ICOT Technical Report: TR-161

TR-161

A Theorem Prover based on Connection Graph and its Implementation by Prolog

by
Toshiro Minami
(Fujitsu Ltd.)

March, 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Theorem Prover based on Connection Graph and its Implementation by Prolog

by

Toshiro MINAMI

Associate Research Staff, Fundamental Informatics Section, International Institute for Advanced Study of Social Information Science, Fujitsu Limited.

ABSTRACT

This paper presents a theorem prover for the first-order predicate logic using connection graph introduced by Kowalski.

The theoretical basis of the prover is taken from the Sickel's "clause interconnectivity graph" method, which is one of the connection graph method. The theorem prover has an additional feature about search strategies, namely static and dynamic. This distinction is made in expectation of increasing the overall efficiency of the prover.

This paper also shows how the prover has been implemented. The implementation language of the prover is Prolog, where distinctive features such as unification and backtracking are fully utilized. The prover consists of four steps; (1) translation from an input formula to the clausal form, (2) conversion from the clausal form to the connection graph, (3) solution searching using the connection graph, and (4) execution of the actual resolutions. In step (1), we need only two sub-steps by using the unification feature in Prolog, while the ordinary algorithm will require six sub-steps to realize this function. Consequently, step (1) requires less computational time.

1. Introduction

Robinson's resolution principle [Robinson 65] is one of the most important methods for the automated theorem proving. This method is more efficient than those which execute the procedures based on the Herbrand's theorem directly. But it is not efficient enough, for the unrestricted applications of it may generate many irrelevant clauses. Therefore many refinements of the resolution principle have been proposed to increase the efficiency, such as semantic resolution, locking resolution, linear resolution, unit resolution, input resolution, and set-of-support resolution, etc. (see, e.g. [Chang 73],[Loveland 78]).

There are some refinements which use graphs to get some informations for planning the resolutions. Kowalski [Kowalski 75] proposed a refinement using a graph, which is called the connection graph. According to Kowalski, a graph which is called the initial connection graph is constructed first, and then in the proof process some new nodes and links are added, and some nodes and related links are deleted. The process terminates in two cases. The first one is the case when the empty clause, the clause which has no links, is created as a new node, which means the resolution process succeeds. The other one is the case when no links are left in the graph so that it is impossible for the process to go further, which means the process fails to find the solution. Sickel proposed another method based on the graph, which she called the clause interconnectivity graph (CIG). CIG is basically the same to the Kowalski's initial connection graph, and it will also be called the connection graph in this paper. The difference from Kowalski's method is that the connection graph is created only once. It is used only for the search of the refutation path, and the tree structure which represents the search paths, called the solution tree, is created in the search process. The solution tree represents the information how to execute the resolutions to get the empty

clause by the resolution process.

The purpose of this paper is to develop a resolution-based first-order theorem prover. The prover takes the Sickel's method as its theoretical basis. We do not take the Kowalski's method because it seems to require much space and time to create connection graphs during the search process. The most important difference of the prover from Sickel's is that it provides two kinds of strategies. They are called static strategy and dynamic strategy. The former is the strategy applied in the process making links of the connection graph, while the latter is the strategy used when the solution search process traverses the graph.

We also show how to implement the first-order theorem prover based on the method mentioned above. As the implementation language, Prolog [Clocksin 81] is used. This language has convenient features, such as unification or backtracking, to the implementation of a theorem prover.

The rest of the paper is organized as follows : In Section 2 we describe some of the search strategies, which determine the selection of the unifiable complementary literals. Each strategy has two aspects, namely, static and dynamic. In Section 3 we describe how to implement the theorem prover. It consists of four steps, translating an input into the clausal form, generating the connection graph, solution searching, and the resolution tree printing. And in Section 4, concluding discussions are made.

2. Search Strategies for Connection Graph Method

Search strategies strongly affect the efficiency of theorem provers. In this section we will discuss about search strategies, especially about those of connection graph methods. We call an algorithm which determines how to

search the solution a search strategy. There have been proposed many strategies (see [Chang 73], [Loveland 78], and [Sato 81]) as the refinements of the resolution principle, as listed below :

(1) Deletion Strategies

Any strategy which deletes clauses, such as subsumed clauses, tautologies, and those which contain pure literals, is called deletion strategy.

(2) Restriction Strategies

Any strategy which restricts the application of resolution strategies to some fixed forms is called restriction strategy. Linear, semantic, and locking resolutions are the typical restriction strategies.

(3) Preference Strategies

Any strategy which determines the preference order of the paths to be searched is called preference strategy. Unit-clause preference is such a strategy.

There are some other strategies which do not belong to the above classes. For example, abstraction strategy [Plaisted 81] is such a kind of strategy.

[Note] We can combine some strategies to get a new one. For example, a combined strategy of deletion and preference strategies is included both in (1) and (3). Connection graph methods allow such combinations of strategies.

We, in this paper, note that a strategy for the connection graph method has two aspects, namely static and dynamic. The static aspect is the part of the strategy which decides which unifiable complementary literals are to be linked and also decides in what order the links should be arranged in the process constructing the connection graph for the given problem. These

decisions are made before the search process starts; so it is called static. The dynamic aspect is the part of the strategy which decides which links are to be searched and also decides in what order the links should be searched in the process which searches the solution of the problem represented by the connection graph. These decisions are made during the search process of the solution and the environment is changing during the process; so it is called dynamic.

As the order of static strategies, we take an inclusion order of the sets of the allowable links of the strategies. The maximum strategy in this order has the following property : Any search strategy can be represented by combining the maximum static strategy and some dynamic strategy. This is because every pair of literals which may be used for the resolution is connected in this maximum strategy, therefore the dynamic strategy can select any such pairs. On the other hand, for any fixed dynamic strategy, it is not possible to prove every problems by any static strategies. Therefore, with respect to descriptive power of the strategies, dynamic strategy is superior to static one.

However when we think about the time-consumption of these two aspects of strategies, the static one is desirable. This is because the static strategy is invoked only once for each of the unifiable complementary literals of the clause set which represents the problem, while the dynamic strategy is invoked every time when the search process has to decide which link is to be chosen. Therefore the time-consumption of the static strategy is relatively small compared with that of the dynamic strategy if the problem is fairly complicated and the search process takes many steps to get a solution.

By these considerations, it is reasonable to describe a strategy by combining the static and dynamic strategies :

(i) Find the static aspect of the strategy. This is a part of the strategy that can be decided by the structure of the clause set. It is

described as the static strategy.

(ii) And the rest of the strategy is to be described as the dynamic strategy.

Now we will see the static and also dynamic aspects of some of the strategies. As Kowalski [Kowalski 79-1] pointed out, top-down and bottom-up search strategies can be expressed by the static strategies. Some parts of the strategies mentioned earlier in this section can be represented by the static strategies, and the rest by the dynamic one.

(1) Deletion strategies can be represented as static strategies to some extent. When applied to the input clauses, these strategies can be represented by the static strategy. For the clauses created during the resolutions, there are no ways to apply the static strategies. That is, the dynamic strategies are required.

For the deletion strategy for the subsumed clauses, when a clause A subsumes another clause B in the clausal form, we delete all links connected to B, which is equivalent to deleting clause B itself. This is the static aspect of the deletion strategy.

A tautology link is a link which creates a tautology as the resolvent corresponding to it. Some of the links are determined as tautology links by the form of the literals. Deletion of such tautology links is the static aspect and is implemented in the prover. Some other links, on the other hand, have the possibility to create the tautologies when they are instantiated. These are called the potential tautology links. A potential tautology link may create a tautology, which depends on the environment of the value assignments to the variables. The environment is not known in the process of generating the connection graph.

The strategy which deletes all the pure literals can be represented completely as the static strategy. It is implemented in such a way that the clauses which include literals with no links are deleted.

(2) Restriction and deletion strategies are almost the same in static and dynamic aspects. This is because a restriction strategy deletes some search paths, while a deletion strategy deletes clauses, which is equivalent to deleting all the paths which connect these clauses. In other words, they make some decisions whether a path should be deleted or not, from the information about the path.

[Note] Some of the strategies, such as locking resolution, require extra information about links. In this case we have to extend the representation of the connection graph so that it includes such information.

(3) Preference strategies, such as unit-clause preference, can be represented as static strategies, as far as they use only the static information in determination of the preference. Unit-clause preference is one of such strategies, since whether or not a clause is unit is decided by a static information, namely the form of the clause. This strategy is implemented in the current version of the prover.

[Note] Since the links of a literal of the connection graph are represented by a list of links, they have natural orders in their representations. This order can be interpreted as that it represents the preference of the connection links. Thus any preference

strategy can be represented in our connection graph by making the order of the links reflecting the preference. However the search process is free to use this order. Therefore, as the total strategy about the preference of the links, the dynamic strategy determines the final preference order of the links.

3. Implementation

We give a description of an implementation of the prover in this section. The implementation language is Prolog (see, for example, [Clocksin 81]). Prolog has special features such as unification and backtracking, and it is very suited for implementing theorem provers.

The prover consists of four major steps of processes shown as follows :

(1) Transformation of an input formula to the clausal form.
(2) Create the connection graph from the clausal form.
(3) Find the solution by searching the connection graph.
(4) Print the result as the resolution tree.

Each of the steps will be described in the subsections from 3.2 to 3.5. First, in the next subsection, we will show how the data used in the prover are represented.

3.1. Data Representations

Data in the prover are represented as follows :

(i) A variable is represented by the Prolog variable, a constant by the Prolog constant, and a function by the Prolog functor.

(ii) An atomic formula is represented by the Prolog term of a functor followed by a parenthesized arguments.

(iii) Logical formulas are represented as follows :

A and B	conjunction
A or B	disjunction
A => B	implication
A <= B	(equivalent to B => A)
A <=> B	logical equivalence
~ A	negation
all(X,A)	universally quantified formula
some(X,A)	existentially quantified formula

(iv) A clause is represented by

$a_1,a_2,\ldots,a_n$ <- $b_1,b_2,\ldots,b_m$.

This formula has the equivalent meaning to

b_1 and b_2 and ... and b_m => a_1 or a_2 or ... or a_n.

A clausal form (clause set) is represented by the list of the representations of the clauses.

(v) A connection graph is represented by a list of linked clauses, where a linked clause is represented by the following format :

<clause name>: [<linked literal>,...] <- [<linked literal>...]

Here a linked literal is represented by

<literal name>: <literal>, [<link>,...] ,

and a <link> is represented by

<clause name> - <literal name>

which indicates the link to which the literal is connected.

Here is an example of formulas printed by the prover.

[Example-1]

```
================  problem  ================
conclusion: X=<X
   premise: X=<Y<=>all(Z,elm(Z,X)=>elm(Z,Y))
```

The premise part of this problem indicates the definition of the set-order, i.e. inclusion, and the conclusion part indicates that a set is a subset of itself.

3.2. Translating Input Formula to Clausal Form

This step translates each of the input formulas, the premise and the conclusion, to the clausal form. Before translating the conclusion, it is changed to the closed formula, and its negation is translated. This pre-translation is necessary because the free variables of the conclusion part must be existentially quantified. The premise part is translated directly. After the translation it appends them into a clause set, which would be used by the next step in sub-section 3.3. If an empty clause is included in the clause set, the prover does not proceed any further; it terminates successfully. This is the case when the premise does not hold or when the conclusion holds without the premise.

Ordinarily the translation consists of several steps, namely translating the formula to the closed formula, expanding the implication or equivalence and moving the outer negation to internal, renaming the conflicting variable names, moving the internal quantifiers to the outermost part of the formula, introducing the Skolem functions, and lastly translating the formula to the

clausal form.

In the prover of this paper, we take a new algorithm to implement the translation. Owing to the unification feature of Prolog, it can be realized in two steps, namely the transformation and assignment of Skolem functions, as described as follows :

(i) Input formula is translated into the corresponding clausal form. The arguments for the Skolem-functions in the clausal form can not be determined during the translation. So, the Skolem-functions are represented by the corresponding variables. These variables are assigned to the actual Skolem-functions in the next step.

(ii) After the first step, the free variables in the input formula are obtained. In this step, the actual forms of the Skolem-functions are created, and they are assigned to their corresponding variables in the clausal form obtained in the first step.

[Note] If the inputs are restricted to closed formulas, only the first step is required to translate them to the clausal form.

Since the method of the prover is different from the ordinary method, it will be necessary to explain more about the translation algorithm. The algorithm consists mainly of the translation rules of the formula to the clausal form. Some of the translation rules used in the first step (i) are :

(1) If the formula A is an atom, then it is translated to

A <- .

If A contains free variables, they are added to the free-variable list.

(2) Logical constant

true

is translated to the empty set of clauses, while

false

is translated to the empty clause :

<- .

(3) Formulas of the form

all(X,F) and all(X,G),

some(X,F) or some(X,G),

A <= B,

A => B,

A <=> B

are translated respectively to the results of the translations of

all(X, F and G),

some(X, F or G),

~B or A,

~A or B,

(~A or B) and (A or ~B).

(4) A formula of the form

~A

is translated to the result of the "negative-translation" of "A". Where negative-translation rules translate a formula "A" to the clausal form which represents the negation of "A". For example, we get the following clausal form :

<- A

as the result of the negative-translation of an atom "A".

(5) A formula of the form

A and B

is translated as follows :

(i) Translate A to the clausal form A'.

(ii) Translate B to the clausal form B'.

(iii) Make the union of A' and B'. If the same clauses appear in both A' and B', then one clause is left, and the other is deleted.

Since a clausal form is a set of clauses, which semanically means the conjunction of its clauses, the conjunctive operation for clausal forms is represented by the set-union of the forms. Since a set is represented by the list of clauses in the prover, the operation is the set-append of the lists of clauses.

(6) A formula of the form

A or B

is translated as follows :

(i) Translate A to the clausal form A'.

(ii) Translate B to the clausal form B'.

(iii) Make the "or-union" of A' and B'.

Here "or-union" is an operation defined as follows :

The or-union of the clauses

$A_1,\ldots,A_n$ <- $B_1,\ldots,B_m$ and

$C_1,\ldots,C_p$ <- $D_1,\ldots,D_q$

is

$A_1,\ldots,A_n,C_1,\ldots,C_p$ <- $B_1,\ldots,B_m,D_1,\ldots,D_q$.

This is because the clauses above is equivalent to the formulas

A_1 or ... or A_n or $\tilde{}B_1$ or ... or $\tilde{}B_m$ and

C_1 or ... or C_p or $\tilde{}D_1$ or ... or $\tilde{}D_q$

respectively, so their disjunction is

A_1 or ... or A_n or C_1 or ... or C_p

or $\tilde{}B_1$ or ... or $\tilde{}B_m$ or $\tilde{}D_1$ or ... or $\tilde{}D_q$.

For the clause sets

$M = \{M_1,...,M_m\}$ and

$N = \{N_1,...,N_n\}$,

the or-formula

M or N

represents the formula

(M_1 and ... and M_m) or (N_1 and ... and N_n) ,

which is equivalent to the formula

(M_1 or N_1) and ... and (M_1 or N_n) and

..... and

(M_m or N_1) and ... and (M_m or N_n).

That is, this is the conjunction of all or-unions of the combinations of two clauses one of which is in M and the other in N. Therefore, the clause set of "M or N" is :

$\{or(M_1,N_1),...,or(M_1,N_n),...,or(M_m,N_1),...,or(M_m,N_n)\}$,

where "$or(M_i,N_j)$" is the or-union of the clauses M_i and N_j.

(7) The universally quantified formula

all(X, F)

is translated to the result of the translation of the subformula "F". In this translation the bound-variables list for "F" is obtained by adding the variable "X" to the bound-variables list for "all(X,F)".

The bound-variables list is used to know if a variable is free or is bounded by a universal quantifier. If a variable is not in the bound-variables list, then it is considered to be a free variable, and is added to the free-variable list. It is also used when translating the formula of the form "some(X,F)". See the case (8) for this form of translation.

(8) The translation of the form

some(X,F)

is done as follows :

The translation is applied to the subformula "F" in the following context;

the variable X is added to the rename list, whose member is of the form "X-XX", where "XX" is the new name for "X". In addition to it, "XX-AV" is added to the Skolem list, which indicates "XX" is a variable to be instantiated later to some Skolem function, and its arguments include the variable list "AV". "AV" is the bound-variables list, which is the list of the variables which appear as bound variables of its universal quantifiers. The final arguments of the Skolem functions consist of the variables in "AV" and free variables of the input formula. The result of the translation of "F" in such a context is the result of translating the formula "some(X,F)".

[Note] It is noted in (4) that we have another set of rules, i.e., negative translation rules, which correspond one to one to the above rules (1)-(8). The descriptions, however, are mostly repetitive and thus omitted.

We also note that, in the case of (1), a variable "X" in "A" is, in fact, renamed to "XX" if "X-XX" appears in the rename list of the translator.

Now we consider the second step (ii). The algorithm of this step is as follows :

When step (i) is finished, we obtain three kinds of information; clausal form which includes variables in the position of Skolem functions, the list of variables corresponding to the Skolem functions with the corresponding bound variables, and the list of free variables used in the clausal form. For each variables corresponding to the Skolem functions, we make a new function structure. The functor name of the structure is created as a new name, which is preceded by "sk" to indicate that it is a name of the Skolem function. The argument of the structure consists of the member of the bound variables and by the free variables. Each of these Skolem functions are assigned to the corresponding variables in the clausal form.

The following is an example printed in the translation step of the prover. Negation of conclusion and premise are translated independently.

[Example-2]

```
    ========= translation to clausal form =========
clausal form (conclusion): 0.150002s
[ 1]      <- sk01=<sk01
clausal form (   premise): 0.650002s
[ 1]  X=<Y <- elm(sk02(X,Y),Y)
[ 2]  Z=<A,elm(sk02(Z,A),Z) <-
[ 3]  elm(B,C) <- D=<C,elm(B,D)
```

These are the clausal forms obtained from the problem in Example-1 by applying this step. The conclusion part is translated to one clause, while the premise part is translated to three clauses. The function beginning with "sk" is a Skolem-function. First, we will explain the translation process by the premise part of Example-2. The logical formula "X=<Y <=> all(Z,elm(Z,X)=>elm(Z,Y))" is given to the translation procedure. The procedure recognizes that the formula is of the form "A <=> B". From the definition of

"<=>", "A<=>B" is equivalent to "(~A or B) and (A or ~B)". So the problem is to translate the formula "(~(X=<Y) or all(Z,elm(Z,X)=>elm(Z,Y))) and (X=<Y or ~all(Z,elm(Z,X)=>elm(Z,Y)))". This formula is of the form "A and B". In this case, the translation rule is: translate both A and B, and join the result. The first part of the formula is "~(X=<Y) or all(Z,elm(Z,X)=>elm(Z,Y))". It is of the form "A or B". The translation rule of this form is: translate both A and B, and combine the result. The first part of this formula is "~ X=<Y". This is translated to the clause "<- X=<Y". The free variables of this formula are X and Y. The second part of the or-formula is "all(Z,elm(Z,X)=>elm(Z,Y))". This is translated to "elm(Z,Y) <- elm(Z,X)". By combining "<- X=<Y" and "elm(Z,Y) <- elm(Z,X)", we get the clause "elm(Z,Y) <- X=<Y,elm(Z,X)". This is the result of the translation of "~(X=<Y) or all(Z,elm(Z,X)=>elm(Z,Y))". This result is the third clause of the premise part of Example-2. The first and the second clauses for the premise part come from the second part of the and-formula.

Since the formula given as the conclusion is an atom "X=<X", the negation of its closure is "~all(X,X=<X)". By applying step (i), it is translated to "<- XX=<XX", and we get the variable list "[XX-[]]" for Skolem functions and the empty list for free variables. By these data, we create a Skolem function name "sk01", which has no arguments. Then we assign the function "sk01" to the variable "XX" in the closed formula, and obtain "<- sk01=<sk01" as the result of translation of the conclusion.

It should be noted that the decreasing of the number of steps, i.e., from 6 steps in ordinary method to 2 steps in this paper, results in the decrease of translation time. In our experience, the new algorithm is about two times faster than the ordinary one.

3.3. Converting the Clausal Form to the Connection Graph

The representation of the connection graph of the prover is different from Sickel's. All the clauses and literals are uniquely named, and to each literal a list of links to the unifiable complementary literals is added. A link is represented by a pair of clause name and literal name. Each clause with the name and link information is stored separately in the Prolog database. During the process, for each positive (or negative) literal the links to the unifiable negative (or respectively positive) literals are also added. Some of the links are deleted by the static strategy. As the default deletion strategy, irrelevant clauses, i.e. those including the literal which has no links to any of the literals which are not included in the clause, are deleted. These clauses are of no use to find the solution, for such a literal mentioned above has no way to be resolved. Furthermore if such clauses exist, we may spend much time in the irrelevant searches for the solution of the literals in the clause. Moreover, the tautology links, which always generate tautology clauses (i.e. clauses which contain the same literals as both the positive and the negative literals) are also deleted.

By the bottom-up (or top-down) strategy, the link from the negative (or respectively positive) literal to the positive (or respectively negative) one is deleted. These two strategies have been implemented. Other strategies can be implemented easily. The most important problem we have now is how to find the useful strategies.

The following example is the result of generating the connection graph corresponding to the clausal form shown in Example-2.

[Example-3]

```
    ========== create connection graph ==========
connection graph : 1.83333s

clause literal literal_body          connection_links
```

```
-------------------------------------------------
[c01]   101     sk01=<sk01
                                ==>  c02-102  c03-104
[c02]   102     X=<Y
                                ==>  c01-101  c04-107
        103     elm(sk02(X,Y),Y)
                                ==>  c03-105  c04-106
[c03]   104     Z=<A
                                ==>  c01-101  c04-107
        105     elm(sk02(Z,A),Z)
                                ==>  c02-103  c04-108
[c04]   106     elm(B,C)
                                ==>  c02-103  c04-108
        107     D=<C
                                ==>  c02-102  c03-104
        108     elm(B,D)
                                ==>  c03-105  c04-106
```

The bracketed names are the clause-names, while the names beginning with "l" are the literal names. They are unique in the clause set. The list of clause names and literal names connected by "-" followed by the arrow "=>" indicates the connection link. Note that there are no indications whether the literal is positive or negative. This is because only the link information is used in the search process.

3.4. Solution Search

During the search process of the solution, an environment graph is constructed, which represents the state of the value assignments at that time. When a new instance of a clause is needed, the clause is obtained from the Prolog database and is instantiated by the appropriate value assignments to the variables. It may be needed to assign values to the variables in the environment graph, at that time. And the instantiated clause is added to the environment graph to get the new environment of unification. At the same time, the search tree for the solution is constructed to keep the unification path to the solution. It is used later when the search process succeeds and the resolution process is called. In the current version, the environment graph

is used only for the search process. In the resolution step, only the search tree is used and the unification environment is reconstructed by the actual resolution process. The success of the resolution process is guaranteed since the search process has been terminated successfully.

Solution search process is organized as follows :

(1) The step selects one clause from the clause set as the start clause. The list of all literals in the start clause is given to step (2). If step (2) terminates with failure, then this step selects another clause. If no clauses are left, then whole of the solution search process terminates with failure. If step (2) terminates with solution tree, then step (1), and consequently whole of the search process, terminates successfully. The solution tree is given to the resolution process to be described in sub-section 3.5.

(2) A list of literals is given to this step. This step calls step (3) repeatedly to find a solution tree for each of the literals. If one of the literals has no solutions, then step (2) terminates with failure. The order of selecting a literal in the list depends on the dynamic strategy of the prover.

(3) A literal is given to this step. This step searches a solution tree which represents a refutation process of the literal. More precisely, a link of the literal is chosen by the dynamic strategy of the prover. The link, in effect, indicates a clause and one of its literal. Then the clause body is taken from the Prolog database, and appropriately instantiated by unification. If the unification fails for this link, we choose another link of the literal under consideration by Prolog backtracking. Here, if no links are left, step (3) terminates with failure. If the unification succeeds, the step makes a list of the residual literals of the link. Here a residual literal of the link is a literal in the clause other than the one connected by the link. Step

(2) is recursively called with the list of the residual literals to search for the solution. Step (3) successfully terminates in two cases. The one is when the literal has no residual literals, and the other one is the case when the link forms a merge loop (described below).

As noted, the dynamic strategy is invoked in selecting a link for a new search. The algorithm which takes the order of the connection links as the preference of the links is implemented as the dynamic strategy.

[Note] In the above process, we may encounter two kinds of loops, namely merge loops and tautology loops. A merge loop is a path which begins with a literal and terminates with the same literal of the same instance of the clause in which it is included. A tautology loop is a loop which comes back to the same clause which has been appeared previously, but not to the same literal of the clause. Merge loops and tautology loops are tested each time when a new link is chosen. A merge loop corresponds to the derivations of some of the input resolutions and factoring, and is also considered to be a termination of the search in the current path. On the other hand, since tautology loop does not contribute to the solution search, we simply reject to use it.

In the next example, we will show how the prover searches the connection graph given by Example-3 and finds a solution. We will also show the solution tree obtained by this solution search process.

[Example-4]

```
    ========== solution search ==========
------------------- start clause  c01 : 101 -------------------
 1  c01-101=>c02-102
          new : 102  sk01=<sk01
                103  elm(sk02(sk01,sk01),sk01)
  2  c02-103=>c03-105
```

```
          new :  104   sk01=<sk01
                 105   elm(sk02(sk01,sk01),sk01)
  3  c03-104=>c01-101
  ***    merge loop   ***
```

This output shows us how the search proceeds in the solution search process. The search begins at the literal "101" in the clause "c01". First the search uses the link from the literal "101" to the literal "102". Since no clauses named "c02" are found in the current environment graph, a new clause instance is created. Next, it uses the link between "103" and "105", and creates a new clause. In the last step, the process reaches the literal "101". The literal is the same as the literal of the same instance in the start clause. Therefore, a merge loop is detected and the process terminates. In this case no backtracking occurs.

```
------------ solution tree ------------
c01-101 => c02-102          ** merge loop (head) **
           c02-103 => c03-105
                      c03-104 => c01-101   ** merge loop (tail) **
```

This indicates the tree structure of the final solution. In this case the solution is of the form of a single merge loop.

Since we use Prolog, it becomes very easy to implement the selection of alternatives. Backtracking is essentially required in such processes.

3.5. Resolution

This step displays the result of the solution which was found in the previous step as the resolution trees. This step is guaranteed to terminate since the previous step, which holds the unification environment of the resolution, terminates successfully.

The solution tree (or sub-tree) which does not include merge loops corresponds to the input resolutions. For example, let the solution tree be of the following form :

```
(1)      L1 => L2
(2)            L3 => L4
(3)                  L5 => L6
(4)            L7 => L8
```

The corresponding resolution is as follows :

```
 <- L1
    L2, L3 <- L7
--------------------- (1)  (L1 and L2 are unified.)
        L3 <- L7
  L5 <- L4
--------------------- (2)  (L3 and L4 are unified.)
  L5 <- L7
     <- L6
--------------------- (3)  (L5 and L6 are unified.)
     <- L7
  L8 <-
--------------------- (4)  (L7 and L8 are unified.)
     <-
```

For the merge loops, the next example will show how the corresponding resolutions are made.

[Example-5]

```
    ======= resolution tree print ======

----- resolution tree -----

[c01]      <- 101[sk01=<sk01]
[c02]            102[X=<Y] <- 103[elm(sk02(X,Y),Y)]
[c03]                         104[X=<Y],105[elm(sk02(X,Y),X)] <-
                 ------  unify  103 = 105  ----------------------
[n01]            104[X=<X],102[X=<X] <-
                 ------  factoring  102 = 104  ------------------
[n02]            102[X=<X] <-
       ------  unify  101 = 102  --------------------------------
[n03]      <-

resolved : 1.43334s
```

Following to the solution tree obtained by the previous step, the actual resolution process is displayed. There are three links in the merge loop.

Therefore there is only one intermediate link except for the terminal two links. Corresponding to the intermediate link, there is an input resolution which unifies the links "103" and "105". After this resolution two literals are left, namely the literals "104" and "102". These literals correspond to the two terminal links of the merge loop, namely "c01-101 => c02-102" and "c03-104 => c01-101". By the factoring which unifies these two literals, we get a literal which is the complementary literal to the terminal literal of the merge loop. By the resolution corresponding to the terminal link "c01-101 => c02-102" (or it is equivalent to the resolution corresponding to "c03-104 => c01-101", since the factoring unifies the literal "102" and "104"), the literal of the name "101" is erased. In this example, since the start clause includes only one clause "101", the empty clause is created, and the resolution process terminates.

The prover has a feature called the "answer" predicate. By the actual resolution displayed by the prover, we can get all the process of the value assignments to the variables. When we want to know only the values of some specific variables, the result displayed to us is too complicated. The answer predicate is provided for such an objective. For example :

The problem given by

```
prove(some(X,p(X) and ans(X)), p(a))
```

makes the following print.

```
-------- value(s) of ans-predicate(s) ------
= a
```

This indicates that the final solution is found when the value of the variable X, which appears in the input as an argument of the ans(wer) predicate, is "a". We can use more than one ans predicates. In this case, if we use the ans predicate as, for example :

```
ans(argument_of_p=X),
```

the meaning of the printed value becomes clear.

4. Conclusions

In this paper we have at first discussed about the strategies of the solution search for the first-order predicate logic theorem provers based on the resolution principle.

The prover which uses the connection graph in the solution search process has two kinds of strategies, namely static and dynamic. By the static strategies we can constrain the space to be searched by subsequent dynamic strategies. To be more specific, some of the links may be deleted by way of the static strategies, so that these links can never be used during the solution search process.

For each pairs of unifiable complementary literals, the static strategy procedure is invoked only once, so that the time consumption is relatively small in the total process. The dynamic strategies, on the other hand, would be invoked many times during the search, and the relative contribution to the total time consumption of the proof process is large. But in principle, any strategies representable by the static one can also be represented by the dynamic one, and the convese is not true. From this observation about the static and dynamic strategies, we conclude that the more the stategies are described as static one, the better, provided that it is realizable at all.

We also described an implementation method of the prover. Two formulas representing the problem are given, namely, the conclusion and the premise. These formulas are translated to the clausal form, and then a connection graph is generated. Some links between unifiable complementary literals are deleted by the static strategies. After the connection graph has been obtained, the solution search process begins. This process searches the connection graph to obtain a solution tree. During the process, an environment graph which represents the assignments of the variables in the actual

resolution process is produced. Such environment graph is used to decide whether a link of the connection graph actually corresponds to the resolution. If the search process terminates successfully, the resolution process is invoked and it prints the actual resolution tree. In this process, the solution tree is used, together with the clausal form saved in the database. The prover also has the so-called answer-predicate feature. The arguments of the "ans" predicates are displayed and the final values of their variables are shown to the user.

Using the special features in Prolog, such as unification and backtracking, the implementation became easier, and the program became more efficient. For example, in translation process, we used a data structure using unification to improve the efficiency of the process, and search process was easily implemented by using the backtracking feature of Prolog.

For the further improvement of the prover, the investigations of the strategies appropriate to problems in some specific domain would be most important. The followings are some of the problems left for the future research :

(1) Make some qantitative evaluation of the prover.

(2) Introduce a strategy description language, and let the user to express the strategies easier.

(3) Introduce "type" to increase the naturalness of the expression of the formula given to the prover.

Acknowledgements

The author would like to acknowledge the encouragement of Dr. Tosio Kitagawa, the president of his institute and Dr. Hajime Enomoto, the director of his institute. He also acknowledge to his colleagues, Hiroyuki Kano and Hajime Sawamura, for their useful suggestions.

This work is part of a major R & D project of the Fifth Generation Computer, conducted under a program set up by the MITI.

References

[Bibel 82] Bibel, W. : Automated Theorem Proving, Vieweg, 1982.

[Chang 73] Chang, C.L. ,Lee, R.C.T. : Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

[Clocksin 81] Clocksin, W.F. , Mellish, C.S : Programming in Prolog, Springer-Verlag, 1983.

[Kowalski 75] Kowalski, R. : A Proof Procedure based on Connection Graphs, JACM, 1975.

[Kowalski 79-1] Kowalski, R. : Logic for Problem Solving, Artificail Intelligence Series 7, North Holland, 1979.

[Kowalski 79-2] Kowalski, R. : Algorithm = Logic + Control, CACM, Vol.22, No. 7, July 1979.

[Loveland 78] Loveland, D.W. : Automated Theorem Proving, North-Holland, 1978.

[Plaisted 81] Plaisted, D.A. : Theorem Proving with Abstraction, Artificial Intelligence 16, 1981.

[Robinson 65] Robinson, J.A. : A Machine-Oriented Logic based on the Resolution Principle, JACM, Vol. 12, 1965.

[Sato 81] Sato, T. : Resolution-Based Theorem Proving, Journal of Information Processing Society of Japan, Vol. 22, No. 11, 1981, (in Japanese).

[Sickel 76] Sickel, S. : A Search Technique for Clause Interconnectivity Graphs, IEEE Trans. on computers, Vol. C-25, No. 8, Aug. 1976.