ICOT Technical Report: TR-037

TR-037

A Methodology for Implementation of A Knowledge Acquisition System

by

H. Kitakami, S. Kunifuji,

T. Miyachi and K. Furukawa

December, 1983

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 METHODOLOGY FOR IMPLEMENTATION OF A KNOWLEDGE ACQUISITION SYSTEM

H.KITAKAMI, S.KUNIFUJI, T.MIYACHI and K.FURUKAWA,

Institute for New Generation Computer Technology (ICOT)
Mitakokusai Building (21F)
1-4-28, Mita, Minato-ku, Tokyo, 108, Japan

ABSTRACT

This paper describes an investigation conducted on a Knowledge Acquisition System for a Knowledge Base System, and discusses a conceptual configuration and implementation method for some of the mechanisms in this system. These include a meta inference mechanism, induction mechanism, knowledge assimilation mechanism and knowledge accommodation mechanism. These mechanisms should enable this system to accumulate the knowledge suited to the user's purposes. The induction mechanism will be realized by improving upon Shapiro's system. The discussion of the implementation method for the induction mechanism attempts to explain speeding up strategy, a strategy designed mainly to prevent recomputation by retaining refuted hypotheses.

These mechanisms include the manipulation of facts, rules and integrity constraints as knowledge.

Finally, the authors present certain execution traces of this system.

1. INTRODUCTION

To approximate human mental processes, a computer system must be equipped with various mechanisms. One is a mechanism for systematically storing and managing human knowledge. This mechanism is known as the knowledge base system [1]. As part of our research, we are studying methodologies for implementing a knowledge acquisition system [2,3,4]. Knowledge acquisition means the function of collecting knowledge by assimilation [5,15,16] and accommodation in a knowledge base.

In this paper, knowledge stored in a knowledge base is regarded as (i) facts (individual facts), (ii) rules (general rules) and (iii) integrity constraints (integrity constraints on facts and rules) ; and the knowledge base format, as a deductive question-answering system for relational databases.

Relational databases which support SEQUEL, QUEL or other data manipulation languages as a deductive function handle mainly facts as knowledge [6,7,8,9]. Knowledge base, however, must also treat rules as knowledge on a full scale basis. In order to realize this, we feel that more powerful database (knowledge base) management techniques have to be investigated in predicate logic.

This paper discusses the conceptual configuration and methodology for the implementation of a knowledge-based knowledge acquisition system.

The results of this system's implementation in Prolog are also reported. The programs were all interpreted under the Edinburgh Prolog-10 [10].

2. CONFIGURATION OF THE KNOWLEDGE ACQUISITION SYSTEM

This section discusses the conceptual configuration of a knowledge acquisition system. Knowledge in the knowledge base will basically take the form of Horn clauses. In order to systematically acquire expert knowledge, a knowledge acquisition system repeats assimilation and accommodation in the knowledge base. The user then monitors the knowledge base using a deductive question answering mechanism. These operations enable this system to accumulate knowledge suited to the user's purposes.

Figure 1 illustrates the concept of a knowledge acquisition system constructed for knowledge base systems.

This system includes a meta inference mechanism which is defined as an amalgamation of object language (Prolog) and meta language. The theoretical study of meta inference is described by Kowalski et al [11,12]. Implementation of this mechanism is discussed in the next chapter.

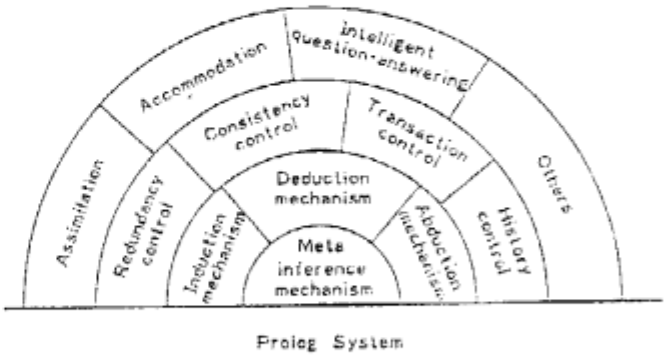

Figure 1. Concept of Knowledge Acquisition System.

The deduction mechanism is used to prove certain kinds of clauses in the knowledge base. Clauses mean Horn clauses in this system.

The induction mechanism is treated as a model inference drawn from Shapiro's research [20,21] in Prolog. It can play a role in the following knowledge acquisition operations :

(1) Creation of the knowledge suited to user's intentions.
(2) Revision of existing knowledge (mainly rules and facts).
(3) Self-organization of existing knowledge.

We are capable of obtaining useful results for (1) and (2) now, but (3) requires further research.

The abduction mechanism is the most difficult mechanism in this system, and includes an analogical reasoning mechanism and so on. This mechanism requires further research.

History control is the function for solving various past, present and future propositions by recording the assimilation and accommodation processes in the knowledge base.

Redundancy control is the function for eliminating redundancy, and is carried out at the user's discretion each time new knowledge is acquired by the knowledge base.

Consistency control is the function for defense against inconsistencies in the knowledge base.

Assimilation is used to store knowledge suited to the user's intentions in the knowledge base. User's intentions are defined by integrity constraints. In this case, existing knowledge is not revised by assimilating external knowledge.

Accommodation is used to revise existing knowledge using external knowledge supplied by the user, which is always regarded as being correct in this operation. Some accommodation methods are listed below:

(1) Revisions of existing knowledge based on facts provided by the user;
(2) Revisions carried out directly by the user;
(3) Revisions carried out by combining methods (1) and (2);
(4) Deletion of existing knowledge which subsequently becomes unnecessary. For details, refer to [13].

When two or more assimilations and accommodations are used as a single command, the operation in the single command may not satisfy intermediate integrity constraints. Transaction control is required to provide facilities for this purpose.

Intelligent question-answering [17] provides a knowledge base question-answering capability based on meta predicates. From this result, the user can decide which of mechanism, the assimilation or accommodation, to apply the knowledge base.

Other functions include those designed to prove inclusion and equality related to knowledge base rules. These are important functions for effective utilization of knowledge bases built in other fields. The call-graph for predicates realized previous mechanisms is shown in APPENDIX-1.

3. META INFERENCE AND DEDUCTION MECHANISM

The meta inference and deduction mechanism can be realized by means of the "demo" predicate and the "deduce" predicate, respectively. A "demo" predicate is used to solve the following problems:

(1) Prove ground clauses.
(2) Prove general clauses.
(3) Prove goals (Prolog interpreter in Prolog).
(4) Others.

This chapter discusses the implementation of "demo" predicate and "deduce" predicate. The "deduce" predicate is implemented to solve problems (1) and (2). Ground clauses don't contain variables but general clauses contain a variable in the argument. The "deduce" predicate is used to implement knowledge assimilation described in chapter 5. The "demo" predicate can solve problem (3).

Figure 2 illustrates an implementation of the "deduce" predicate to prove clauses (1) and (2). These clauses are assumed to take the following form:

(A) Head: -Goals.
(B) Head.

However, "Head" represents one predicate and "Goals" consists of the logical sum (alternatively, logical OR) and logical product (alternatively, logical AND) of predicates. "Goals" is allowed to contain the cut operator and the system predicate. Hereafter, the clause means knowledge because it is the primitive unit for knowledge assimilation and accommodation in this system.

The "deduce" predicate in Figure 2 has three arguments. The first argument gives the name list, KBNL, of the knowledge bases to be used; the second argument specifies the ground/general clauses to be proved; and the third argument specifies identification (ID) of knowledge in the existing knowledge base which is not useful to solve this problem.

The "demo" predicate in Figure 2 is known as a Prolog interpreter [16,21] in Prolog and is extended by the authors using Kowalski's idea [11,12]. This "demo" predicate has four arguments. The first and second arguments have the same specifications as the "deduce" predicate; the third argument gives the following information to control the resolution process:

(1) Temporary knowledge (Goals) used in this process. Each Goal of the Goals gives a knowledge defined in the knowledge base.
(2) Identification (ID) of non refered knowledge.
(3) Control variable to manipulate the cut operator in the goals.

(4) Maximum number of resolution steps.

The fourth argument returns the following resolution results:

(1) Truth value (true/overflow).
(2) Proof tree, if truth value is overflow.

The "select_variable_list" predicate is the predicate that selects the variables from the clause given by the user.

The "skolem" predicate provides constants (unique in the system) to the variables.

The "clause_kb" predicate is the predicate that searches for the "goals" which correspond to the "Head" from the knowledge base list, KBNL. The "next_clause" predicate searches for the "goals" which corresponds to the "Head" from the knowledge base, KBN.

The physical structure of knowledge is as follows:

```
KBN(P1,P2,P3,Clause).
```

Each piece of knowledge in the knowledge base is linked by a before pointer P3 and forward pointer P2. P1 is the address pointer of the clause. The knowledge identifier ID is defined by these three pointers [P1,P2,P3]. These pointers control the stored order of knowledge as procedure in the knowledge base.

```
/* Deduce ground/general Clause */

deduce(KBNL,Clause,Cond):-
  verify( (select_variable_list(Clause,Variable_list),
   skolem(Variable_list),
   (Clause=(P:-Q)->demo(KBNL,P,[0,Cond,Cut,50],[true,[]]);
    demo(KBNL,Clause,[[],Cond,Cut,50],[true,[]])) ).

verify(P):- \+(\+(P)).

demo(KBNL,true,[Res,Cond,Cut,Count],[true,[]]):-!.
demo(KBNL,Goals,[Res,Cond,Cut,0],[overflow,[]]):-!.
demo(KBNL,!,[Res,Cond,Cut,Count],[Result,Stack]).
demo(KBNL,!,[Res,Cond,cut,Count],[Result,Stack]).
demo(KBNL,(P;Q),[Res,Cond,Cut,Count],[Result,Stack]):-!,
    (demo(KBNL,P,[Res,Cond,Cut,Count],[Result,Stack]) ;
     demo(KBNL,Q,[Res,Cond,Cut,Count],[Result,Stack])).
demo(KBNL,(P,Q),[Res,Cond,Cut,Count],[Result,Stack]):-!,
    demo(KBNL,P,[Res,Cond,Value,Count],[Result1,Stack1]),
    (Value=cut,Cut=cut,Result=Result1,Stack=Stack1,! ;
     Result1=true->
     demo(KBNL,Q,[Res,Cond,Cut,Count],[Result,Stack]) ;
     Result=Result1,Stack=Stack1).
demo(KBNL,P,[Res,Cond,Cut,Count],[Result,Stack]):-
    system(P)->P,Result=true,Stack=[] ;
    Count1 is Count-1,
    clause_kb(KBNL,P,ID,Q,[Res,Cond]),
    demo(KBNL,Q,[Res,Cond,Cut,Count1],[Result1,Stack1]),
    (Cut=cut,!,fail ;
     Result1=true->Result=true,Stack=[] ;
     Result=overflow,Stack=[(P:-Q)|Stack1]).

clause_kb((KBN,KBNL),Head,ID,Goals,[Res,Cond]):-!,
    ( clause_kb(KBN,Head,ID,Goals,[Res,Cond]) ;
      clause_kb(KBNL,Head,ID,Goals,[Res,Cond]) ).
clause_kb(KBN,Head,ID,Goals,[Res,Cond]):-
    next_clause(KBN,ID,Clause,Cond),
    ( Clause=(Head:-Goals) ;
      Clause=Head,Goals=true  ) .
clause_kb(KBN,Head,[],true,[Res,Cond]):-
    Res\==[],unify(Res,Head).

/* Unify between                                    */
/* one of user defined predicates and a predicate */
unify((C,CL),P):-!,
    (unify(C,P);unify(CL,P)) .
unify(P,P).
```

Figure 2. A Prolog Program for the "deduce" Predicate for Proof of Ground/General Clauses.

4. INDUCTION MECHANISM

The induction mechanism in Figure 1 will be treated as a problem concerning inductive model inference from facts. The model inference mechanism is realized by improving upon Shapiro's system [20,21].

This section discusses our research into model inference, and presents the relations between "demo" predicate and the model inference. We also discuss a speeding up strategy for the model inference.

4.1 Model inference

Figure 3 shows an incremental model inference algorithm presented by Shapiro, where Fn=<OBSn,V>, n=1,2,...,m,(m:constant) is a sequence of observation data and the observation sentence "OBSn" is a fact and "V" is the truth value of "true" or "false".

```
Sfalse:={□}, Strue:={ }.
L0:={□}, mark □ "false".
repeat
  Read the next observation data Fn=<OBSn,V> and add OBSn to Sv.
 repeat
   while ( Derive OBS from conjecture Lp, OBS belongs to Sfalse ) do
     By contradiction backtracing,
     discover a refuted hypothesis H and mark it "false".
     Delete H from Lp.
   while ( ~(Derive OBSi from conjecture Lp), OBSi belongs to Strue ) do
     By a refinement operation applied to the hypothesis marked "false",
     discover new hypothesis Hi which is satisfiable to derive Hi to OBSi
     and add Hi to Lp ( Lq = Lp ^ Hi, q=p+1 ).
 until ( Neither of the while loops is entered ).
 output Lp.
forever.
```

Figure 3. An Incremental Model Inference Algorithm.

The set of discovered hypotheses is the procedures (set of rules) programmed in Prolog. Before this algorithm reads the first observation data, which has a "true" observation sentence, the conjecture "L0" consists of contradiction "□" only. After this algorithm reads it, conjecture "Lp", (p>0) consists of contradiction "□" and hypotheses. In order to discover the new hypothesis, this algorithm searches the refinement graph. The search method on the refinement graph is a breadth-first search and the refinement graph is generated by a refinement operation.

The mechanism to derive "OBS" from "Lp", shown in Figure 4, is the meta inference mechanism. This mechanism was implemented by meta predicate "solve" in Shapiro's system. Our system expands this predicate "solve" into the "model_demo" predicate shown in Figure 4. In this way, the physical separation among the knowledge base can be obtained for the model inference system.

Figure 5 shows a Prolog program which applies a "model_demo" predicate to Figure 3. The predicate "model_inference" has recursive structure. This predicate extends from the first "repeat" procedure to the "forever" procedure shown in Figure 3. The predicate "model_inference1" also has recursive structure.

This predicate extends from the second "repeat" procedure to the "until" procedure. The "Too Weak" part in the predicate "model_inference1" covers up to the first "while" procedure. The "Too Strong" part in "model_inference1" covers up to the second "while" procedure. The observation data stored in the internal database can be found by the predicate "fact".

```
model_demo(KBNL,Goals):-
   demo(KBNL,Goals,[[],[],Cut,50],[Result,Stack]),
   (Result\==true,! ; true),
   (Result=overflow ->nl,
   stack_overflow(KBNL,Goals,Stack),
   model_demo(KBNL,Goals) ; true).
```

Figure 4. A Prolog Program for The "model_demo" Predicate.

```
model_inference(KBN,IC_name,Concept) :-
   nl,ask_for('Next fact(sentence,true/false) or end ',Fact),
   ( Fact=end ;
   ( Fact=check -> model_inference1(KBN,IC_name,_) ;
     ( Fact=(P,V),verify(P=Concept) ;
     nl,write('* Error Concept Form.'),fail ),!,
     Fact=(P,V), (V=true ; V=false)  ->
     assert_fact(P,V), model_inference1(KBN,IC_name,P) ;
     write('!Illegal input'), nl ),!,nl,
   model_inference(KBN,IC_name,Concept) ).
model_inference(KBN,IC_name,Concept):-
   nl,ask_for('Next fact(sentence,true/false) or end ',Fact),
   ( Fact=end ;
   ( Fact=check->model_inference1(KBN,IC_name,_) ;
     Fact=(P,V),(V=true;V=false) ->
     assert_fact(P,V),model_inference1(KBN,IC_name,P) ;
     write('!Illegal input'),nl ),!,nl,
   model_inference(KBN,IC_name,Concept) ).

model_inference1(KBN,IC_name,P) :-
   write('Checking fact(s)...'), ttyflush,
   ( fact(P,false),model_demo(KBN,P) ->
       nl, false_solution(KBN,IC_name,P),
       model_inference1(KBN,IC_name,_) ;    % (Too Strong)
     fact(P,true),\+model_demo(KBN,P) ->
       nl,missing_solution(KBN,IC_name,P),
       model_inference1(KBN,IC_name,_) ;    % (Too Weak)
     write('no error found.'),nl ).
```

Figure 5. A Prolog Program for Model Inference System.

The refinement operation in Figure 3 , which takes a clause "p" (alternatively, generating hypothesis) as an input and generates a clause "q" (alternatively, generated hypothesis) as an output, is one of the following four cases:

(1) q ::= a(X1,X2,X3,...,Xn) if p ::= □, where "a" is an inductive generalized predicate symbol with n arguments.

(2) q ::= unify Xi and Xj for p(X1,X2,X3,...,Xn), where "Xi" and "Xj" are distinct variables.

(3) q ::= instantiate Xi to t(Y1,Y2,...,Ym) for p(X1,X2,X3,...,Xn), where "t" is a functor with m arguments.

(4) q ::= (Head:-Goals,Gn) and p ::= (Head:-Goals), where "Gn" is a user defined predicate, or recursively defined predicate whose symbol name is the same as that for the given "Head's" symbol name.

4.2 Speeding up strategy

The speeding up strategy for model inference is summarized as follows:

(1) Type (structural) specifications of predicate arguments.
The Specified type can be used to infer a model without consideration of other types.

(2) Specifications of the user defined predicate.
The user defined predicate can be used to quickly generate hypotheses with simple structure in short time.

(3) I/O specifications of predicate arguments.
If the I/O specifications are used to select hypothesis whose output arguments are controllable for input arguments, this hypothesis has useful structure as model.

(4) Selecting a hypothesis with a useful structure.
If the hypothesis has a "Goals" part, any predicates of the "Goals" are not equal in structure.
If the hypothesis has recursive structure, the "Head" and recursive predicate of the hypothesis also are not equal in structure, where the recursive predicate belongs to the "Goals" part in the hypothesis.

(5) Preventing recomputation by retaining refuted hypotheses.

(6) Others.

Strategies (1)-(4) have already been discussed in the section on the model inference system [20], but the first discussion of (4) was not enough.

This section will discuss strategy (5) in Figure 6. We assume that the search strategy of the refinement graph in Figure 5 is a breadth-first search and our goal is to search for hypothesis "Hj". If the hypothesis "H6" is refuted by a false sentence, the system can search for a new hypothesis, "Hj", without the need for recomputation from "H0" by retaining hypotheses , "H2" --- "H4". These hypotheses are stored into the knowledge base by executing the predicate "insert_knowledge". The predicate "insert_knowledge" is mainly implemented by system predicate "assert". In order to mark the refuted hypothesis "H6" with a "false" statement, the system also stores this hypothesis in the knowledge base "refuted_hypothesis". The new hypothesis is discovered from generated hypotheses, "H5" --- "Hj" --- , using generating hypotheses, "H2" --- "H6", retained in the knowledge base. The predicate to execute this procedure is shown in the call-graph in APPENDIX-1. In the call-graph, the predicate "current_status" is the predicate to find the generating hypothesis for the "true" observation

sentence. The predicate "set_new_status" is the predicate to retain refining status, "H2" --- "Hk", for the "true" observation sentence. The results of this experiment are shown later.

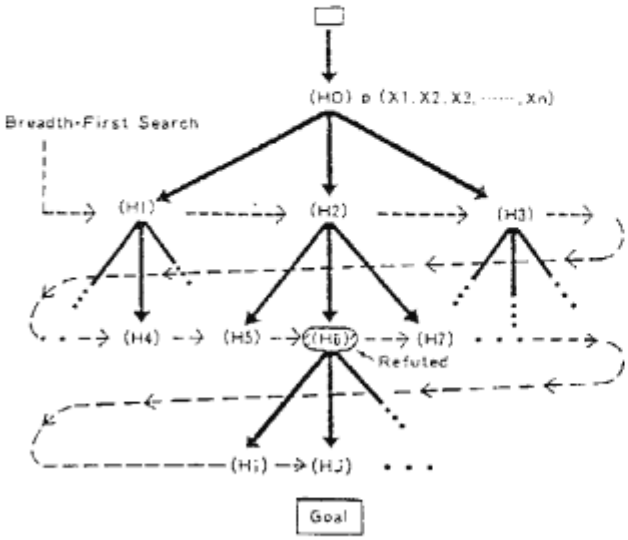

Figure 6. The Speeding up Strategy in Refinement Graph.

5. KNOWLEDGE ASSIMILATION

This section describes an algorithm for assimilation as shown in Figure 1 on the basis of Kowalski's idea [12,13]. In our implementation, the algorithm is predicated on the following major assumptions:

(1) The input knowledge is constructed by predicates defined in the knowledge base.
(2) The knowledge base under consideration is consistent with integrity constraints. (Integrity constraints are also consistent.)
(3) Knowledge (rules, facts and integrity constraints) in the knowledge base are described in Prolog.
(4) The system adds the knowledge to be assimilated just after the existing knowledge in the knowledge base.

The assimilation algorithm is organized as follows:

(A1) If the integrity constraints concerned can't be proved from the knowledge based on the action that input knowledge has been inserted into the knowledge base, restore the knowledge base to its original state because it is inconsistent.
(A2) Remove redundancy from the knowledge base on the assumption that input knowledge has been inserted into the knowledge base. However, redundancy removal processing depends on user's intention.
(A3) If the integrity constraints concerned can't be proved from the knowledge base on the grounds that redundunt knowledge has been removed from the knowledge base, restore the knowledge base to its original state (In (A2), redundancy may be removed.).
(A4) If input knowledge qualifies for insertion under all of procedures (A1) and (A3) , insert the input knowledge into the knowledge base.

Figure 7 shows a Prolog program for assimilation by this algorithm. The predicate "assimilate" has three arguments. The first argument gives the name, KBN, of the knowledge base which stores facts and rules. The second argument gives the name, IC_name, of the knowledge base which stores integrity constraints. The third argument specifies input knowledge.

```
/* (A1) : Is ["Input"+"KBN"] inconsistent with "ICs"  ?          */
assimilate(KBN,IC_name,Input):-
     insert_knowledge(KBN,Input,ID1),
     assert(inserted(KBN,ID1,Input)),
     IC_name\==[],
     head_part(Input,Head),
     check_inconsistency(IC_name,ID2,KBN,insert,Head),
     restore_kb(KBN),
     message(insert_inconsistent,IC_name,ID2,Input).

/* (A2) : Is ["New_kb" = "Input"+"KBN"] redundant  ?             */
assimilate(KBN,IC_name,Input):-
     head_part(Input,Head),
     mgt(Head,Concept),
     query_eliminate_redundancy,
     assert(eliminate_redundancy),
     eliminate_redundancy(KBN,Concept),fail.

/* (A3) : Is ["New_kb"] inconsistent with "IC"  ?                */
assimilate(KBN,IC_name,Input):-
     IC_name\==[],
     eliminate_redundancy,
     head_part(Input,Head),
     check_inconsistency(IC_name,Id,KBN,delete,Head),
     restore_kb(KBN),
     message(insert_inconsistent,IC_name,Id,Input).

/* (A4) : Post_processing.                                       */
assimilate(KBN,IC_name,Input):-
     post_processing(KBN,Input),
     message(assit_success,Input).

/* Check inconsistency in the knowledge base with ICs.           */
check_inconsistency(IC_name,ID,KBN,Action_Mode,Head):-
     next_clause(IC_name,ID,(inconsistent(Actions,Head):-Goals),[]),
     member(Action_Mode,Actions),
     meta_demo(KBN,Goals).

/* Take out head part from the clause.                           */
head_part(Clause,Head):-Clause=(Head:-Goals),!.
head_part(Clause,Head):-Clause=Head.

/* Eliminate redundancy */
eliminate_redundancy(KBN,ID):-
   next_clause(KBN,ID1,Clause,[ID]),
   decuce(KBN,Clause,[ID1]),
   update_knowledge(KBN,[],ID1),
   assert(updated(KBN,ID1,Clause)),
   message(redundant,KBN,Clause),
   fail.
```

The "mgt" predicate is the predicate defined below:

```
mgt(P,P0):-functor(P,F,N),functor(P0,F,N).
```

Figure 7. A Prolog Program for Knowledge Assimilation.

In Figure 7, the predicate "insert_knowledge" and "update_knowledge" are predicates to insert

and update knowledge in the knowledge base, respectively. The predicate "assert" is used to control the assimilation procedure. The predicate "check_inconsistent" is the predicate to check inconsistency in the knowledge base with integrity constraints named "IC_name". This predicate has five arguments. The first and third argument give the name, IC_name, of integrity constraints and the name, KBN, of the knowledge base, respectively. If this predicate discovers inconsistency using an integrity constraint, and returns the identifier, ID, of the integrity constraint to the second argument. The fourth argument specifies a user action, "insert", "delete" or "update" to check the integrity constraints. The fifth arguments gives the concept for "Head" checked by integrity constraints. The predicate "restore_kb" is the predicate to restore the knowledge base to its original state. The predicate "query_redundancy_elimination" asks for a decision concerning whether or not do eliminate redundancy. The predicate "eliminate_redundancy" is the predicate to eliminate redundancy in a given concept. The syntax of integrity constraints is as follows form:

```
inconsistent(Action_list,Concept):- Condition.
```

User's action, "Action_list", for the "Concept" is inconsistent, if the "Condition" is "true". The "Action_list" gives some actions ("insert", "delete" and "update") for the knowledge base. The "Condition" gives some goals which also include the meta predicate if necessary.

6. KNOWLEDGE ACCOMMODATION

Knowledge accommodation for a knowledge base is similar to the debugging of logic programming, and includes functions for the revision of existing knowledge from facts and the direct replacement of existing knowledge with new knowledge. For this operation, this accommodation is necessary in the following procedures:

(1) Revisions of existing knowledge in the knowledge base.
(2) Redundancy elimination (depends on user's intention).
(3) Knowledge base recovery by means of discovering contradictions.

Knowledge accommodation can be executed using the following predicates (For example, see APPENDIX-2):

```
accommodate( KBN, IC_name, AK).
```

The first argument gives the name, KBN, of the knowledge base to be accommodated; the second argument gives the name, IC_name, of the knowledge base to be specified by integrity constraints for "KBN"; the third argument specifies the most general form, AK, of the knowledge to be accommodated. If the "AK" knowledge is not in the knowledge base, the execution of this "accommodate" predicate fails because it is meaningless. In this case, we may try to create new knowledge from several facts.

In order to create new knowledge from facts, this system has the following predicate which is similar to the "accommodate" predicate:

```
create( KBN, IC_name, CK).
```

The first two arguments have the same specifications as the "accommodate" predicate. The third argument specifies the most general form, CK, of the knowledge to be created.

7. EXAMPLES OF KNOWLEDGE ACQUISITION PROGRAM EXECUTIONS

This chapter presents several kinds of examples using the well-known building block problem. One is a knowledge assimilation/accommodation problem and the other is a model inference problem used in the previously described speeding up strategies. In addition to these problems, this chapter presents an example of DCG (Definite Clause Grammar).

7.1 Problem setting

Figure 8 shows an arrangement of building blocks. Suppose that towers are built on floor "a", using building blocks "b", "c",..., "h" and "i". Building block "f" is rectangular, the others square. Assume further that building blocks "j" and "k" are in hand, characterized by the absence of data related to "j" and "k" in the predicate called the relation "on". "j" is rectangular, "k" square.

The Prolog program in Figure 9 shows the location relations of the building blocks presented in Figure 8. Each building block is represented by the predicate "block"; their stack relations, by the predicate "on".

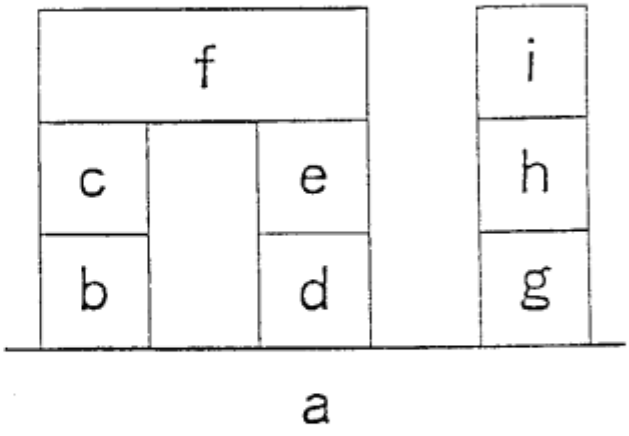

Figure 8. Arrangement of Building Blocks.

The facts in this knowledge base are the predicates "rectangular_block", "square_block", "floor", and "on"; and the rules are the

"floor", and "on"; and the rules are the predicates "block" and "tower". The knowledge base is named "build".

```
/* rectangular blocks */

floor(a).
rectangular_block(f).
rectangular_block(j).

/* square blocks */

square_block(b).
square_block(c).
square_block(d).
square_block(e).
square_block(g).
square_block(h).
square_block(i).
square_block(k).

/* block */

block(X):-
   square_block(X);
   rectangular_block(X);
   floor(X).

/* on(X,Y) : X is on Y */

on(b,a).
on(d,a).
on(c,b).
on(f,c).
on(e,d).
on(f,e).
on(g,a).
on(h,g).
on(i,h).

/*   tower(X,Y) : The tower consists of block X */
/*             on top of tower Y.               */

tower(X,Y):-tower1(X,Y), Y\==[].
tower1(X,[]):-floor(X).
tower1(X,[Y|Z]):-block(X),on(X,Y),tower1(Y,Z).
```

Figure 9. Knowledge Base Represented Location Relations of Building Blocks.

The following restrictions are assumed as integrity constraints (ICs).

(1) "floor(a)" is never deleted.
(2) "rectangular_block" is supported by two or more "towers".

Figure 10 shows these relations. This knowledge base is named "ic".

```
/* Integrity Constraints */

inconsistent([delete],floor(_)):- \+meta_demo(build,floor(a)).

inconsistent([delete,insert,update],on(_,_)):-
rectangular_block(X),
setof_kb(build,[Y],(tower(X,Y),me(Y,[])),S), length(S,N),N<2.


The "setof_kb" and "meta_demo" predicates are defined below:

setof_kb(KBNL,X,Goals,Set):-setof(X,meta_demo(KBNL,Goals),Set).
meta_demo(KBNL,Goals):-demo(KBNL,Goals,[[],[],Cut,50],[true,[]]).
```

Figure 10. Integrity Constraints on Building Block Knowledge Base.

From the foregoing, the program is executed to solve the following examples:

(Example 1) Problems of integrity constraints related to assembly of building blocks.
(Example 2) Accommodation of the predicate "above".
(Example 3) Model inference problem of "arch".

In addtion to these examples, we try to execute the following example:

(Example 4) Synthesis of the simple DCG (Definite Clause Grammar). In order to make inferences based on as few observation data as possible, an integrity constraint is defined below:

```
inconsistent([insert],s(_,_)):-
                    s(X,[]),exist_variable(X).
```

We assume that the sentences and dictionary of DCG are stored in the knowledge base as "dcg". The sentence predicate of DCG is named "s".

Results of the execution are given in the APPENDIX-2. Shapiro's model inference system cannot solve Example 3 under Edinburgh Prolog-10 because the marking shown in Figure 3 is not sufficient for discovering new hypothesis. Our system can solve it from 1 fact and 1 integrity constraint. We can also solve Example 4 from 2 facts and 1 integrity constraint. Shapiro's report [21] solves a similar example using a specific refinement operation which is DCG oriented, but our system solves it using a general refinement operation and integrity constraints.

8. CONCLUSION

We have discussed a conceptual configuration for a knowledge-based knowledge acquisition system and presented results of its implementation in Prolog. The implementation parts in this system are meta inference, deduction, assimilation, accommodation, deductive question_answering mechanism and model inference system improved for speed. We have also obtained useful results for some execution of the system.

All programs are interpreted under the Edinburgh Prolog-10 on a DEC2060.

Our future research plans will concern the following subjects:

(1) A method for implementing knowledge acquisition which allows input of two or more clauses of the knowledge, instead of only one as discussed here, in each processing run.
(2) Generalization and speeding up of redundancy removal.
(3) Generalization and speeding up of inductive model inference.
(4) Inductive inference for integrity constraints from facts.
(5) The problem of proving equality between concepts in knowledge bases.
(6) Speeding up of processing through introducing a parallel computation mechanism.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for many valuable comments, which have greatly improved both the content and the presentation of this paper, and wish to thank Director K.Fuchi, ICOT Research Center for providing us with the opportunity for this research. The authors gratefully acknowledge the useful comments made by Prof. S.Ohsuga of Tokyo University, Prof. S.Arikawa of Kyushu University, Lecturer Y.Tanaka of Hokkaido University and Dr. E.Y.Shapiro of Weizmann Institute of Science. Finally the authors also thank Researcher M.Asou, First Research Laboratory of ICOT, Researcher H.Hirakawa, Second Research Laboratory of ICOT and Dr. T.Yokomori, of Fujitsu Limited, for their useful suggestions.

REFERENCES

[1] Fuchi, K.: Problem Solving and Inference Mechanism, Information Processing Society Journal Vol. 19 No.10 (in Japanese) (1978).
[2] McDermott, J.: ANA: An Assimilating and Accommodating Production System, Tech. Rept. CMU-CS-78-156, Carnegie-Melon University, Dept. of Computer Science (1978).
[3] Grumbach, A.: Knowlegde Acquisition in Prolog, Proceedings of the First International Logic Programming Conference in France (1982).
[4] Shimura, M.: Knowledge Acquisition and Learning, The Journal of the Institute of Electronics and Communication Engineers of Japan (in Japanese) AL80-46 (1980).
[5] Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, S. and Yokota, H.: A Knowledge Assimilation Method for Logic Databases, Research Institute for Mathematical Sciences(RIMS) of Kyoto University, Symposium on "A Study in Theory and Practice of Model Representation and its Construction" Seminar report (in Japanese) (1983).
[6] Chamberlin, D.D. et al.: SEQUEL 2: A Unified Approach to Definition, Manipulation, and Control, IBM J. RES. DEVELOP. (1976).
[7] Stonebraker, M. et al.: The Design and Implementation of INGRES, Memo. No. ERL-M 577, Univ. of California, Berkeley (1976).
[8] Makinouchi, A., Tezuka, M., Kitakami, H. and Adachi, S.: The Optimization Strategy for Query Evaluation in RDB/V1, The 5th International Conference on VLDB (1981).
[9] Kitakami, H., Makinouchi, A., Tezuka, M. and Adachi, S.: Optimization Method in the Relational Database System RDB/V1, Transactions of Information Processing Society of Japan, Vol.24, No.3 (1983).
[10] Bowen, D.L.: DEC-10 PROLOG USER'S MANUAL, University of Edinburgh (1981).
[11] Bowen, K.A., Kowalski, R.A.: Amalgamating Language and Meta Language in Logic Programming, School of Computer and Information Sciences, University of Syracuse (1981).
[12] Kunifuji, S., Asou M., Sakai K., Miyachi, T., Kitakami, H., Yokota, H., Yasukawa, H. and Furukawa, K.: Amalgamation of Object Knowledge and Meta Knowledge in Prolog and its Application, Information Processing Society of Japan, Research Committee Material, "Knowledge Engineering and Artificial Intelligence" (in Japanese) (1983).
[13] Kitakami, H., Miyachi, T., Kunifuji, S. and Furukawa, K.: The Concept of Knowledge Base System KAISER, 26th National Conference of Information Processing Society of Japan (in Japanese) (1983).
[14] Eswaran, K.P. and Chamberlin, D.D.: Functional Specification of a System for Data Base Integrity, Proc. 1st VLDB Conf. (1975).
[15] Kowalski, R.A.: Logic for Problem Solving, Elsevier North Holland, Inc., pp. 239-246 (1979).
[16] Kowalski, R.A.: Logic as a Database Language, Department of Computing, Imperial College (1981).
[17] Furukawa, K.: On Intelligent Access to Data Bases, Information Processing Society Journal, Vol.23 No.10 (in Japanese) (1982).
[18] Coelho, H., Cotta, J.C. and Pereira, L.M.: How to Solve It with PROLOG (2nd. edition), Laboratório Nacional de Engenharia Civil, Lisboa (1980).
[19] Mitchell, T.M.: Version Spaces: An Approach to Concept Learning, Stanford CS Report, STAN-CS-78-711 (1978).
[20] Shapiro, E.Y.: Inductive Inference of Theories From Facts, Technical Report 192, Department of Computer Science, Yale University (1981).
[21] Shapiro, E.Y.: Algorithmic Program Debugging, An ACM Distinguished Dessertation 1982, The MIT Press.
[22] Doyle, J.: Truth Maintenance System for Problem Solving, AI-TR-419 (1978).

APPENDIX-1

Call-graph for assimilate, accommodate, model inference (show the major predicates used) and retrieve. The " * " marked predicates are the predicate to be implemented by Shapiro.

APPENDIX-2

Execution traces of a user running this system.

<< Example 1 >>

This example assimilates the two instances, on(j,i) and on(k,i), with the integrity constraints "ic".

```
$ ?- list_all(ic).

(inconsistent([delete],floor(X)):- \+meta_demo(build,floor(a))).    ...
(inconsistent([delete,insert,update],on(X,Y)):-
     rectangular_block(Z),
     setof_kb(build,[U],(tower(Z,U),mc(U,[])),Y1),length(Y1,U1),U1<2).

yes
$ ?- list(build,on(_,_)).

Listing of on(X,Y):
    on(b,a).
    on(d,a).
    on(c,b).
    on(f,c).
    on(e,d).
    on(f,e).
    on(g,a).
    on(h,g).
    on(i,h).

yes
$ ?- assimilate(build,ic,on(j,i)).

* The Input_Knowledge "on(j,i)"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([delete,insert,update],on(_0,_1)):-
       rectangular_block(_2),
       setof_kb(build,[_3],(tower(_2,_3),mc(_3,[])),_4),
       length(_4,_5),_5<2)".

yes
$ ?- assimilate(build,ic,on(k,i)).

* Eliminate Redundancy ? (y/n) n.

* The Input_Knowledge "on(k,i)"  was assimilated.

yes
$ ?- retrieve(build,on(X,Y),[X,Y]).

[b,a]
[c,b]
[d,a]
[e,d]
[f,c]
[f,e]
[g,a]
[h,g]
[i,h]
[k,i]

yes
$ ?- list(build,on(_,_)).

Listing of on(X,Y):
    on(b,a).
    on(d,a).
    on(c,b).
    on(f,c).
    on(e,d).
    on(f,e).
    on(g,a).
    on(h,g).
    on(i,h).
    on(k,i).

yes
```

<<Example 2>>

In order to efficiently define "above", this example uses both the "assimilate" and "accommodate" predicates.
The first step tries to insert an integrity constraint and to assimilate the "above" knowledge. The second step deductively tests the correctness of the "above" instance. As the "above" concept is not enough, the accommodation of the "above" concept is executed by the "accommodate" predicate in the third step.
The final step deductively infers all the instances of 'above'.

```
$ ?- insert_knowledge(ic,
$ ?- [(inconsistent([insert],above(_,_)):- above(X,X)),_].

yes
$ ?- list_all(ic).

(inconsistent([delete],floor(X)):- \+meta_demo(build,floor(a))).
(inconsistent([delete,insert,update],on(X,Y)):-
     rectangular_block(Z),
     setof_kb(build,[U],(tower(Z,U),mc(U,[])),Y1),length(Y1,U1),U1<2).
(inconsistent([insert],above(X,Y)):- above(Z,Z)).

yes
$ ?- assimilate(build,ic,(above(X,Y):-on(X,Y))).

* Eliminate Redundancy ? (y/n) n.

* The Input_Knowledge "(above(_0,_1):-on(_0,_1))"  was assimilated.

yes
$ ?- assimilate(build,ic,(above(X,Y):-on(X,Z),on(Z,Y))).

* Eliminate Redundancy ? (y/n) n.

* The Input_Knowledge "(above(_0,_1):-on(_0,_2),on(_2,_1))"
 was assimilated.

yes
$ ?- retrieve(build,above(X,Y),[X,Y]).

[b,a]
[c,a]
[c,b]
[d,a]
[e,a]
[e,d]
[f,b]
[f,c]
[f,a]
[f,e]
[g,a]
[h,a]
[h,g]
[i,g]
[i,h]

yes
$ ?- accommodate(build,ic,above(_,_)).

* Initialize parameter(y/n)? n.

* The Knowledge_base "solutions" was created on Core_Space.

* The Knowledge_base "rejected" was created on Core_Space.

Next fact(sentence,true/false) or end ? above(i,a),true.
Checking fact(a)...
Error: missing solution above(i,a). diagnosing...

Query: on(i,a)? n.

Query: on(h,a)? n.
Error diagnosed: above(i,a) is uncovered.

Searching for a cover to above(i,a)...
Refining: (above(X,Y):-true)
Refining: (above(X,Y):-on(X,U))
Refining: (above(X,Y):-on(X,U),on(X,W))
Refining: (above(X,Y):-on(X,U),on(U,W))
Refining: (above(X,Y):-on(X,U),above(U,W))
Checking: (above(X,Y):-on(X,U),above(U,Y))
Found clause: (above(X,Y):-on(X,U),above(U,Y))
    After searching 20 clauses.
Listing of above(X,Y):
    (above(X,Y):-on(X,Y)).
    (above(X,Y):-on(X,U),on(U,Y)).
    (above(X,Y):-on(X,U),above(U,Y)).

Checking fact(a)...no error found.

Next fact(sentence,true/false) or end ? end.
```

```
* Purge Facts(y/n) ? y.

* The Knowledge_base "solutions" was dropped from Core_Space.

* Purge refuted theory(y/n) ? y.

* The Knowledge_base "refuted_hypothesis" was dropped from Core_Space.

* Purge refinement status(y/n) ? y.

* Eliminate Redundancy ? (y/n) y.

* The Knowledge "(above(_0,_1):-on(_0,_2),on(_2,_1))"
 was eliminated from Knowledge_Base "build" .

* Construction of the Knowledge "above"
 has been finished.

yes
$ ?- list(build,above(X,Y)).

Listing of above(X,Y):
    (above(X,Y):-on(X,Y)).
    (above(X,Y):-on(X,U),above(U,Y)).

yes
$ ?- retrieve(build,above(X,Y),[X,Y]).

[b,a]
[c,a]
[c,b]
[d,a]
[e,a]
[e,d]
[f,a]
[f,b]
[f,c]
[f,d]
[f,e]
[g,a]
[h,a]
[h,g]
[i,a]
[i,g]
[i,h]

yes
```

<< Example 3 >>

In order to inductively infer the "arch" concept, the first step executes to insert a meta-rule, that is an integrity constraint. The second step executes the "create" predicate to infer the "arch" concept. The final step deductively infers all the instances of "arch".

```
$ ?- insert_knowledge(ic,
$ ?- (inconsistent([insert],arch(_,_,_)):- arch(X,Y,Y)),_).

yes
$ ?- list_all(ic).

(inconsistent([delete],floor(X)):- \+meta_demo(build,floor(a))).
(inconsistent([delete,insert,update],on(X,Y)):-
     rectangular_block(Z),
     setof_kb(build,[U],(tower(Z,U),ne(U,[])),Y1),length(Y1,U1),U1<2).
(inconsistent([insert],arch(_0,_1,_2)):- arch(_3,_4,_4))

yes
$ ?- create(build,ic,arch(_,_,_)).

* Initialize parameter(y/n)? n.

* The Knowledge_base "solutions" was created on Core_Space.

* The Knowledge_base "rejected" was created on Core_Space.

Next fact(sentence,true/false) or end ? arch(f,[c,b,a],[e,d,a]),true.
Checking fact(a)...
Error: missing solution arch(f,[c,b,a],[e,d,a]). diagnosing...
Error diagnosed: arch(f,[c,b,a],[e,d,a]) is uncovered.

Searching for a cover to arch(f,[c,b,a],[e,d,a])...
Refining: (arch(X,Y,Z):-true)
Refining: (arch(X,[Y|Z],U):-true)
Refining: (arch(X,Y,[Z|U]):-true)
Refining: (arch(X,Y,Z):-tower(X,V))
Refining: (arch(X,[Y|Z],[U|V]):-true)
Refining: (arch(X,[Y,Z|U],V):-true)
Refining: (arch(X,[Y|Z],U):-tower(X,W))
Refining: (arch(X,[Y|Z],[U|V]):-true)
Refining: (arch(X,Y,[Z,U|V]):-true)
Refining: (arch(X,Y,[Z|U]):-tower(X,W))
Refining: (arch(X,Y,Z):-tower(X,V),tower(X,Y1))
Checking: (arch(X,Y,Z):-tower(X,V),tower(X,Z))
Found clause: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
    after searching 26 clauses.
Listing of arch(X,Y,Z):
    (arch(X,Y,Z):-tower(X,Y),tower(X,Z)).

* The Knowledge "(arch(_0,_1,_2):-tower(_0,_1),tower(_0,_2))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],arch(_0,_1,_2)):- arch(_3,_4,_4))".

Checking fact(a)...
Error: missing solution arch(f,[c,b,a],[e,d,a]). diagnosing...
Error diagnosed: arch(f,[c,b,a],[e,d,a]) is uncovered.

Searching for a cover to arch(f,[c,b,a],[e,d,a])...
Refining: (arch(X,Y,Z):-tower(X,V),tower(X,X1))
Checking: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Refuted: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Checking: (arch(X,Y,Z):-tower(X,Z),tower(X,Y))
Found clause: (arch(X,Y,Z):-tower(X,Z),tower(X,Y))
    after searching 4 clauses.
Listing of arch(X,Y,Z):
    (arch(X,Y,Z):-tower(X,Z),tower(X,Y)).

* The Knowledge "(arch(_0,_1,_2):-tower(_0,_2),tower(_0,_1))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],arch(_0,_1,_2)):- arch(_3,_4,_4))".

Checking fact(a)...
Error: missing solution arch(f,[c,b,a],[e,d,a]). diagnosing...
Error diagnosed: arch(f,[c,b,a],[e,d,a]) is uncovered.

Searching for a cover to arch(f,[c,b,a],[e,d,a])...
Refining: (arch(X,Y,Z):-tower(X,V),tower(X,X1))
Checking: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Refuted: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Checking: (arch(X,Y,Z):-tower(X,Z),tower(X,Y))
Refuted: (arch(X,Y,Z):-tower(X,Z),tower(X,Y))
Refining: (arch(X,[Y,Z|U],[V|W]):-true)
Refining: (arch(X,[Y|Z],[U,V|W]):-true)
Refining: (arch(X,[Y|Z],[U|V]):-tower(X,X1))
Refining: (arch(X,[Y,Z|U],[V|W]):-true)
Refining: (arch(X,[Y,Z,U|V],W):-true)
Refining: (arch(X,[Y,Z|U],V):-tower(X,X1))
Refining: (arch(X,[Y|Z],U):-tower(X,W),tower(X,Y1))
Refining: (arch(X,[Y|Z],[U,V|W]):-true)
Refining: (arch(X,[Y,Z|U],[V|W]):-true)
Refining: (arch(X,[Y|Z],[U|V]):-tower(X,X1))
Refining: (arch(X,[Y|Z],[U,V|W]):-true)
Refining: (arch(X,Y,[Z,U,V|W]):-true)
Refining: (arch(X,Y,[Z,U|V]):-tower(X,X1))
Refining: (arch(X,Y,[Z|U]):-tower(X,W),tower(X,Y1))
Refining: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Checking: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),rectangular_block(X))
Found clause: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),rectangular_block(X))
    after searching 37 clauses.
Listing of arch(X,Y,Z):
    (arch(X,Y,Z):-tower(X,Y),tower(X,Z),rectangular_block(X)).

* The Knowledge
 "(arch(_0,_1,_2):-tower(_0,_1),tower(_0,_2),rectangular_block(_0))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],arch(_0,_1,_2)):- arch(_3,_4,_4))".

Checking fact(a)...
Error: missing solution arch(f,[c,b,a],[e,d,a]). diagnosing...
Error diagnosed: arch(f,[c,b,a],[e,d,a]) is uncovered.

Searching for a cover to arch(f,[c,b,a],[e,d,a])...
Refining: (arch(X,Y,Z):-tower(X,Y),tower(X,Z))
Checking: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),rectangular_block(X))
Refuted: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),rectangular_block(X))
Checking: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),Y\==Z)
Found clause: (arch(X,Y,Z):-tower(X,Y),tower(X,Z),Y\==Z)
    after searching 4 clauses.
Listing of arch(X,Y,Z):
    (arch(X,Y,Z):-tower(X,Y),tower(X,Z),Y\==Z).

Checking fact(a)...no error found.

Next fact(sentence,true/false) or end ? end.

* Purge Facts(y/n) ? y.

* The Knowledge_base "solutions" was dropped from Core_Space.

* Purge refuted theory(y/n) ? y.

* The Knowledge_base "refuted_hypothesis" was dropped from Core_Space.

* Purge refinement status(y/n) ? y.

* Eliminate Redundancy ? (y/n) n.

* Construction of the Knowledge "arch"
 has been finished.

yes
```

```
$ ?- list(build,arch(_,_,_)).

Listing of arch(X,Y,Z):
    (arch(X,Y,Z):-tower(X,Y),tower(X,Z),Y\==Z).

yes
$ ?- retrieve(build,arch(X,Y,Z),[X,Y,Z]).

[f,[c,b,a],[e,d,g]]
[f,[e,d,a],[c,b,g]]

yes
```

<< Example 4 >>

In order to synthesize the DCG, the first step executes to insert a meta-rule, that is an integrity constraint. The second step synthesizes the DCG from 2 facts using the "create" predicate. The final step deductively infers all the instances of "s".

```
$ ?- list_all(dcg).

n([hermia|_0],_0)
n([lysander|_0],_0)
vi([walks|_0],_0)
vt([loves|_0],_0)

yes
$ ?- insert_knowledge(ic,
$ ?- (inconsistent([insert],s(_,_)):- s(X,[]),exist_variable(X)),_).

yes
$ ?- list_all(ic).

(inconsistent([delete],floor(X)):- \+meta_demo(build,floor(a))).
(inconsistent([delete,insert,update],on(X,Y)):-
   * rectangular_block(Z),
     setof_kb(build,[U],(tower(Z,U),member(U,[X|Y])),Y1),length(Y1,U1),U1<2).
(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2)).

yes
$ ?- create(dcg,ic,s(_,_)).

* Initialize parameter(y/n)? n.

* The Knowledge_base "solutions" was created on Core_Space.

* The Knowledge_base "rejected" was created on Core_Space.

Next fact(sentence,true/false) or end ? s([hermia,walks],[]),true.
Checking fact(s)...
Error: missing solution s([hermia,walks],[]). diagnosing...
Error diagnosed: s([hermia,walks],[]) is uncovered.

Searching for a cover to s([hermia,walks],[])...
Refining: (s(X,Y):-true)
Checking: (s(X,[]):-true)
Found clause: (s(X,[]):-true)
    after searching 4 clauses.
Listing of s(X,Y):
    s(X,[]).

* The Knowledge "(s(_0,[]):-true)"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2))".

Checking fact(s)...
Error: missing solution s([hermia,walks],[]). diagnosing...
Error diagnosed: s([hermia,walks],[]) is uncovered.

Searching for a cover to s([hermia,walks],[])...
Refining: (s(X,Y):-true)
Checking: (s(X,[]):-true)
Refuted: (s(X,[]):-true)
Refining: (s([X|Y],Z):-true)
Checking: (s([X|Y],[]):-true)
Refuted: (s([X|Y],[]):-true)
Refining: (s(X,[]):-true)
Checking: (s([X|Y],[]):-true)
Refuted: (s([X|Y],[]):-true)
Refining: (s([X,Y],Z):-n(X,U))
Refining: (s([X,Y|Z],U):-true)
Checking: (s([X,Y|Z],Z):-true)
Refuted: (s([X,Y|Z],Z):-true)
Checking: (s([X,Y|Z],[]):-true)
Refuted: (s([X,Y|Z],[]):-true)
Refining: (s([X|Y],Z):-vi(Y,V))
Checking: (s([X|Y],Z):-vi(Y,Z))
Found clause: (s([X|Y],Z):-vi(Y,Z))
    after searching 16 clauses.
Listing of s(X,Y):
    (s([X|Y],Z):-vi(Y,Z)).

* The Knowledge "(s([_0|_1],_2):-vi(_1,_2))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2))".

Checking fact(s)...
Error: missing solution s([hermia,walks],[]). diagnosing...
Error diagnosed: s([hermia,walks],[]) is uncovered.

Searching for a cover to s([hermia,walks],[])...
Refining: (s([X|Y],Z):-vi(Y,V))
Checking: (s([X|Y],Z):-vi(Y,Z))
Refuted: (s([X|Y],Z):-vi(Y,Z))
Refining: (s([X|Y],[]):-true)
Checking: (s([X|Y],[]):-true)
Refuted: (s([X|Y],[]):-true)
Refining: (s([X,Y|Z],[]):-true)
Checking: (s([X,Y|Z],[]):-true)
Refuted: (s([X,Y|Z],[]):-true)
Refining: (s([X,Y],Z):-n(X,U))
Refining: (s([X,Y|Z],U):-true)
Checking: (s([X,Y|Z],Z):-true)
Refuted: (s([X,Y|Z],Z):-true)
Checking: (s([X,Y|Z],[]):-true)
Refuted: (s([X,Y|Z],[]):-true)
Refining: (s([X|Y],Z):-vi(Y,V))
Checking: (s([X|Y],Z):-vi(Y,Z))
Found clause: (s([X|Y],Z):-vi(Y,Z))
    after searching 26 clauses.
Listing of s(X,Y):
    (s(X,Y):-n(X,U),vi(U,Y)).
    s([X,Y,Z|U],U).

* The Knowledge "(s([_0,_1,_2|_3],_3):-true)"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2))".

Checking fact(s)...
Error: missing solution s([hermia,loves,lysander],[]). diagnosing...
Error diagnosed: s([hermia,loves,lysander],[]) is uncovered.

Searching for a cover to s([hermia,loves,lysander],[])...
Refining: (s([X,Y,Z|U],V):-true)
Checking: (s([X,Y,Z|U],U):-true)
Refuted: (s([X,Y,Z|U],U):-true)
Checking: (s([X,Y,Z|U],[]):-true)
Refuted: (s([X,Y,Z|U],[]):-true)
Refining: (s([X,Y|Z],[]):-true)
Checking: (s([X,Y,Z|U],[]):-true)
Refuted: (s([X,Y,Z|U],[]):-true)
Refining: (s([X,Y|Z],U):-n(Z,W))
Checking: (s([X,Y|Z],U):-n(Z,U))
Found clause: (s([X,Y|Z],U):-n(Z,U))
    after searching 7 clauses.
Listing of s(X,Y):
    (s(X,Y):-n(X,U),vi(U,Y)).
    (s([X,Y|Z],U):-n(Z,U)).

* The Knowledge "(s([_0,_1|_2],_3):-n(_2,3))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2))".
```

```
Checking fact(s)...
Error: missing solution s([hermia,loves,lysander],[]). diagnosing...
Error diagnosed: s([hermia,loves,lysander],[]) is uncovered.

Searching for a cover to s([hermia,loves,lysander],[])...
Refining: (s([X,Y|Z],U):-n(Z,W))
Checking: (s([X,Y|Z],U):-n(Z,U))
Refuted: (s([X,Y|Z],U):-n(Z,U))
Refining: (s([X,Y|Z],[]):-true)
Checking: (s([X,Y,Z|U],[]):-true)
Refuted: (s([X,Y,Z|U],[]):-true)
Refining: (s([X|Y],Z):-vt(Y,V),n(V,Y1))
Checking: (s([X|Y],Z):-vt(Y,V),n(V,Z))
Found clause: (s([X|Y],Z):-vt(Y,V),n(V,Z))
    after searching 7 clauses.

Listing of s(X,Y):
    (s(X,Y):-n(X,U),vi(U,Y)).
    (s([X|Y],Z):-vt(Y,V),n(V,Z)).

* The Knowledge "(s([_0|_1],_2):-vt(_1,_3),n(_3,_2))"
 is inconsistent with the Integrity_Constraints
 "(inconsistent([insert],s(_0,_1)):- s(_2,[]),exist_variable(_2))".

Checking fact(s)...
Error: missing solution s([hermia,loves,lysander],[]). diagnosing...
Error diagnosed: s([hermia,loves,lysander],[]) is uncovered.

Searching for a cover to s([hermia,loves,lysander],[])...
Refining: (s([X|Y],Z):-vt(Y,V),n(V,X1))
Checking: (s([X|Y],Z):-vt(Y,V),n(V,Z))
Refuted: (s([X|Y],Z):-vt(Y,V),n(V,Z))
Refining: (s([X|Y],Z):-vt(Y,V),vt(Y,X1))
Refining: (s([X,Y|Z],[]):-true)
Checking: (s([X,Y,Z|U],[]):-true)
Refuted: (s([X,Y,Z|U],[]):-true)
Refining: (s(X,Y):-n(X,U),n(X,W),n(X,Y1))
Refining: (s(X,Y):-n(X,U),n(X,W),vt(U,Y1))
Refining: (s(X,Y):-n(X,U),n(X,W),vt(W,Y1))
Refining: (s(X,Y):-n(X,U),vt(U,W),n(X,Y1))
Refining: (s(X,Y):-n(X,U),vt(U,W),n(W,Y1))
Checking: (s(X,Y):-n(X,U),vt(U,W),n(W,Y))
Found clause: (s(X,Y):-n(X,U),vt(U,W),n(W,Y))
    after searching 28 clauses.
Listing of s(X,Y):
    (s(X,Y):-n(X,U),vi(U,Y)).
    (s(X,Y):-n(X,U),vt(U,W),n(W,Y)).

Checking fact(s)...no error found.

Next fact(sentence,true/false) or end ? end.

* Purge Facts(y/n) ? y.

* The Knowledge_base "solutions" was dropped from Core_Space.

* Purge refuted theory(y/n) ? y.

* The Knowledge_base "refuted_hypothesis" was dropped from Core_Space.

* Purge refinement status(y/n) ? y.

* Eliminate Redundancy ? (y/n) n.

* Construction of the Knowledge "s"
 has been finished.

yes
$ ?- list(dog,s(_,_)).

Listing of s(X,Y):
    (s(X,Y):-n(X,U),vi(U,Y)).
    (s(X,Y):-n(X,U),vt(U,W),n(W,Y)).

yes
$ ?- retrieve(dog,s(X,[]),[X]).

[[hermia,loves,hermia]]
[[hermia,loves,lysander]]
[[hermia,walks]]
[[lysander,loves,hermia]]
[[lysander,loves,lysander]]
[[lysander,walks]]

yes
```