PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

TRANSFORMATIONAL LOGIC PROGRAM SYNTHESIS

Taisuke SATO
Electrotechnical Laboratory,
1-1-4, Umezono, Sakura-mura,
Niihari-gun, Ibaraki, 305
JAPAN

Hisao TAMAKI
Ibaraki University, 4-12,
Nakanarusawa, Hitati-si,
Ibaraki, 316 JAPAN

ABSTRACT: A new approach to logic program synthesis from the first order specifications is presented. Our synthesis process starts with a specification for p(X) of the form p(X)<->formula(X), where formula(X) is a first order formula and p(X) is an atomic formula. We assume that the predicate symbol "p" does not occur in formula(X). We also assume that every predicate in formula(X) is already defined by some logic program S (a set of definite clauses) and given an interpretation by the least model m(S) of S. Then, the interpretation of formula(X) by m(S) denotes a relation. Our objective is to demonstrate a method for synthesizing a logic program for "p" that computes the same relation as denoted by formula(X). The method relies on a negation technique which takes "the complement" of a given program. This technique enables us to synthesize not only a program for the negative specification but also one for the universally quantifed specification without induction. Synthesis of an N-queens program is given as an illustrative example.

1. INTRODUCTION

Although a lot of effort has been devoted to the problem of program synthesis, it remains a challenging problem. One of the reasons is the semantic gap between specification languages and programming languages. In this respect, it is advantageous to deal with the problem within the logic programming paradigm because logic programs are not only executable but also highly declarative.

In this paper we propose a new approach to logic program synthesis from the first order specifications. Our synthesis process starts with a specification of the form p(X)<->formula(X), where formula(X) is a first order formula and p(X) is an atomic formula. We assume that the predicate symbol "p" does not occur in formula(X). We also assume that every predicate in formula(X) is already defined by some logic program S (a set of definite clauses) and given an interpretation by the least model m(S) of S. Then, the interpretation of formula(X) by m(S) denotes a relation. Our objective is to demonstrate a method for synthesizing a logic program for "p" that computes the same relation as denoted by formula(X). The method relies on a "negation technique" [Sato 84] that eliminates negations and universal quantifiers from formula(X).

The negation technique derives a program S' that computes the relation ~q(X) from a program S for q(X) when S satisfies certain conditions. In order to satisfy these conditions which will be described later, an equivalence preserving program transformation [Tamaki 84] is often performed before the application of the negation technique.

Since negation technique plays an major role in our approach, we first describe it in section 3 and its application in section 4. Then we present a sample synthesis, i.e. the synthesis of an N-queens program in section 5.

2. BACKGROUND

We assume that programs and formulas in this paper are written in a many sorted first order language L. We fix the language L and use U to denote the Herbrand universe, i.e. the set of all ground (variable free) terms in L. By convention, terms beginning with upper case letters are variables and those beginning with lower case letters are constants, function symbols or predicate symbols.

A definite clause is a formula of the form p0 <- p1 & ... & pm (m>=0) where pi (0=<i=<m) is an atom (atomic formula). A logic program S is a finite set of definite clauses. The meaning of a program S is defined as the set of all ground atoms provable from S. We define:

success(S) =
  { p | p is a ground atom in L and S |- p }.

Success(S) is called the success set of S. We use failure(S) to denote the finite-failure set of S as defined by Apt and van Emden [Apt et al. 84]. Roughly speaking, it is the set of all ground atoms p such that SLD-refutation of <-p fails in a finite number of steps. The set, failure(S), is a subset of all of the ground atoms unprovable from S. We define an interpretation m(S) over U as follows:

```
  for any ground atom p in L
  m(S) |= p    iff (if-and-only-if)
               p is in success(S).
```

m(S) thus defined becomes the least model over U of S. That is, for any model I over U of S, if m(S) |= p then I |= p holds for every ground atom p. Since success(S) and m(S) are essentially the same, we use them interchangeably. In light of our program semantics, two programs S1 and S2 are equivalent iff m(S1) = m(S2).

3. NEGATION TECHNIQUE

The negation technique is a kind of program transformation. It is a procedure to derive a program S' from a given program S such that:

(1) Predicate names in S and those in S' have one-to-one correspondence. If we present the correspondence as, p in S <---> p' in S', p and p' have the same arity k(>0).

(2) For any ground atom p(t1,...,tk) and p'(t1,...,tk),

```
  not(S |- p(t1,...tk)) if S' |- p'(t1,...,tk)
```

If S' satisfies (1) and (2), it is called a dual program of S. Moreover, when "if" in (2) can be replaced "iff", we call S' a complement program of S.

Successful computations by S' mimick failed computations by S. In this sense S' can be regarded as the procedural negation of S. We would like to show the negeation technique by an example for saving sapce. Details are described in [Sato 84]. Given the program:

```
  S = { mem(H,[H|L]), mem(X,[H|L])<-mem(X,L) }
                                       ...(3-1)
```

where [a|b] stands for the term cons(a,b). The second argument of mem is a list and mem(a,b) is intended to mean that a is a member of list b. As stated before, S defines the binary relation 'mem' over U through m(S) so that the '~mem' relation is also defined. Since no clause in S has internal variables (an internal variable is one occurring only in the body of a clause such as Y in a(X)<- b(X,Y)), we are able to obtain a program for '~mem' by applying the negation technique to S in the following way. First we apply steps 1-6 to each predicate in S.

[STEP 1] Construct an IFF-definition [Apt et al. 82] of the selected predicate. We obtain,

```
  mem(A,B)<->
    (exist H,L)(<A,B>=<H,[H|L]>) or
    (exist X,H,L)(<A,B>=<X,[H|L]>&mem(X,L))
                                       ...(3-2)
```

where "=" means the syntactic identity in U and <a,b>=<c,d> is a shorthand for (a=b)&(c=d). We sometimes consider <a1,...,am> as a vector of terms for convenience.

[STEP 2] Negate both sides of the IFF-definition. <a,b>=/=<c,d> is a shorthand for ~((a=b)&(c=d)).

```
  ~mem(A,B)<->
   (all H,L)(<A,B>=/=<H,[H|L]>) &
   (all X,H,L)(<A,B>=/=<X,[H|L]> or ~mem(X,L))
                                       ...(3-3)
```

[STEP 3] Transform every conjunct on the right side of the result of [STEP 2] which has the form,

```
 (all X1,...,Xn)(<A1,...,Ak>=/=<t1,...,tk> or
               ~p1 or ... or ~pm)
```

(m>0) to,

```
 (all X1,...,Xn)<A1,...,Ak>=/=<t1,...,tk>
    or (exist X1,...,Xn)
        (<A1,...,Ak>=<t1,...,tk> & ~p1(X))
    ...
    or (exist X1,...,Xn)
        (<A1,...,Ak>=<t1,...,tk> & ~pm(X)).
```

In this case we obtain,

```
 ~mem(A,B)<->
    (all H,L)(<A,B>=/=<H,[H|L]>)&
       {(all X,H,L)(<A,B>=/=<X,[H|L]>) or
        (exist X,H,L)
              (<A,B>=<X,[H|L]>)&~mem(X,L))}
                                       ...(3-4)
```

[STEP 4] Transform the right hand side to a disjunctive form.

```
  ~mem(A,B)<->
     {(all H,L)(<A,B>=/=<H,[H|L]>)&
         (all X,H,L)(<A,B>=/=<X,[H|L]>)} or
     {(all H,L)(<A,B>=/=<H,[H|L]>)&
         (exist X,H,L)
            (<A,B>=<X,[H|L]>)&~mem(X,L))}
                                       ...(3-5)
```

[STEP 5] At this step we consider, for example, (all H,L)(<A,B> =/= <H,[H|L]>) as defining a new unary predicate with a vector argument <A,B>. In addition, we assume that there is a logic program which computes that predicate. This is possible and sound because for any ground term a and b, (all H,L)(<a,b>=/=<H,[H|L]>) holds iff the two term vectors <a,b> and <H,[H|L]> are ununifiable and ununifiability is a recusive relation over U. (Practically speaking, this is computed by "negation as failure rule" [Clark 78] such as not(<a,b>=<H,[H|L]>)). For convenience, we introduce a parameterized predicate ununi(<A1,...,Ak> , <t1,...,tk>) (k>0) where <A1,...,Ak> is an argument and a vector term <t1,..,tk> is a parameter. We stipulate that for any ground term ai(1=<i=<k), ununi(<a1,...,ak>, <t1,...,tk>) holds iff <a1,...,ak> and <t1,...,tk> are ununifiable. We eliminate universal quantifiers using this predicate. Note that the result has the form of an IFF-definition. We get,

```
~mem(A,B)<-> (ununi(<A,B>,<H,[H|L]>)
                & ununi(<A,B>,<X,[H|L]>) or
             ((exist X,H,L)(<A,B>=<X,[H|L]>
                & ununi(<A,B>,<H',[H'|L']>)
                & ~mem(X,L)).
                                        ...(3-6)
```

[STEP 6] Transform the IFF-definition given as the output of [STEP 5] to a clause set S'. Then simplify goals containing "ununi" predicates using the property of "ununi". For example, ununi(<A,B>,<H,[H|L]>) & ununi(<A,B>,<X,[H|L]>) equals ununi(B,[H|L]) wheih in turn equals B=[] since the variable B must have a list as its value.

[STEP 7] After processing all predicates in S with [STEP 1] to [STEP 6], collect the resulting clause sets and let S' be the union of them. Then regard the negated predicate symbol as a new predicate name. S' is the output of the negation technique applied to S. Below we consider "~mem" as a new predicate name.

```
S' = {  ~mem(A,[]),
        ~mem(A,[H|L])
            <- ununi(A,H) & ~mem(A,L) }
                                        ...(3-7)
```

Next we give an outline of a proof that S' is not only a dual but also a complementary program of S. Let '~mem' be the least relation defined by (3-7). It satisfies the IFF-definition of '~mem' (3-6) and this also satisfies (3-5) and (3-4) in turn. Now we note that for any ground term a,

```
(all X)(a=/=t(X) or ~p(a,X))
   <-> (all X)(a=/=t(X)) or
       (exist X)(a=t(X) & ~p(a,X))
```

holds over U because in U, X is determined uniquely by a=t(X) when a is given. It follows from this fact and (3-4) that the relation '~mem' defined by success(S') satisfies (3-3). (If some clause in S has an internal variable, this argument is not valid). Therefore, the relation

```
R = { mem(a,b) | ~mem(a,b) is not in
         success(S'). a,b are ground terms.}
```

satisfies (3-2), which implies two facts.

[Fact 1] R is a super set of success(S).

This is because success(S) is the least relation that satisfies (3-2). If ~mem(a,b) is in success(S'), mem(a,b) is not in R by the definition. Therefore, it follows from [Fact 1] that if ~mem(a,b) is in success(S'), mem(a,b) is not in success(S). Thus, S' is proved to be a dual program of S.

[Fact 2] R has no intersection with failure(S) [Apt 82].

To show that the dual program S' is also the complementary program of S, we introduce the following definition. We say that a program S is dichotomous iff for every k-ary predicate p (k>0) occurring in S and ground term a1,...ak, p(a1,...,ak) belongs either to success(S) or to failure(S). In other words if an SLD-tree [Apt 82] with root <-p(a1,...,ak) is always finite for every ground atom p(a1,...,ak), then S is dichotomous.

S, i.e. the mem program (3-1), is dichotomous. Thus it follows from [Fact 1] and [Fact 2] that R coincides with success(S). In other words, ~mem(a,b) is in success(S') iff mem(a,b) is in success(S). Thus we have proved that S' is a complementary program of S. Generalization of the arguments up to this point leads to the following theorem.

[THEOREM 1]
If every clause in a program S has no internal variables, then the procedure [STEP 1] to [STEP 7] gives a dual program of S. In addition, if S is dichotomous, it is also a complementary program of S.

We say that a clause p0(X) <- p1(X,Y) &...& pm(X,Y) in a program S with an internal variable Y has a function part p1 iff p1(X,Y) defines a partial function from X to Y (that is, whenever S |- p1(a,b) and S |- p1(a,b') then b = b' holds for any ground term a,b,b') and there is a predicate ndom(X) called a non-domain predicate for p1 such that S |- ndom(a) iff not(S |- (exist Y) p1(a,Y)). Even when there are clauses with internal variables in S, we can obtain a dual/complementary program for S by the procedure similar to that described above if the clauses have a function part [Sato 84].

4. DOUBLE NEGATION TECHNIQUE

The double negation technique is used to derive a program from a universally quantified specification. We again illustrate the idea with an example. Let the specification be:

```
arl(L,N)<->(all X)(mem(X,L)-> 1=<X=<N)
                                        ...(4-1)
```

where mem(X,L) is defined by (3-1). arl(L,N) is intended to mean that every element in list L is between 1 and N. A logic program for arl(L,N) is derived as follows. First take the logical negation of both sides of the specification.

```
~arl(L,N)<->(exist X)(mem(X,L)& ~(1=<X=<N))
                                        ...(4-2)
```

Then we consider "~arl" as a new predicate symbol and (4-2) as a specification for ~arl(L,N). (4-2) is apparently satisfied by a logic program:

```
{ ~arl(L,N)<- mem(X,L)& ~(1=<X=<N)
   clauses for "mem", "=<" }.            ...(4-3)
```

If we can apply the negation technique to (4-3), we will obtain a correct program for the specification (4-2). However, the existence of the internal variable X in (4-3) is an obstacle to the application (see theorem 1). We use the logic program transformation system [Tamaki 84] to eliminate internal variables. This system has two basic transformations. One is "unfolding of a goal" that means one step symbolic execution of a goal. The other, "folding of goals", replaces a procedure body (goals) by a procedure call (a clause head). The system preserves program equivalence. That is, if a program S1 is transformed to a program S2 by the system then m(S1) = m(S2) holds. Transformation of (4-3) by the system containing one unfolding and one folding operation yields an equivalent program that includes,

```
{ ~arl([H|L],N)<- ~(1=<H=<N),
  ~arl([H|L],N)<- ~arl(L,N)
   clauses for ">"  }.                   ...(4-4)
```

This program is guaranteed to be equivalent to (4-3) with respect to "~arl". Moreover it has no internal variables so that we can apply negation technique. We get,

```
{ arl([],N),
  arl([H|L],N)<-(1=<H=<N)&arl(L,N)
  clauses for "=<"  }.                   ...(4-5)
```

Suppose that a specification p(X1,...,Xn) <-> formula(X1,...,Xn) is given in conjunction with a program S0 which defines the primitive predicates used in formula(X1,...,Xn). And suppose that we have successfully synthesized a program S1 for p(X1,...,Xn). We say that S1 is partially correct with respect to the specification if for any ground term ai (1=<i=<n), m(S1) |= p(a1,...,an) implies m(S0) |= formula(a1,...,an). When the equivalence, m(S1) |= p(a1,...,an) iff m(S0) |= formula(a1,...,an) holds, we say that S1 is totally correct with respect to the specification or S1 realizes the specification p(X1,...,Xn) <-> formula(X1,..,Xn).

We show that (4-5) is totally correct with respect to (4-1). First (4-5) is a complementary program of (4-4) because (4-4) is dichotomous. Second (4-4) is equivalent to (4-3) with respect to "~arl" and (4-3) is totally correct with respect to (4-2). Therefore, the complement of the "arl" relation computed by (4-5) satisfies (4-2). In other words the relation computed by (4-5) satisfies (4-1) Q.E.D.// In addition, since (4-5) is obtained regardless of the contents of the 2-ary predicate (1=<H=<N), (4-5) remains totally correct even when (1=<H=<N) is replaced by some other predicate. This fact will be used in section 5.

As this example shows, the problem of synthesizing a program for a universally quantified specification like p(X)<->(all Y)formula(X,Y) can be solved in three steps. The first step is the logical negation of a given specification and its realization by a program S1. The second is the transformation of S1 to an equivalent program S2 to which negation technique is applicable. The third is the application of the negation technique to S2. This method is called the double negation technique. If the transformation from S1 to S2 is successful, the result of the double negation technique is partially correct with respect to the given specification. In addition, if S2 is dichotomous, the result becomes totally correct.

5. SYNTHESIS OF AN N-QUEENS PROGRAM

In this section we demonstrate a synthesis of an N-queens program which searches for a "mutually non-attacking arrangement of N queens" on an N by N chess board. We assume that the primitive relations defined by the following self-explanatory program are available.

[Given]

```
{ len([],0),      len([H|L],N+1)<-len(L,N),
  mem(H,[H|L]), mem(X,[H|L])<-mem(X,L),
  ap([],X,X),     ap([H|X],Y,[H|Z])<-ap(X,Y,Z),
  add(0,X,X),     add(X+1,Y,Z+1)<-add(X,Y,Z),
  check(N,A,B)<-add(A,M+1,B)&~(N=M+1),
  check(N,A,B)<-add(B,M+1,A)&~(N=M+1) }
```

Here X+1 stands for s(X) and "s" is a successor function. Using these predicates we specify the N-queens problem in a top-down manner as follows.

[Specifications]

```
queen(L,N)<->ar(L,N)&safe(L)             ...(5-1)
  ar(L,N)<->
     len(L,N)&(all X)(mem(X,L)->within(X,N))
                                         ...(5-2)
    within(X,N)<->
       (exist Y,Z)(add(1,Y,X)&add(Z,X,N))
                                         ...(5-3)
safe(L)<->
   (all A,B,N)(dif(A,B,L,N)->check(N,A,B))
                                         ...(5-4)
  dif(A,B,L,N)<->
    (exist X,Y,Z,W)
       (ap(X,[A|Y],Z)&ap(Z,[B|W],L)&
        len([A|Y],N)                     ...(5-5)
```

queen(L,N) means that list L is an answer to the N-queens problem and it is defined by ar(L,N) and safe(L). ar(L,N) means that list L has length N and each integer in L satisfies within(X,N). within(X,N) means that 1=<X=<N. safe(L) means that any pair of queens in list L are mutually non-attacking. It is defined using dif(A,B,L,N) which means that the distance between A and B in list L is N. Every argument is supposed to have an appropriate sort.

Our synthesis process proceeds in a bottm up manner, i.e. in the order (5-3), (5-2), (5-5), (5-4) and finally (5-1). We first start with (5-3).

```
within(X,N)<->
  (exist Y,Z)(add(1,Y,X)&add(Z,X,N)) ...(5-3)
```

Program synthesis of this type which has the form p(X) <-> (exist Y) q(X,Y) is straightforward. A program { p(X) <- q(X,Y), clauses for "q" } realizes the specification so that (5-3) is realized by:

```
{ witin(X,N)<-add(1,Y,X)&add(Z,X,N),
  clauses for "add" }.
```

This is optimized by an equivalence preserving transformation system [Tamaki 84]. After one unfolding and one folding operation, We obtain:

```
{ within(N,N)<-add(1,Y,N),
  within(X,N+1)<- within(X,N),
  clauses for "add" }                    ...(5-3')
```

Next we move to the synthesis of (5-2).

```
ar(L,N)<->
  len(L,N)&(all X)(mem(X,L)->within(X,N))
                                          ...(5-2)
```

This requires the synthesis of ar1(L,N) <-> (all X)(mem(X,L) -> within(X,N)) which has been done already in section 4 using the double negation technique. The result is (4-5) where we identify (1=<X=<N) with within(X,N). Therefore, a program,

```
{ ar(L,N)<-len(L,N)&ar1(L,N),
    clauses for "len" } U (4-5) U (5-3')
```

constitutes a totally correct program with respect to the specification (5-2). An efficient program, however, is more desirable. So we start the equivalence preserving transformation process by setting,

```
ar2(L,N,X)<->
  (exist M)len(L,M)&add(M,X,N)&ar1(L,N).
```

It is obvious that ar2(L,N,0) <-> len(L,N) & ar1(L,N) holds. After 3 unfolding operations and 1 folding operation, we reach a totally correct program with respect to the specification (5-2).

```
{ ar(L,N)<-ar2(L,N,0),
  ar2([],N,N),
  ar2([H|L],N,X)<- within(H,N)&ar2(L,N,X+1)
  within(N,N)<-add(1,Y,N),
  within(X,N+1)<- within(X,N),
  clauses for "add" }                    ...(5-2')
```

After having realized (5-2) as a program, we move to the synthesis of (5-4). First we realize the specification (5-5) by the program:

```
{ dif(A,B,L,N)<-
    ap(X,[A|Y],Z)&ap(Z,[B|W],L)&len([A|Y],N),
  clauses for "ap" and "len" }
```

Second we transform this program to an equivalent one that has no internal variables. This requires 7 unfolding operations, 3 folding operations and 1 introduction of a new predicate dif1(B,L,N) whose definition is

```
dif1(B,L,N)<->
   (exist Y,W)ap(Y,[B|W],L)&len(Y,N).
```

```
{  dif(A,B,[H|L],N)<-dif(A,B,L,N)
   dif(A,B,[A|L],N+1)<-dif1(B,L,N),
   dif1(B,[B|L],0),
   dif1(B,[H|L],N+1)<-dif1(B,L,N) }
                                          ...(5-5')
```

We next apply the double negation technique to (5-4). First we consider the specification,

```
~safe(L)<->
   (exist A,B,N)(dif(A,B,L,N)&check(N,A,B))
                                          ...(5-4')
```

and realizes it by a program that has no internal variables. This requires 2 unfolding operations, 3 folding operations and 1 introduction of a new predicate ~safe1(A,L,X) whose definition is ~safe1(A,L,X) <-> (exist B,N,M) dif1(B,L,N) & add(N,X,M) & ~check(M,A,B) (safe1(A,L,X) <-> (all B,N,M) (dif1(B,L,N) & add(N,X,M) -> check(M,A,B)). Here we consider ~safe, ~safe1, safe1 as new predicate symbols. The resulting program is

```
{ ~safe([H|L])<-~safe(L),
  ~safe([H|L])<-~safe1(H,L,1),
  ~safe1(A,[B|L],X)<-~check(X,A,B),
  ~safe1(A,[H|L])<-~safe1(A,L,X+1) }.
                                          ...(5-4'')
```

Note that this program has no internal variables. In addition, it is dichotomous. By applying the negation technique to this program, we finally obtain,

```
{ safe([]),
  safe([H|L])<-safe1(H,L,1)&safe(L),
  safe1(A,[],N),
  safe1(A,[B|L],N)<-
    check(N,A,B)&safe1(A,L,N+1),
  clauses for "check" }            ...(5-4''')
```

which is totally correct with respect to the initial specification (5-4). Thus we have realized the specifications (5-2), (5-4) by (5-2'), (5-4''') respectively. Now, the top-most specification (5-1) is realized by

```
{ queen(L,N)<-ar(L,N)&safe(L) }
              U (5-2') U (5-4''') ...(5-1')
```

Although (5-1') is totally correct with respect to (5-1), it is too brute a generate-and-test program. We attempt an improvement by transformation. The transformation begins with a new predicate

```
queen1(L,N,X)<->ar2(L,N,X)&safe(L).
```

After 3 unfolding operations, 3 folding operations and 1 introduction of a new predicate queen2(H,N,L,Y) <-> within(H,N) & safe1(H,L,Y), we reach the following program

[Synthesized Program]

```
{ queen(L,N)<-queen1(L,N,0),
  queen1([],N,N),
  queen1([Q|L],N,X)<-
    queen1(L,N,X+1)&queen2(Q,N,L,1)
  queen2(Q,N,[],X)<- within(Q,N),
  queen2(Q,N,[B|L],X)<-
    queen2(Q,N,L,X+1)&check(X,Q,B),
  within(N,N)<-add(1,Y,N),
  within(X,N+1)<- within(X,N),
  clauses for "add", "check" ... } ...(5-1'')
```

When queen(L,N) is called with specified N, the program chooses queens (an integer Q, 1=<Q=<N) and place them on the chess bord one by one. Whenever a queen is placed, it checks whether or not it is mutually non-attacking to the existing queens on the board.

To derive this program, 17 unfolding operations, 12 folding operations, 4 introductions of new predicates and 2 applications of the negation technique were required. We do not evaluate whether or not the derived program is worthy of those costs. However, we emphasize that it is guaranteed to be not only partially correct but also totally correct with respect to (5-1). For during the synthesis process, every intermediate program to which the negation technique was applied was dichotomous, and the transformation system preserves program equivalence.

6. DISCUSSION

We have illustrated a transformational approach to logic program synthsis based on the negation technique. It is summarized as follows. Suppose that a(X), b(X), c(X,Y) are predicates defined by some logic program S through its least model. Then, a specification for a predicate p(X) in the left side is realized by a program in the right side.

```
p(X)<->a(X) & b(X)
        ----------> { p(X)<-a(X)&b(X) } U S

p(X)<->a(X) or b(X)
        ---> { p(X)<-a(X), p(X)<-b(X) } U S

p(X)<->(exist Y)c(X,Y)
        -------------> { p(X)<-c(X,Y) } U S

p(X)<->~a(X)
        ------------> By negation technique

p(X)<->(all Y)c(X,Y)
        -----> By double negation technique
```

When a specification p(X)<->formula(X) is given, a partially or totally correct program with respect to the specificaton can be synthesized by recursive application of this table to the subformulas of formula(X) with the help of the equivalence preserving transformation as seen in section 5. Our method has the following interesting features.

First our system does not suffer from nondeterminacies caused by the deduction in a formal system as compared with the deductive approach to logic program synthesis [Clark et al. 77], [Eriksson et al. 82], [Hansson et al. 79], [Hogger 81]. Instead, we have to cope with nondeterminacies in the transformation process in the double negation technique or those in optimization. But the skeletal process is deterministic and has no need for a search process such as 'guess step' in [Bibel 80].

Second the system synthesizes not only a function but also a (nondeterministic) program for a relation and it does not require any existence proof of the object to be synthesized unlike [Manna 81], [Sato 79]. Induction plays only a secondary role in our approach though it may be used to establish, for example, (X+Y)+Z = X+(Y+Z).

Third it has simple and clear semantic basis which elucidates the meaning of the synthesis process. For example, every predicate introduced during the synthesis has a first order specification. Such specification can be helpful in the optimization stage.

On the other hand since our method is a one-to-one mapping from a subformula in the given specification to a program, we may lose opportunities to shorten the path to the final program by 'macro processing' of the specification. Moreover the output program tends to have a flavor of generate-and-test so that the subsequent optimization becomes very important as is exemplified in section 5.

We hope that the synthesis method presented here will contribute one step toward a (semi-)automatic programming environment which logic programming aspires to achieve.

ACKNOWLEDGEMENT: The authers are grateful to Monica Strauss for detailed corrections. Thanks are due also to the members of Machine Inference Section of Electrotechnical Laboratory and the working groups of the Fifth Generation Computer Project.

REFERENCES:

Apt,K.R. and van Emden,M.H.,"Contributions to the Theory of Logic Programming", JACM 29-3, 1982.

Bibel,W.,"Syntax-Directed, Semantics-Supported Program Synthesis", Artificial Intelligence 14, North-Holland, 1980.

Clark,K.L. and Tarnlund,S-A.,"A First Order Theory of Data and Programs", Proc. of IFIP 77, North-Holland, 1977.

Clark,K.L.,"Negation as Failure", in Logic and Database, Plenum Press, New York, 1978.

Eriksson,A. and Johanson,A.L.,"Computer_Based Synthesis of Logic Programs", Proc. of 5th International Symposium on Programming, Lec. Note in Comp. Sci. 137, Springer, 1982.

Hansson,A. and Tarnlund,S-A.,"A Natural Programming Calculus", Proc. of 6th IJCAI, 1979.

Hogger,C.J.,"Derivation of Logic Programs", JACM Vol. 28 No. 2, 1981.

Manna,W. and Waldinger,R.,"Deductive Synthesis of the Unification Algorithm", STAN-CS-81-855, Dep. of Comp. Sci, Stanford Univ., 1981.

M.Sato.,"Towards A Mathematical Theory of Program Synthesis", Proc. of 5th IJCAI, 1979.

T. Sato,"Negation Technique", to appear in ICOT technical report TR-038, 1984.

H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs", 2nd logic programming conference, Uppsala, Sweden, 1984.