TM-0059

PROBLEMS IN DEVELOPING AN EXPERIMENTAL SYSTEM ABLE TO REUSE EXISTING PROGRAMS

by
Yoshiaki Nagai, Eiki Chigira, Masakazu Kobayashi
(Hitachi, Ltd.)
Kouichi Furukawa
(ICOT)

April, 1984

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

ABSTRACT

This paper discusses an experimental system for reusable software engineering.

Software productivity can be greatly improved if a "software recycling" system is developed which, given a requirement specification, extracts required program parts, and combines them into a new program.

As a first step toward this goal, an expert system is developed which contains knowledge of skilled program designers as well as standard program parts.

In this paper a behavior model of a designer is first presented who wants to reuse existing software. Based on the model functions are defined which are necessary to support the designer. Behavioral models are studied in detail. This functional analysis leads to the use of a blackboard-type inference engine as the problem solver of the system, since the inference engine must have complex problem solving facilities.

This paper also discusses the use of a knowledge programming language/system, called Mandala, to implement the experimental system.

Finally we also consider the syntax of production rule for the inference engine in accordance with the syntax of Mandala.

1 Introduction

Recently, reusable software engineering technology has attracted public attention as a very reliable tool capable of enhancing software productivity. This is because the reuse of existing software will bring about many favorable effects, such as streamlining development procedures and test work, simplifying maintenance,etc. Reusable software engineering will be a key technology for software development in the future.

Future reusable software engineering systems are expected to be able to :

1) use only parts of existing programs in accordance with the specifications of new software.

2) synthesize these parts.

3) modify these parts.

Research on these activities is proceeding in ICOT. We should be able to apply this research to the development of the first.

However, this paper concentrates on fundamental steps necessary to realize the seconds.

As a first step, we must recognize the following givens :

1) Standard program parts have already been stored in a data base system.

2) Suitable parts must be selected from the system for a new program.

3) These parts must be put together to form a new program.

Generally, a skilled program designer must utilize this process. Therefore, we seek to develop an expert system with the knowledge of such a designer in order to implement these functions.

We first present a behavior model of a man who wants to reuse existing software. We then study certain functions necessary to implement an experimental system which will allow him to do this. Next, we develop detailed models for this system

by functional analysis.

In module design, we developed some programs to realize the functions.

Finally, we discuss problems, particularly how to generate the blackboard-type inference engine coded in Prolog, how to merge the Mandala and blackboard-type inference engine, and how to represent the format of a production rule.

2 Cognitive Study of Software Reuse and Necessary Functions

2.1 Cognitive model

We shall construct a model in which a skilled designer wishes to produce a new program specification reusing existing program parts. Suppose that the domain of the new program is business use.

(1) A program designer designs a 'rough' specification of a new program.

(2) He marks groups of parts applicable to the new specification.

(3) He selects one part from each group.

(4) He modifies these parts.

In this model, there are two decision processes in the design program, one in which program parts are selected and modified, and another in which a program logic is configured.

This model shows a procedure of program design after the system was already designed. Therefore, the information input to and output from, the functions of, and the file names used by the system are fixed. In designing a 'rough' specification, program functions must be fixed, while keeping in mind the relation between input and output information.

Next, the designer confirms the logical flow of the program functions, simultaneously marking the program part in accordance with the required specification. Alternative program part combinations are generated in the process, and a few are selected by trial and error.

Third, the designer studies the logic of the program parts and determine which parts meet his "expectations". In this process, the logic of the new program is fixed and the new specification completed. A model illustrating these processes is shown in Fig.1.

2.2 Functions necessary for the system.

The concept of a system reusing program parts will be discussed in this section based on the model described above.

The model is divided into two submodels:

1) determining the part to be reused,

2) modifying this part.

(1) determining the program part to be reused

An expert system must have at least a mechanism for reasoning and the relevant knowledge in order to be able to select a program part.

The first output from this inference engine is a logic chart involving a whole specification for the new program. This output shows that the system reasons what kind of program is to be solved.

The second output from the engine is the reusable program parts. When the expert system comes to understand the program, it reasons how the problem is to be solved. It then, outputs the reusable program parts.

(2) modifying the program part

Modification of program parts selected by a system with the knowledge of an experienced program designer will be considered.

In the reconstruction of any program part, man illustrates the problem, finds the difference from the standard part, writes down the new information, and finds related information. He then modifies the part.

With some information, text or sentences may be more useful than illustrations to

display the relationship between information represented by sentences and illustrations.

To implement this system, the following two functions are necessary.

(1) Simultaneous display of illustration, text and sentence for part specification.

(2) Logical connection of information between picture and sentence.

These concepts are shown in Fig.2.

3 Operational Model of First Implementation

We shall present an example of a file collation program produced by selecting the proper parts from many standard program parts, modifying them, and combining them into one program.

(1) Questions from the system(initial questions)

Fig. 3 shows a frame of first communication between man and system. As the answer of 'outline of process' was 'file collation', the question 'names of file' is reasoned.

The system will construct a logic chart through these questions and answers.

(2) Logic chart output

As for the arrangement of a logic chart, the system reasons the functional model that performs the collation of files.

Therefore, the system must have the knowledge necessary for combining any parts.

Fig. 4 shows the logical flow of the collation program.

It is impossible to determine the program part with regard to the matching process of any records for file collation when the conditions of matching process are not fixed.

(3) Question for program part selection

In this example, it is necessary to input the numbers of the key word for matching

and of 'control break' for writing an error list.

After receiving this information, the system determines the best program part.

In this reasoning process, the system uses the knowledge it has for selecting program parts.

If we input information to the effect that there are two pieces of data in one transaction file record in accordance with a matching key word and none for writing the error list, we get the illustration shown in Fig.5.

(4) Support for program part modification

We consider the operational model of the program part editor as an example of the program part 'read a transaction file'.

The standard specification of the part is shown in Fig.6 and 7. We shall add any output item to the part. In this case, the program designer writes some items, such as 'name of article, volume, unit price, sum', in the display with text type. The system will display this data in the logic chart box.

This completes a new specification for a program part, and the experimental system will have accomplished its goal.

To implement this operational model by computer, the system must have modules of inference engine, knowledge and editor for modifying program parts.

We use the Mandala as an engine. However, we intend to develop a production system which processes certain knowhow.

The editor for program part modification will be constructed with logic chart editor, program part specification editor, and man machine interface.

These modules are illustrated hierarchically in Fig.8.

4 Problems in First Implementation

The following problems have arisen in the design and implementation of the

system:

1) How to implement a blackboard-type inference engine coded in relational language such as Prolog.

2) How to design a blackboard-type engine consistent with the specification of Mandala.

3) How to implement the syntax of knowledge represented by production rules.

We shall discuss each problem.

(1) Implementation of the blackboard-type engine code in relational language.

The blackboard-type inference model features

1) easy systemization of problem solving knowledge,

2) the ability to reason a complex problem by hypothesis on the blackboard.

We use the blackboard-type inference engine for complex problem solving.

A blackboard-type inference engine is constructed with forward chaining, backward chaining, and reasoning using classified knowledge in accordance with the hypothesis on the blackboard.

Backward reasoning is considered mechanically to contain forward reasoning, so we operated rule-interpreter in a blackboard-type engine using backward chaining.

The rule interpreter and rule are shown in Fig.9.

(2) Designing the blackboard-type inference engine consistent with Mandala.

For implementation, we wanted to describe the blackboard-type inference engine, which is a rule-oriented system, in the manner of the object-oriented system Mandala. We considered the configuration shown in Fig.10.

Our proposed system is composed of Problem Solver, Program Parts Base, and Explanation Handler. Problem Solver consists of three components : a unit world, which includes methods for production rule interpreter ; a group of unit worlds for production rule (knowledge sources), which is connected to the interpreter with

is_a link in Mandala; and a group of problem solver instances.

Program Parts Base is a kind of program part data base for a reusable software system. Program parts are described and stored in the hierarchy of the system with is_a link in Mandala. Explanation Handler explains the inference process through the journal after inference or anytime the user wants to know.

Our system is initiated when the manager of Problem Solver receives a message about a problem and a request for problem solving. At this time, an instance of problem solver (Problem Solver 1), decided by the user before inference, is created, and interprets production rules in the unit world. According to the reasoning, some required problem solver instances of knowledge sources are created and begin reasoning concurrently. On inference, Problem Solver sends a message for refer or update when the interpreter refers, updates, or creates instances of program parts. Program Parts Base, which receives the message, searches the corresponding program part object, changes the state, and replies to Problem Solver.

The instance, created in Problem Solver unit world (for example,Problem Solver 1), sets some commonly shared variables when it proceeds to reason.

Thus, several problem solver instances can reason concurrently.

The area of these commonly shared variables is called blackboard, a kind of short term memory, and on the blackboard some hypotheses are constructed in the reasoning process. The blackboard is a commonly shared area, so it must be able to merge update requirements from problem solver instances. The effective merging technique on the blackboard will lead to an extremely powerful blackboard-type inference engine when the PIM (Parallel Inference Machine) is developed in ICOT in the near future.

(3) Syntax of rule

We shall describe a production rule syntax in Mandala. We shall also consider the representation of a knowledge source and rules consistent with the unit world of

Mandala.

Replace the name of a knowledge source with that of a unit world according to the representation of the unit world in Mandala. The knowledge should describe the following.

⟨knowledge source⟩ ::= class (⟨knowledge source name⟩)

⟨rules⟩...

⟨knowledge source name⟩ ::=⟨identifier⟩

⟨rules⟩ ::=⟨knowledge source name⟩(⟨is_a relation⟩, true)

⟨knowledge source name⟩(⟨rule⟩, true)

⟨knowledge source name⟩(trigger([⟨level name⟩,...]),true)

⟨knowledge source name⟩(consequence([⟨level name⟩,...]),true)

⟨is_a relation⟩ ::= ⟨object name⟩ is_a ⟨object name⟩

⟨object name⟩ ::=⟨identifier⟩

⟨level name⟩ ::=⟨identifier⟩

Rules are composed of knowledge source characteristics, production rules, user predicate definitions, etc. The characteristics of knowledge source for example, could be hierarchical relation of knowledge sources, levels, or objects on a blackboard referring to a left hand side of a rule

The unit worlds of knowledge sources and rule interpreter are connected with the link of 'is_a' link in Mandala.

The levels on a blackboard are described as follows.

⟨knowledge source name⟩(trigger ([⟨level name⟩,...]), true).

The levels and objects on a blackboard referring to the right hand side of rules are described as follows.

⟨knowledge source⟩(consequence([⟨level name⟩,...]),true).

The production rule has the following syntax.

⟨knowledge source name⟩ (⟨rule⟩,true)

This rule has a functor of four arguments. These are the name, the left-hand side, and right-hand side of the rule and the data to execute the next knowledge source. Using these, we get a rule description method consistent with the unit world of Mandala.

⟨rule⟩ ::= rule(⟨rule name⟩, ⟨Lhs⟩, ⟨Rhs⟩, ⟨control information⟩)

⟨rule name⟩ ::= ⟨identifier⟩

⟨Lhs⟩ ::= ⟨Lhs phrase⟩,...

⟨Lhs phrase⟩ ::= ⟨object⟩

| ⟨user predicate⟩

⟨Rhs⟩ ::= ⟨Rhs phrase⟩,...

⟨Rhs phrase⟩ ::= add_obj (⟨object⟩)

| add_attr (⟨object⟩)

| rep_attr (⟨object⟩)

| set_goal (⟨object⟩)

| add_frame (⟨frame name⟩,[⟨slot⟩,...])

| remove_frame (⟨frame name⟩)

| add_slot (⟨frame name⟩, ⟨slot⟩)

| chg_slot (⟨frame name⟩, ⟨slot⟩)

| remove_slot (⟨frame name⟩, ⟨slot⟩)

⟨object⟩ ::= (⟨object name⟩, ⟨attribute⟩)

⟨attribute⟩ ::= ⟨attribute name⟩ (⟨attribute value⟩)

⟨object name⟩ ::= ⟨identifier⟩

⟨attribute name⟩ ::= ⟨identifier⟩

⟨attribute value⟩ ::= ⟨identifier⟩

5 Conclusion

We discussed the problems involved in implementing a system able to reuse existing programs coded in a relational language.

We described a cognitive model of a skilled program designer producing a new program by reusing existing software. We also described the functions of the the system.

In addition, we examined program part reuse based on the model and constructed modules in detail. In the last part, we examined a rule interpreter in the blackboard-type inference engine to implement the system. We also discussed specifications of the inference engine consistent with the unit world of Mandala.

Finally, we described a syntax of rules.

We are going to develop a system able to reuse existing software under the specifications mentioned above.

6 Acknowledgement

The authors wish to express their thanks for the practical advice and guidance received from Mr.Yoshihiko Aoyama, Department Manager, Systems Development Labolatory, Hitachi, Ltd. They also acknowledge the help on the technical problems of knowledge engineering given by Mr.Hirokazu Ihara, Deputy Manager of Systems Development Laboratory, Hitachi Ltd.

7 References

(1) Furukawa,K., A.Takeuchi, S.Kunifuji:

Mandala: A Concurrent Prolog Based Knowledge Programming

Language/System, IPSJ, Nov.1983.

(2) Nii,H.P., N.Aiello: AGE(Attempt to Generalize):

A Knowledge-Based Program for Building Knowledge-Based Program,

IJCAI-79, 1979

(3) Hayes-Roth,F., D.A.Waterman, D.B.Lenat:

Building Expert Systems, Addison - Wesley, 1983

(4) Sowa,J.F.:

Conceptial Structures - Information Processing in Mind and Machine,

Addison - Wesley, 1984

(5) Shapiro,E.Y., A.Takeuchi:

Object Oriented Programming in Concurrent Prolog, ICOT Technical Report,

April, 1983

(6) Shapiro,E.Y.:

A Subset of Concurrent Prolog and Its Interpreter, ICOT TR-003 Jan. 1983

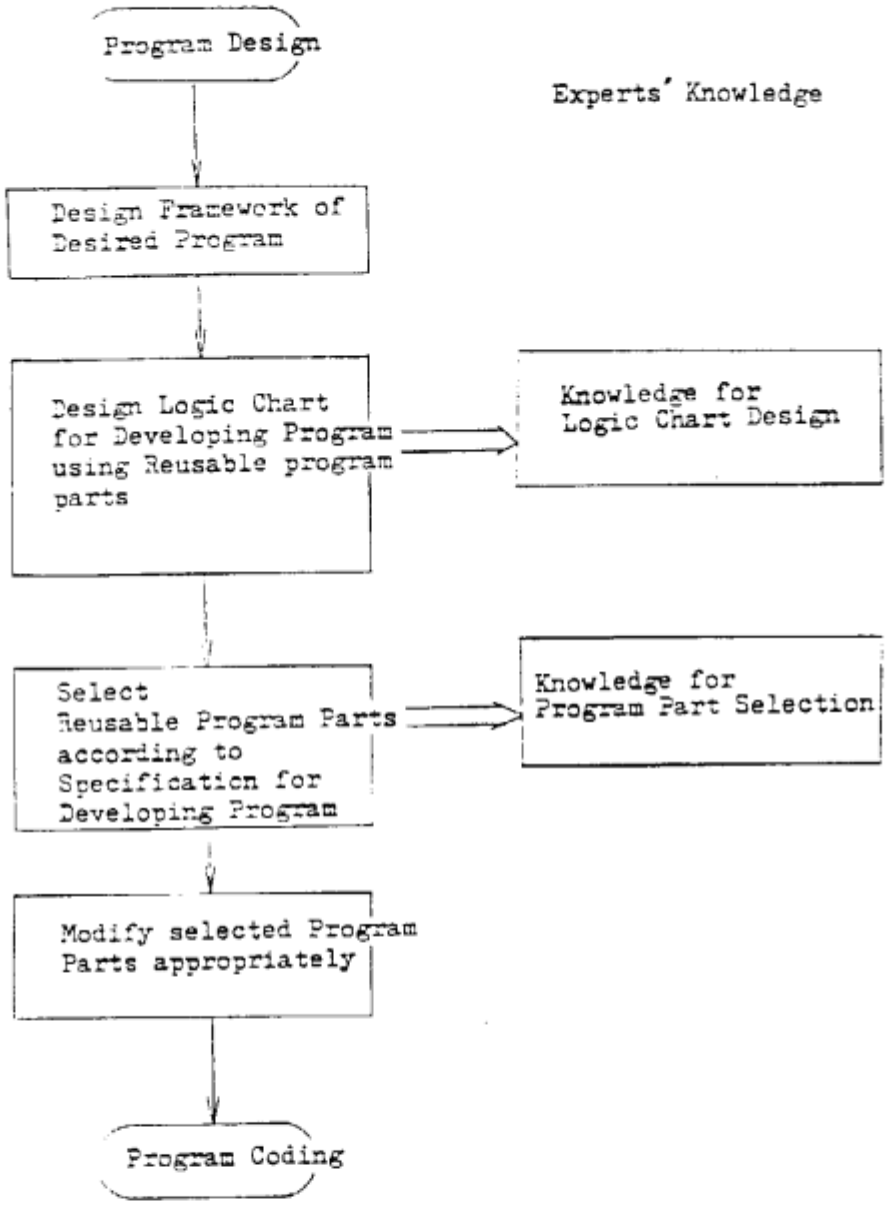

Figure 1. Operation Model for Reusing Program Parts and Experts' Knowledge

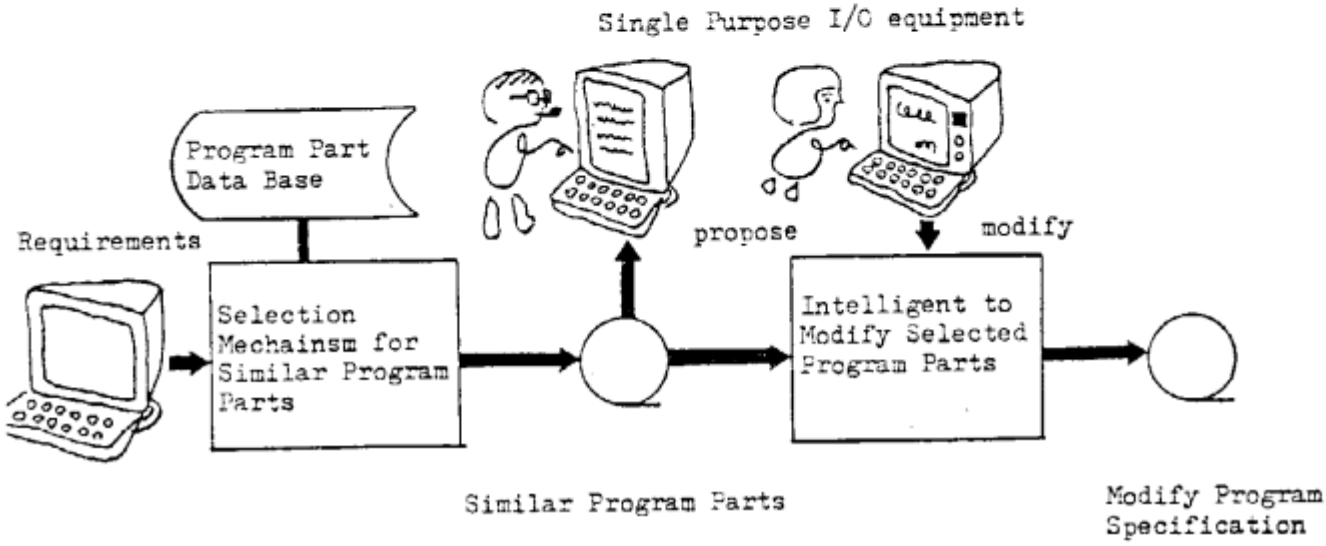

Figure 2. Image for Fundamental Experimentation of Reusable Software Engineering System

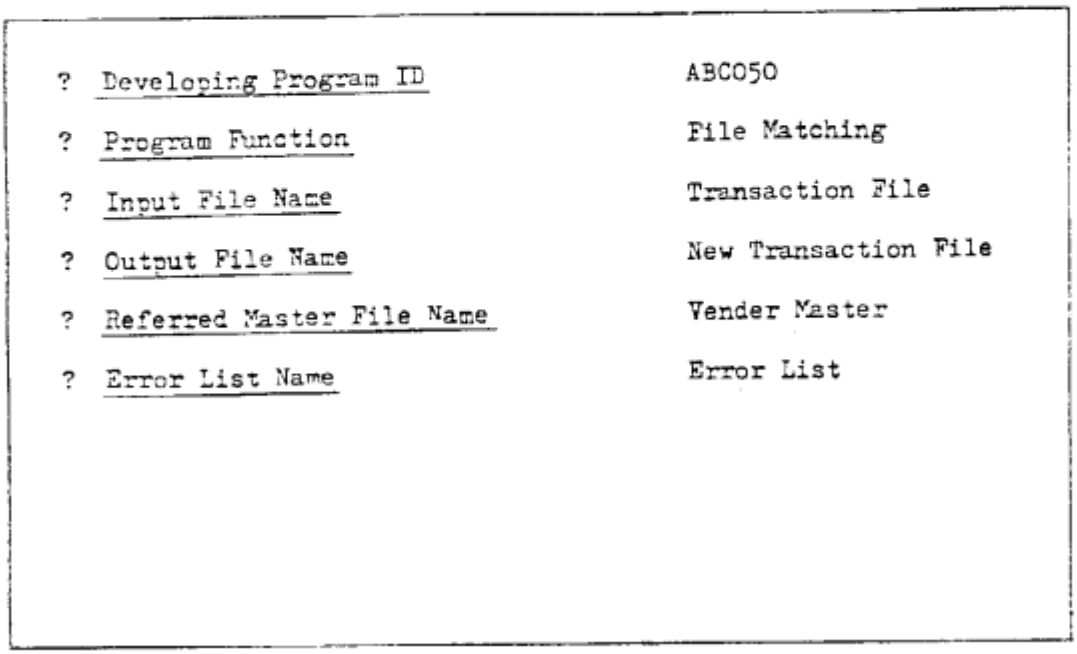

Figure 3. Initial Action of System
(underlined parts are questions from the System)

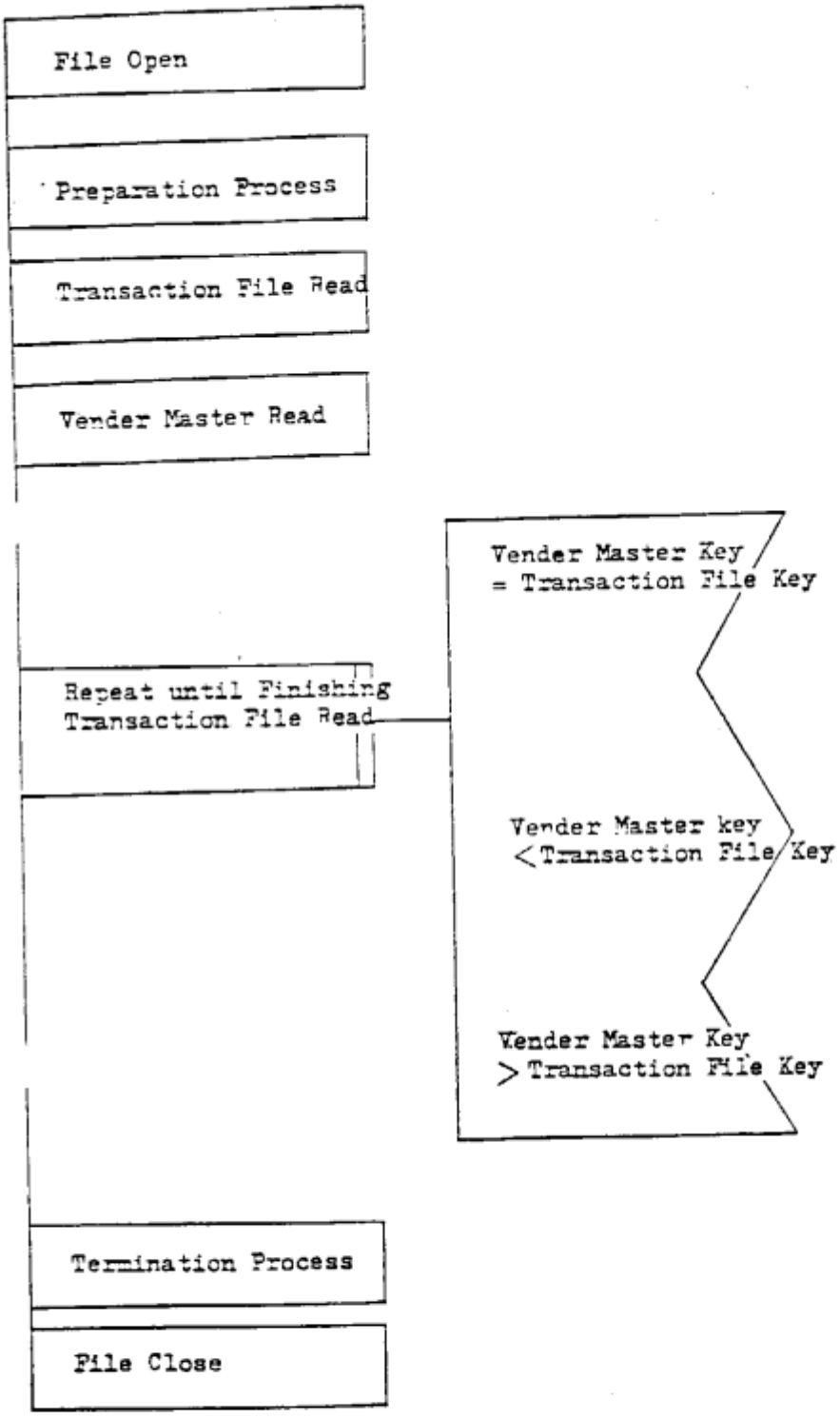

Figure 4. New Program Logic

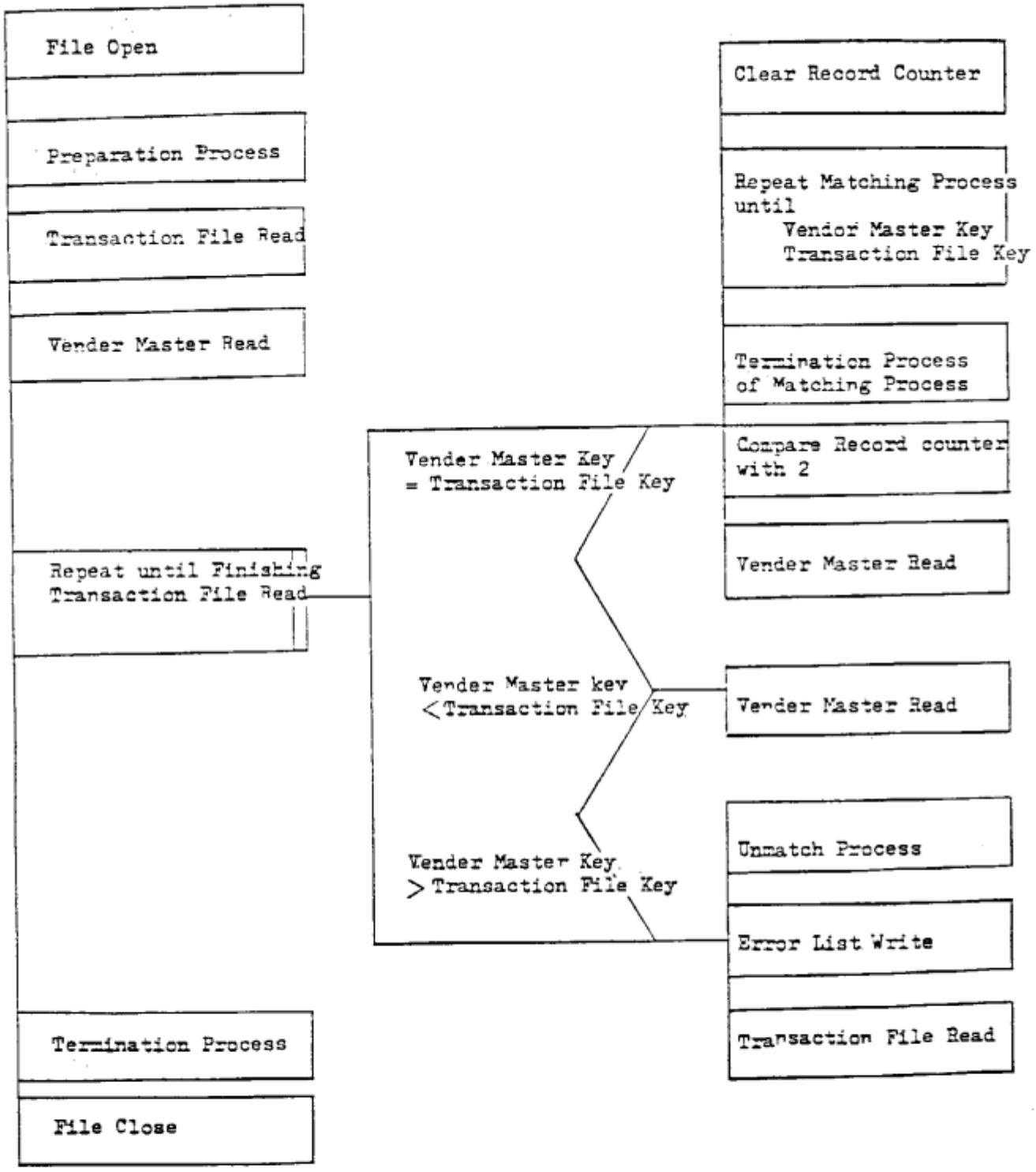

Figure 5. Global View of File Matching Program Part(New Program Logic)

Transaction File Read

Function This Part Reads Transaction File.
It Gives High-Value to Matching Key Area when it finds End of File.

Data Definition

Input

Transaction File

```
RECORDING MODE IS F

LABEL RECORD STANDARD

BLOCK CONTAINS 00080 CHARACTERS

DATA RECORD TRANS-REC
```

Output

Trans-rec

```
02  TRANS-KEY IS PICTURE X(03)

02  TRANS-MEI IS PICTURE X(30)

02  DATE

    03  YY IS PICTURE 9(02)

    03  MM IS PICTURE 9(02)

    03  DD IS PICTURE 9(02)

02  FILLER IS PICTURE X(41)
```

Figure 6. Program Parts Specification(before modification)

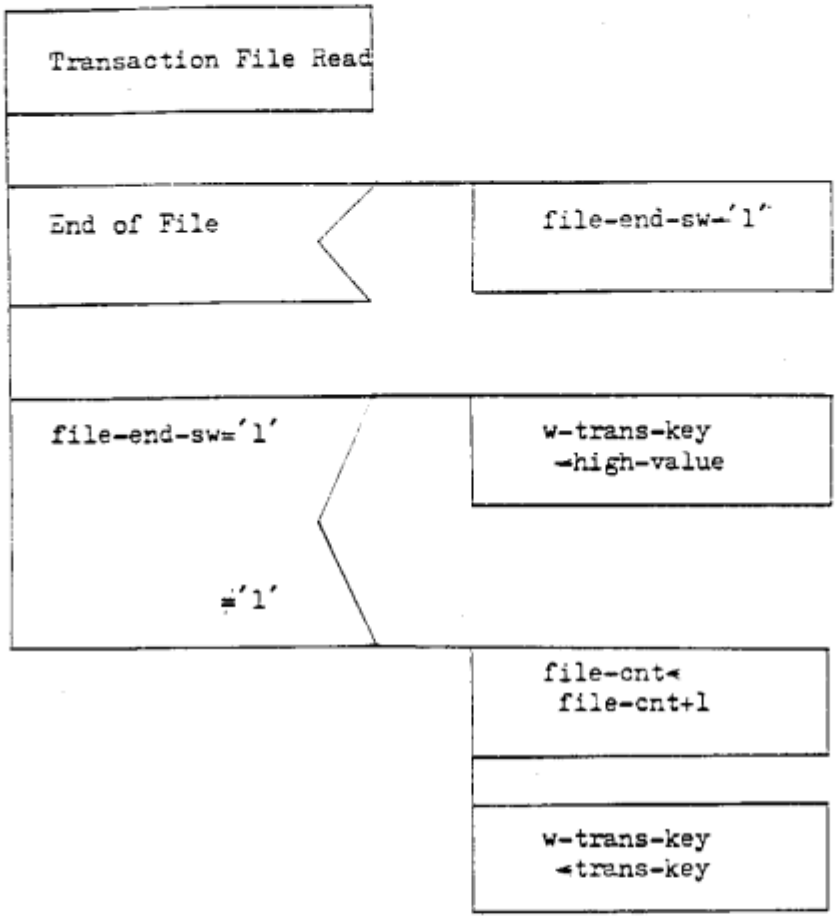

Figure 7. Program part Specification (befor modification)

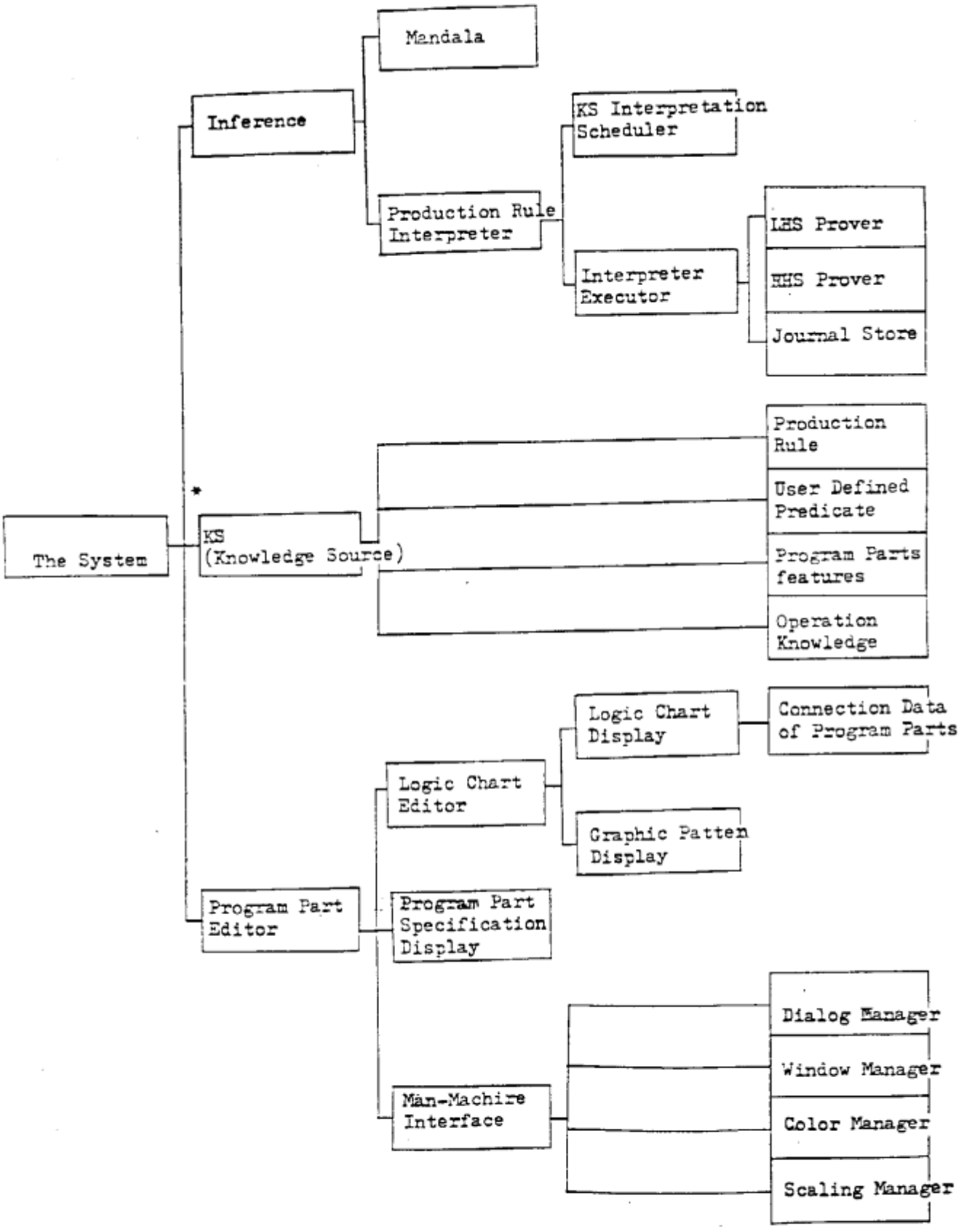

Figure 8. Module Decomposition Tree of
Reusable Software Engineering System for Fundamental Experimentation

* treat as a kind of module

```
/*   interpreter   */

go :-
   seeing(Input),
   see(lta4),
   read(LTM),
   seen,
   see(Input),
   reconsult(rule4),
   prompt(_,'> '),
   repeat,
      nl,
      write('nani o shoumei shitai desuka?'),
      nl,
      read(X),
      produce(X,LTM,R),
      result(X,R),
   until(X).

until(nashi).

result(nashi,_) :-
   !.
result(X,t) :-
   write('kotae  '),
   write(X),
   write('  wa'),
   write(' shoumei saremashita!'),
   nl.
result(X,f) :-
   write('kotae  '),
   write(X),
   write('  wa'),
   write(' shoumei fukanou desu!'),
   nl.

produce(nashi,_,_) :-
   !.
produce(X,LTM,R) :-
   prod([X],[X],LTM,R),
   !.

prod([],_,_,t).
prod([H|T],[Y],LTM,t) :-
   prod1([H],[Y],LTM,t),
   prod(T,[Y],LTM,t).
prod(_,_,_,f).

prod1([X],_,LTM,t) :-
   member(X,LTM),
   output([X,'(',in,'LTM',')',nl]),
   !.
prod1([X],[Y],LTM,t) :-
   not((rule(_,r:C=>A),
        act(A,STM,[X],np),
        recognize(C,STM,np))),
   not(X=Y),
   question(X,R),
```

Figure 9-1. Experiment Classification of Production Rule (1/2)

(Production Rule Interpreter)

```
    exit(R).
prod1([X],[Y],LTM,t) :-
    X @< a,
    rule(c0,r:C=>A),
    act(A,STM,[X],P),
    recognize(C,STM,P),
    prod(STM,[Y],LTM,t).
prod1([X],[Y],LTM,t) :-
    X @>= a,
    rule(c1,r:C=>A),
    act(A,STM,[X],P),
    recognize(C,STM,P),
    prod(STM,[Y],LTM,t).
exit(hai).

question(X,R) :-
    repeat,
        write(X),
        write(' wa tadashii desuka?'),
        nl,
        read(R),
    until1(R),
    !.
until1(hai).
until1(iie).

act([],X,X,_).
act([H|T],OldSTM,STM,P) :-
    action(H,STM1,STM,P),
    act(T,OldSTM,STM1,P),
    !.
action(insert(X),L,[X|L],_).
action(insert(X),[Y|L1],[Y|L],_) :-
    action(insert(X),L1,L,_).
action(delete(X),[X|L],L,_).
action(replace(X,Y),[X],[Y],_).
action(replace(X,Y),[X|L],[Y|L],_).
action(output(P),X,X,P) :-
    call(output(P)).
action(output(_),X,X,np).
action(E,X,X,_) :-
    call(E).
recognize([],_,_).
recognize([H|T],STM,P) :-
    condition(H,STM,P),
    recognize(T,STM,P),
    !.

condition(insta(X),STM,_) :-
    member(X,STM).
condition(output(P),_,P) :-
    call(output(P)).
condition(output(_),_,np).
    call(E).

output([]).
output([nl|T]) :-
    !,
    nl,
    output(T).
output([H|T]) :-
    tab(1),
    write(H),
    output(T).
```

Figure 9-1. (2/2)

```
/*   rules   */

rule(c0,r:
        [insta(a),
         output([a,nl])]
      =>
        [replace(a,b),
         output([b,'<='])].
rule(c0,r:
        [insta(s),
         insta(b),
         output['b,s',nl])]
      =>
        [delete(s),
         delete(b),
         insert(c),
         output([c,'<='])]).
rule(c0,r:
        [insta(h),
         output([h,nl])]
      =>
        [replace(h,e),
         output([e,'<='])].
rule(c0,r:
        [insta(c),
         insta(z),
         output['c,z',nl])]
      =>
        [replace(c,e),
         delete(z),
         output([e,'<='])].
rule(c0,r:
        [insta(e),
         output([e,nl])]
      =>
        [replace(e,f),
         output([f,'<='])].

rule(c1,r:
        [insta(y),
         output([y,nl])]
      =>
        [replace(y,z),
         output([z,'<='])].
rule(c1,r:
        [insta(w),
         insta(x),
         output['w,x',nl])]
      =>
        [delete(w),
         delete(x),
         insert(y),
         output([y,'<='])].
rule(c1,r:
        [insta(u),
         output([u,nl])]
      =>
        [replace(u,v),
```

Figure 9-2. Experiment Classification of Production Rules (1/2)
(Production Rules)

```
            output([v,'<='])]).
rule(c1,r:
          [insta(v),
           output([v,nl])]
        =>
          [replace(v,x),
           output([x,'<='])]).
rule(c1,r:
          [insta(b),
           output([b,nl])]
        =>
          [replace(b,w),
           output([w,'<='])]).

/* LTM */
[a,u].
```

Figure 9-2. (2/2)
(Production Rule , Initial Values)

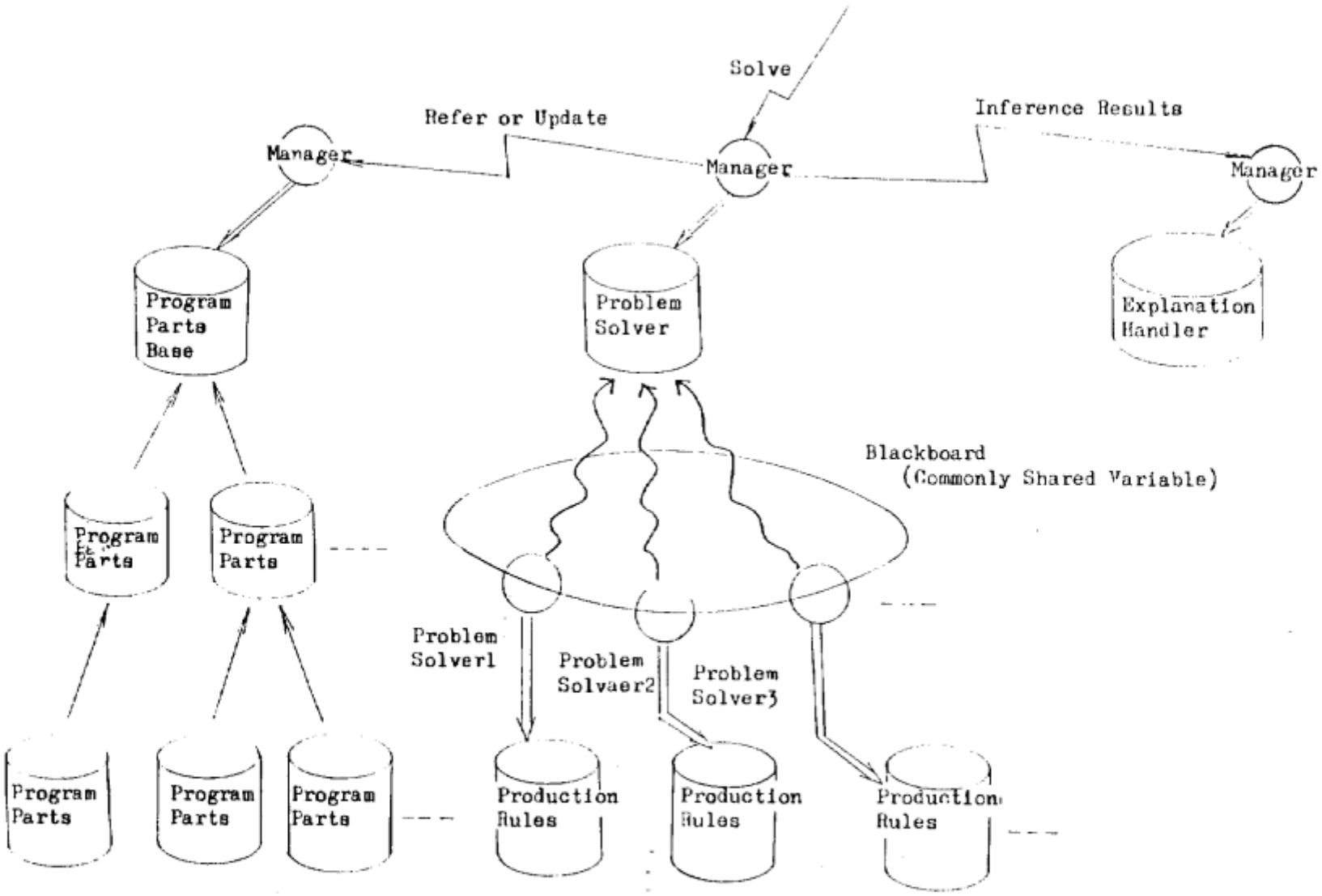

Figure 10. Blackboard-Type Inference Engine on Mandala