ICOT Technical Report: TR-188

TR-188

A Framework for Interactive Problem Solving based on Interactive Query Revision

by
M. Ohki, A. Takeuchi
and K. Furukawa

June, 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 Framework for Interactive Problem Solving based on Interactive Query Revision

Masaru Ohki, Akikazu Takeuchi and Koichi Furukawa

ICOT Research Center,
Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 1-4-28, Mita,
Minato-ku, Tokyo, 108, Japan

Abstract

Logic programming has been widely used because of the clearness of its semantics and its extensibility. Many inference systems have been proposed using a logic programming framework. But few of these have studied logic based man-machine interaction, apart from systems based on incremental query [Emden 1985]. Incremental query allows users to enter a part of queries incrementally instead of entering the whole query at once, as in Prolog. In this paper we investigate essential concepts of interactive problem solving and generalize incremental query further. And we propose a new query model for logic programming, which we call interactive query revision. Interactive query revision allows a user to modify queries and hypotheses and to act as a part of the inference engine, in addition to entering queries incrementally. We apply interactive query revision to an interactive LSI layout system.

1. Introduction

It has been claimed that logic programming provides a powerful framework for building inference systems. The theorem-proving capability of logic programming languages is the starting point for logic-based inference systems. The methodology for realizing inference in an expert system by computation in logic programming languages was discussed in an early work on a logic-based expert system [Clark 1982].

Basically, an inference system consists of an inference engine and a rule base. An inference engine is the kernel of the system performing basic inference. A rule base is a database of inference rules. The important characteristics of inference systems are usually all realized in their inference engines. They are the explanation facility, handling of certainty factors, query facilities used when some information is missing, multiple rule bases, frames and so on.

Logic programming has provided new concepts not only for basic inference mechanisms but also for such extended features as those listed above, which are essential in inference systems. In APES [Hammond 1983], explanation facilities are realized by meta-level operations on the proof tree. Shapiro proposes an efficient debugging method for rules [Shapiro 1983a] and a meta-interpreter to handle certainty factors [Shapiro 1983b]. Sergot [Sergot 1983] introduced an open world assumption in his logic-based expert system in order to realize query facilities naturally if inference fails without additional information. Several researchers [Nakashima 1982, Poole 1985, Bowen 1985] introduce multiple theory models to form bases for hypothetical inference. Frame-based representation of knowledge in a logic programming language was introduced in CIL [Mukai 1985], based on situation semantics. Partial evaluation of logic programs [Takeuchi 1985] made possible customization of inference engines and rule compilations. Emden introduced a new computation model called the query interaction model, facilitating a mixed languages environment [Emden 1984].

These are just some of the many proposals for inference engines. However, almost all of them are related to logic-based inference mechanisms and, as yet, apart from [Emden 1984], no studies have directly confronted the problem of logic-based models of environments including man-machine interaction. Emden introduced a new standpoint from which logic can be seen as an interaction language and his concept of the incremental query represents a new type of query for logic programs. Emden also proved that incremental query can be a strong basis when building universally accessible interfaces for logic programs such as spreadsheets [Emden 1985].

In practical applications, many problems cannot be given as a set of goals at once. To model the whole human problem solving task, it has to be considered in two modes, horizontal and vertical modes. The horizontal mode represents the top-level tasks of problem solving such as detailed specification of the problem, only roughly specified at first, and examination of the solutions under several variations of the original problem. It often happens that the problem to be solved is too weakly specified. A user has to specify the problem in more detail in order to

solve the problem by computer. Since there are many possibilities in describing the details of the problem, the user has to try to specify the problem in various ways. Furthermore the problem to be solved changes over time as the user observes the solutions under several variations of the original problem and comes to better understand the properties of the problem. In fact, problems people have in mind are very vague and can only be specified weakly. This is where the horizontal mode of problem solving comes into play. At the moment this phase is performed by the users themselves.

The vertical mode of problem solving involves solving a problem that is well specified in the horizontal mode. Many concepts have been invented to model this kind of problem solving task in the computer, such as common sense reasoning, ambiguous reasoning using certainty factors, hypothetical reasoning, default reasoning,qualitative reasoning and so on.

As stated above, there are many contributions from logic programming to the vertical mode of problem solving, but only a few address the horizontal mode. When a transformed problem cannot be solved or its solution is not satisfiable, the user has to change the detail of the problem specification added during transformation. Thus, the problem specification task is still the heavier task, even if the computer assists users in the vertical mode. We believe that "logic for problem solving" has to be extended to "logic for interactive problem solving."

In this paper, we present a new query model for logic programs to assist users in the horizontal mode of problem solving and establish ideal man-machine interaction for interactive problem solving. Our new query model is called interactive query revision model. It is based on the previous work, incremental query, of Emden et al. [Emden 1984, 1985]. They suggested that incremental query could use for interactive problem solving. But they did not investigate interactive problem solving in detail because incremental query aimed at the implementation of spreadsheet. We investigate essential concepts for interactive problem solving and we propose interactive query revision model, which realizes the concepts.

In Section 2 we describe the difficulties of current problem solving systems from the viewpoint of interaction between man and machine. In Section 3, we introduce the interactive query revision model for Prolog [Bowen 1982] and outline an implementation. Section 4 shows how interactive query revision is applied to an interactive LSI layout system and how it solves the problems described in Section 2.

2. Difficulties of problem solving and new concepts to solve them

(1) Observation 1:

The user does not know exactly what he wants in the solution.

In general, users do not know what they want exactly, however, they do know what they do not want. It is difficult for users to fully specify the problem, but easy for them to say "no" when they see unacceptable 'solutions'. For example, when a user starts to lay out an LSI, he does not often know the detailed specification and he determines these details as he goes along. If he found the LSI unsatisfactory, he may design it over again or add new constraints. In order to support such users, we need the concept of "open constraints" allowing users to add new constraints incrementally. The incremental query model is the first to realize open constraints. In the incremental query model, when the solution obtained based on the constraints entered up to a given point is unsatisfactory, the user can add constraints that exclude the solution proposed at that point as new increments.

(2) Observation 2:

The user wants to know the relation between solutions and constraints/hypotheses.

There are two ways in which a problem is said to be weakly specified. The first is that the goal to be solved is vague. The second is that the rules and facts are incomplete and contain hypotheses. If the problem initially given is weakly specified, the solutions of the problem conceptually form a set, each element of which is a solution of the strictly specified version of the original problem. There is no way to obtain the whole set of solutions for a weakly specified problem, since computers can only solve strictly specified problems. For example, in the case of LSI layout a user may want to design the best LSI among those satisfying the weak specification. Instead of directly solving the weakly specified problem, we propose the concept of "variation of solution" to allow the user to see the variations in solutions resulting from slightly modifying parts of constraints or hypotheses and select the most desirable from among them. Now, even if the user has only a vague problem in mind, he can examine solution variations with respect to several constraints or several hypotheses and find the most acceptable candidate in the set of solutions.

(3) Observation 3:

The user does not perform a function as part of the inference engine.

The inference engine is usually incomplete. In fact, it is impossible to build the perfect inference engine based on current technology. For example, a system for laying out LSIs may not include information on their sale conditions. But this is often important to designers. Instead of making a perfect inference engine, we propose a concept of the "combination of

computer and man." It provides a mechanism to allow users to act as a part of inference engine. Even when the system fails to solve a part of problem, appropriate action by the user can keep the system going until it succeeds in solving the problem. This mechanism is convenient when the problem to be solved partially exceeds the range of the rule base. Of course, even if the user has the opportunity to act as a part of the inference engine, he may not solve the relevant part of the problem. Nevertheless, it is clear that a mechanism to add the power of the human brain will significantly enrich the system.

3. Interactive Query Revision

3.1 Interactive Query Revision Model

A query of Prolog is a set of goals such as the following:

```
?- p,..,q.
```

When this is entered, the Prolog system solves it and replies yes with answer substitution if the goals succeed, or no if they do not. A user cannot interact with the Prolog system in the course of computation except at input/output, an extralogical feature of the Prolog system.

The incremental query model expands the Prolog query model so that queries can be incrementally added. The idea of incremental query is that instead of entering the whole goals, a user is allowed to enter parts of the goals incrementally as he sees the intermediate solution(substitution) of some of the queries entered so far. The incremental queries are of the following form: ('???-' is the prompt for the incremental queries.)

```
???- p.
    <answer substitution for "p">
???- q.
    <answer substitution for "p and q">
???- r.
    <answer substitution for "p and q and r">
```

where "p","q" and "r" are queries entered by the user and <answer substitution>'s are responses from the system. These queries are equivalent the Prolog goals "p,q,r". <answer substitution for "p and q and r"> is equivalent to the answer substitution of Prolog for the goals "p,q,r". In incremental query, when a new query, Qi, is given, the current (last) answer substitution Si-1 is applied to Qi. The resultant query, (Qi)Si-1, is then solved and the updated answer substitution, Si, is returned. If the query Qi cannot be solved with the current substitution,Si-1, backtracking occurs. The system tries to find alternative substitution S'i-1 by re-solved Qi-1 with Si-2, so that (Qi)S'i-1 can be successfully solved. A variant of incremental query allowing the user to cancel parts of the queries already entered is given in [Emden 1985], and its implementation in Prolog is also described.

The interactive query revision model carries the idea of incremental query a step further. It allows a user to enter a query incrementally, insert a new query between previous queries, remove parts of queries or partially replace them. It also allows the user to add hypotheses to the program, and remove them or replace them in the program incrementally. These functions realize two of the concepts stated in Section 2, "open constraint" and "variation of solution." Whatever queries or hypotheses parts are modified, the logical validity of the solution is preserved in the interactive query revision model. Suppose the following queries

```
???- p. ???- q. ???- r.
    (Queries are written in the same line.)
```

have been given. If a user replaces the query "q" with "q1", the above queries change as follows:

```
???- p. ???- q1. ???- r.
```

They are solved as if equivalent to Prolog goals "p,q1,r". If a hypothesis is added to the program after the queries "p" and "q" are solved:

```
???- p. ???- q. ???- <added hypothesis>.
```

the solution of these queries is equal to the solution of the Prolog goals "p,q" in the program to which the hypothesis is added.

We introduce the special command, "user_interaction," to realize the third concept, "combination of computer and man." It allows users to help the system by giving a solution for a part of the problem in the form of queries, and allows them to act as part of the inference engine. Assigning a solution to an unbound variable, which cannot be determined by the system, is an example of an interaction with a user. Once the mechanism recognizing the user as a part of inference engine is introduced, it is obviously necessary that interactions with a user are treated in the same manner as other logical constructs. This means that on backtracking the system needs to be able to query the user for alternatives of the previous interaction. This kind of interaction with the users is logical and entirely different from conventional input/output, which is extralogical. When the "user_interaction" command is entered, a user is asked for the queries he has in mind at that moment. Suppose the following sequence of queries:

```
???- p. ???- user_interaction. ???- r.
```

where "q1" and "q2" are entered at "user_interaction." It is handled as if equal to the following sequence of queries:

```
???- p. ???- (q1;q2). ???- r.
```

These goals,"q1" and "q2", are connected with disjunction, so the user can give alternative choices or instructions to the system. These alternatives are selected one by one when backtracking occurs. If all alternatives are exhausted, the user is asked to enter other alternatives. If the user replies with another alternative, say "q3", the query sequence is then as follows:

```
?- p. ?- (q1;q2;q3). ?- r.
```

If the user wants to backtrack beyond "user_interaction," it is possible to pass to enter an alternative.

Interactive query revision is independent of inference systems since it assists users in the horizontal mode of problem solving. Therefore, we can use various inference systems operating in the vertical mode with the interactive query revision model.

3.2 Implementation

We have implemented the interactive query revision model based on the incremental query model. First we briefly describe an implementation of incremental query [Emden 1985].

The key point of the implementation of incremental query is the stack of elements consisting of an incrementally-entered query and its environment. The environment is a set of pairs of variable and value before the query is executed. The stack is used to simulate the backtracking mechanism so that incremental query can be processed just as in Prolog. An example of a stack is given in Figure 1. When a new query is added and fails under the current environment, the system starts to simulate backtracking. It moves back along the stack one by one, gets goals, and makes new goals by combining the previous goals and the current query. The new goals are solved as the Prolog goals. If the goals fail, the system resumes backtracking.

Incremental queries	Stack token forms	environments
X=1	('X'=1)	('X',_)
Y=2	('Y'=2)	('X',1),('Y',_)
Z=3	('Z'=3)	('X',1),('Y',2),('Z',_)

Figure 1 An example of incremental queries the content of the stack

We have enhanced incremental query to implement interactive query revision. We added the following mechanisms to modify queries: (1) editing a stack to insert, remove and replace queries, (2) recalculating modified queries, (3) reforming a new stack. Modification of hypotheses is currently implemented by modifying the program in the global database of Prolog. When hypotheses are modified, for example, when a hypothesis is added to a program, all queries must be recalculated because it is difficult to know which queries can use the hypothesis.

The "user_interaction" command takes in solutions inferred by the user in query form. When backtracking occurs, the user interaction is also redone. If all solutions inferred by the user have been tried, the user is asked to enter other alternatives. The user can enter one of the following:

(1) a new alternative
Add a new alternative to current alternatives.

(2) pass
Leave the set of alternatives as it is at that moment and propagate backtracking beyond the "user_interaction" command.

(3) temporarily close
Leave the set of alternatives as it is during execution of the current query and propagate backtracking beyond the command.

(4) close
Leave the set of alternatives as it is and propagate backtracking beyond the command.

4. Application of interactive query revision to an LSI layout system

An LSI chip layout is a typical example of problem solving using computers. The system considered here lays out three kinds of component blocks, CPU, ROM and RAM, on LSI, as shown in Figure 2. It is written in CIL [Mukai 1985] and has two types of knowledge. One is the knowledge to lay out an LSI and the other is the knowledge about the component blocks. CIL is a logic programming language. Compared with Prolog, it is augmented with the "freeze" primitive [Colmerauer 1982] and an association list. An association list is denoted by "{A1/V1,..,An/Vn}", where Ai is an attribute name and Vi is its associated value. Unification of two association lists only succeeds if the values of their common attributes can be unified, in which case their attributes are merged. Otherwise, unification fails. Access to an attribute A of an association list X is denoted by the primitive function "X!A". The "freeze" primitive suspends goals with respect to a variable until the variable(called the frozen variable) becomes instantiated. A question mark, "?", is attached to a frozen variable.

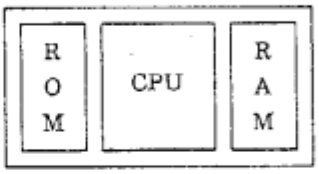


Figure 2 An example of LSI chip layout

The rules for LSI layout are described as constraints using the "freeze" primitive of CIL. One of the rules for laying three component blocks on an LSI is shown in Figure 3. An LSI chip is described as an association list containing LSI type, right, center and left component, width(w) and length(l) of the LSI, and its cost. The width and length of a component block are determined by constraints in "arrangement31_constraint." It expresses conditions such as, if the length of the left component is not given it is set equal to the length of the LSI. If

```
arrangement(LSI,[A,B,C]) :-
    !, arrangement3(LSI,[A,B,C]).
arrangement3(LSI,[A,B,C]) :-      arrange3([A,B,C],[Right,Center,Left]),
    LSI = {type/lsi31,right/Right,center/Center,left/Left,w/W,l/H,cost/Cost},
    cost/Cost,W,H,[Right,Center,Left]),
    arrangement31_constraint(Right,Center,Left,W,H).
arrangement31_constraint(Right,Center,Left,W,H) :-
    constraint_ge(H,Left!l),
    constraint_ge(H,Center!l),
    constraint_ge(H,Right!l),
    constraint_add(W,Center!w,Right!w,Left!w).
```

Figure 3 Part of the knowledge for LSI chip layout

```
ram(RAM) :-
    Type = ram4, Access = 200, PerSpace = 6, PerCost = 2,
    template_ram(RAM,Type,Access,PerSpace,PerCost).
template_ram(RAM,Type,Access,PerSpace,PerCost) :-
    RAM = {type/Type,w/W1,l/H1,capacity/Capacity,
    r_capacity/R_Capacity, access_time/Access,cost/Cost},
    (W1?) :> 0, (H1?) :> 0,
    or_freeze([([H1,W1],ram1(Capacity,H1,W1,R_Capacity,PerSpace)),
          ([Capacity,H1],ram2(W1,H1,Capacity,R_Capacity,PerSpace)),
          ([Capacity,W1],ram2(H1,W1,Capacity,R_Capacity,PerSpace))]),
    Cost :== (R_Capacity?) * PerCost.
```

Figure 4 Part of the knowledge on RAM

it is specified, its value is checked to make sure it is less than or equal to the length of the LSI. The widths of components and the LSI are determined by the constraint that the width of the LSI is greater than or equal to sum of the widths of components. Figure 4 shows one of the rules for a component RAM. A RAM is also described as an association list containing RAM type, width and length, requested capacity and actual capacity, access time and cost. The reason why a RAM association list contains actual capacity as an attribute is that actual capacity is not always equal to requested capacity, because its shape is restricted to a rectangle. If any two of the values for width, length and requested capacity are specified, the rest is determined from them. The cost of the RAM can be determined if actual capacity is given.

Let us start to lay out an LSI chip, even though we do not completely know all the constraints. The constraints for LSI layout we have at first are as follows:

(1) The size of the LSI chip is roughly 30mm wide and 20mm long.
(2) The LSI has at least CPU and ROM.
(3) The most efficient layout should be produced.

The scenario in which we lay out an LSI chip is as follows:

(1) First, CPU and ROM are arranged on the LSI with 30mm wide and 20mm long.
(2) RAM is then placed in the remaining space.
(3) The LSI layout system does not possess knowledge on the reasonable cost/performance ratio, which is one of sale conditions of the LSI, so we pose it to the system by "user interaction."
(4) If a proposed layout fails, we try to hypothesize a new type of ROM.
(5) Using the new type of ROM we continue to lay out by trial and error to improve the design.

Now let us lay out the LSI chip. Some configurations of LSI are shown in Figure 5. Statements begun by "%" explain the course of design, and statements parenthesized by "(" and ")" are other comments.

```
| ?- iq.
        % First we try to set  the width and the length of LSI to 30mm
        %and 20mm respectively.
??- LSI!w=30,LSI!l=20.
LSI = {w/30,l/20}          ("{ }" is an association list.)
                (Figure 5 (1) shows the chip at this point.)
        % Arrange two components, CPU and ROM, on the LSI.
???--  arrangement(LSI,[CPU,ROM]).
                :          (A display of values of variables is omitted.)
        % Assign  substantial CPU and ROM to variables CPU and ROM
        %respectively.
???--  cpu(CPU),rom(ROM).
                :
                (Figure 5 (2) shows the chip at this point.)
        % Attempt to set ROM capacity to 300 bytes.
```

```
???--  ROM!capacity=300.
LSI = {w/30,l/20,cost/3200,type/lsi2,right/{cpu1},left/{rom1}}
CPU = {cost/2000,w/1,l/20,type/cpu1,perform/100}
ROM = {cost/600,capacity/300,access_time/100,w/15,l/20,type/rom1,r_capacity/300}
        % The first version of layout was obtained.
        %What is the space-utilization efficiency of the LSI?
???--  space(LSI,Space).
                :
Space = 53
                (Figure 5 (3) shows the chip at this point)
        % Space-utilization efficiency is 53 %, it is very low.
        %It is necessary to redo the layout. First, let us take a look at
        %all queries so far using the list_queries command.
???--  list_queries.
----------------------
[0]  space(LSI,Space)
----------------------
[1]  ROM!capacity=300
----------------------
[2]  cpu(CPU),rom(ROM)
----------------------
[3]  arrangement(LSI,[CPU,ROM])
----------------------
[4]  LSI!w=30,LSI!l=20
        % Attempt to change the capacity of ROM.  This attempt is an example
        %of "variation of solutions."  It is performed by the "replace_query"
        %command.  We replace an old query ROM!capacity=300 by a new query
        %ROM!capacity=500.  The argument of the command  indicates
        %the number of the query in the queries list above.
        %When the query is replaced, all queries after that query are
        %re-solved. That is, [1] and [0] in the above list are solved again.
???--  replace_query(1).
 >> Enter a query :  ROM!capacity=500.
                :
Space = 86
        % There is still space left. Now let us try to place RAM on the LSI.
        %We replace "arrangement(LSI,[CPU,ROM])" by
        %"arrangement(LSI,[CPU,ROM,RAM])".
???--  replace_query(3).
 >> Enter a query :  arrangement(LSI,[CPU,ROM,RAM]).
                :
        % Assign substantial RAM to a variable RAM.
???--  ram(RAM).
                :
        % Add a constraint specifying reasonable cost/performance ratio
        %from the point of view of expert designers
        %before we specify the RAM capacity.  Definition of the reasonable
        %ratio differs with designers and over time, and this system
        %does not have such knowledge. Enter constraints using
        %"user interaction" so that we can add alternative constraints
        %on backtracking.
???--  user_interaction.
 >> Enter comment for user interaction :
 'reasonable cost/performance ratio'
 >> Enter user_interaction query(or close/pass/temp_close) :
 CPU!perform=100,ge(RAM!capacity,100),le(LSI!cost,4500).
                ("ge" is "greater than or equal to,"
                 and "le" is "less than or equal to.")
 >> Enter user_interaction query(or close/pass/temp_close) :
 CPU!perform=150,ge(RAM!capacity,50),le(LSI!cost,7000).
 >> Enter user_interaction query(or close/pass/temp_close) :
 pass.
                :
        % Now make the RAM capacity 100 bytes.
???--  RAM!capacity=100.
                :
Space = 96
                (Figure 5 (4) shows the chip at this point.)
        % A layout is completed. But there is still space left.  Since we
        %should use the LSI space efficiently, we try to increase
        %the RAM capacity.
```

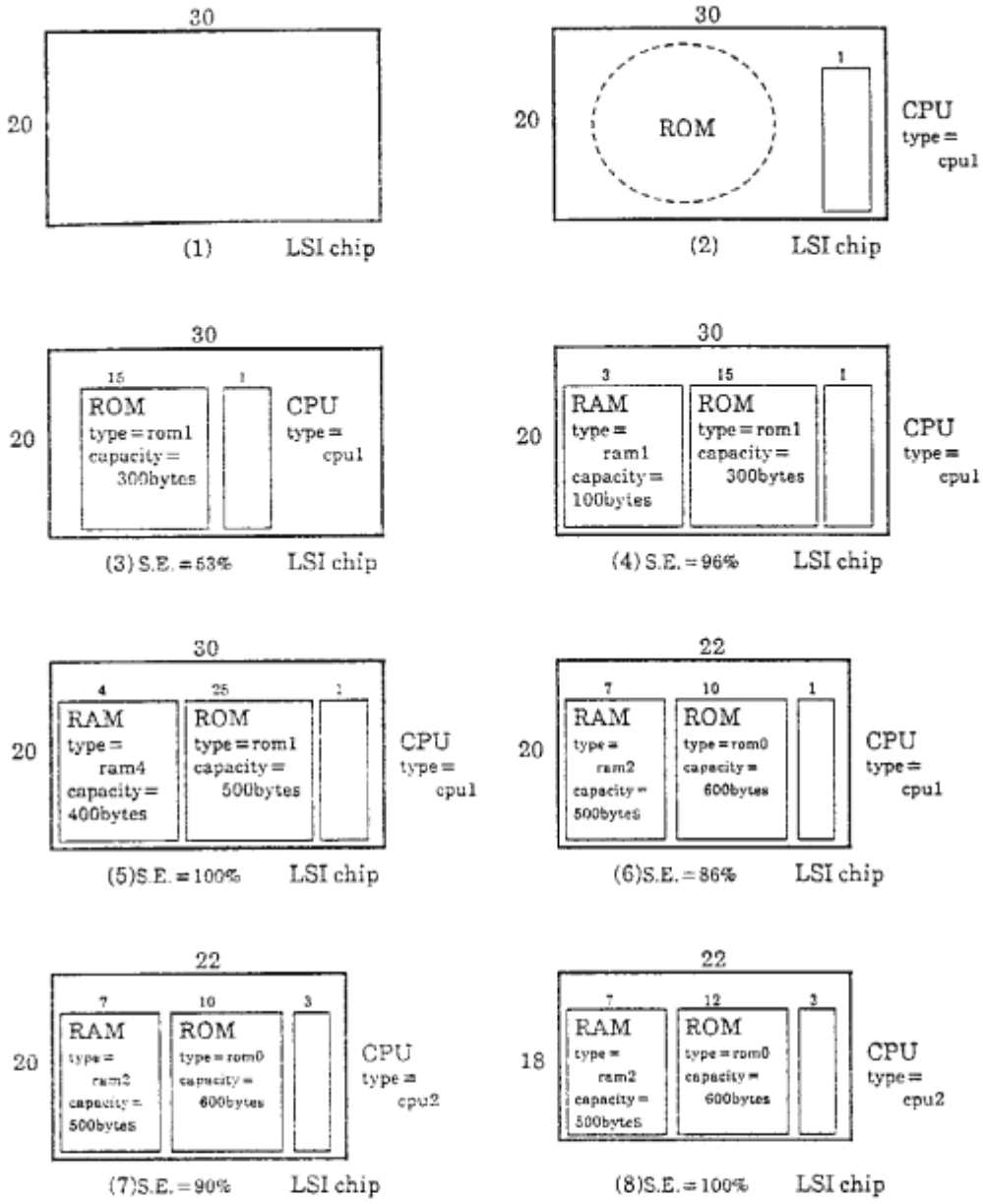
```
???--  replace_query(0).
 >> Enter query :  RAM!capacity=200.
        % Backtracking is invoked because the replaced queries fail.
        %At "user_interaction," we are asked whether we have other
        %alternative constraints.  We enter an alternative constraint as it
        %occurs to us just now looking at indeterminate solutions.
 ** Comment of user_interaction = reasonable cost/performance ratio
 >> Enter user_interaction query(or close/pass/temp_close) :
 CPU!perform=100,ge(RAM!capacity,200),le(LSI!cost,5000).
        %Backtracking occurs again and we are
        %asked still more for alternative constraints, but we temporarily
        %close interaction because we do not have any alternatives.
        %Backtracking propagates to queries before the "user_interaction."
 ** Comment of user_interaction = reasonable cost/performance ratio
 >> Enter user_interaction query(or close/pass/temp_close) :
 temp_close.
LSI = {w/30,l/20,cost/4080,type/lsi31,center/{rom1},right/{ram2},center/{rom1}}
CPU = {cost/2000,l/20,w/1,type/cpu1,perform/100}
ROM = {cost/1000,capacity/500,access_time/100,l/20,w/25,
                                                type/rom1,r_capacity/500}
RAM = {cost/480,capacity/200,access_time/100,l/20,w/3,type/ram2,r_capacity/240}
Space = 96
        % The LSI layout system selected another type of RAM.  There is
        %space left yet.  Attempt to increase the RAM capacity further.
???--  replace_query(0).
 >> Enter query :  RAM!capacity=400.
        % Backtracking occurs, but we do not have any alternative
        %conditions. Close interaction.
 ** Comment of user_interaction = reasonable cost/performance ratio
 >> Enter user_interaction query(or close/pass/temp_close) :
 close.
LSI = {w/30,l/20,cost/4560,type/lsi31,center/{rom1},right/{cpu1},left/{ram4}}
CPU = {cost/2000,l/20,w/1,type/cpu1,perform/100}
ROM = {cost/1000,capacity/500,access_time/100,l/20,w/25,
                                                type/rom1,r_capacity/500}
RAM = {cost/960,capacity/400,access_time/200,l/20,w/4,type/ram4,r_capacity/480}
Space = 100
                (Figure 5 (5) shows the chip at this point.)
        % We try to improve the layout.
                :
                (Several queries are omitted.)
        % First, we attempted to reduce the size of the LSI to 22mm wide
        %and 20mm long.  The system laid out LSI using other
        %components.  So space was generated.  We tried to increase the RAM
        %capacity to 500 bytes.  There was still space left.
        %We also tried to increase the ROM to 600 bytes.  But the access time
        %of RAM became 200ns.  The access time must be less than
        %or equal to 100ns.  So we added a new constraint for the access time.
        %We did not know the detailed specification relating to the
        %access time at first, but it was clear that the solution was
        %unsatisfactory.  We removed the unsatisfactory solution by
        %adding a new constraint. Now ROM access time became 200ns also.
                :
        %We add a new constraint that the access time of ROM must be less than
        %or equal to 100ns.
???--  le(ROM!access_time,100).
I cannot solve the following queries.
-----------------------
[0]  le(ROM!access_time,100)
-----------------------
[1]  le(RAM!access_time,100)
-----------------------
[2]  RAM!capacity=500
-----------------------
[3] user( CPU!perform=100,ge(RAM!capacity,100),le(LSI!cost,4500);
     CPU!perform=150,ge(RAM!capacity,50),le(LSI!cost,7000);
     CPU!perform=100,ge(RAM!capacity,200),le(LSI!cost,5000))
-----------------------
```

```
[4]  ram(RAM)
----------------------
[5]  space(LSI,Space)
----------------------
[6]  ROM!capacity=600
----------------------
[7]  cpu(CPU),rom(ROM)
----------------------
[8]  arrangement(LSI,[CPU,ROM,RAM])
----------------------
[9]  LSI!w=22,LSI!l=20
        % The layout failed.  But, we want to know how the layout changes
        %if we provide a new type of ROM.  We add the new type of ROM
        %as a hypothesis using the "add_hypo" command.
???--  add_hypo.
 >> Enter hypothetical clause (or end) :
 rom(ROM) :- template_rom(ROM,rom0,100,3,1).
        % We use a template clause for ROM to define a new ROM,
        %whose type, access time, capacity per space and capacity
        %per cost are rom0, 100ns, 3 and 1 respectively.
 >> Enter hypothetical clause (or end) :
 end.

LSI = {w/22,l/20,center/{rom0},cost/4160,type/lsi31,right/{cpu1},left/{ram2}}
CPU = {cost/2000,l/20,w/1,type/cpu1,perform/100}
ROM = {cost/600,capacity/600,access_time/100,l/20,w/10,type/rom0,r_capacity/600}
RAM = {cost/1120,capacity/500,access_time/100,l/20,w/7,type/ram2,r_capacity/560}
Space = 81
                (Figure 5 (6) shows the chip at this point.)
        % Attempt to reduce the cost of the LSI.
???--  le(LSI!cost,4000).
                :
Space = 90
                (Figure 5 (7) shows the chip at this point.)
        % Attempt to reduce its cost further.
???--  le(LSI!cost,2800).
I cannot solve the following queries.
                :
        % The layout failed.  How would it be if we reduced the size of
        %the LSI.
???--  replace_query(11).
 >> Enter query :  LSI!w=22,LSI!l=18.
LSI = {w/22,l/18,center/{rom0},cost/2752,type/lsi31,right/{cpu2},left/{ram2}}.
CPU = {cost/700,l/18,w/3,type/cpu2,perform/100}
ROM = {cost/648,capacity/600,access_time/100,l/18,w/12,type/rom0,r_capacity/648}
RAM = {cost/1008,capacity/500,access_time/100,l/18,w/7,type/ram2,r_capacity/504}
Space = 100
                (Figure 5 (8) shows the chip at this point.)
        % We may try to get a far better LSI while changing conditions.
                :
                (Several queries are omitted.)
                :
        %We tried to reduce the cost and the size.  But we cannot get
        %a better LSI than the previous one.
        %We ought to be content with the previous layout.
```

We have completed a LSI layout, though we did not know the full detailed specification for the LSI at first. CPU and ROM were laid out first on the LSI chip, but we found that there was space left. So we placed a RAM on the LSI. Next, we modified the constraints while looking at the intermediate solutions to get a better LSI. Modification was easy using interactive query revision. We also wanted to take account of the cost/performance ratio, but the system does not have such knowledge because it differs with designers and over time. We input it to the system using "user interaction" keeping an eye on intermediate solutions. It might be difficult for even an expert designer to find it without considering the intermediate solutions. When a layout failed, we added one hypothesis to know how solutions varied using a new type of ROM. Finally, we got a satisfactory LSI. If we had not used interactive query revision, we would have had to enter many sets of goals to the LSI chip layout system and we might not have been able to include knowledge the system did not have, such as the reasonable cost/performance ratio.

It may not be easy to see the output of variables in this example. But it can be easily enhanced by connecting a special output subsystem, like the incremental query system connected to the spreadsheet interface described in [Emden 1985].



S.E. = Space-utilization Efficiency

Figure 5 LSI Configuration

5. Concluding remarks

In this paper we discussed the contribution of logic programming to inference systems and pointed out that one phase of problem solving has been ignored, which is quite indispensable from the viewpoint of interactive problem solving. The limitations of current problem solving systems are obtained from the following observations: (1) The user does not exactly know what he wants in the solution. (2) The user wants to know the relation between solutions and constraints/hypotheses. (3) The user does not perform a function as part of the inference engine. An alternative query model called the interactive query revision model was introduced to resolve these issues. Using an LSI chip layout system, it was shown how our query model has achieved man-machine interaction in a satisfactory manner, and how it can solve a problem interactively under cooperation between the user and the computer.

We developed our query model on a Prolog system. Since this query model is general, we plan to combine it with a powerful inference system that can perform various inference functions such as common sense reasoning, ambiguous reasoning using certainty factors, hypothetical reasoning, default reasoning, qualitative reasoning and so on.

Acknowledgment

We wish express our thanks to Prof. M.H. van Emden for introducing us to this research field. And we wish express our thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity of doing this research in the Fifth Generation Computer Systems Project at ICOT. We would also like to thank Kuniaki Mukai, who patiently taught us CIL, and the ICOT research staff.

References

[Bowen 1982] D.L.Bowen, L.M.Pereira, F.C.N.Pereira and D.H.D.Warren: User's guide to DECsystem-10 Prolog. Dept of Artificial Intelligence, University of Edinburgh (1982).

[Bowen 1985] K.Bowen, T.Weinberg: A Meta-Level Extension of Prolog, Technical Report CIS-85-1, Syracuse University (1985).

[Colmerauer 1982] A.Colmerauer: Prolog II: Reference Manual and Theoretical Model,Internal Report, Groupe Intelligence Artificielle, Universite d'Aix-Marseille II, (1982).

[Emden 1984] M.H.Emden: Logic as an Interaction Language, Proc. of 5th Conf. Canadian Soc. for Computational Studies in Intelligence (1984).

[Emden 1985] M.H.Emden, M.Ohki, A.Takeuchi: Spreadsheet with Incremental Queries as a User Interface for Logic Programming, ICOT Technical Report (1985).

[Mukai 1985] K.Mukai, H.Yasukawa: Complex Indeterminates in Prolog and its Application to Discourse Models, New Generation Computing, 3, pp441-466 (1985).

[Nakashima 1982] H.Nakashima: Prolog/KR - languages features, Proc. of the First International Logic Programming Conference, (1982).

[Poole] D.Pools, R.Aleliunas, R.Goebel: Theorist: a logical reasoning system for defaults and diagnosis, Waterloo University (1985).

[Sergot 1983] Marek Sergot: A Query-The-User for Logic Programming, Proc. of the European Conference on Integrated Computing Systems, P. Degano and E. Sandewall (eds.), North Holland, 1983.

[Shapiro 1983a] E.Shapiro: Algorithmic Program Debugging, The MIT Press, 1983

[Shapiro 1983b] E.Shapiro: Logic Programs with Uncertainties: A Tool for Implementing Rule-based Systems, Proc. of IJCAI'83, 1983

[Takeuchi 1985] A.Takeuchi, K.Furukawa: Partial Evaluation of Prolog Programs and its Application to Meta Programming, Proc. of Logic Programming Conference'85 (Tokyo), (1985). .