TM-0496

A Fast Prolog-based Inference Engine KORE/IE

by
T. Shintani

May, 1988

A Fast Prolog-based Inference Engine KORE/IE

Toramatsu Shintani
International Institute for Advanced Study of Social Information Science (IIAS-SIS)
FUJITSU LIMITED, 140 Miyamoto, Numazu-shi, Shizuoka 410-03, Japan
e-mail: tora@iias.fujitsu.junet

ABSTRACT

In this paper, we propose a fast Prolog-based inference engine KORE/IE (Knowledge Oriented Reasoning Environment / Inference Engine) and discuss some features of the system. The main features provided by KORE/IE are (1)a mechanism of an inference engine on KORE, (2)cooperative problem solving among rule bases, and (3)a speedy inference mechanism. On Prolog, it is important to speed up inference engines without sacrificing flexible power for rule expressions. It is essential for constructing large scale application programs on the systems. In order to speed up the inference mechanism and realize the flexible rule expressions, we take advantage of a speedy refutation mechanism, partial evaluation techniques and fast searching for heads of clauses. At present, KORE/IE is implemented on C-Prolog and Quintus Prolog, and its efficiency of inferences is comparable with that of OPS5 on Franz Lisp.

1. Introduction

KORE/IE (Knowledge Oriented Reasoning Environment / Inference Engine) can function not only as the inference engine module of KORE(Shintani 1986), but also as an independent production system. The basic functions of KORE/IE are based on those of OPS5(Forgy 1981), and the system provides a rule-oriented programming environment (i.e. a tracer for rules, a stepper for inferences, and so on) in a similar manner as OPS5. Furthermore, its functions are extended by using the advantages of Prolog. As the system adopts a mechanism of a pure production system, the rules can be represented efficiently by considering the inference mechanism. Inference is attained by performing a sequence of operations called the recognize-act cycle:

(1)Matching process: Determine a CS(conflict set) which includes rules
  whose LHSs (left-hand sides) have matched the current contents of working memory.
(2)Conflict resolution process: Select one rule from CS; if CS is empty,
  halt the recognize-act cycle.
(3)Act process: Perform the actions in the RHS (a right-hand side) of the
  selected rule.
(4)Go to (1).

The components of a pure production system consist of WM(working memory), PM(production memory), and PSI(production system interpreter). In KORE/IE, PSI's functions are provided by Prolog system itself and the remaining components are realized by Prolog programs as shown in Fig.1. This

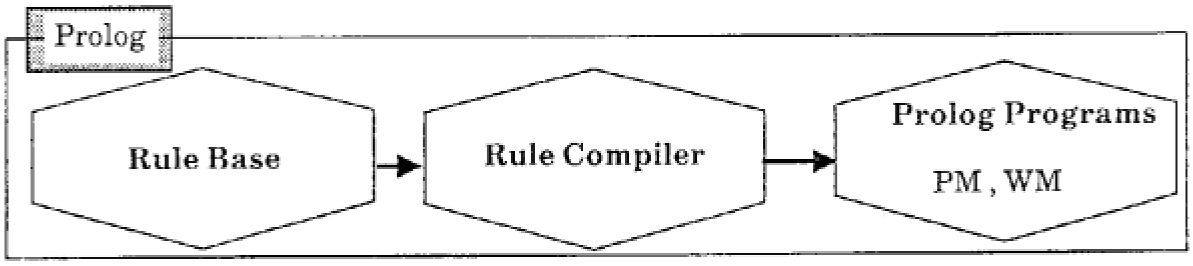


Fig.1. Components of the KORE/IE .

architecture of the KORE/IE is the most important feature in speeding up the system because recognize-act cycles can be made faster by utilizing the fast refutation mechanism which is a basic computation mechanism for Prolog systems.

A Prolog system provides powerful pattern matching (that is, unification) and a flexible backtracking mechanism for constructing inference engines. Furthermore, the system itself can be utilized as a powerful inference engine based on a refutation mechanism(Kowalski 79).

Generally, in order to realize an inference system, we must construct an interpreter for executing inference rules. However, we can realize such a system simply by using a Prolog system as an inference engine in which the non-unit Prolog clauses are considered to be rules of the system and the rules are executed directly by the Prolog system. These clauses can be easily obtained by transforming the rules into the Prolog program. In logic programming, as a very straightforward method, this method is usually adopted to realize inference systems (e.g. BUP (Matsumoto 1983)).

There is an alternative method based on partial evaluation (Futamura 1983) of Prolog programs. In this method, an inference engine is partially evaluated with respect to inference rules and the result generates a specialized Prolog program for the rules. Then the inferences are realized by executing the program. This method is adopted in order to build an interpreter for the rules. This program can execute rules efficiently (Takeuchi 1985). The main advantage is easy maintenance of the rules. Its efficiency is due to the fast refutation mechanism of Prolog systems.

The above two methods for constructing inference systems do not actively exploit the internal mechanisms (e.g. recognize-act cycles, making and modifying WM, conflict resolution) of the system. We can further speed up the systems by making the internal mechanisms faster.

In this paper, we propose a new method for constructing a speedy inference system based on the techniques of the existing two methods mentioned above. In order to speed up the inference and realize the flexible rule expressions, we take advantage of features of Prolog systems, such as a speedy refutation mechanism, partial evaluation techniques and speedy searching for heads of clauses. The two existing methods directly transform rules into Prolog programs (that is, non-unit clauses), however the main feature of our approach corresponds to speeding up internal mechanism of the inference. At present, using the new method, KORE/IE is implemented on C-Prolog and Quintus Prolog, and its speed of the inference is comparable with that of OPS5 on Franz Lisp.

This paper consists of five sections. In Section 2 we sketch functions provided by KORE/IE. In Section 3, the main part of this paper, a new method for building an inference system is presented. In Section 4, experimental results of its performance are described. In Section 5, some concluding remarks are presented.

2. KORE/IE

2.1. The rule

KORE/IE provides functions for rule-oriented problem solving and knowledge representation in KORE. Independently it functions as a production system like OPS5. KORE/IE realizes a flexible and readable rule description by adopting the syntax of a term description in Prolog. A rule consists of (1)the name of the rule, (2)the symbol ":", (3)the symbol "if", (4)the LHS of the rule, (5)the symbol "then", (6)the RHS of the rule, and (7)the symbol ".". The rule is expressed as follows;

Rule_name: if *Conditions*$_1$ & *Conditions*$_2$ ...
then *Actions*$_1$ & *Actions*$_2$ ... .

where the Rule__name is the name of the rule. ConditionN is a condition element of the rule. It is called an LHS pattern. ActionN is a rule action, which is used to change WM, to execute Prolog programs, and so on. It is called an RHS action. The symbol "&" is a delimiter. For example, a rule can be expressed as follows;

```
on__floor:
      if goal(status=active, type=(on;move), object__name=X) &
         monkey(on \== floor)
      then
```

```
modify(2, on=floor) &
modify(1, status=satisfied).
```

As shown in the example, the LHS is composed of LHS patterns which correspond to compound terms of Prolog. The patterns are called declarative information in KORE. The patterns are described by a unified description for representing declarative information in KORE subsystems. For example, a WM is described by a set of declarative information, and it corresponds to a relational database which is managed by KORE/DB, a subsystem of KORE which provides functions for managing databases.

The LHS pattern consists of (1)a class name (which corresponds to a functor name of a term in Prolog) and (2)its arguments. The arguments are a sequence of one or more slot-value pairs. The pairs correspond to terms which have functors of arity 2 (e.g. =, ==, \==, =<, >=). The arguments are called slot descriptions. The slot descriptions are used in the same manner as the basic predicates for comparison of terms in Prolog. For example, "on \== floor" in the above example tests if the value of the slot "on" obtained from the current contents of WM and the value "floor" are not literally identical. In Prolog programs, the goals in the body of a clause are linked by the operator "," and the operator ";" which can be interpreted as conjunction ("and") and disjunction ("or") respectively. In the same manner as Prolog programs, we can also use a conjunctive and a disjunctive slot description in the slot description as follows:

```
Conjunctive slot description:
        Slot = (Restriction1, Restriction2, ..., RestrictionN)
Disjunctive slot description:
        Slot = (Value1; Value2; ...; ValueN)
```

The right side of a conjunctive slot description consists of some restrictions (or Prolog goals) which are used to indicate that the slot value in a WM element must satisfy the restrictions simultaneously. The right side of a disjunctive slot description consists of some values (or Prolog goals) which are used to specify that any of the contained values is acceptable as a match. Thus, "type=(on;move)" in the above example will match either "on" or "move".

In a rule, we can use a functional notation to enhance the readability of the rule. For example, the following predicate definition returns the result to the first argument "Result":

```
add_one(Result, Number) :- Result is Number + 1.
```

This definition can be used as a function in a slot description as follows;

```
slot = add_one(X).
```

The functional notation can be executed as a normal Prolog program since a rule containing functional notations is compiled into Prolog programs.

2.2. Mechanisms for cooperative problem solving

A cooperative problem solving can be applicable among rule bases in KORE/IE. It appears that the cooperation among KORE/IE rules corresponds to cooperative problem solving in the blackboard model(Lesser 1977). However, cooperative problem solving in KORE/IE can be realized efficiently without building a meta control mechanism, since a basic Prolog computation mechanism (that is, the refutation mechanism) can be used. The internal mechanism for cooperative problem solving on rule bases in KORE/IE can be considered as generating optimum Prolog programs from rules in the rule bases and executing the programs in Prolog. In order to generate the programs, a rule compiler is applied. The rule compiler translates a rule into a uniform Prolog program. The program consists of an LHS program for the LHS and an RHS program for the RHS, and is realized as flat non-unit clauses of Prolog as shown in Fig.2.

A change in WM is transformed to a question for the LHS program. Then, applying the question by using the Prolog refutation mechanism corresponds to executing the matching process of the recognize-act cycle and generates instantiations as a result of the question (for further details, see Section 3). An instantiation is an object in a conflict set and an ordered pair of a rule name and a list of WM elements

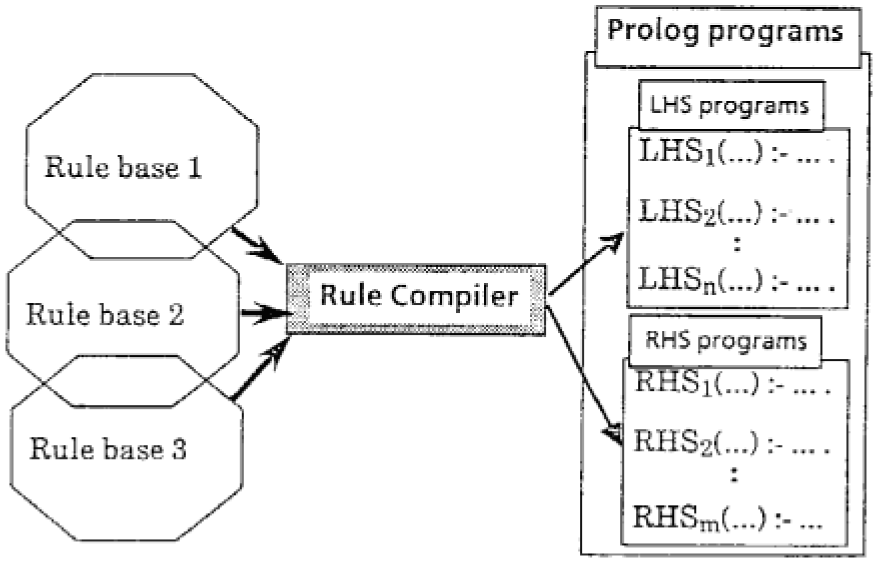


Fig.2. Rule compilation.

satisfying the rule's LHS. The instantiations are generated for every rule base and conflict sets are also generated for every rule base. A unique conflict resolution strategy for each rule base is applied to conflict sets for each rule base. After the conflict resolution, one instantiation for each rule base is selected for executing the RHS programs for each rule base. Execution of the RHS programs is achieved by applying questions for RHS for each rule base. The question is generated by transforming the selected instantiation. Applying the question in Prolog corresponds to performing an act process of the recognize-act cycle. An order for invoking rule bases is determined automatically according to the order of their compilation. Otherwise, the order can be determined explicitly by showing the order in the rule bases.

3. Speeding up inference engines

Inferences in KORE/IE are realized by performing recognize-act cycles. In order to speed up the inferences, we need to speed up a sequence of operations in the recognize-act cycle, which are matching, conflict resolution, and act processes. Generally, the matching process consumes more time than the other processes, and influences the efficiency of the inferences. In a straightforward pattern matcher all the WM elements are compared against all the LHS patterns for every rule on each cycle. In order to avoid iteration over the elements on each cycle and to realize an efficient matching process, the Rete Match algorithm (Forgy 1982) is used in OPS5. The algorithm was developed to eliminate extra work in the unoptimized pattern matcher. In the algorithm, LHS's are compiled into a tree-structured sorting network by linking nodes together which test the slot values, and a matching process is realized by passing tokens into the network in which information of previous matching is stored. The token is a description of WM changes.

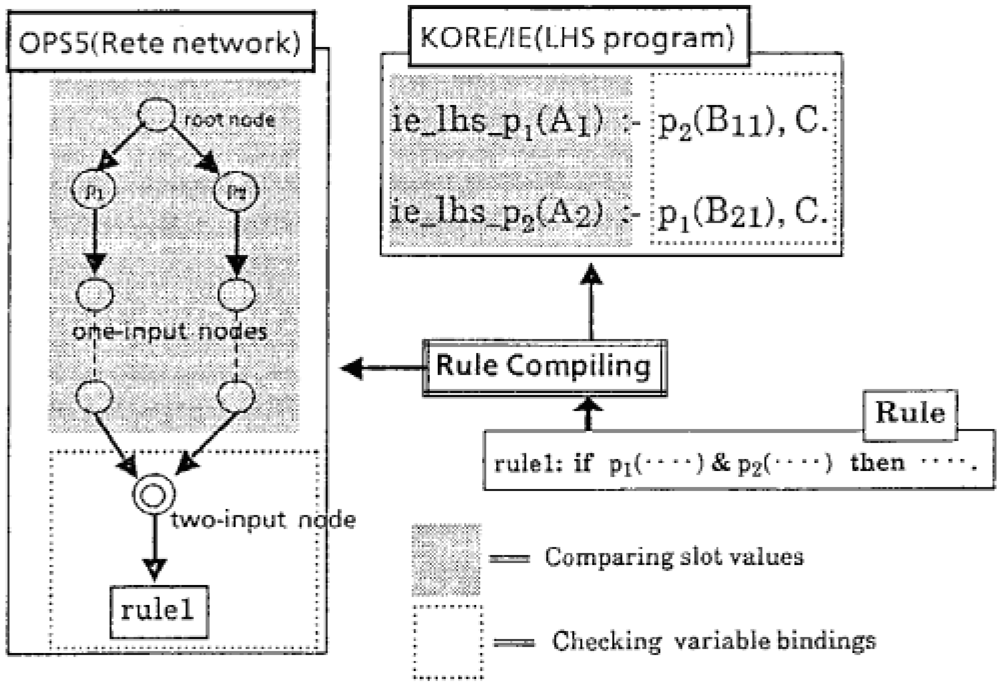
However, in logic programming, it is difficult to implement the kind of network structure efficiently. It requires appropriate data structures (e.g. pointers) to construct the network. Therefore, in order to speed up an inference engine, we make use of an efficient refutation mechanism in Prolog to match the techniques of the Rete Match algorithm which avoids iterating on matching processes.

3.1. The LHS program

An LHS program is a specialized program for matching process and is generated by transforming rules into Prolog programs which include information about variable bindings in the LHS. The programs are used for computing a conflict set. In the transformation, we use a technique for partial evaluation of Prolog programs. We call the transformation LHS compilation of rules. The technique is useful for generating the specialized program from rule descriptions and declarative data for defining skeletons of LHS patterns. The data can be asserted by using "literalize" command in KORE/IE. In the compilation, an LHS is transformed into Prolog programs by processing information about variable bindings. In a strict sense, it appears that the technique is not the partial evaluation (Futamura 1983), but it essentially utilizes the partial evaluation of Prolog programs which can evaluate parts of a program without some special evaluation scheme such as lazy evaluation. However, in a broad sense, it is a kind of partial evaluation since the LHS program is generated as a matching program from the data and the LHS pattern in which the unification mechanism of the Prolog system itself is considered a program required by the partial evaluation. In this approach the Prolog system corresponds to the PSI (production system interpreter). Thus, in order to realize fast-inferences effectively, we need to utilize the advantages of Prolog systems and optimize the Prolog programs generated by the compilation. We utilize the following functions of Prolog systems: (1)fast head searching; clauses are hash-indexed according to functor name and its arity in the head of the clauses and (2)powerful backtracking for the search.

Function (2) can be used as a control mechanism for finding several instantiations at the time when the WM is changed. Using this function, the system can check all the LHS program according to changes of WM and then generate instantiations without building a particular control mechanism.

Function (1) is particularly important in speeding up inferences and can provide functions for realizing a root node of the network in the Rete Match algorithm. The root node is an entrance to the network and receives only the changes of WM and passes them to its successors. The node is used for utilizing only changes of WM in the matching process. In LHS programs, the functions of the root node can be realized easily by naming the functor name in the head of the clause according to the class name of the LHS pattern, because the clause is hash-indexed on the name. Each clause of the LHS program corresponds to a path from a root node to a terminal node (that is, production node) in the network because an LHS pattern is realized by using a Prolog clause. In the clause, functions of one-input nodes (which are used for comparing slot values) and two-input nodes (which are used for checking variable bindings) in the network are realized simply by using the refutation mechanism of the Prolog system as shown in Fig.3-1.



OPS5(Rete network)
root node
p_1
p_2
one-input nodes
two-input node
rule1
KORE/IE(LHS program)
$ie\_lhs\_p_1(A_1)$:- $p_2(B_{11})$, C.
$ie\_lhs\_p_2(A_2)$:- $p_1(B_{21})$, C.
Rule Compiling
Rule
rule1: if $p_1(\cdots)$ & $p_2(\cdots)$ then ⋯.
▨ = Comparing slot values
⬚ = Checking variable bindings

Fig.3-1. LHS program and Rete network.

Fig.3-2 is given to clarify the description in Fig.3-1. In the LHS program, for example, the skeleton of the LHS programs is generated by the LHS compilation. The functor "ie_lhs_class1" is named according to the class name "class1" of the LHS pattern. A_i represents a sequence of arguments in the head; A_i consists

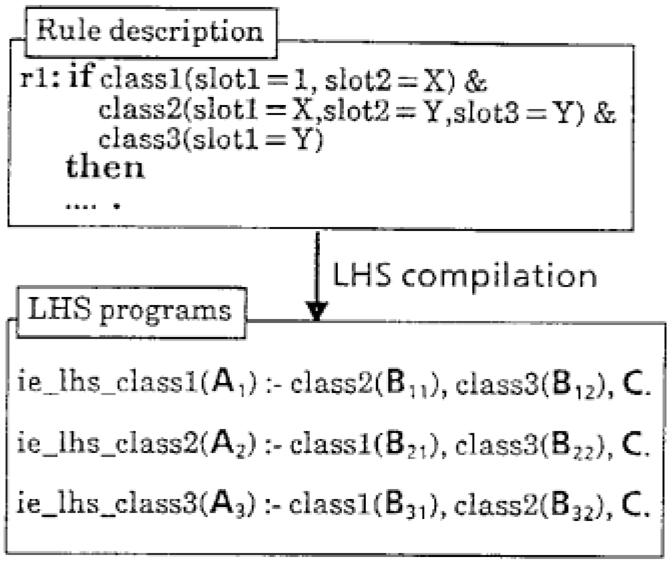

Rule description
r1: if class1(slot1 = 1, slot2 = X) &
class2(slot1 = X,slot2 = Y,slot3 = Y) &
class3(slot1 = Y)
then
.... .
LHS compilation
LHS programs
ie_lhs_class1(A₁) :- class2(B₁₁), class3(B₁₂), C.
ie_lhs_class2(A₂) :- class1(B₂₁), class3(B₂₂), C.
ie_lhs_class3(A₃) :- class1(B₃₁), class2(B₃₂), C.

Fig.3-2. The LHS program.

of a rule name, slot values, information of variable bindings , and so on. The bodies of the clauses in the LHS programs represent the other LHS patterns requested for satisfying the rule. The B_{ij} represents slot values of the other LHS patterns. The C represents constraints for variable bindings between A_i and B_{ij}. Details will be described in Section 3.2. By using the LHS program, We can perform a matching process according to changes of WM elements without iterations over the WM elements. A change of a WM element is transformed into a question for executing the LHS program as mentioned in Section 2.2. In Fig.3-2, for example, if a new WM element, whose class name is "class1", is built and added to WM, the first clause in the LHS programs is called and executed by performing the following question;

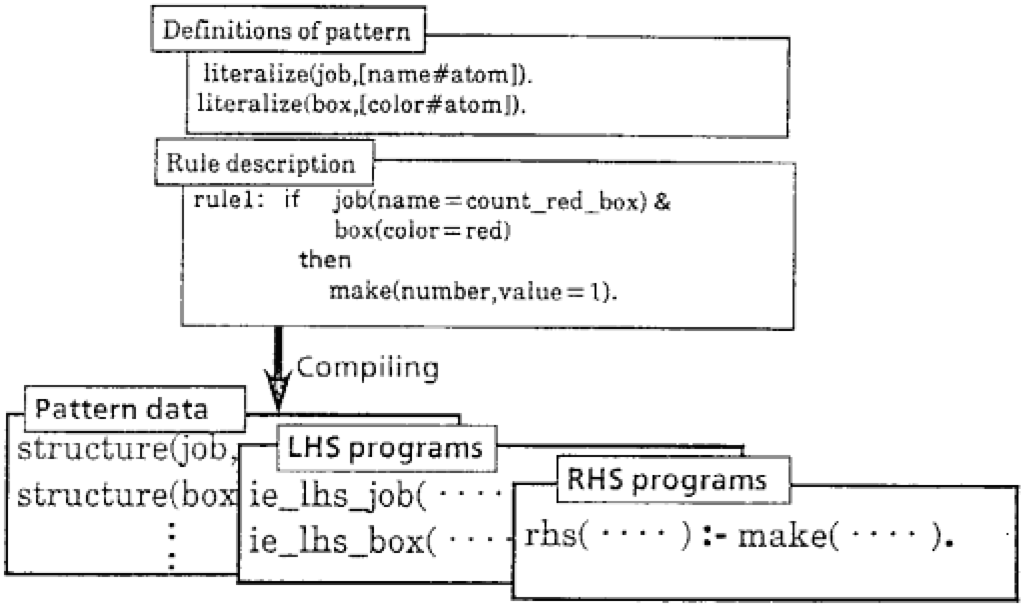
?- ie_lhs_class1(A_1).

where A_1 can be determined by the new WM element.

A feature of the LHS program is that the matching process is run by the speedy refutation mechanism in Prolog systems, which is started by the question generated by the change of WM. The process in the Rete Match algorithm is performed by passing a token into the network. The token is an ordered pair of a tag (that is, + and - which indicate adding to WM and deleting from WM respectively) and a WM element. In the LHS programs, the matching process is realized without building a particular mechanism by directly using the refutation mechanism. The refutation mechanism is a basic computation mechanism in Prolog systems, hence we can utilize the efficiency of head searching in Prolog systems. In the Rete Match algorithm, when rules are compiled into the network, the order of nodes has direct effects upon efficiency of the matching process. The LHS program is not affected by the order since the efficiency of unification between terms is not quite affected by the order of the arguments. The network reduces memory use by using the structural similarity of rules. In the LHS program, memory use is not efficient since the clauses are generated for every LHS pattern. Improving the efficiency of memory use is a subject for a future study.

3.2. Rule compilation

In KORE/IE we speed up inferences by compiling (or transforming) the rules into Prolog programs. The compilation is realized by the rule compiler of KORE/IE. In the compilation, in order to speed up each step of a recognize-act cycle, the programs are generated according to LHS and RHS of rules as shown in Fig.4. We call the compilation for the LHS LHS compilation, and the compilation for the RHS RHS compilation. LHS compilation generates Prolog programs for speeding up the matching process as mentioned in Section 3.1. RHS compilation generates Prolog programs for speeding up the action process.



```
literalize(job,[name#atom]).
literalize(box,[color#atom]).
```

```
rule1:  if    job(name=count_red_box) &
              box(color=red)
           then
              make(number,value=1).
```

```
structure(job,
structure(box
    :
```

```
ie_lhs_job( · · · ·
ie_lhs_box( · · ·
```

```
rhs( · · · · ) :- make( · · · · ).
```

Fig.4. Generating Prolog programs by compiling a rule.

In Fig.4 the patterns define class names, slot names and types of the slot values in the LHS patterns. This definition is realized by using the "literalize" command in KORE/IE and is used for standardizing LHS patterns by the compiler. By standardization of LHS patterns, positions of slots in LHS patterns are fixed, and this contributes to speeding up processing of LHS patterns. For example, the rule descriptions in Fig.5 is compiled into (1)the pattern data in Fig.6 defined by the literalize commands, (2)the LHS programs in Fig.7 and (3)the RHS programs in Fig.8.

```
literalize(monkey, [at, on, holds]).
literalize(goal, [status, type, object__name, on, to]).
'At__Monkey': if
        goal + goal(status=active, type=at, object__name=nothing, to=P1) &
        monkey + monkey(on=floor, at\== P1, holds=nothing)
    then
        nl & write('Walk to ') & write(P1) & nl & nl &
        modify(monkey, at=P1) &
        modify(goal, status=satisfied).
```

Fig.5. An example of a rule description.

In Fig.6, the fact "structure" is used to keep information about standardized LHS patterns where the first and second arguments represent the class name and the database name respectively. The database name is used by KORE/DB. The third, fourth and fifth arguments represent the number of slots, the list of slot names ,and the list of types for slot values, respectively. The positions in the list which is the fourth argument correspond to the positions in the list which is the fifth argument. In KORE/IE, the class "start" is defined by a default class for convenience. The fact "position" keeps the positions of slots of LHS patterns which are defined by the literalize commands. The fact "lhs__class" is used to keep the functor names in the heads of LHS programs. The pattern data are used to standardize patterns in KORE/IE and enable to speed up referring to slot values.

As shown in Fig.7, the compiler generates LHS programs. The number of the programs (or clauses) corresponds to the number of LHS patterns. Then, by using the programs, we can perform the matching process according to changes of WM as mentioned in Section 3.1. The programs can also check the variable bindings in the process. For example, let us see the LHS program, which is the first clause in Fig.7, according to the LHS pattern named "monkey", which is the second pattern in Fig.5. The LHS pattern can

```
structure(goal,goal,6,[time_tag,status,type,object_name,on,to],[number,non,non,non,non,non]).
structure(monkey,monkey,4,[time_tag,at,on,holds],[number,non,non,non]).
structure(start,start,2,[time_tag,order],[number,number]).

position(start,time_tag,number,1).
position(start,order,non,2).
position(monkey,time_tag,,number,1).
position(monkey,at,non,2).
position(monkey,on,non,3).
position(monkey,holds,non,4).
position(goal,time_tag,,number,1).
position(goal,status,non,2).
position(goal,type,non,3).
position(goal,object_name,non,4).
position(goal,on,non,5).
position(goal,to,non,6).

lhs__class(start,ie_lhs_start).
lhs__class(monkey,ie_lhs_monkey).
lhs__class(goal,ie_lhs_goal).
```

Fig.6. An example of pattern data.

```
ie_lhs_monkey('At_Monkey',r1,[A,B,B],[C,D],1,D,A,floor,nothing) :-
                goal(C,active,at,nothing,E,B),
                A \== B.

ie_lhs_goal('At_Monkey',r1,[A,B,B],[C,D],1,C,active,at,nothing,E,B) :-
                monkey(D,A,floor,nothing),
                A \== B.
```

Fig.7. An example of an LHS program.

be named by using the operator "+". The argument in the head of the clause can be shown as in Fig.7-1.

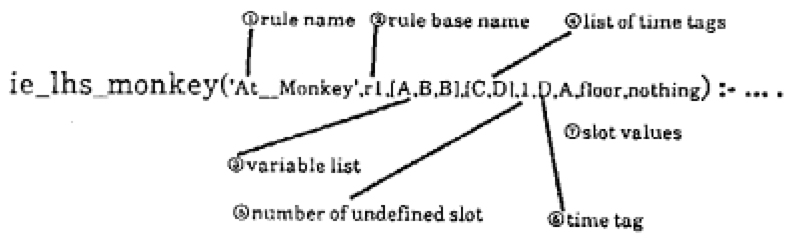

①rule name
②rule base name
④list of time tags
ie_lhs_monkey('At__Monkey',r1,[A,B,B],[C,D],1,D,A,floor,nothing) :- ... .
⑦slot values
③variable list
⑤number of undefined slot
⑥time tag

Fig.7-1. The head of the LHS program.

Argument (3) in Fig.7-1 is used to represent a variable list which keeps all the variables occurred in the LHS patterns in the rule. In the example, the variable "B" corresponds to the variable "P1" in the LHS of the rule. The variable "A" is a dummy variable used in the slot description "at\== *P1*" of the second LHS pattern. The internal representation of the slot description is "at = (*X* \== *P1*)" where the dummy variable "*X*" corresponds to the variable "A". Argument (4) is a list of time tags which indicate satisfied WM elements in the matching process. Argument (5) is the number of undefined slots which are unused (or unfilled) slots in the LHS pattern in spite of definition in the literalize command. The number is used as information for conflict resolutions. Argument (6) is a time tag which indicates a WM element matched the LHS pattern. The rest of the arguments including Argument (7) is a sequence of slot values where the values are arranged in the order defined by the literalize command. In Fig. 7-1, as the slot "at" of the class "monkey" is undefined, a dummy variable "*A*" is placed at the position for the slot. Checking for variable bindings between LHS patterns is realized in the body of the program. In the example, it is the last goal "A \== B".

In Fig.8, the lists "[A,B,B]" and "[C,D]" in the head represent a list of variables occurred in the LHS program and a list of time tags respectively. The other variables "E","F","G" are used as flags for cooperative problem solving with the other KORE subsystems.

```
rhs('At_Monkey',r1,[A,B,B],[C,D],E,F,G) :-
            nl,
            write('Walk to '),
            write(B),
            nl,
            nl,
            modify(E,F,D,[at=B],G),
            modify(E,F,C,[status=satisfied],G),
            !.
```

Fig.8. An example of an RHS program.

The LHS programs are called by actions (that is, make, modify, and remove) which change WM. Let us take an example. The make command is defined as in Fig.9, which is used for building a new WM element and adding it to the WM. For example, the command is used as follows;

make(goal(status = active, type = at)).

As shown in Fig.9, in a process of the make command, to begin with, a time tag is given in part (1) and the input pattern is standardized in part (2). Then, in part (3) a new WM element is added to WM. In part (4), a question is generated by using the information from (1),(2), and (3). By executing the question instantiations can be composed in part (5).

```
make(Fact) :-
  (1)  Fact =.. [F|Atrs],
       Time_tag is cputime,
  (2)  structure(F,_,_,[_|ANL],[_|ATL]),
       reforming_make(F,Atrs,ANL,ATL,Reformed),
  (3)  FACT =.. [F,Time_Tag|Reformed],
       assert(FACT),
  (4)  lhs_class(F,FF),
       Call =.. [FF,Rule_Name,Rule_Base,VL,Instantiation,
                                   NUS,Time_Tag|Reformed],
  (5)  (call(CALL),
        strategy_rec(Rule_Base,_,Strategy),
        ass(Rule_Base,Rule_Name,Instantiation,NUS,Strategy,VL),
        fail ;
        true),
       !.
```

Fig.9. A Prolog definition of the make command.

The RHS program is also executed, by generating a question like the LHS program, as shown Fig.10. Fig.10 shows a definition of an inference stepper in KORE/IE. The RHS program is called in the definition. To begin with, in part (1) an instantiation is selected as a result of the conflict resolution. In part (2), a question is generated by using the information (1). By executing the question, the action process is realized.

4. Experimental results

We conducted a few experiments to see how fast KORE/IE runs. It is generally considered that the speed for executing programs in Prolog systems is considerably slower than the speed in Lisp systems. To

```
running(N,off,Rule_base) :-
        now_running(Rule_Base,running),
        strategy_rec(Rule_Base,_,Strategy),
        retract(cs(Rule_Base,[[Rule_name1,Time_Tags1,STT1,NUS1,Var_List1]|CS])),
    (1) conflict_resolution(Strategy,CS,
                                    [Rule_Name1,Time_Tags1,STT1,NUS1,Var_List1],
                                    [Rule_Name,Time_Tags,_,_,Var_List],New_CS),
        asserta(cs(Rule_Base,New_CS)),
        ie_to_eden(Rule_Name,Rule_Base,EDEN_Mode),
    (2) rhs(Rule_Name,Rule_Base,Var_List,Time_Tags,off,EDEN_Mode,Time_Tags),
        NN is N - 1,
        !,
        running(NN,off,Rule_Base).
```

Fig.10. A Prolog definition of the inference stepper.

begin with, KORE/IE on C-prolog is compared with OPS5 on Franz Lisp. It is well known that OPS5 is the fastest system among production systems on interpretive languages. Likewise it is well known that C-prolog is one of the slowest Prolog systems. Table 1 shows the comparison of the OPS5 on Franz Lisp (Opus 38.79) with KORE/IE on C-Prolog (version 1.4). For this test, we used a standard bench mark system which is also used in (Brekke 1986). The system is "Monkey and Bananas" which consists of 27 rules (Brownston 1985). In Table 1, the OPS5(compiled) is a compiled OPS5 system by using the Lisp compiler. The OPS5(interpreted) is a system which is loaded simply into Lisp system. The method of using OPS5(interpreted) corresponds to the method of KORE/IE on C-Prolog. As is evident in Table 1, KORE/IE is sufficiently faster than OPS5(interpreted) and the speed of KORE/IE is comparable with that of OPS5(compiled). It should be noted that the speed of KORE/IE is improved effectively if we use a more efficient Prolog system.

VAX11/780
OPS5 : Franz Lisp (Opus 38.79)
KORE/IE : C-prolog (version 1.4)

Table 1.

	OPS5 (Compiled)	OPS5 (interpreted)	KORE/IE
Rule Execution time(Sec)	1.5	61.5	7.0

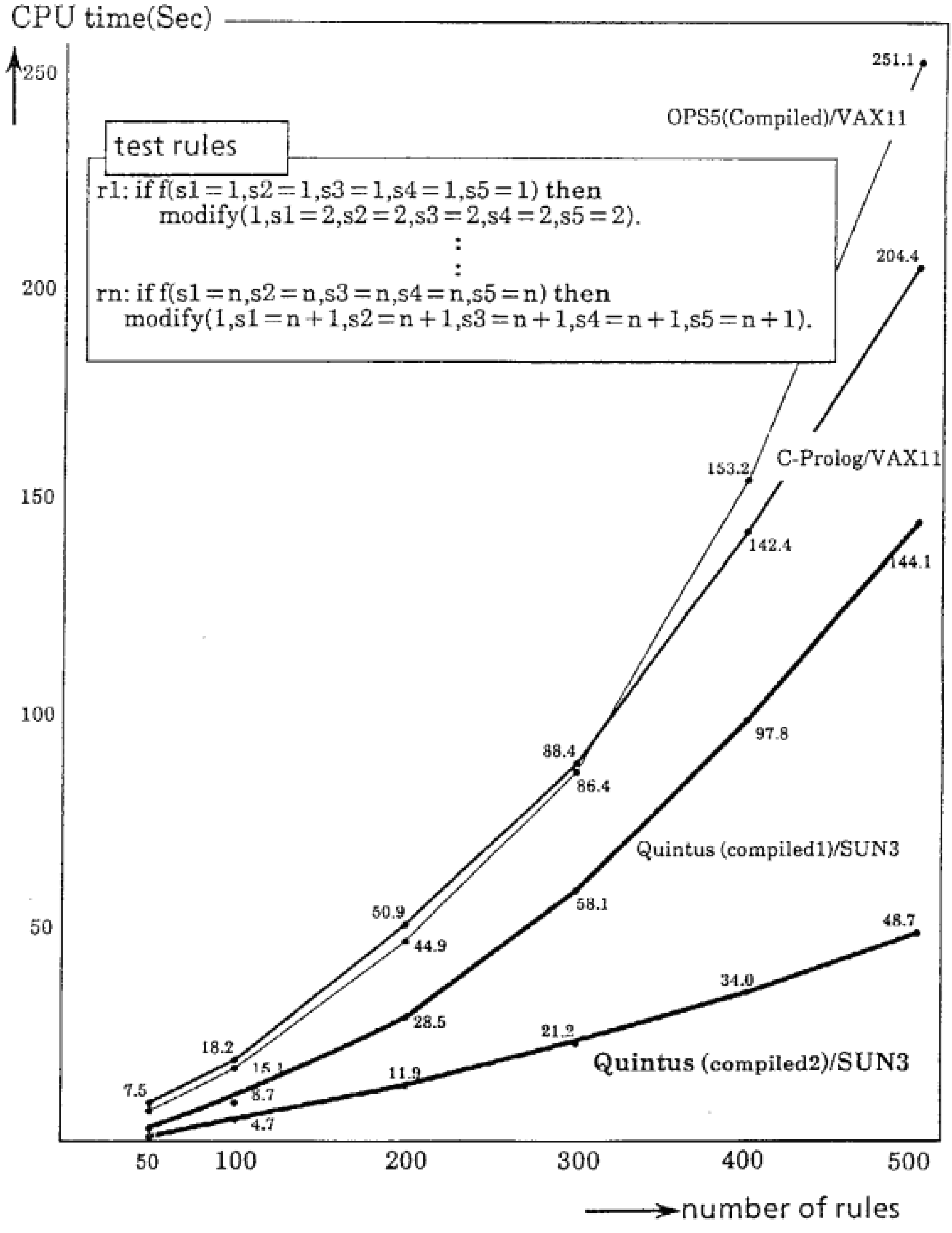
Table 2 shows the performances of KORE/IE on C-prolog and Quintus Prolog where we used the same example as for Table 1. The Table 2, the Quintus(compiled1) is where the KORE/IE system is compiled using a Prolog compiler. The Quintus(compiled2) is a system where LHS programs, RHS programs and pattern data generated by the compiler are also compiled using a Prolog compiler. What is evident from the table is that the performance of KORE/IE is improved effectively by using the Prolog compiler. However, since the mechanism for inputs and outputs in Prolog systems is inefficient, it seems that the improvement reaches a limit.

SUN3/52m
KORE/IE : C-prolog (version 1.4)
Quintus Prolog (Release 1.6)

Table 2.

	Quintus (Compiled2)	Quintus (Compiled1)	C-Prolog
Rule Execution time(Sec)	2.0	2.5	4.1

Now, we test with an example without using inputs and outputs. As shown in Fig.11, the example is rules that modifies WM successively according to the number of rules. In the graph, the vertical axis indicates CPU time (seconds) required for each system, and the horizontal axis indicates the number of rules. The graph shows performances of Quintus(compiled2) and Quintus(compiled1) on SUN3/52m, and the performances of KORE/IE (the C-prolog version) and OPS5(compiled) on VAX11/780. As is evident from the graph, the Quintus(compiled2) realizes the efficiency of head searching in Prolog systems. The time required is linearly proportional to the number of rules. It seems that Quintus(compiled2) attains ideal performance in the Prolog system. The performance of the KORE/IE system on VAX11/780 is equal or superior to that of OPS5 on VAX11/780.



```
r1: if f(s1=1,s2=1,s3=1,s4=1,s5=1) then
      modify(1,s1=2,s2=2,s3=2,s4=2,s5=2).
                    :
                    :
rn: if f(s1=n,s2=n,s3=n,s4=n,s5=n) then
   modify(1,s1=n+1,s2=n+1,s3=n+1,s4=n+1,s5=n+1).
```

Fig.11. Evaluation of KORE/IE.

5. Conclusions

In this paper, we have proposed and discussed the following advantages of KORE/IE; (1) a mechanism for an inference engine for KORE, (2)cooperative problem solving among rule bases, (3)an efficient inference mechanism. By using (1) and (2), we have shown the flexible and powerful mechanisms of KORE/IE. The (3) is the most important subject in logic programming and also the main part of this paper. In KORE/IE, we have realized a speedy inference mechanism without restricting rule expressions by using advantages of Prolog systems. Specifically speaking, in order to realize the full speedy mechanism, we have utilized the features of the inference mechanism and the efficient refutation mechanism of Prolog systems. Namely, by compiling rules into optimized Prolog programs, recognize-act cycles are transformed into execution of the Prolog programs based on the speedy refutation mechanism. The programs consist of LHS programs and RHS programs. By using the LHS program we can realize an efficient matching process. It can be seen from the experimental results that the efficiency is superior to that of the Rete Match algorithm.

Acknowledgment

The author would like to acknowledge the continuing guidance and encouragements of Dr. Tosio Kitagawa, the president of IIAS-SIS, and Dr. Hajime Enomoto, the director of IIAS-SIS. The author is deeply grateful to Dr. Mitsuhiko Toda, IIAS-SIS, for reading the draft of this paper and giving him many valuable comments. Also many thanks to Mr. Yoshinori Katayama and Mr. Kunihiko Hiraishi, IIAS-SIS, for their useful comments. The author is also very grateful to Mr. Hiromichi Futagami for his computer programming with great competence and enthusiasm.

This research has been carried out as a part of Fifth Generation Computer Project.

References

Brownston L, Farrell E K, and Martin N(1985).Programming expert system in OPS5. Addison-Wesley

Brekke B(1986) Benchmarking Expert System Tool Performance. Ford Aerospace Tech Note

Forgy CL(1981) OPS5 User's Manual. CMU-CS-81-135,July

Forgy CL(1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence 19: 17-37

Futamura Y (1983) Partial Computation of Programs, Lecture Notes in Computer Science 147, Springer-Verlag

Kowalski R (1977) Logic for Problem Solving. Elservier North Holland :49-74

Lesser VR and Erman LD (1977) A retrospective view of the HEARSAY-II architecture. Proc. IJCAI 5:790-800

Matsumoto Y, Tanaka H, and Kiyono M (1983) BUP: A Bottom-up Parser Embedded in Prolog. New Generation Computing 1,No.2

Shintani T, Katayama Y, Hiraishi K, and Toda M (1986) KORE: A Hybrid Knowledge Programming Environment for Decision Support based on a Logic Programming Language. Lecture Notes in Computer Science 264, Logic Programming '86 :22-33

Takeuchi A and Furukawa K (1985) Partial Evaluation of Prolog Programs and Its Application to Meta Programming. in Kuger, H.-J.(ed.): Information Processing 86, Dublin, Ireland 415-420., North-Holland