programs. In unfold/fold transformation, the efficiency improvement is mainly the result of finding the recursive definition of a predicate, by performing folding steps. Introduction of auxiliary predicates often allows folding steps. Thus, invention of new predicates is one of the most important problems in program transformation.

On the other hand, unfold/fold transformation is often utilized for logic program synthesis. In those studies, unfold/fold rules are used to eliminate quantifiers by folding to obtain definite clause programs from first order formulae. However, in most of those studies, unfold/fold rules were applied nondeterministically and general methods to derive definite clauses were not known.

We have studied logic program transformation and synthesis method based on unfold/fold transformation and have obtained the following results.

(1) We investigated a strategy of logic program transformation based on unfold/fold rules [Kawamura 1991]. New predicates synthesized automatically to perform folding. We also extended

this method to incorporate goal replacement transformation [Tamaki and Sato 1984].

(2) We showed a characterization of classes of first order formulae from which definite clause programs can be derived automatically [Kawamura 1992]. Those formulae are described by Horn clauses extended by universally quantified implicational formulae. A synthesis procedure based on generalized unfold/fold rules [Kanamori and Horiuchi 1987] is given, and with some syntactic restrictions, those formulae successfully transformed into equivalent definite clause programs.

5 Conclusion

We have overviewed research and development of parallel automated reasoning systems at ICOT. The constituent research tasks of three main areas provided us with the following very promising technological results.

(1) **Parallel Theorem Prover and its implementation techniques on PIM**

We have presented two versions of a model-generation theorem prover MGTP implemented in KL1: MGTP/G for ground models and MGTP/N for non-ground models. We evaluated their performance on the distributed memory multi-processors Multi-PSI and PIM.

Range-restricted problems require only matching rather than full unification, and by making full use of the language features of KL1, excellent efficiency was achieved from MGTP/G.

To solve non-range-restricted problems by the model generation method, however, MGTP/N is restricted to Horn clause problems, using a set of KL1 meta-programming tools called the Meta-Library, to support the full unification and the other functions for variable management.

To improve the efficiency of the MGTP provers, we developed RAMS and MERC methods that enable us to avoid redundant computations in conjunctive matching. We were able to obtain good performance results by using these methods on PSI.

To ease severe time and space requirements in proving hard mathematical theorems (such as condensed detachment problems) by MGTP/N, we proposed the lazy model generation method, which can decrease the time and space complexity of the basic algorithm by several orders of magnitude. Our results show that significant saving in computation and memory can be realized by using the lazy algorithm.

For non-Horn ground problems, case splitting was used as the basic seed of OR parallel MGTP/G. This kind of problem is well-suited to MIMD machine such as Multi-PSI, on which it is necessary to make granularity as large as possible to minimize communication costs. We obtained an almost linear speedup for the n-queens, pigeon hole, and other problems on Multi-PSI, using a simple allocation scheme for task distribution.

For Horn non-ground problems, on the other hand, we had to exploit the AND parallelism inherent to conjunctive matching and subsumption. We found that good performance and scalability were obtained by using the AND parallelization scheme of MGTP/N.

In particular, our latest results, obtained with the MGTP/N prover on PIM/m, showed linear speed-up on condensed detachment problems, at least up to 128 PEs. The key technique is the lazy model generation method, that avoids the unnecessary computation and use of time and space while maintaining a high running rate.

The full unification algorithm, written in KL1 and used in MGTP/N, is one hundred times slower than that written in C on SPARCs. We are considering the incorporation of built-in firmware functions to bridge this gap. But developing KL1 compilation techniques for non-ground models, we believe, will further contribute to parallel logic programming on PIM.

Through the development of MGTP provers, we confirmed that KL1 is a powerful tool for the rapid prototyping of concurrent systems, and that parallel automated reasoning systems can be easily and effectively built on the parallel inference machine, PIM.

(2) **Applications**

The modal logic prover on MGTP/G realizes two advantages. The first is that the redundancy elimination and parallelization of MGTP/G directly endow the prover with good performance. The second is that direct representation of tableaux rules of modal logic as hyper-resolution clauses are far more suited to adding heuristics for performance. This prover exhibited excellent benchmark results.

The basic idea of non-monotonic and abductive systems on MGTP is to use the MGTP as an meta-interpreter for each system's special properties, such as nonmonotonic provability (*negation as failure*) and the *consistency* of abductive explanations, into formulae having a kind of modality such that MGTP can deal with them within classical logic. The extra requirements for these special properties are thus reduced to "generate-and-test" problems of model candidates that can be efficiently handled by MGTP

through the case-splitting of non-unit consequences and rejection of inconsistent model candidates.

We used MGTP for the application of program synthesis in two ways.

In one approach, we used *Realizability Interpretation*(an extension of *Curry-Howard Isomorphism*), an area of constructive mathematics, to give executable meaning to the proofs obtained by efficient theorem provers.

Two systems, MGTP and PAPYRUS, both developed in ICOT, were used for experiments on sorting algorithms to obtain practical insights into our approach. We performed experiments on sorting algorithms and Chinese Reminder problems and succeeded in obtaining ML programs from MGTP proofs.

To obtain parallel programs, we proposed a new logic μ, that is a constructive logic extended by introducing new operators μ and $\natural$. Operator μ is a fixpoint operator on formulae. We can express the non-bounded repetition of inputs and outputs with operators μ and $\natural$. Furthermore, we showed a method of synthesizing "program" as a parallel process, like CCS, from proofs of logic μ. We also showed the proof of consistency of Logic μ and the validity of the method to synthesize "program".

Our other approach to synthesize parallel programs by MGTP is the use of temporal logic, in which specifications are modeled by finite state diagrams, as follows.

1) Specification description language Ack, based on a transition system.
2) Graphical representation in Ack.
3) Ack interpreter by MGTP.

It is not necessary to describe all state transitions concretely using Ack, because several state transitions are deduced from one expression by theorem proving in temporal logic. Therefore, we can obtain a complete specification from an ambiguous one.

Another approach is to use term rewriting systems(Metis). MENDELS ZONE is a software development system for parallel programs. The target parallel programming language is MENDEL, which is a textual form of Petri Nets, that is translated into the concurrent logic programming language KL1 and executed on Multi-PSI.

We defined the decomposition rules for data-flow diagrams and subsequently extracted programs. Metis provides an experimental environment for studying practical techniques by equational reasoning, of implement, and test. The kernel function of Metis is the Knuth-Bendix (KB) completion procedure. We adopt Metis to verify the components of Petri Nets.

Only the skeletons of Petri Net structures are automatically retracted (slots and the KL1 codes of methods are ignored) since our verification and synthesis are applicable to a bounded net. The verification tool verifies whether Petri Nets satisfy given temporal logic constraints.

(3) Advanced Inference and Learning

To extend the reasoning power of AR systems, we have taken logical, computational, and empirical approaches.

In the logical approach, *analogical reasoning*, considered to be at the very core of human problem-solving, has been analyzed formally and a mechanism for analogical reasoning has been explored. In this approach, our objective was to clarify a general relationship between those analogical factors T, B, S and P under a given theory $\mathcal{A}$, as formally as possible. Determining the relationship between the analogical factors would answer these problems once and for all. We clarified the relationship and formulated a general solution for them all.

In the computational approach, we studied the *inventing of new predicates*, one of the most serious problems in the learning of logic programs. We proposed a polynomial time learning algorithm for the class of simple deterministic languages, based on nonterminal invention and contradiction backtracing. Since the class of simple deterministic languages includes regular languages, the result is a natural extension of our previous work. We have also investigated the application of *minimally multiple generalization* to the constructive learning of logic programs. Recently, Arimura proposed the notion of minimally multiple generalization (*mmg*) . We are now investigating an efficient constructive learning method that uses *mmg*.

In the empirical approach, we have studied automated programming, especially, the *logic program transformation and synthesis* method based on an unfold/fold transformation, a well-known means of deriving correct and efficient programs. We investigated a strategy for logic program transformation based on unfold/fold rules. New predicates are synthesized automatically to perform folding. We also extended this method to incorporate a goal replacement transformation.

We also showed a characterization of the classes of first order formulae, from which definite clause programs can be derived automatically. These formulae are described by Horn clauses, extended by universally quantified implicational formulae. A synthesis procedure based on generalized unfold/fold rules

is given, and with some syntactic restrictions, these formulae can be successfully transformed into equivalent definite clause programs.

These results contribute to the development of FGCS, not only in AI applications, but also in the foundation of the parallel logic programming that we regard as being the kernel of FGCS.

Acknowledgment

The research on automated reasoning systems was carried out by the Fifth Research Laboratory at ICOT in tight cooperation with five manufactures. Thanks are firstly due to who have given support and helpful comments, including Dr. Kazuhiro Fuchi, the director of ICOT, and Dr. Koichi Furukawa, the deputy director of ICOT. Many fruitful discussions took place at the meetings of Working Groups: PTP, PAR, ANR, and ALT. We would like to thank the chair persons and all other members of the Working Groups. Special thanks go to many people at the cooperating manufacturers in charge of the joint research programs.

References

[Fuchi 1990] K. Fuchi, Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Hasegawa *et al.*, 1990] R. Hasegawa, H. Fujita and M. Fujita, A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, in *Italy-Japan-Sweden Workshop '90,* ICOT TR-588, 1990.

[Fujita and Hasegawa 1990] H. Fujita and R. Hasegawa, A Model Generation Theorem Prover in KL1 Using Ramified-Stack Algorithm, ICOT TR-606, 1990.

[Hasegawa 1991] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan, in Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[Hasegawa *et al.*, 1992] R. Hasegawa, M. Koshimura and H. Fujita, Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers, ICOT TR-751, 1992.

[Koshimura *et al.*, 1990] M. Koshimura, H. Fujita and R. Hasegawa, Meta-Programming in KL1, ICOT-TR-623, 1990 (in Japanese).

[Manthey and Bry 1988] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, in *Proc. of CADE 88, Argonne, illinois,* 1988.

[Nitta *et al.*, 1992] K. Nitta, Y. Ohtake, S. Maeda, M. Ono, H. Ohsaki and K. Sakane, HELIC-II: a legal resoning system on the parallel inference machine, in *Proc. of FGCS'92*, Tokyó, 1992.

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

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

[McCune and Wos 1991] W.W. McCune and L. Wos, Experiments in Automated Deduction with Condensed Detachment, Argonne National Laboratory, 1991.

[Overbeek 1990] R. Overbeek, Challenge Problems, (private communication) 1990.

[Wilson and Loveland 1989] D. Wilson and D. Loveland, Incorporating Relevancy Testing in SATCHMO, Technical Report of CS(CS-1989-24), Duke University, 1989.

[Fitting 1983] M. Fitting, *Proof Methods for Modal and Intuitionistic Logic*, D.Reidel Publishing Co., Dordrecht 1983.

[Fitting 1988] M. Fitting, "First-Order Modal Tableaux", *Journal of Automated Reasoning*, Vol.4, No.2, 1988.

[Koshimura and Hasegawa 1991] M. Koshimura and R. Hasegawa, "Modal Propositional Tableaux in a Model Generation Theorem Prover", In *Proceedings of the Logic Programming Conference'91*, Tokyo, 1991 (in Japanese).

[Smullyan 1968] R.M. Smullyan, *First-Order Logic*, Vol 43 of Ergebnisse der Mathematik, Springer-Verlag, Berlin, 1968.

[Arima 1991] J. Arima, A Logical Analysis of Relevance in Analogy, in *Proc. of Workshop on Algorithmic Learning Theory ALT'91*, Japanese Society for Artificial Intelligence, 1991.

[Arima 1992] J. Arima, Logical Structure of Analogy, in *FGCS'92*, Tokyo, 1992.

[Kohda and Maeda 91a] Y. Kohda and M. Maeda, Strategy Management Shell on a Parallel Machine, IIAS RESEARCH Memorandum IIAS-RM-91-8E, Fujitsu, October 1991.

[Kohda and Maeda 1991b] Y. Kohda and M. Maeda, Strategy Management Shell on a Parallel Machine, in poster session of ILPS'91, San Diego, October 1991.

[Maeda *et al.*, 1990] M. Maeda, H. Uoi, N. Tokura, Process and Stream Oriented Debugger for GHC programs, Proceedings of Logic Programming Conference 1990, pp.169-178, ICOT, July 1990.

[Maeda 1992] M. Maeda, Implementing a Process Oriented Debugger with Reflection and Program Transformation, in *Proc. of FGCS'92*, Tokyo, 1992.

[Smith 1984] B.C. Smith, Reflection and Semantics in Lisp, Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, pp.23-35, ACM, January 1984.

[Sugano 1990] H. Sugano, Meta and Reflective Computation in Logic Programs and its Semantics, Proceedings of the Second Workshop on Meta-Programming in Logic, Leuven, Belgium, pp.19-34, April, 1990.

[Sugano 1991] H. Sugano, Modeling Group Reflection in a Simple Concurrent Constraint Language, *OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[Tanaka 1991] J. Tanaka, An Experimental Reflective Programming System Written in GHC, *Journal of Information Processing*, Vol.14, No.1, pp.74-84, 1991.

[Tanaka and Matono 1992] J. Tanaka and F. Matono, Constructing and Collapsing a Reflective Tower in Reflective Guarded Horn Clauses, in *Proc. of FGCS'92*, Tokyo, 1992.

[Arimura 1991] H. Arimura, T. Shinohara and S. Otsuki, Polynomial time inference of unions of tree pattern languages. In S. Arikawa, A. Maruoka, and T. Sato, editors, *Proc. ALT '91*, pp. 105–114. Ohmsha, 1991.

[Ishizaka 1989] H. Ishizaka, Inductive inference of regular languages based on model inference. *International journal of Computer Mathematics*, 27:67–83, 1989.

[Ishizaka 1990] H. Ishizaka, Polynomial time learnability of simple deterministic languages. *Machine Learning*, 5(2):151–164, 1990.

[Ling 1989] X. Ling, Inventing theoretical terms in inductive learning of functions - search and constructive methods. In Zbigniew W. Ras, editor, *Methodologies for Intelligent Systems, 4*, pp. 332–341. North-Holland, October 1989.

[Muggleton and Buntine 1988] S. Muggleton and W. Buntine, Machine invention of first-order predicates by inverting resolution. In *Proc. 5th International Conference on Machine Learning*, pp. 339–352, 1988.

[Plotkin 1970] G.D. Plotkin, A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pp. 153–163. Edinburgh University Press, 1970.

[Burstall and Darlington 1977] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", J.ACM, Vol.24, No.1, pp.44–67, 1977.

[Kanamori and Horiuchi 1987] T. Kanamori and K. Horiuchi, "Construction of Logic Programs Based on Generalized Unfold/Fold Rules", Proc. of 4th International Conference on Logic Programming, pp.744–768, Melbourne, 1987.

[Kawamura 1991] T. Kawamura, "Derivation of Efficient Logic Programs by Synthesizing New Predicates", Proc. of 1991 International Logic Programming Symposium, pp.611 – 625, San Diego, 1991.

[Kawamura 1992] T. Kawamura, "Logic Program Synthesis from First Order Logic Specifications", to appear in International Conference on Fifth Generation Computer Systems 1992, Tokyo, 1992.

[Tamaki and Sato 1984] H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs", Proc. of 2nd International Logic Programming Conference, pp.127–138, Uppsala, 1984.

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

[de Kleer 1986] J. de Kleer, An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[Fujita and Hasegawa 1991] H. Fujita and R. Hasegawa, A model generation theorem prover in KL1 using a ramified-stack algorithm. In: *Proceedings of the Eighth International Conference on Logic Programming* (Paris, France), pp. 535–548, MIT Press, Cambridge, MA, 1991.

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

[Inoue 1988] K. Inoue, Problem solving with hypothetical reasoning, in *Proc. of FGCS'88*, pp. 1275–1281, Tokyo, 1988.

[Inoue 1991a] K. Inoue, Extended logic programs with default assumptions, in *Proc. of the Eighth International Conference on Logic Programming* (Paris, France), pp. 490–504, MIT Press, Cambridge, MA, 1991.

[Inoue 1991b] K. Inoue, Linear resolution for consequence-finding, To appear in: *Artificial Intelligence*, An earlier version appeared as: Consequence-finding based on ordered linear resolution, in *Proc. of IJCAI-91*, pp. 158–164, Sydney, Australia, 1991.

[Inoue *et al.*, 1992a] K. Inoue, M. Koshimura and R. Hasegawa, Embedding negation as failure into a model generation theorem prover, To appear in *CADE 92*, Saratoga Springs, NY, June 1992.

[Inoue *et al.*, 1992b] K. Inoue, Y. Ohta, R. Hasegawa and M. Nakashima, Hypothetical reasoning systems on the MGTP, ICOT-TR 1992 (in Japanese).

[Loveland 1978] D.W. Loveland, *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[Ohta and Inoue 1990] Y. Ohta and K. Inoue, A forward-chaining multiple context reasoner and its application to logic design, in: *Proceedings of the Second IEEE International Conference on Tools for Artificial Intelligence*, pp. 386–392, Herndon, VA, 1990.

[Ohta and Inoue 1992] Y. Ohta and K. Inoue, A forward-chaining hypothetical reasoner based on upside-down meta-interpretation, in *Proc. of FGCS'92*, Tokyo, 1992.

[Poole *et al.*, 1987] D. Poole, R. Goebel and R. Aleliunas, Theorist: a logical reasoning system for defaults and diagnosis, In: Nick Cercone and Gordon McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pp. 331–352, Springer-Verlag, New York, 1987.

[Stickel 1991]
M.E. Stickel, Upside-down meta-interpretation of the model elimination theorem-proving procedure for deduction and abduction, ICOT TR-664, 1991.

[Constable *et al.*, 1986] R.L. Constable et al, *Implementing Mathematics with the Nuprl Proof Development System,* Prenticd-Hall, NJ, 1986.

[Hayashi and Nakano 1988] S. Hayashi and H. Nakano, *PX: A Computational Logic,* MIT Press, Cambridge, 1988.

[Harper *et al.*, 1987] R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, in *Symposium on Logic in Computer Science,* IEEE, pp. 194-204, 1987.

[Pfenning 1988] F. Pfenning, Elf: A Language for Logic Definition and Verified Meta-Programming, in *Fourth Annual Symposium on Logic in Computer Science*, IEEE, pp. 313-322, 1989.

[Takayama 1987] Y. Takayama, Writing Programs as QJ Proof and Compiling into Prolog Programs, in *Proc. of IEEE The Symposium on Logic Programming '87*, pp. 278-287, 1987.

[Howard 1980] W.A. Howard, "The formulae-as-types notion of construction", in Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, pp.479-490, 1980.

[Martin 1982] P. Martin-Löf, "Constructive mathematics and computer programming", in Logic, Methodology, and Philosophy of Science VI, Cohen, L.J. et al, eds., North-Holland, pp.153-179, 1982.

[Sato 1986] M. Sato, "QJ: A Constructive Logical System with Types", France-Japan Artificial Intelligence and Computer Science Symposium 86, Tokyo, 1986.

[Milner 1989] R. Milner, "Communication and Concurrency", Prentice-Hall International, 1989.

[Honiden *et al.*, 1990] S. Honiden et al., An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, in *The Journal of Real-Time Systems*, 1990.

[Honiden *et al.*, 1991] S. Honiden et al., An Integration Environment to Put Formal Specification into Practical Use in Real-Time Systems, in *Proc. 6th IWSSD*, 1991.

[Ohsuga *et al.*, 91] A. Ohsuga et al., A Term Rewriting System Generator, in *Software Science and Engineering*, World Scientific ,1991.

[Ohsuga *et al.*, 90] A. Ohsuga et al., Complete E-unification based on an extension of the Knuth-Bendix Completion Procedure, in *Proc. of Workshop on Word Equations and Related Topics*, LNCS 572, 1990.

[Uchihira *et al.*, 90a] N. Uchihira et al., Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, in *Proc. of 23th HICSS*, 1990.

[Uchihira *et al.*, 90b] N. Uchihira et al., Verification and Synthesis of Concurrent Programs Using Petri Nets and Temporal Logic, in *Trans. IEICE*, Vol. E73, No. 12, 1990.