AN INTELLIGENT CAI SYSTEM BASED ON LOGIC PROGRAMMING

KAZUHISA KAWAI

February 1986

Graduate School of Engineering Science
Osaka University

ACKNOWLEDGEMENTS

First and foremost, the author would like to express his sincerest gratitude to Professor Jun'ichi Toyoda, this thesis supervisor. Prof. Toyoda has had a great influence on what the author accomplished in these years, and what he knows today. He has been a constant source of invaluable advice and encouragement throughout the course of the research for this thesis.

Special thanks are also due to Prof. T. Fujisawa, Prof. T. Kasami, Prof. K. Takashima, Prof. N. Tokura and Prof. K. Torii for their invaluable guidance through the course of author's undergraduate and graduate studies.

The author wishes to thank Prof. O. Kakusho of the Institute of Scientific and Industrial Research for his invaluable suggestions and encouragements.

The author wishes to acknowledge valuable discussions with Assistant Prof. K. Uehara. He also escorted the author's first steps in Prolog.

The author is also grateful to Assistant Prof. R. Mizoguchi, Assistant Prof. M. Yanagida, Assistant Prof. T. Yamaguchi and the colleagues of Prof. Toyoda's laboratory and Prof. Kakusho's laboratory, in particular, Mr. M. Ganke, Mr. H. Kinoh and Mr. Y. Nakamura.

ABSTRACT

Kazuhisa Kawai, Graduate School of Osaka University

February 1986

Doctoral Thesis: An Intelligent CAI System Based on Logic Programming

The problems studied in this thesis are concerned with educational applications of the computer technology and, in particular, centered on ICAI systems based on logic programming. Educational applications of the computer technology are classified into two categories according to the role of the computer. First, a computer instructs the students as a human teacher. ICAI systems are included in this category. Second, the student uses a computer as learning tools. One of the typical systems of this category is LOGO which provides an environment for learning the skills of problem-solving. The main components of an ICAI system are the expertise, the student model and tutoring strategies. The student model manages what the student does and does not understand, and the performance of an ICAI system depends largely on how well the student model approximates the human student. We propose a new framework for ICAI systems which uses inductive inference for constructing the student model from the student's behavior. In the framework, both the expertise and the student model are represented as Prolog programs, which enables to express meta-knowledge that is the student's knowledge of how to use his knowledge. The framework is also implemented in Prolog. Since the construction of the student model is performed independently of the expertise,

the framework is domain-independent. Therefore, ICAI systems for any subject area can be built with the framework. Two actual ICAI systems, for programming in Prolog and for chemical reaction equations, are constructed with the framework.

For the purpose of acquiring programming skills, the following two learning schemes are needed; to give the knowledge for the syntax and basic techniques of the language and to let the students practise programming in the language. These two schemes correspond to two categories of educational applications of the computer technology. From this point of view, we develop a new ICAI system for programming in Prolog that has both teaching aids and the learning environment. Teaching aids are constructed with the framework for ICAI systems together with the domain-specific knowledge for the expertise of programming in Prolog and for tutoring strategies. The learning environment has a parser to diagnose the student's program and a visual Prolog interpreter.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Educational applications of the computer technology are classified into two categories according to the role of the computer. First, a computer instructs the students as a human teacher. Electronic page-turners, frame-oriented CAI (Computer-Assisted Instruction) systems and ICAI (Intelligent CAI) systems are included in this category [1][2]. Second, the student uses a computer as learning tools. One of the typical systems of this category is LOGO [3] which provides an environment for learning the skills of problem-solving. Systems using games and simulations as learning tools are also classified into this category [1].

The main components of an ICAI system are the expertise, the student model and tutoring strategies. The student model manages what the student does and does not understand, and the performance of an ICAI system depends largely on how well the student model approximates the human student. We propose a new framework for ICAI systems which uses inductive inference for constructing the student model from the student's behavior. In the framework, both the expertise and the student model are represented as Prolog programs, which enables to express meta-knowledge that is the student's knowledge of how to use his knowledge. The framework is also implemented in Prolog. Since the construction of the student model is performed independently of the expertise, the framework is domain-independent. Therefore, ICAI systems for any subject area can be built with the framework. Two actual ICAI systems, for programming in

Prolog and for chemical reaction equations, are constructed with the framework.

For the purpose of acquiring programming skills, the following two learning schemes are needed; to give the knowledge for the syntax and basic techniques of the language and to let the students practise programming in the language. These two schemes correspond to two categories of educational applications of the computer technology. In this thesis, they are called the interactive teaching system and the environmental learning system respectively. From this point of view, we develop a new ICAI system for programming in Prolog that has both teaching aids and the learning environment. Teaching aids are constructed with the framework for ICAI systems together with the domain-specific knowledge for the expertise of programming in Prolog and for tutoring strategies. The learning environment has a parser to diagnose the student's program and a visual Prolog interpreter.

In chapter 2 we describe a framework for ICAI systems based on logic programming. In the framework, student modelling can be considered as a process of inducing the whole understandings from the student's behavior. Hence, it is possible to model any kind of the student's understandings using a general inductive inference on the model representation language Prolog.

In chapter 3 we present two specific ICAI systems, for programming in Prolog and for chemical reaction equations, constructed with the framework. These applications demonstrate the domain-independency of the framework. In order to construct an specific ICAI system on the framework, three kinds of the domain-specific knowledge are needed; the expertise, tutoring

strategies, and the knowledge for interface. In the chapter, three kinds of the knowledge for each ICAI system are described and experiments of them are mentioned.

Chapter 4 is the closing chapter, in which we summarize the content of this thesis and point out the way to future research.

CHAPTER 2

A FRAMEWORK FOR ICAI SYSTEMS

2.1 Outline of the Framework

The overall configuration of the framework for ICAI systems is shown in Fig.1. The expertise module gives problems and comments to the student one by one. The student's responses are evaluated by the expertise and used to construct the student model. Shapiro's inductive inference algorithm called MIS (Model Inference System) [4], which synthesizes a Prolog program from the given facts, is applied to build the student model from the student's behavior. In modelling stage, the tutoring module chooses the next problem or comment to be presented. When the model construction is completed, the student's misconceptions are extracted from the student model using PDS (Program Diagnosis System) which is also developed by Shapiro [4]. The student's misconceptions are identified as bugs in the Prolog program representing the student model. The identified misconceptions are sent to the tutoring module. The tutoring module chooses the next problem or remedial comment suited for the misconceptions and indicates it to the expertise module.

The framework is quite independent of the subject area. Any specific ICAI system can be built based on the framework with the specific expertise, tutoring strategies and the knowledge for interface.

Both the framework and its application systems are implemented on the super mini-computer MV/8000II in MV-Prolog. The programming language MV-Prolog is an extended version of C-Prolog on the VAX systems [5]. It has the useful interface

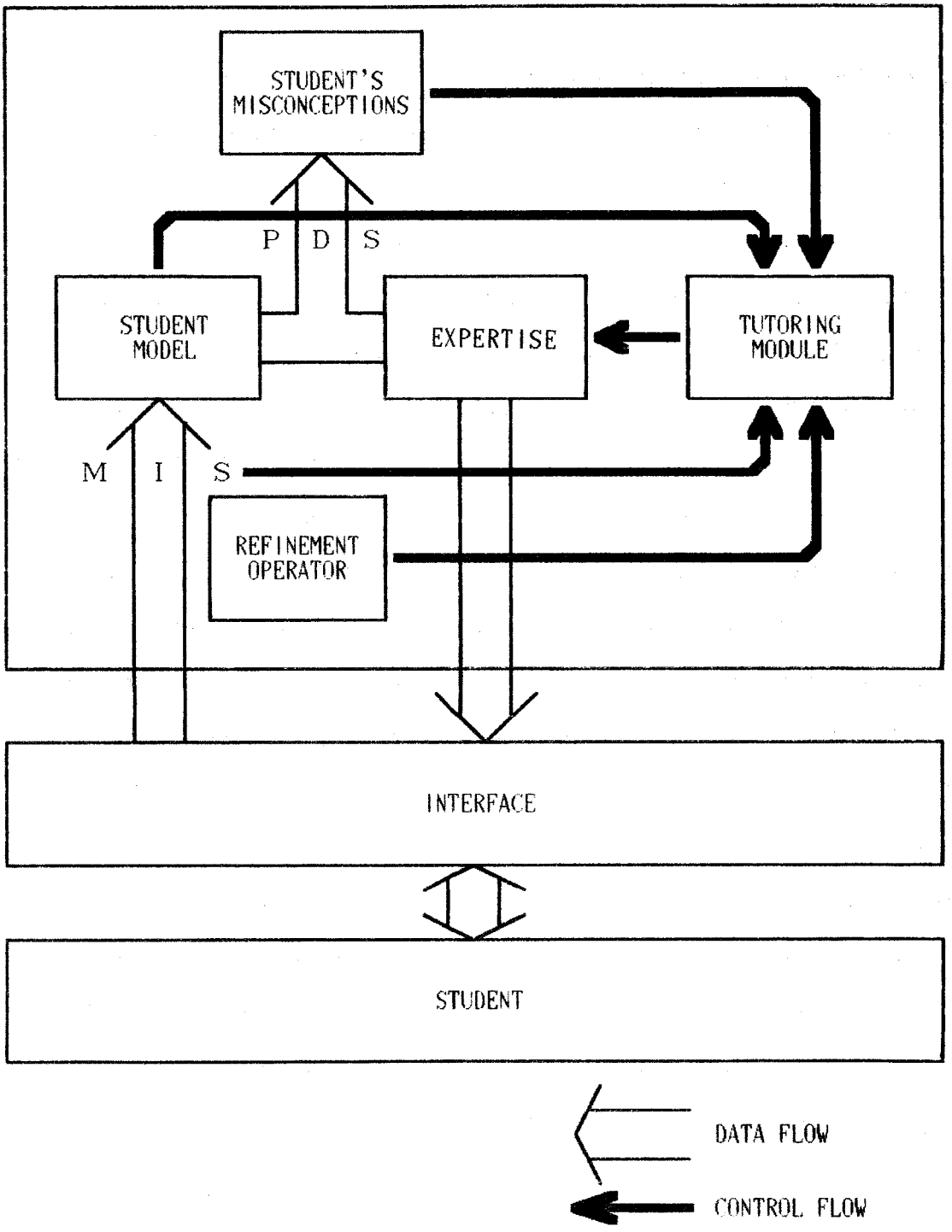


Fig.1 Diagram of the framework for ICAI systems

including windows, menus and graphics.

2.2 Knowledge Representation Based on Logic Programming

Prolog has the following distinctive features as a knowledge representation language [6].

1) Deductions on the representation are guaranteed correct.

2) The derivation of new facts from old ones are mechanized.

Therefore, Prolog is one of the most practical language for the knowledge information processing. In particular, we look upon as important that Prolog has an ability to represent the procedural knowledge. Both the expertise and the student model are not simple data-bases of course material, but include procedural knowledge concerning causal or relational reasoning, deduction and problem-solving. Prolog has a capability to represent both data and procedure as a program. Prolog is also suitable for inductive inference. Therefore, we employ Prolog for representing the expertise and the student model.

Some students can solve the basic problems but cannot solve the applied problems. They have the specific knowledge but cannot use it correctly. In order to model this type of misconceptions, the model (knowledge) representation must be possible to manage the meta-knowledge which is the knowledge of how to use the knowledge. For example, we consider a chemical reaction of an acid and a salt.

(1) soluble salt + acid
    -> precipitated salt + acid

(2) salt with volatile acid + unvolatile acid
    -> salt with unvolatile acid + volatile acid

Each of the above equations represents the basic knowledge. Equation (1) represents that when a soluble salt reacts with an acid, it forms a precipitated salt and an acid. Equation (2) also represents that a salt with an unvolatile acid and a volatile acid are formed by the reaction of a salt with a volatile acid and an unvolatile acid.

Solving the applied problem of a chemical reaction of an acid and a salt, the above basic knowledge must be used in correct manner. An expert solves the applied problem using the above basic knowledge as follows:

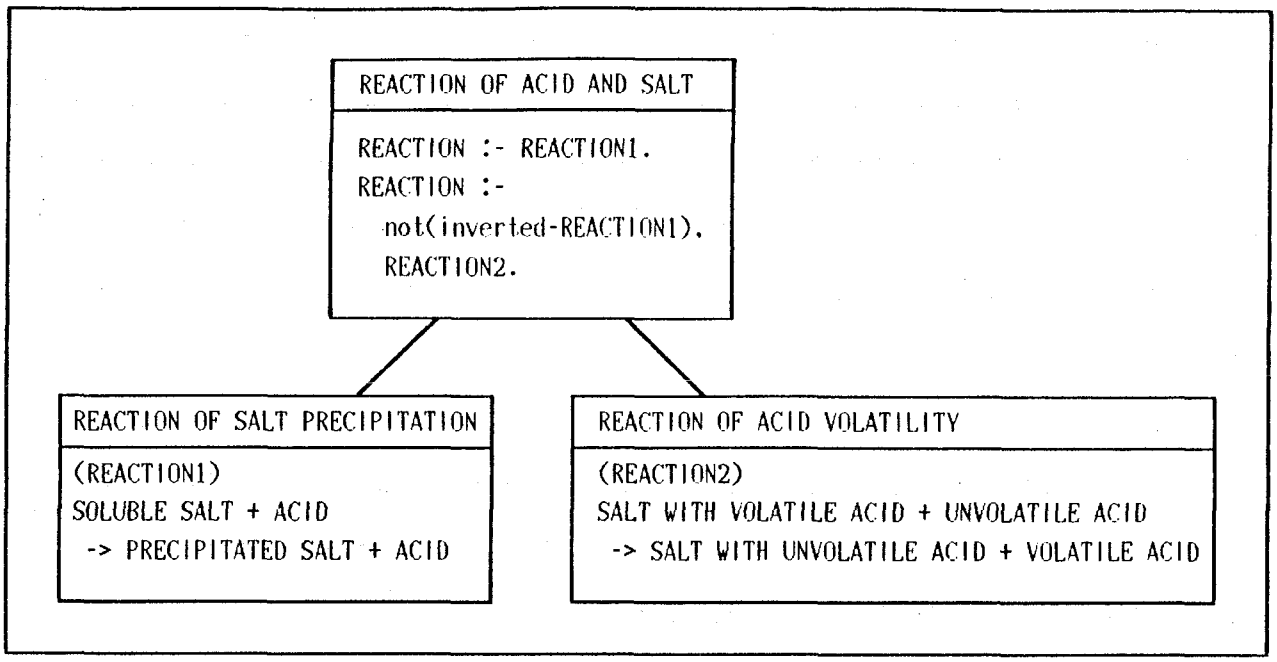
i) if a precipitated salt is formed, then apply the equation (1).

ii) if a reactant is a precipitated salt and a product is a soluble salt, then apply the equation (1) in the opposite direction.

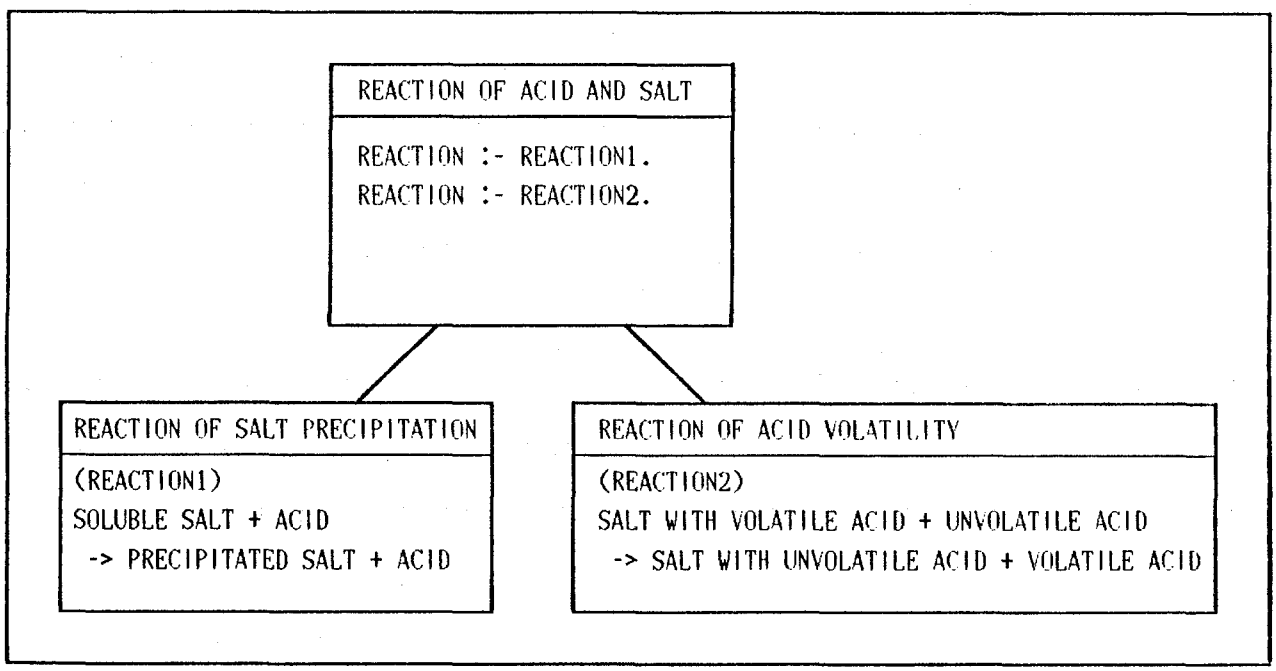
iii) if conditions of the equation (2) are satisfied, then equation (2) is applied.

This expertise is the meta-knowledge of the knowledge for chemical reactions of salt precipitation (equation (1)) and for chemical reactions of acid volatility (equation (2)).

The framework incorporates a hierarchical structure in model representation, so that the meta-knowledge is represented in the meta-level world. Figure 2 illustrates the hierarchy of the above example. A meta-level world named reaction of acid and salt represents the meta-knowledge of the basic knowledge that is represented as two worlds named reaction of salt precipitation and reaction of acid volatility. Figure 2-(b) shows the model of a student who cannot solve applied problems. The student's



(a) Expertise



(b) Student model

Fig.2 Schematic representation of knowledge for reaction of acid and salt

misconception of the incorrect use of knowledge is modelled as the bug in the meta-level world.

The hierarchical structure of the expertise contributes to forming the hierarchy of the expertise that is usually represented as chapters and sections. This hierarchical structure gives the formalism of the expertise a modularity. Moreover, restricting the range of student modelling within one world makes the modelling efficient. The hierarchical structure is implemented by giving each predicate an extra argument indicating the world which includes the predicate.

A teacher can organize course material according to its hierarchical structure, because no restriction on the identification of the hierarchical structure is imposed in the framework. The important point is that the framework has a facility of representing hierarchical material and that a teacher can use it very easily.

2.3 Student Modelling Based on Inductive Inference

2.3.1 Related Works

The major components of an ICAI system are following three modules [2]:

1) Expertise module; this module has the knowledge of the subject matter, and use it to generate problems, to evaluate student's replies and to reply to questions from the student.

2) Student model module; this module generates the student model which manages what each student does or does not understand, and how he obtains the solution.

3) Tutoring module; according to the student model, this

module chooses the next problem or the remedial comment, and instructs them to the expertise module. Also, this module controls overall system behavior.

Figure 3 illustrates the three components and the mutual relations. Since the system performance depends on how closely the student model approximates the student's status, the student model is considered to be the most important one. In particular, what kind of student's misconceptions the student model can manage determines the educational effect. Although a student's mistake is either a wrong solution or a missing solution, the student's misconceptions that are the sources of the mistake are divided into three categories:

1) A lack of knowledge; a student does not have a specific knowledge of course material. He, for example, does not know the capital of Brazil.

2) The incorrect knowledge; a student has the incorrect knowledge. For example, he believes that the capital of Brazil is Rio de Janeiro.

3) The incorrect use of knowledge; a student has the specific knowledge, nevertheless, he cannot use it or uses it incorrectly.

Most traditional ICAI systems have employed either the overlay or bug approach to modelling. In the overlay approach [7], the student's understanding is represented as a subset of the expertise. This modelling marks each knowledge according to whether a piece of evidence indicates the student has it or not. In this approach, only one category of misconceptions, a lack of knowledge, can be modelled, and the others, the incorrect

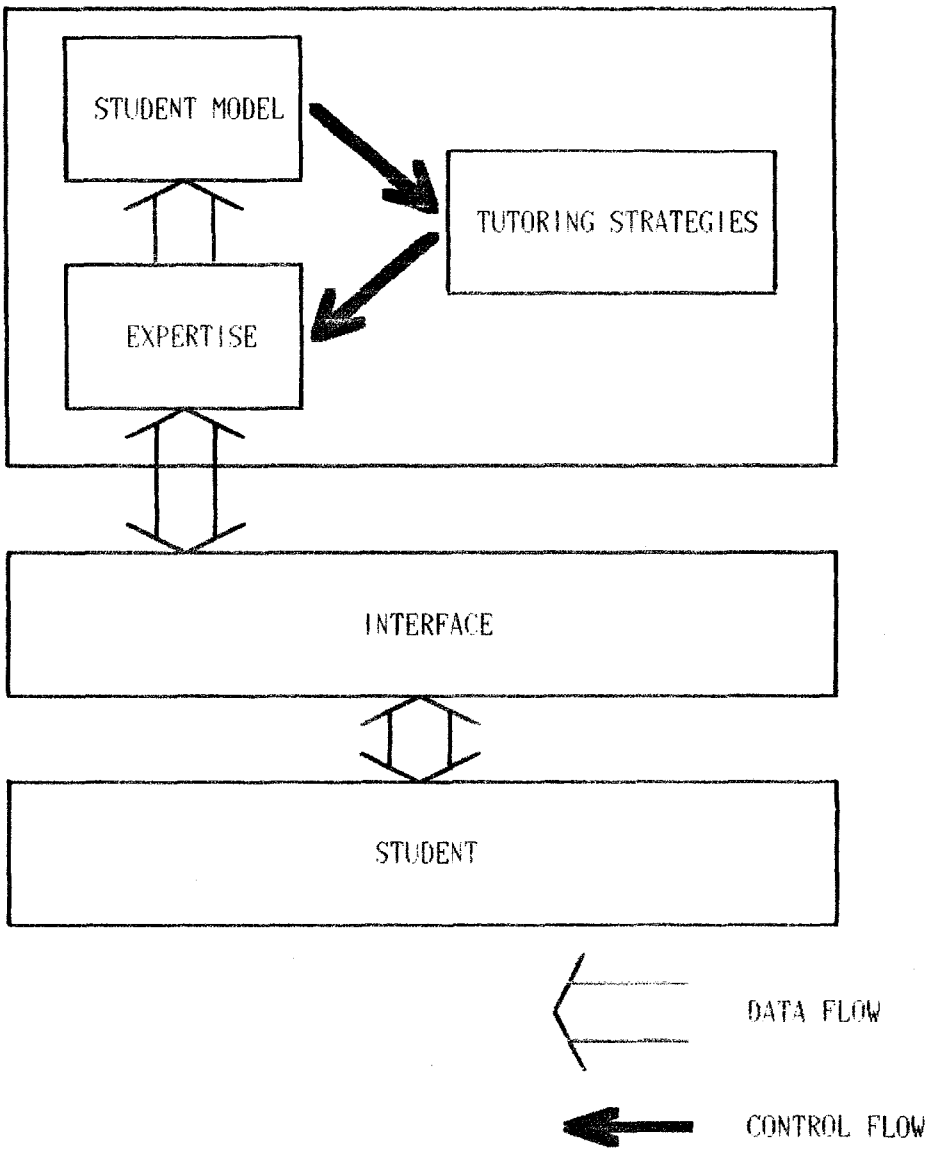


Fig.3 Block diagram of general ICAI systems

knowledge and the incorrect use of knowledge, cannot be done. In the bug approach [8], the student's knowledge is represented as a perturbation of or derivation from the expertise. The bug model has a good representability and can manage all of three categories of misconceptions. The modelling, however, depends so heavily on the expertise contents that it is difficult to make the modelling process independent of course material.

We develop a new framework for ICAI systems based on logic programming and inductive inference. The student model is constructed by a general inductive inference to model all of three categories of student's misconceptions.

2.3.2 Student Modelling by MIS

Since the model representation language of the framework is Prolog, the model inference algorithm MIS can be used as model-constructor. MIS induces the Prolog program which satisfies all of the given facts. In modelling of the framework, facts are obtained from problems and student's replies. The induction process is a repetition of making hypothesis and verifying it until obtaining a complete program. When MIS generates a clause, MIS gives the oracle [4], who knows everything on the target program, the following three kinds of queries concerned with the clause.

(1) What predicates are used in the body of the clause.

(2) The argument type of each predicate.

(3) The validity of each predicate.

The refinement operator [4] of MIS generates a new clause as a hypothesis using the oracle replies to these queries. In the

framework, the most general refinement operator is employed so as to synthesize all of the program that is indicated by the oracle. A hypothesis for the target program is set up, and then it is verified whether all of the given facts are satisfied by the hypothesized program. If there is an unsatisfied fact, PDS identifies the bug of the hypothesized program. It also gives the oracle a type of queries as follow:

(4) The validity of each goal that is executed.

A clause identified as a bug is removed, and a new hypothesis is set up. In student modelling the oracle is the student, because he is the one who knows everything about himself. Thus, the oracle queries are raised to the student.

2.3.3 Pedagogical Validity of Student Modelling

In this section, we describe the student modelling by MIS in detail, and show the pedagogical validity of the modelling.

First, we consider the range of the model that can be constructed. The range of the program MIS can synthesize is determined by a refinement operator. Since the most general refinement operator is employed in the framework, every clause indicated by the oracle can be synthesized. Nevertheless, the predicate that is not provided in the system is not useful as the oracle reply for the query (1) described in the previous section. For example, let us consider that the system provides only a predicate "precipitated" and that the student use the predicate "deposited" as a reply for the oracle query (1). In this case, MIS constructs the student model that use the predicate "deposited". However, the system does not identify "deposited"

with "precipitated". Thus, the clause that has the predicate "deposited" is identified as a bug i.e. a misconception. As this example, MIS can synthesize any clause that is indicated by the oracle, nevertheless, the clause with the predicate not provided in the system is not effective pedagogically. Therefore, MIS synthesizes the program that uses only predicates provided in advance. Though the oracle query (1) does not have to be raised in this synthesizing, the system raises it to the student for the modelling efficiency and the pedagogical effectiveness of the query as described below. This modelling approach has a same representability as the bug approach. Furthermore, this approach can synthesize any clause that can be obtained the combination of provided predicates, although the bug approach must provide all pieces of knowledge that is used in modelling.

Second, let us consider the efficiency of the student modelling. The efficiency of program synthesis by MIS depends on both the given facts and the refinement operator. The most effective fact is of the difference between the synthesizing program and the target program. In the student modelling, the target program is the human student understandings. Since the human student understandings are almost same as the expertise, the most effective fact is of the difference between the constructing student model and the expertise. Therefore, in order to model efficiently, we give the priority to the problem that is answered contrary when it is solved with the expertise and the student model. Though the framework has a general refinement operator, the student modelling can be performed more efficiently using the specific refinement operator. When the

actual ICAI system is built with the framework, it is useful to employ the domain-specific refinement operator.

The third problem is the termination of the modelling. Usually it is impossible to determine the exact termination of the student modelling. However, it is also impossible for a human teacher to do so. In the framework, it is regarded as the termination that the student model is not changed during a certain number of problems. The number is decided by a teacher.

The fourth problem is the pedagogical validity of the oracle queries. Since the query (2) described in the previous section must be needed to the transformation of the external form and the internal form, the reply to the query is provided in the system and the query need not be raised to the student. The query (4) is inherently same as the query (3). Thus, we consider the remaining two queries i.e. query (1) and query (3). We demonstrate that they are pedagogically valid using the chemical reaction equations teaching.

query (1): queried predicates represent the inference process or reason for the concept that is represented as the head predicate of the hypothesized clause. Hence, the query can be transformed to the question about the reason as Fig.4-(A). The query is raised when the predicates are needed, that is, a student uses a new concept and the system models it. Thus, the query is raised exactly after the student uses the new concept.

query (3): this query is about the subconcept of currently teaching concept. Thus, it is a simple basic problem as illustrated in Fig.4-(B).

Finally, we consider the change of student's status. In

(1) Pb(NH(3))(2) + H(2)CO(3) -> PbCO(3) + 2HNH(3)
I: no.
Tell the reason why you answered 'no'?
I: Pb(NH(3))(2) is not a salt. ----- (A)
I: ***

Is PbCO(3) precipitated?
I: yes. ----- (B)

Fig.4 Examples of queries set to student

order to synthesize the complete program by MIS, all of the given facts and the oracle replies must be consistent. This means that the student may not change his understandings during modelling. However, it often happens that the student finds his misconception and changes his replies. Therefore, the modelling of the framework makes the student indicate his change of understandings. When a student finds his misconceptions, he indicates his change of understandings to the system and replies the correct answer. Then, the framework removes all of the student's replies related to the indicated change and resumes the modelling with new replies.

2.3.4 Meta-Knowledge Modelling

The student model forms the hierarchy as well as the expertise. Since construction of the student model is done in bottom up way and the modelling in the worlds which are lower than the current world is already completed, the student model for the lower knowledge can be assumed to have almost the same structure as the expertise. Therefore, we can restrict the range of student modelling within the current world so as to improve the modelling efficiency. When the meta-level world is taught, if the student has a misconception categorized in incorrect use of knowledge then MIS synthesizes a Prolog program that has a bug corresponding to the misconception.

2.4 Extraction of Misconception by PDS

PDS can identify bugs in a Prolog program that behaves incorrectly. In extracting misconceptions, the program given to

PDS is the student model and the bug identified by PDS is the student's misconception. The oracle is the expertise, so the oracle queries are never raised to the student. The identified bug is either a missing clause or a false clause. The former and the latter correspond to a lack of knowledge and the incorrect knowledge, respectively. The bug in a program of the meta-level world corresponds to the incorrect use of knowledge. Hence, all of three categories of student's misconceptions described in section 2.3.1 can be extracted using PDS.

2.5 Expertise Module

The expertise module uses the expertise to generate problems, to evaluate student's replies and to reply to questions from the student. Figure 5 illustrates these functions. A part enclosed with bold lines is the domain-specific knowledge.

A student's reply is evaluated by executing the Prolog goal that represents the pair of the problem and the student's reply.

There are two types of questions from students, "yes/no" and "what/how". "Is AgCl a precipitated salt?" is an example of the "yes/no" question. This type of questions is transformed into a Prolog goal and the goal is executed. If the execution succeeds, the module answers "Yes, it is". If not, the module answers "No, it isn't". "What is the precipitation?" is an example of the "what/how" question. This type of questions is answered by displaying the contents of the file for answering.

The expertise of the framework is articulate, that is, the expertise can solve the problem in the same manner as the expert does [2]. Hence, the simple explanation by the expertise is

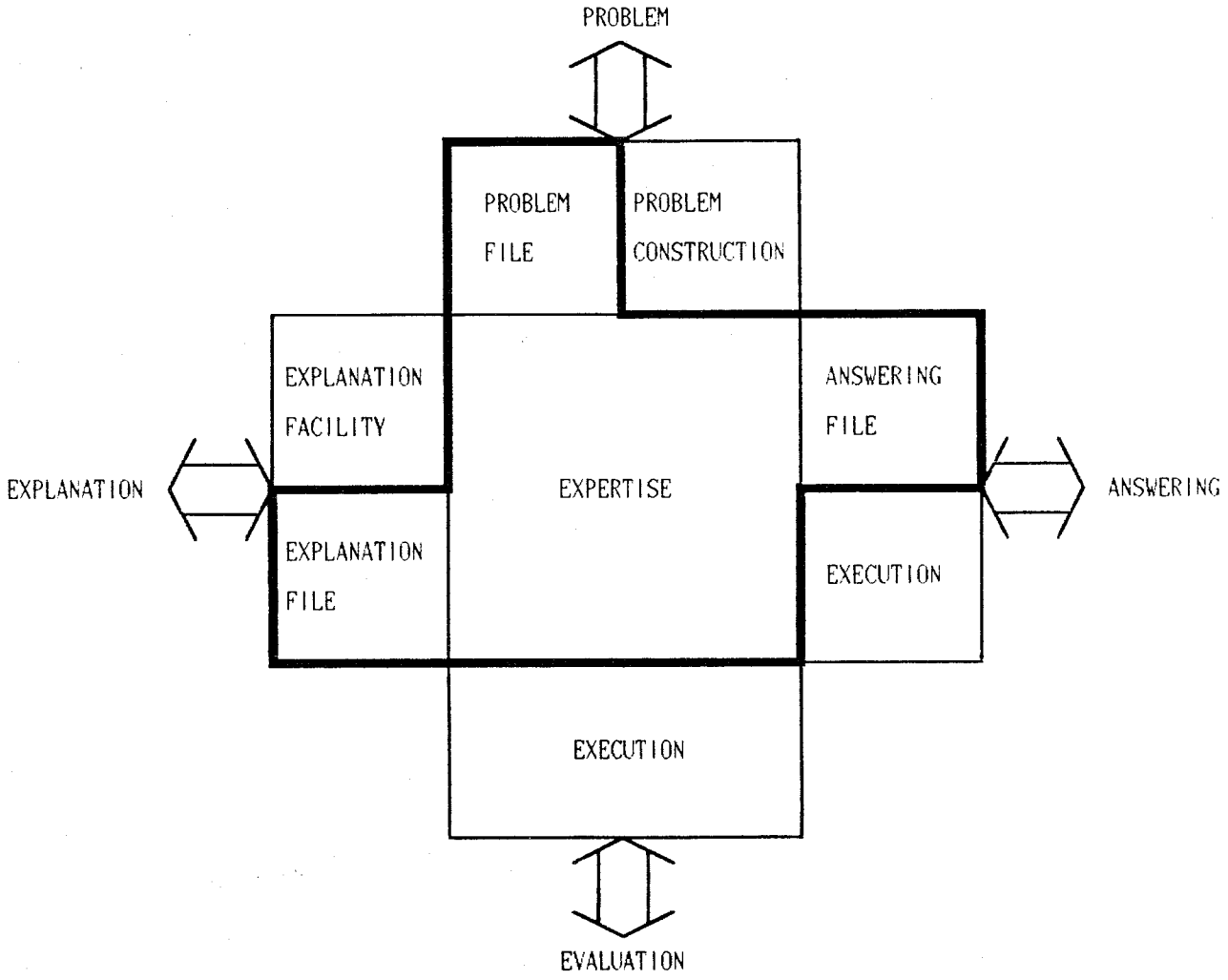


Fig.5 Illustration of expertise module

useful and understandable to students. In addition to this explanation facility, the expertise module can display the contents of the file for explanation. The file is provided for each Prolog clause of the expertise.

The problems for students are classified into two categories. First of them is the problem that is provided by the teacher as the file. Problems in this category are displayed as well as the explanation and answering. Second of them is the problem that is made by the problem-construction facility of the expertise module. The problem is constructed by the Prolog retrieval in the expertise.

2.6 Tutoring Module and Tutoring Strategies

The tutoring module controls the whole system based on two kinds of tutoring strategies. One is the modelling strategy, and the other is the retraining strategy. In this section, we explain the function of tutoring module along the ICAI system execution. First of all, the system gives explanations and problems to the student and constructs the student model using MIS. In this stage, the tutoring module instructs to the expertise module which explanation or problem is given to the student according to the modelling strategy. The modelling strategy is represented as a list. Each element of the list is an instruction to give an explanation, to set a problem and activate MIS, or to extract the misconception using PDS. When problems are set to a student, problems are solved by both the expertise and the student model, and then the problem which is differently answered is given to the student prior to other

problems. This is aimed at efficient modelling and at finding misconceptions by the student. When the student modelling reaches a certain extent and the model is not changed during a certain number of problems, the student modelling is regarded to terminate and then the student's misconceptions are extracted by PDS. The tutoring module receives the extracted misconceptions and instructs to the expertise module that the remedial problems and comments for retraining are given to the student. The way how to retrain the student is indicated by the retraining strategy. The retraining strategy controls the retraining method based on the type of the student's misconception and a piece of the expertise corresponding to the misconception. The retraining methods employed in the framework are as follows;

i) Correction: A piece of the expertise which corresponds to the student's misconception is displayed.

ii) Remedial exercises: Problems which will be answered incorrectly because of the misconception are given to the student in order to make him aware of the misconception by himself.

iii) Socratic tutoring [2].

2.7 Interface Module

The interface module mutually transforms an internal form and an external form that is suitable to students. Though the natural language interface is the most favourite one and has been studied by many researchers [2][9][10], many complicated problems still remains unsolved. In the framework, the interface module only performs the transformation between a Prolog form and a

simple sentence (natural language). For example,

precipitated(Ag,Cl) <=> AgCl is precipitated.

precipitated(Ag,Cl) <=> AgCl is deposited.

precipitated(Ag,Cl) <=> AgCl is settled.

The dictionary for these transformations is provided as Prolog clauses and it is changeable for each ICAI system.

CHAPTER 3

SPECIFIC SYSTEMS CONSTRUCTED WITH THE FRAMEWORK

We develop two specific ICAI systems constructed with the framework for the purpose of demonstrating the domain-independency of the framework. One of them is an ICAI system for programming in Prolog, and the other is an ICAI system for chemical reaction equations.

3.1 An ICAI System for Programming in Prolog

3.1.1 Basic Design

As described in chapter 1, educational applications of the computer are classified into two categories according to the role of the computer. They are the interactive teaching system and the environmental learning system. These two systems have the suitable course materials and the suitable teaching periods. The interactive teaching system is appropriate to the subject which can be taught by instruction. The environmental learning system suits the subject that must be learned by the student himself. In teaching of the course material, the game or simulation learning is used to let the student have an interest in the subject at the early stage of the learning period, and then the interactive teaching system instructs him the knowledge for the subject. In the final period, his understanding is made deeper using the environmental learning system. This is the one of the most effective teaching courses. In teaching of programming language, the same course is also effective. At first the student tries to execute sample programs in a textbook, and then he is taught the syntax and basic programming techniques of the

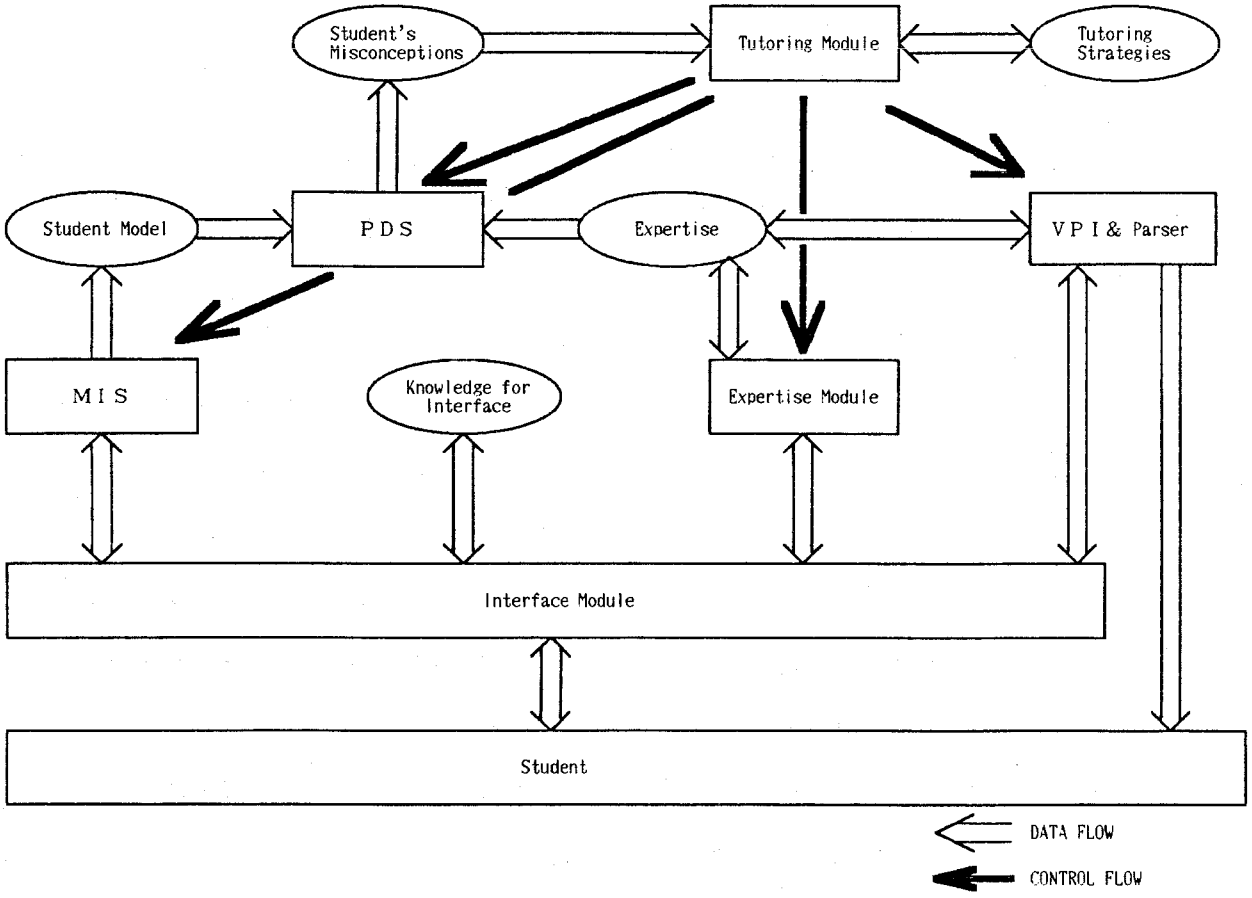


Fig.6 Block diagram of the ICAI system for programming in Prolog

language, and finally he makes many programs and learns the programming paradigm through programming. This section is concerned with an ICAI system for programming in Prolog [11] constructed on the basis of the above observations. It teaches the knowledge for programming in Prolog by instruction and provides the environment where the student can learn the programming using his knowledge. The interactive teaching (sub) system of the ICAI system is built on the framework for ICAI systems described in chapter 2 by embedding the knowledge for programming in Prolog in it. The environmental learning (sub) system has a visual Prolog interpreter called VPI and a parser. VPI interprets the program and displays the execution process in the form of a computation tree. The parser analyzes the program to detect and indicate its error if any. The major modules (boxes) and the knowledge (ellipses) of the ICAI system are shown in Fig.6.

3.1.2 Curriculum of Programming in Prolog

3.1.2.1 The Goal of Instruction

The ICAI system assumes that the target students are undergraduates (seniors) who have experiences of programming in procedural languages, e.g., FORTRAN, Pascal and C. The goal of tutoring is to enable them to understand and write Prolog programs. For this purpose the followings must be taught and learned.

1) Syntax and execution mechanism of Prolog
2) Functions of built-in predicates
3) Useful programming techniques

4) Logic programming paradigm

3.1.2.2 Curriculum of Programming in Prolog

The curriculum of the ICAI system is summarized in Table 1. Each item consists of more than ten explanations and problems. Each of items from 1) to 6) is taught by the interactive teaching system, and the item 7) is learned in the environmental learning system.

1) Program examples

Prolog is a programming language which can manage objects and relationships between them. The ICAI system shows sample programs and gives the students a feeling for programming in Prolog.

2) Syntax

The ICAI system teaches the Prolog syntax and data-structures. Many different Prolog systems are available recently, and each of them has the different syntax. The ICAI system teaches the syntax of MV-Prolog.

3) Unification

A variable is either instantiated or uninstantiated. A variable which is instantiated to an object never changes. This property, called single assignment, of the variable is one of the most important and distinguished properties of Prolog. Moreover, the matching between constants, the substitution of variables and the shared variables are taught in this item.

4) Interpreter

The behavior of a Prolog interpreter is taught in this item. The interpreter searches a unifiable clause in the depth first

Table.1 Curriculum of the ICAI system

Items	Contents
1) Program examples	sample programs
2) Syntax	syntax and data-structure
3) Unification	single assignment, matching, substitution, shared variables
4) Interpreter	depth first search, backtracking
5) Built-in predicates	functions and usage of built-in predicates
6) Programming techniques	list recursion, numerical recursion, difference list
7) Logic programming paradigm	relation, logic, non-determinacy

way and unifies the goal and the head of the clause. If a unifiable clause cannot be found, a backtracking occurs. These processes of the interpreter are referred to as the non-determinacy of Prolog. Moreover, the ICAI system uses VPI for displaying the interpretation process.

5) Built-in predicates

In this item, functions of and how to use built-in predicates are taught.

6) Programming techniques [12]

More than ten basic programming techniques are taught in this item, e.g., the list recursion, the numerical recursion and the cut-fail combination. The instruction is made through programming and error correcting practice.

7) Logic programming paradigm

This item is taught in the environmental learning system. The student makes many programs in the learning environment and learns the logic programming paradigm. Problems for programs are set by the ICAI system or prepared by the student himself.

3.1.3 System Implementation

3.1.3.1 Expertise for Programming in Prolog

In this section, we describe the expertise and the student modelling for each item mentioned in section 3.1.2.2.

1) Program examples

In teaching this item, the ICAI system only gives the explanations to the student and does not construct the student model from his behavior. Explanations are stored in the files and are given to the student one by one according to the tutoring

strategy.

2) Syntax

The expertise of the syntax is represented as a set of the Prolog clauses. The following is an example of a clause;

```
variable(i_model,syntax_atomic,12,X,Z) :-
        capital_letter(i_model,_,_,X,Y),
        character_list(i_model,_,_,Y,Z).
```

This clause means that the string represented by the fourth argument and the fifth argument as a difference list is the syntactic element represented by the predicate name. The knowledge is organized in a hierarchical structure in order to deal with the meta-knowledge and to obtain the modularity. The expertise for each item consists of from several to more than ten worlds. The second argument of the head indicates the world name in which the clause is included. The third argument of the head means ID number of the clause in the world. The first argument indicates that the clause is a piece of the expertise (ideal_model).

The student model is constructed by MIS which synthesizes a Prolog program from given facts. The facts are obtained from pairs of a problem and a student's reply. We describe how to obtain the facts from problem-reply pairs in the ICAI system. All of the problems in the ICAI system are represented in the following form:

```
(Q) Choose the correct variables.
        1. var  2. Var  3. _xyz  4. 345  5. _
```

The problem is prepared in the file and the fact for each choice is also prepared. For example, the fact

variable(s_model,syntax_atomic,_,[v,a,r],[])

is prepared for the element 1. var. The truth value is determined according to the student's replies. If he chooses an element as the correct one, the fact prepared for the element is determined as true. If not, the prepared fact is decided as false. A set of pairs of prepared facts and their truth values is sent to MIS.

The oracle queries raised by MIS in the student modelling takes one of the following forms:

* Tell the reason why var is a variable.
* Is var a string?

They are transformed from the oracle queries in Prolog representation to the questions in natural language by the interface module.

3) Unification

The expertise for unification of two elements is represented as follows using a substitution list.

i) Fetch the value of the first element out of the substitution list.

ii) Fetch the value of the second element out of the substitution list.

iii) case (fetched value of the first element,
fetched value of the second element)

iii.1) (var, var) shared variable

iii.2) (var, non-var) substitution

iii.3) (const, const) equality

iii.4) (compound term, compound term)
functor = functor

arity = arity

unification of each argument pair.

A substitution list is a list of pairs each of which is a pair of variable and its value. Both the variable and its value are represented as difference lists. The Prolog program representing the above knowledge is shown in Fig.10-a). The clauses whose head predicates are "unification" correspond to cases of the knowledge. (Figure 10 lists the first and second clauses.) Roles of the first, second, and third arguments of the clauses are the same as those of clauses for the item 2) syntax. The elements to be unified are represented as difference lists that are expressed by the pairs of fourth and fifth arguments and of sixth and seventh arguments. The rest arguments stand for substitution lists. The eighth argument stands for the current substitution list. The unification is performed with the substitution list represented as the eighth argument, and then the resulting substitution list is returned as the ninth argument. Examples of the problems and queries are shown in Fig.9 and Fig.11.

4) Interpreter

The expertise for the Prolog interpreter is constructed with four elements, that is, a sequence of goals, a sequence of clauses, a substitution list and the environment. A sequence of goals is the conjunction of Prolog goals which will have to be satisfied. A program is represented as a sequence of clauses. The order of the sequence is same as the order in which the programmer inputs the clauses. A substitution list is used for maintaining the values of variables in the program. The Prolog

interpreter realizes the non-determinacy by a depth first search for the unifiable clause and a backtracking mechanism. The status of the interpreter must be preserved for the backtracking. Preserved items are a sequence of goals, an unified clause and a substitution list. They are called the environment and preserved using a push down stack. Using the above four elements, the expertise for the Prolog interpreter, i.e., how the Prolog interpreter behaves, is summarized as follows:

i) Choose the first goal from a sequence of goals as the current goal.

ii) Choose a clause unifiable with the current goal from a sequence of clauses.
If there is not a unifiable clause, goto vi).

iii) Push down the environment into the stack.

iv) Create a new sequence of goals.

v) Goto i).

vi) Pop up the environment from the stack.

vii) Choose an alternative clause unifiable with the goal.
If there is not an alternative clause, goto vi).

viii) goto iii).

5) Built-in predicates

Functions of and how to use built-in predicates are taught in this item. Since the unified knowledge representation for functions of built-in predicates is not completed in the current ICAI system, the expertise represented in Prolog is not provided and the student model is not constructed. Functions are taught using explanations prepared in files and some easy exercises. Some usages of built-in predicates are taught in the next item 6)

programming techniques. The expertise for them is represented in Prolog. Nevertheless, the expertise for most of usages are not represented in Prolog as well as that for functions, and they are also taught using canned explanations and simple exercises. It requires further investigation to represent the expertise for this item in Prolog and to construct the student model.

6) Programming techniques

The list recursion, the numerical recursion, the cut-fail combination and so on are taught in this item. Here, we describe the expertise and the student modelling for the list recursion. The program using this technique has the following seven characteristics.

i) One of the arguments of the head predicate is a dotted pair of variables, i.e., [Variable1|Variable2].

ii) One of the goals of the body has the same predicate as the head.

iii) The arity of the goal is same as the arity of the head.

iv) The goal has an argument Variable2 at the same position as the argument of the characteristic i).

v) There is a fact that has the same predicate as the goal of the characteristic ii).

vi) The arity of the fact is same as that of the goal of the characteristic ii).

vii) The fact has an empty list, i.e., [], as an argument at the same position of the characteristic iv).

The expertise for this technique is expressed as the conjunction of Prolog goals corresponding to the above characteristics. The student model is constructed based on the

student's program using the technique. The student's program is analyzed to examine whether each characteristic is used. Since all of the characteristics can be detected by the syntactic analysis, the program is parsed using DCG [13] rules in the expertise. If a characteristic is detected in the program, a fact corresponding to the characteristic is sent to MIS. MIS constructs the student model as the conjunction of goals that are generalized expressions of the facts.

7) Logic programming paradigm

In this item the student makes programs in the environmental learning system. Problems for programming are prepared in files. And, the student can exercise the programming for the problem prepared by himself. Executions of programs are performed by VPI.

Before the execution of the student's program, the program is parsed based on the expertise for Prolog syntax and the syntax error is diagnosed if any. If a syntax error is detected in the program, the error is displayed and the explanation for it is given. In this case the program is not interpreted by VPI. The details of VPI are described in the next section.

3.1.3.2 VPI for Learning Environment

The ICAI system provides the student with an environment where he can learn the logic programming paradigm through programming in Prolog and executing the program on the visual Prolog interpreter named VPI. VPI is a new Prolog interpreter which displays an execution process of a program as the growing and reducing of a computation tree. Some other visual Prolog

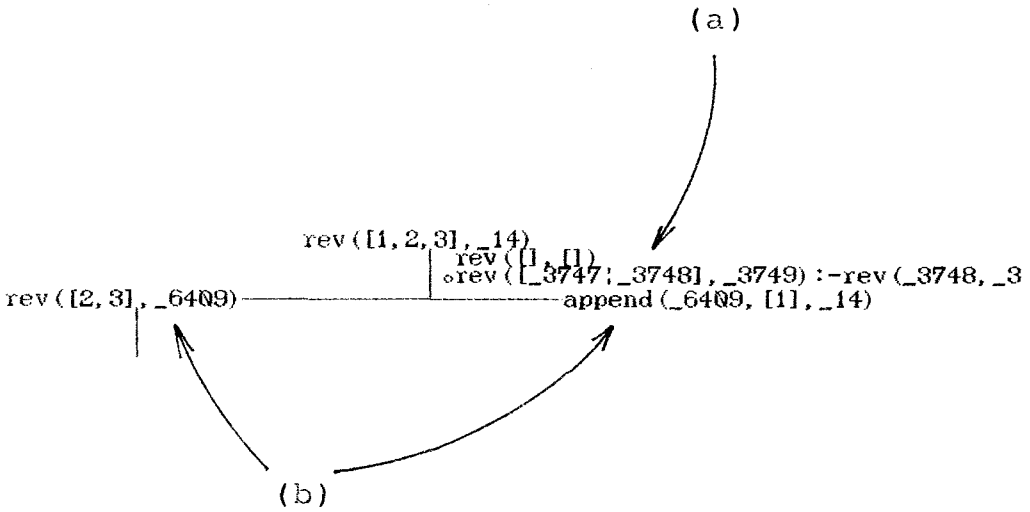


Fig.7 The portion of proof tree displayed by VPI

interpreters are proposed in [14] and [15].

The most complicated processes of a Prolog execution are a backtracking and an instantiation of a variable by unification. In particular, the backtracking over many goals at a time is the most complicated. And, the case where one of the shared variables is instantiated to a constant is rather difficult to understand. In this case, other shared variables are also instantiated to the constant. VPI has facilities for displaying these processes in a clearly understandable way.

A procedure call and a backtracking are visualized by a dynamically changing computation tree. When a goal is executed, the clauses which have the same head predicate as the goal are displayed under the goal, and then the unifiable clauses are marked (Fig.7(a)). New goals which are derived from the unified clause are displayed as children nodes of the computation tree (Fig.7(b)). If a backtracking occurs, the backtracked goals are eliminated, i.e., the computation tree is reduced. In the case of shallow-backtracking, that is, there is an alternative clause, the alternative clause is marked and then new goals are displayed as children nodes.

When a variable is instantiated, the display of the variable is changed to the instantiated value. If the variable is shared with other variables, the display of them are also changed to the instantiated value.

3.1.3.3 Tutoring Strategies

The tutoring module controls the whole of system behaviors as described in section 2.6. The modelling strategy is prepared

for each item.

```
[...,
 explanation(file,syntax_term1),
 problem(file,syntax_term1),
 pds(file,syntax_term1),
 retraining,
 ...]
```

The above is a part of the modelling strategy for the item 2) syntax. "explanation(file,syntax_term1)" instructs the expertise module to display an explanation in the file named syntax_term1.epl. "problem(file,syntax_term1)" also instructs it to display a problem in the file named syntax_term1.prb and sends MIS a set of facts representing the student's replies. The tutoring module has no relation to the queries raised by MIS. They are presented to the student directly. "pds(file,syntax_term1)" indicates the detection of the student's misconceptions. The goals prepared in the file named syntax_term1.pds are executed with the expertise and the student model. If the solution of a goal by the expertise is different from that of the goal by the student model, then the goal is sent to PDS and bugs of the student model are detected. "retraining" indicates that the tutoring module instructs the expertise module to give the student remedial comments and problems for his misconception.

Moreover, the tutoring module activates VPI and the parser in teaching item 7) logic programming paradigm.

3.1.3.4 Interface Module

The interface module transforms an internal form into external form and vice versa as described in section 2.7. Moreover, The interface module enables a friendly interaction between the student and the ICAI system using extended facilities of MV-Prolog such as menus and windows. Table 2 summarizes commands and their functions. A student can input any command at any time when the ICAI system is in input mode. Commands are displayed at the bottom of the screen as a menu and selected by the cursor. Explanations, problems, hints and so on are displayed in the different windows. Figure 8 shows the layout of the windows.

Table.2 Commands and their functions

Commands	Functions
next	advance to the next problem or explanation
QA	typewrite an answer or a question
skip	skip to the next world
back	back to the world before
bend	back to the previous world
menu	select the next world out of the menu
hint	display a hint for the problem
detail	display a more detailed explanation
ans	display an answer of the problem
help	help-message for commands
prolog	execution of a Prolog goal
end	terminate the tutoring system

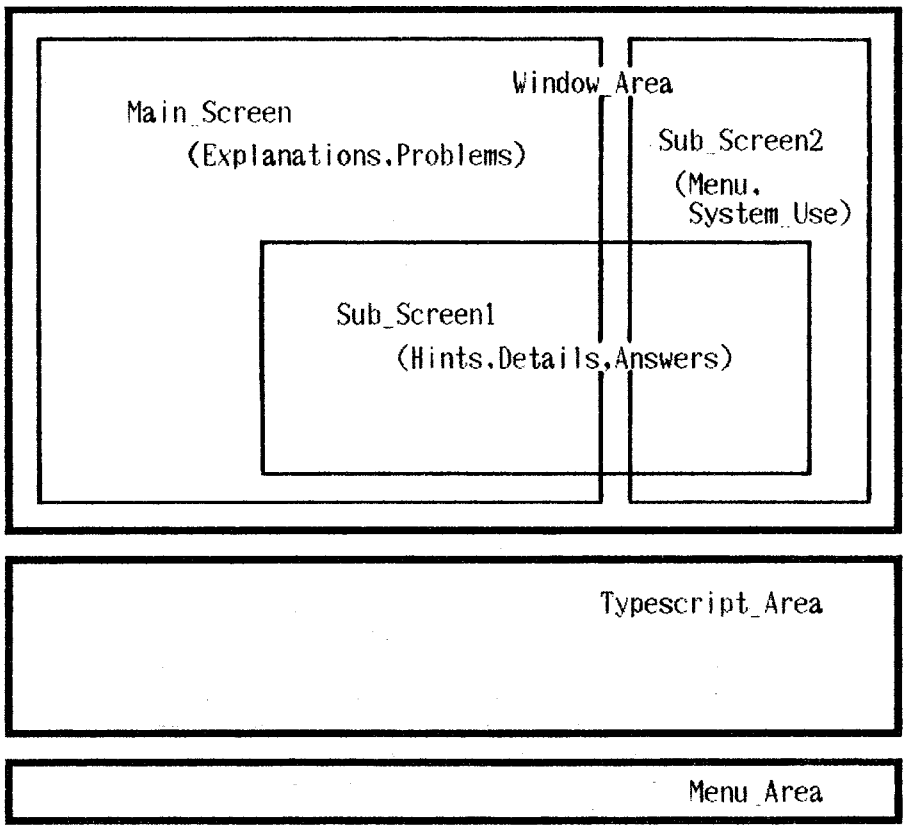


Fig.8 Screen design

3.1.4 Experiments

In Fig.9, the student requests hints on a problem of the item 3) unification. The problem is displayed in the "ICAI" window placed at the upper left hand of the screen, and hints are also displayed in the "HINT" window placed at the middle. The hint in Fig.9 are concerned with an explanation for the unification process. When the student answers No.1 and 2 as unifiable terms (the correct answer is No.1 and 3), the student model is constructed as shown in Fig.10-b) after raising the queries for reasons to the student (Fig.11). Considering the student's replies for these queries, it can be thought that he does not know that if the fetched value of a variable is another variable then the latter variable's value must be fetched, because he replies that "The value of Y is X." and that "The value of X is 1." Comparing the constructed student model shown in Fig.10-b) with the expertise shown in Fig.10-a), this misconception is modelled as a lack of the goal "fetch_value" (indicated by (A) in Fig.10).

Figure 12 shows a screen when the student's program is analyzed and its error is detected in teaching the list recursion of the item 6) programming techniques. The ICAI system analyzes the program based on the syntax rules for the list recursion and then detects and indicates that an empty list ([]) is replaced with a variable (X) in the head of the second clause.

Figure 13 shows an example of VPI's output. It is the terminal screen when the goal rev([1,2,3],X) is executed and a solution is obtained.

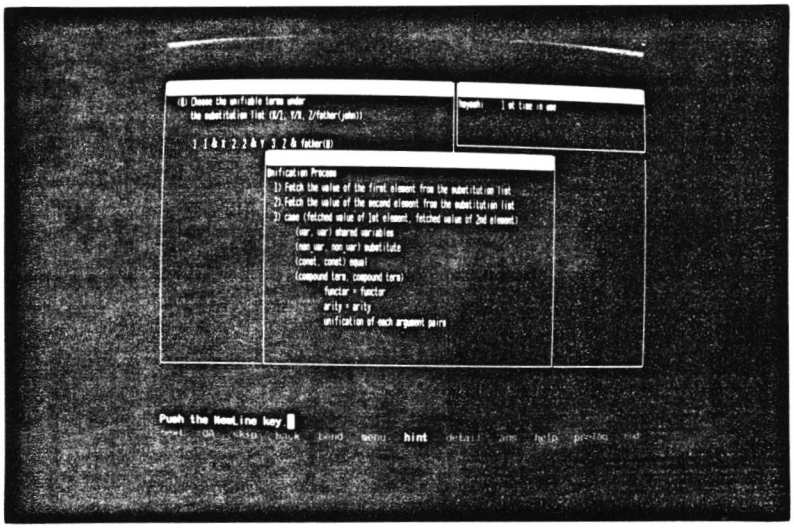


Fig.9 Example of a training session on unification

```
unification(i_model,unify,1,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(i_model,_,_,XV1,XV2),
        variable(i_model,_,_,YV1,YV2),
        shared(i_model,_,_,XV1,XV2,YV1,YV2,S,S0).
unification(i_model,unify,2,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(i_model,_,_,XV1,XV2),
        non_variable(i_model,_,_,YV1,YV2),
        substitute(i_model,_,_,XV1,XV2,YV1,YV2,S,S0).

fetch_value(i_model,unify,11,X1,X2,_,X1,X2) :-
        non_variable(i_model,_,_,X1,X2).
fetch_value(i_model,unify,12,X1,X2,S,X1,X2) :-
        variable(i_model,_,_,X1,X2),
        not(member(((X1,X2) $ _),S)).
fetch_value(i_model,unify,13,X1,X2,S,Z1,Z2) :-
        variable(i_model,_,_,X1,X2),
        member(((X1,X2) $ (Y1,Y2)),S),
        fetch_value(i_model,_,_,Y1,Y2,S,Z1,Z2).          ----- (A)
```

a) Expertise

```
unification(s_model,unify,1,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(s_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(s_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(s_model,_,_,XV1,XV2),
        variable(s_model,_,_,YV1,YV2),
        shared(s_model,_,_,XV1,XV2,YV1,YV2,S,S0).
unification(s_model,unify,2,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(s_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(s_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(s_model,_,_,XV1,XV2),
        non_variable(s_model,_,_,YV1,YV2),
        substitute(s_model,_,_,XV1,XV2,YV1,YV2,S,S0).

fetch_value(s_model,unify,11,X1,X2,_,X1,X2) :-
        non_variable(s_model,_,_,X1,X2).
fetch_value(s_model,unify,12,X1,X2,S,X1,X2) :-
        variable(s_model,_,_,X1,X2),
        not(member(((X1,X2) $ _),S)).
fetch_value(s_model,unify,13,X1,X2,S,Z1,Z2) :-
        variable(s_model,_,_,X1,X2),
        member(((X1,X2) $ (Y1,Y2)),S).                   ----- (A)
```

b) Student model

Fig.10 Knowledge for unification in Prolog

```
Tell the reason why 1 and X are unifiable?
>>> The fetched value of 1 is 1.
>>> The fetched value of X is 1.
>>> 1 is identical with 1.
>>> ***

Tell the reason why 2 and Y are unifiable?
>>> The fetched value of 2 is 2.
>>> The fetched value of Y is X.
>>> X is substituted with 2.
>>> ***
```

Fig.11 Query examples by MIS

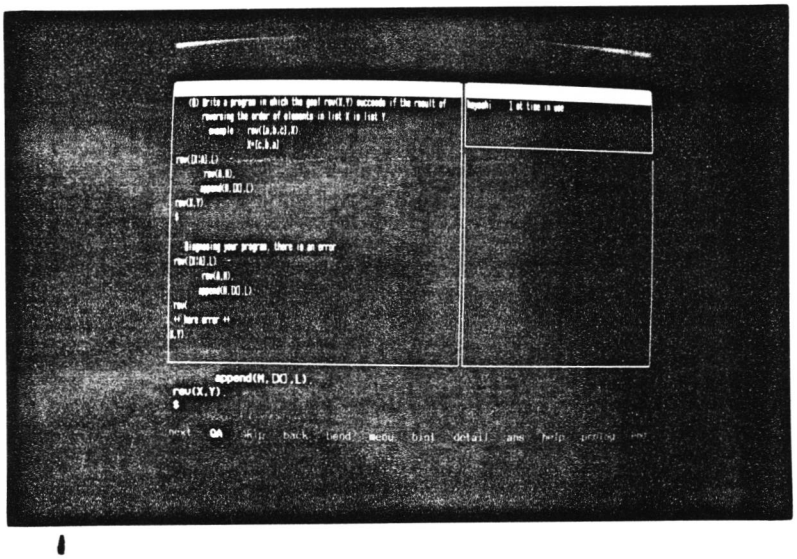


Fig.12 Example of a training session on programming techniques

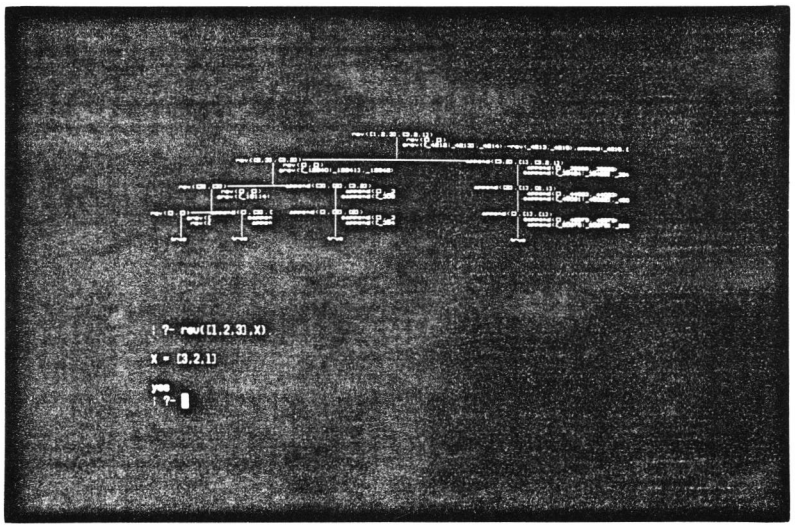


Fig.13 Example of VPI's output

3.2 An ICAI System for Chemical Reaction Equations

We also build an ICAI system for chemical reaction equations on the framework with the domain-specific knowledge. The ICAI system teaches a chemical reaction of an acid and a salt. The expertise of the ICAI system is formalized with eight worlds as illustrated in Fig.14. The most top level world named reaction of acid and salt consists of the meta-knowledge for the two lower level worlds as described in section 2.2. The rest five worlds are constructed based on course material hierarchy.

Figure 15 is an example of teaching the meta-knowledge in the world named reaction of acid and salt. A part of the expertise is shown in Fig.16. The sequence of problems from (1) to (6) is for the modelling. After replying to the problem (6), the constructed model that is a corresponding part of Fig.16 is shown in Fig.17. The first, second and third arguments of a predicate in the expertise and the student model represent, respectively, the flag for distinction of the expertise and the student model, the name of the world in which the predicate is included, and the index to the file for explanation. Predicates "rule", "rule1" and "rule2" represent the knowledge in the top three level worlds. The forth, fifth and sixth arguments of them characterized as "X", "Y" and "Z" mean the following chemical reaction equation.

$$XY + H_xZ \rightarrow XZ + H_xY$$

In the above equation, "XY" is the reactant salt, "H_xZ" is the reactant acid, "XZ" is the product salt and "H_xY" is the product acid. The student does not consider the use of inverted reaction as described in section 2.2, and then he makes a wrong answer to

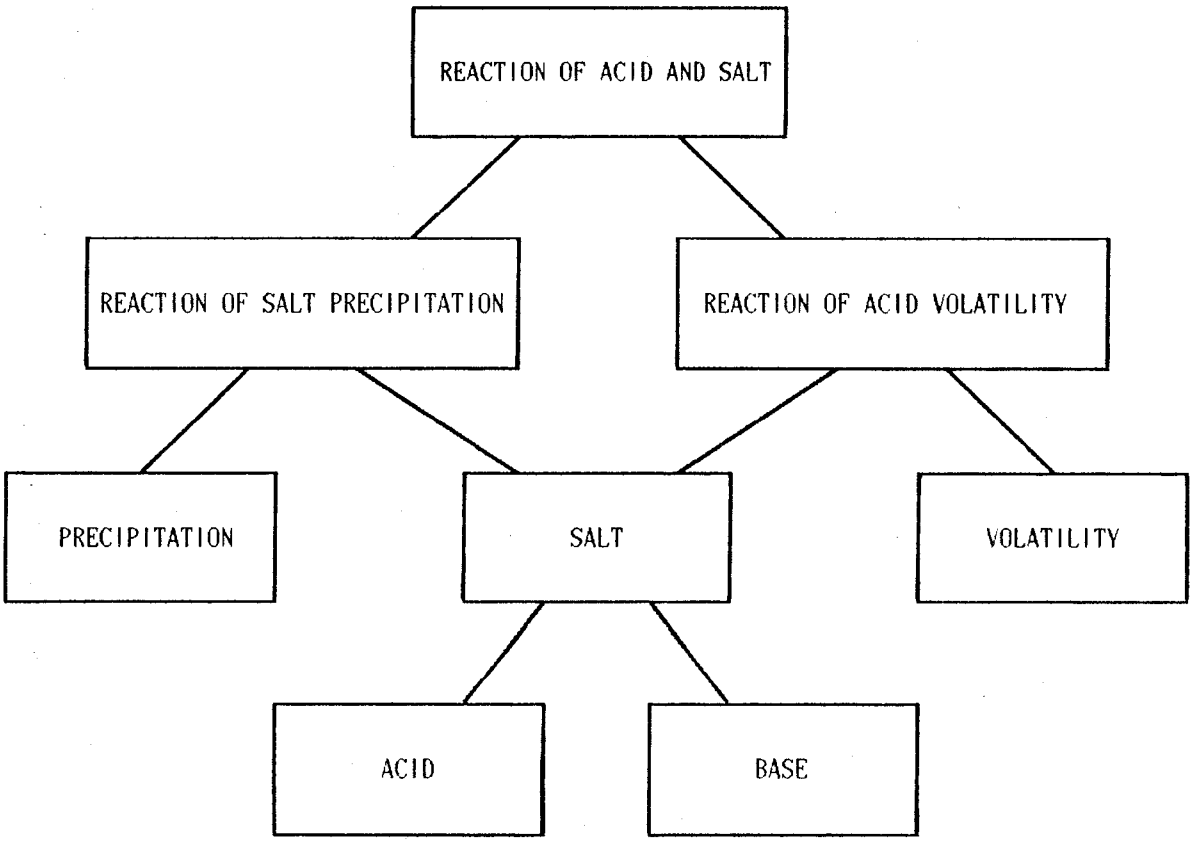


Fig.14 Diagram of knowledge hierarchy

```
Are the following chemical reaction equations correct or not?
If correct, answer 'yes.' If not, answer 'no.'
(1) K(2)CO(3) + H(2)SO(4) -> K(2)SO(4) + H(2)CO(3)
I: yes.
Tell the reason why you answered 'yes'?
I: K(2)SO(4) is not precipitated.
I: H(2)CO(3) is a volatile acid.
I: ***
(2) Na(2)SO(4) + 2HCl -> 2NaCl + H(2)SO(4)
I: no.
(3) Pb(NO(3))(2) + 2HCl -> PbCl(2) + 2HNO(3)
I: yes.
(4) BaCO(3) + 2HCl -> BaCl(2) + H(2)CO(3)
I: no.
(5) Ag(2)SO(4) + 2HCl -> 2AgCl + H(2)SO(4)
I: yes.
(6) 2AgCl + H(2)SO(4) -> Ag(2)SO(4) + 2HCl
I: yes.
(7) PbCl(2) + H(2)SO(4) -> PbSO(4) + 2HCl
I: yes.
(8) CuCl(2) + H(2)SO(4) -> CuSO(4) + 2HCl
I: '**'.
I: no.
Tell the reason why you answered 'no'?
I: CuCl(2) is precipitated.
I: Inverted reaction takes place.
I: ***
(9) Ag(2)CO(3) + H(2)SO(4) -> Ag(2)SO(4) + H(2)CO(3)
I: no.
```

Fig.15 Example of a training trace

```
rule(i_model,'ROAS',1,X,Y,Z) :-
        rule1(i_model,'ROSP',_,X,Y,Z).
rule(i_model,'ROAS',2,X,Y,Z) :-
        not_rule1(i_model,'ROAS',_,X,Z,Y),
        rule2(i_model,'ROAV',_,X,Y,Z).
not_rule1(i_model,'ROAS',3,X,Y,Z) :-
        not(rule1(i_model,'ROSP',_,X,Y,Z)).

rule1(i_model,'ROSP',1,X,Y,Z) :-
        salt(i_model,'SALT',_,X,Y),
        precipitated(i_model,'PRECIPITATION',_,X,Z).

rule2(i_model,'ROAV',1,X,Y,Z) :-
        salt(i_model,'SALT',_,X,Y),
        volatiled(i_model,'VOLATILITY',_,Y),
        unvolatiled(i_model,'VOLATILITY',_,Z).
```

Fig.16 Contents of expertise

```
rule(s_model,'ROAS',1,X,Y,Z) :-
        rule1(s_model,'ROSP',_,X,Y,Z).
rule(s_model,'ROAS',2,X,Y,Z) :-
        rule2(s_model,'ROAV',_,X,Y,Z).

rule1(s_model,'ROSP',1,X,Y,Z) :-
        salt(s_model,'SALT',_,X,Y),
        precipitated(s_model,'PRECIPITATION',_,X,Z).

rule2(s_model,'ROAV',1,X,Y,Z) :-
        salt(s_model,'SALT',_,X,Y),
        volatiled(s_model,'VOLATILITY',_,Y),
        unvolatiled(s_model,'VOLATILITY',_,Z).
```

Fig.17 Constructed student model

the problem (6). The ICAI system constructs the student model which has an incorrect meta-knowledge as shown in Fig.17. Based on this misconception, the ICAI system gives the remedial problems from (7) to (9) that must use the correct meta-knowledge concerned to the inverted reaction. The student finds his misconception, instructs the change of his understanding to the ICAI system (replying '**' to problem (8)), and then he makes the correct answer.

CHAPTER 4

CONCLUSIONS

The problems studied in this thesis are concerned with educational applications of the computer technology, and are, in particular, centered mainly on ICAI systems based on logic programming.

The first problem studied in the thesis is to develop the framework for ICAI systems based on logic programming. On the basis of the observation on the disadvantages of the modelling approaches used in the traditional ICAI systems, we have proposed a new framework for ICAI systems with a powerful modelling scheme. The framework is summarized as follows:

1) Model representation in logic program.

2) Student modelling based on inductive inference.

3) Domain-independence.

The framework will be extended as the general tool for ICAI system building. The problems for this extension are generalizing the tutoring strategies, fulfilling interface facilities and developing the utilities for the teachers.

Further, we described two ICAI systems, for programming in Prolog and for chemical reaction equations, constructed with the framework. The development of them demonstrates the domain-independency of the framework. The ICAI system for programming in Prolog has both the interactive teaching system and the environmental learning system. Both of them are quite important to learn a programming language. The interactive teaching system is constructed with the framework for ICAI systems. The environmental learning system uses a visual Prolog

interpreter VPI. The friendly interface is implemented using windows, menus and graphics. The ICAI system is currently used for educating the undergraduates (seniors) of our laboratories. The evaluation of the system performance through the practical use still remains as the future work.

APPENDIX

Expertise for Programming in Prolog (Listings)

This appendix contains listings of the expertise for programming in Prolog described in section 3.1.3.1.

```
/*************************************************/
/*    Expertise for syntax_primitive             */
/*************************************************/

small_letter(i_model,syntax_primitive,1,[X|Xs],Xs) :-
        member(X,
                [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
capital_letter(i_model,syntax_primitive,2,[X|Xs],Xs) :-
        member(X,
                ['A','B','C','D','E','F','G','H','I','J','K','L','M',
                 'N','O','P','Q','R','S','T','U','V','W','X','Y','Z']).
numeral(i_model,syntax_primitive,3,[X|Xs],Xs) :-
        member(X,[0,1,2,3,4,5,6,7,8,9]).
builtin_predicate(i_model,syntax_primitive,4,[X|Xs],Xs) :-
        builtin(X).
symbol(i_model,syntax_primitive,5,[X|Xs],Xs) :-
        member(X,
                ['+','-','*','/','¥','^','<','>','=','`','~',':','.',
                 '?','@','#','$','&']).
number(i_model,syntax_primitive,6,[X|Xs],Xs) :-
        integer(X).
underline(i_model,syntax_primitive,7,[X|Xs],Xs) :-
        member(X,['_']).
single_quote(i_model,syntax_primitive,8,[X|Xs],Xs) :-
        member(X,['''']).

/*************************************************/
/*    Expertise for syntax_character             */
/*************************************************/

symbol_list(i_model,syntax_character,11,X,Y) :-
        symbol(i_model,_,_,X,Y).
symbol_list(i_model,syntax_character,12,X,Z) :-
        symbol(i_model,_,_,X,Y),
        symbol_list(i_model,_,_,Y,Z).
small_character(i_model,syntax_character,21,X,Y) :-
        small_letter(i_model,_,_,X,Y).
small_character(i_model,syntax_character,22,X,Y) :-
        numeral(i_model,_,_,X,Y).
small_character(i_model,syntax_character,23,X,Y) :-
        underline(i_model,_,_,X,Y).
small_character_list(i_model,syntax_character,31,X,Y) :-
        small_character(i_model,_,_,X,Y).
small_character_list(i_model,syntax_character,32,X,Z) :-
        small_character(i_model,_,_,X,Y),
        small_character_list(i_model,_,_,Y,Z).
character(i_model,syntax_character,41,X,Y) :-
        small_letter(i_model,_,_,X,Y).
character(i_model,syntax_character,42,X,Y) :-
        capital_letter(i_model,_,_,X,Y).
character(i_model,syntax_character,43,X,Y) :-
        numeral(i_model,_,_,X,Y).
```

```
character(i_model,syntax_character,44,X,Y) :-
        underline(i_model,_,_,X,Y).
character_list(i_model,syntax_character,51,X,Y) :-
        character(i_model,_,_,X,Y).
character_list(i_model,syntax_character,52,X,Z) :-
        character(i_model,_,_,X,Y),
        character_list(i_model,_,_,Y,Z).
character_symbol(i_model,syntax_character,61,X,Y) :-
        character(i_model,_,_,X,Y).
character_symbol(i_model,syntax_character,62,X,Y) :-
        symbol(i_model,_,_,X,Y).
character_symbol_list(i_model,syntax_character,71,X,Y) :-
        character_symbol(i_model,_,_,X,Y).
character_symbol_list(i_model,syntax_character,72,X,Z) :-
        character_symbol(i_model,_,_,X,Y),
        character_symbol_list(i_model,_,_,Y,Z).

/*************************************************/
/*    Expertise for syntax_atomic                */
/*************************************************/

variable(i_model,syntax_atomic,11,X,Y) :-
        capital_letter(i_model,_,_,X,Y).
variable(i_model,syntax_atomic,12,X,Z) :-
        capital_letter(i_model,_,_,X,Y),
        character_list(i_model,_,_,Y,Z).
variable(i_model,syntax_atomic,13,X,Z) :-
        underline(i_model,_,_,X,Y),
        character_list(i_model,_,_,Y,Z).
variable(i_model,syntax_atomic,14,X,Y) :-
        underline(i_model,_,_,X,Y).
atom(i_model,syntax_atomic,21,X,Y) :-
        small_letter(i_model,_,_,X,Y).
atom(i_model,syntax_atomic,22,X,Z) :-
        small_letter(i_model,_,_,X,Y),
        character_list(i_model,_,_,Y,Z).
atom(i_model,syntax_atomic,23,X,Y) :-
        symbol_list(i_model,_,_,X,Y).
atom(i_model,syntax_atomic,24,X,W) :-
        single_quote(i_model,_,_,X,Y),
        character_symbol_list(i_model,_,_,Y,Z),
        single_quote(i_model,_,_,Z,W).
constant(i_model,syntax_atomic,31,X,Y) :-
        atom(i_model,_,_,X,Y).
constant(i_model,syntax_atomic,32,X,Y) :-
        number(i_model,_,_,X,Y).

/*************************************************/
/*    Expertise for syntax_term                  */
/*************************************************/

functor(i_model,syntax_term,1,X,Y) :-
        atom(i_model,_,_,X,Y).
compound_term(i_model,syntax_term,2,X,Z) :-
        functor(i_model,_,_,X,['('|Y]),
        argument_list(i_model,_,_,Y,[')'|Z]).
list(i_model,syntax_term,3,[X|Xs],Xs) :-
        member(i_model,_,_,X,[[]]).
list(i_model,syntax_term,4,['['|X],Y) :-
        argument_list(i_model,_,_,X,[']'|Y]).
```

```
list(i_model,syntax_term,5,['['|X],Z) :-
        argument_list(i_model,_,_,X,['|'|Y]),
        argument(i_model,_,_,Y,[']'|Z]).
argument(i_model,syntax_term,6,X,Y) :-
        constant(i_model,_,_,X,Y).
argument(i_model,syntax_term,7,X,Y) :-
        variable(i_model,_,_,X,Y).
argument(i_model,syntax_term,8,X,Y) :-
        list(i_model,_,_,X,Y).
argument(i_model,syntax_term,9,X,Y) :-
        compound_term(i_model,_,_,X,Y).
argument_list(i_model,syntax_term,10,X,Y) :-
        argument(i_model,_,_,X,Y).
argument_list(i_model,syntax_term,11,X,Z) :-
        argument(i_model,_,_,X,[','|Y]),
        argument_list(i_model,_,_,Y,Z).
term(i_model,syntax_term,12,X,Y) :-
        constant(i_model,_,_,X,Y).
term(i_model,syntax_term,13,X,Y) :-
        variable(i_model,_,_,X,Y).
term(i_model,syntax_term,14,X,Y) :-
        list(i_model,_,_,X,Y).
term(i_model,syntax_term,15,X,Y) :-
        builtin_predicate(i_model,_,_,X,Y).
term(i_model,syntax_term,16,X,Y) :-
        compound_term(i_model,_,_,X,Y).

/*************************************************/
/*    Expertise for syntax_program               */
/*************************************************/

predicate(i_model,syntax_program,1,X,Y) :-
        compound_term(i_model,_,_,X,Y).
predicate(i_model,syntax_program,2,X,Y) :-
        functor(i_model,_,_,X,Y).
predicate(i_model,syntax_program,3,X,Y) :-
        builtin_predicate(i_model,_,_,X,Y).
procedure(i_model,syntax_program,4,X,Y) :-
        predicate(i_model,_,_,X,Y).
procedure(i_model,syntax_program,5,X,Z) :-
        predicate(i_model,_,_,X,[','|Y]),
        procedure(i_model,_,_,Y,Z).
command(i_model,syntax_program,6,[":-"|X],Y) :-
        procedure(i_model,_,_,X,['.'|Y]).
rule(i_model,syntax_program,7,X,Z) :-
        predicate(i_model,_,_,X,[":-"|Y]),
        procedure(i_model,_,_,Y,['.'|Z]).
fact(i_model,syntax_program,8,X,Y) :-
        predicate(i_model,_,_,X,['.'|Y]).
clause(i_model,syntax_program,9,X,Y) :-
        rule(i_model,_,_,X,Y).
clause(i_model,syntax_program,10,X,Y) :-
        fact(i_model,_,_,X,Y).


/*************************************************/
/*    Expertise for unification                  */
/*************************************************/

unification(i_model,unify,1,X1,X2,Y1,Y2,S,S0) :-
```

```
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(i_model,_,_,XV1,XV2),
        variable(i_model,_,_,YV1,YV2),
        shared(i_model,_,_,XV1,XV2,YV1,YV2,S,S0).
unification(i_model,unify,2,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        variable(i_model,_,_,XV1,XV2),
        non_variable(i_model,_,_,YV1,YV2),
        substitute(i_model,_,_,XV1,XV2,YV1,YV2,S,S0).
unification(i_model,unify,3,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        constant(i_model,_,_,XV1,XV2),
        constant(i_model,_,_,YV1,YV2),
        XV1 = YV1,
        XV2 = YV2.
unification(i_model,unify,4,X1,X2,Y1,Y2,S,S0) :-
        fetch_value(i_model,_,_,X1,X2,S,XV1,XV2),
        fetch_value(i_model,_,_,Y1,Y2,S,YV1,YV2),
        compound_term(i_model,_,_,XV1,XV2),
        compound_term(i_model,_,_,YV1,YV2),
        functor(i_model,_,_,XV1,['('|XV0]),
        functor(i_model,_,_,YV1,['('|YV0]),
        argument_list(i_model,_,_,XV0,[')'|XV2]),
        argument_list(i_model,_,_,YV0,[')'|YV2]),
        unification(i_model,_,_,XV1,['('|XV0],YV1,['('|YV0],S,S1),
        arg_list_unification(i_model,_,_,XV0,[')'|XV2],YV0,[')'|YV2],S1,S0).

fetch_value(i_model,unify,11,X1,X2,_,X1,X2) :-
        non_variable(i_model,_,_,X1,X2).
fetch_value(i_model,unify,12,X1,X2,S,X1,X2) :-
        variable(i_model,_,_,X1,X2),
        not(member(((X1,X2) $ _),S)).
fetch_value(i_model,unify,13,X1,X2,S,Z1,Z2) :-
        variable(i_model,_,_,X1,X2),
        member(((X1,X2) $ (Y1,Y2)),S),
        fetch_value(i_model,_,_,Y1,Y2,S,Z1,Z2).

shared(i_model,unify,21,X1,X2,Y1,Y2,S,S0) :-
        append([((X1,X2) $ (Y1,Y2))],S,S0).

substitute(i_model,unify,31,X1,X2,Y1,Y2,S,S0) :-
        append([((X1,X2) $ (Y1,Y2))],S,S0).

arg_list_unification(i_model,unify,41,X1,X2,Y1,Y2,S,S0) :-
        argument(i_model,_,_,X1,X2),
        argument(i_model,_,_,Y1,Y2),
        unification(i_model,_,_,X1,X2,Y1,Y2,S,S0).
arg_list_unification(i_model,unify,42,X1,X2,Y1,Y2,S,S0) :-
        argument(i_model,_,_,X1,[','|X0]),
        argument_list(i_model,_,_,X0,X2),
        argument(i_model,_,_,Y1,[','|Y0]),
        argument_list(i_model,_,_,Y0,Y2),
        unification(i_model,_,_,X1,[','|X0],Y1,[','|Y0],S,S1),
        arg_list_unification(i_model,_,_,X0,X2,Y0,Y2,S1,S0).


/**************************************************/
```

```
/*    Expertise for interpreter                    */
/*************************************************/

interpreter(i_model,interpreter,1,[Goals|Prev_goals],Clauses,S,S0) :-
        search_goal(i_model,_,_,Goal,Goals),
        search_clause(i_model,_,_,Goal,[Head|Body],Clauses,S,S1),
        environment(i_model,_,_,
                [Goals|Prev_goals],[Head|Body],S,New_prev_goals),
        new_goals(i_model,_,_,Goals,Body,New_goals),
        interpreter(i_model,_,_,
                [New_goals|New_prev_goals],Clauses,S1,S0).
interpreter(i_model,interpreter,2,[Goals|Prev_goals],Clauses,S,S0) :-
        search_goal(i_model,_,_,Goal,Goals),
        not(search_clause(i_model,_,_,
                Goal,[Head|Body],Clauses,S,S1)),
        backtrack(i_model,_,_,Prev_goals,Clauses,S0).

backtrack(i_model,interpreter,11,[(Goals,Clause,S1)|Prev_goals],Clauses,S0) :-
        search_goal(i_model,_,_,Goal,Goals),
        search_clause_in_backtrack(i_model,_,_,
                Goal,Clause,[Head|Body],Clauses,S1,S2),
        environment(i_model,_,_,
                [Goals|Prev_goals],[Head|Body],S1,New_prev_goals),
        new_goals(i_model,_,_,Goals,Body,New_goals),
        interpreter(i_model,_,_,[New_goals|New_prev_goals],Clauses,S2,S0).
backtrack(i_model,interpreter,12,[(Goals,Clause,S1)|Prev_goals],Clauses,S0) :-
        search_goal(i_model,_,_,Goal,Goals),
        not(search_clause_in_backtrack(i_model,_,_,
                Goal,Clause,[Head|Body],Clauses,S1,S2)),
        backtrack(i_model,_,_,Prev_goals,Clauses,S0).

search_goal(i_model,interpreter,21,Goal,[Goal|_]).

search_clause(i_model,interpreter,31,
                Goal,[Head|Body],[[Head|Body]|Clauses],S,S0) :-
        unification(i_model,_,_,Goal,Head,S,S0).
search_clause(i_model,interpreter,32,
                Goal,[Head|Body],[[Head|Body]|Clauses],S,S0) :-
        not(unification(i_model,_,_,Goal,Head,S,S0)),
        search_clause(i_model,_,_,Goal,[Head|Body],Clauses,S,S0).

environment(i_model,interpreter,41,
                [Goals|Prev_goals],Clause,S,[(Goals,Clause,S)|Prev_goals]).

new_goals(i_model,interpreter,51,
                [Goal|Goals],Body,New_goals) :-
        append(Body,Goals,New_goals).

search_clause_in_backtrack(i_model,interpreter,61,
                Goal,Clause,Clause1,[Clause|Clauses],S,S0) :-
        search_clause(i_model,_,_,Goal,Clause1,Clauses,S,S0).
search_clause_in_backtrack(i_model,interpreter,62,
                Goal,Clause,Clause1,[Clause2|Clauses],S,S0) :-
        not(Clause = Clause2),
        search_clause_in_backtrack(i_model,_,_,
                Goal,Clause,Clause1,Clauses,S,S0).


/*************************************************/
/*    DCG rules for list recursion ([])          */
```

```
/*************************************************/

prog(_9480,_9481,_9482,_9483)-->
        s(_9480,_9481,_9482,_9483),
        f(_9480,_9482,_9483).

s(_9736,_9781,_9510,_9511)-->
        h(_9736,_9781,_9510,_9511),
        [':'],
        ['-'],
        b(_9736,_9781,_9510,_9511),
        ['.'].
h(_9736,_9781,_9510,_9511)-->
        pred(_9736),
        ['('],
        arg1(_9510),
        ['['],
        var0,
        ['|'],
        var1(_9781),
        [']'],
        arg2(_9511),
        [')'].
b(_9736,_9781,_9510,_9511)-->
        conj1,
        pred(_9736),
        ['('],
        arg1(_9510),
        var1(_9781),
        arg2(_9511),
        [')'],
        conj2.
f(_10188,_10094,_10095)-->
        pred(_10188),
        ['('],
        arg1(_10094),
        ['['],
        [']'],
        arg2(_10095),
        [')'],
        end.

/*************************************************/
/*    DCG rules for list recursion ([X|X])   */
/*************************************************/

prog(_9307,_9308,_9309,_9310)-->
        f(_9307,_9309,_9310),
        s(_9307,_9308,_9309,_9310).

f(_9323,_9398,_9399)-->
        pred(_9323),
        ['('],
        arg1(_9398),
        ['['],
        var0,
        ['|'],
        var0,
        [']'],
        arg2(_9399),
```

```
        [')'],
        end.
s(_9675,_9720,_9823,_9824)-->
        h(_9675,_9720,_9823,_9824),
        [':'],
        ['-'],
        b(_9675,_9720,_9823,_9824),
        ['.'].
h(_9675,_9720,_9823,_9824)-->
        pred(_9675),
        ['('],
        arg1(_9823),
        ['['],
        var0,
        ['|'],
        var1(_9720),
        [']'],
        arg2(_9824),
        [')'].
b(_9675,_9720,_9823,_9824)-->
        conj1,
        pred(_9675),
        ['('],
        arg1(_9823),
        var1(_9720),
        arg2(_9824),
        [')'],
        conj2.

/*****************************************************/
/*    DCG rules for numerical generate-test  */
/*****************************************************/

prog(_7287,_7288,_7289,_7290)-->
        f(_7287,_7289,_7290),
        s(_7287,_7288,_7289,_7290).

f(_7303,_7348,_7349)-->
        pred(_7303),
        ['('],
        arg1(_7348),
        fig,
        arg2(_7349),
        [')'],
        end.
s(_7549,_7572,_7709,_7710)-->
        h(_7549,_7572,_7709,_7710),
        [':'],
        ['-'],
        b(_7549,_7572,_7709,_7710),
        ['.'].
h(_7549,_7572,_7709,_7710)-->
        pred(_7549),
        ['('],
        arg1(_7709),
        var1(_7572),
        arg2(_7710),
        [')'].
b(_7549,_7572,_7709,_7710)-->
        conj1,
```

```
        pred(_7549),
        ['('],
        arg1(_7709),
        var2(_7636),
        arg2(_7710),
        [')'],
        conj3,
        var1(_7572),
        equal,
        var2(_7680),
        operator0,
        fig,
        conj2.

/*****************************************************/
/*    DCG rules for list recursion ()        */
/*****************************************************/

prog(_5398,_5399,_5400,_5401)-->
        cla,
        s(_5398,_5399,_5400,_5401),
        cla.

s(_5443,_5488,_5591,_5592)-->
        h(_5443,_5488,_5591,_5592),
        [':'],
        ['-'],
        b(_5443,_5488,_5591,_5592),
        ['.'].
h(_5443,_5488,_5591,_5592)-->
        pred(_5443),
        ['('],
        arg1(_5591),
        ['['],
        var0,
        ['|'],
        var1(_5488),
        [']'],
        arg2(_5592),
        [')'].
b(_5443,_5488,_5591,_5592)-->
        conj1,
        pred(_5443),
        ['('],
        arg1(_5591),
        var1(_5488),
        arg2(_5592),
        [')'],
        conj2.

/*****************************************************/
/*    DCG rules for write all of answers     */
/*****************************************************/

prog(_4021,_4022,_4023,_4024)-->
        s(_4021,_4022,_4023,_4024),
        cla(_4021).

s(_4049,_4391,_4152,_4392)-->
        h(_4049,_4391,_4152,_4392),
```

```
        [':'],
        ['-'],
        b(_4049,_4391,_4152,_4392),
        ['.'].
h(_4049,_4391,_4152,_4392)-->
        pred(_4049),
        ['('],
        arg1(_4152),
        var0,
        arg2(_4152),
        [')'].
b(_4049,_4391,_4152,_4392)-->
        conj1,
        write,
        ['('],
        term,
        [')'],
        conj3,
        fail.

/*****************************************************/
/*    DCG rules for dummy success                    */
/*****************************************************/

prog(_3836,_3837,_3838,_3839)-->
        s(_3836,_3837,_3838,_3839),
        f(_3836,_3838,_3839),
        cla(_3836).

s(_4095,_4096,_4097,_4098)-->
        h(_4095,_4096,_4097,_4098),
        [':'],
        ['-'],
        b(_4095,_4096,_4097,_4098),
        ['.'].
h(_4095,_4096,_4097,_4098)-->
        pred(_3864),
        ['('],
        arg1(_3933),
        term,
        arg2(_3934),
        [')'].
b(_4095,_4096,_4097,_4098)-->
        conj1,
        fail.
f(_4157,_4202,_4203)-->
        pred(_4157),
        ['('],
        arg1(_4202),
        var0,
        arg2(_4203),
        [')'],
        end.

/*****************************************************/
/*    DCG rules for cut-fail combination             */
/*****************************************************/

prog(_3060,_3061,_3062,_3063)-->
        cla,
```

```
        s(_3060,_3061,_3062,_3063),
        cla.

s(_3329,_3330,_3331,_3332)-->
        h(_3329,_3330,_3331,_3332),
        [':'],
        ['-'],
        b(_3329,_3330,_3331,_3332),
        ['.'].
h(_3329,_3330,_3331,_3332)-->
        pred(_3093),
        ['('],
        argument_list(_3110),
        [')'].
b(_3329,_3330,_3331,_3332)-->
        conj1,
        ['!'],
        [','],
        fail.

/*************************************************/
/*    DCG rules for tree recursion               */
/*************************************************/

prog(_7866,_7867,_7868,_7869,_7870,_7871)-->
        f(_7866,_7868,_7869,_7870,_7871),
        s(_7866,_7867,_7868,_7869,_7870,_7871),
        cla(_7866).

f(_7884,_7957,_7958,_7907,_7924)-->
        pred(_7884),
        ['('],
        arg1(_7957),
        pred1(_7907),
        ['('],
        argument_list(_7924),
        [')'],
        arg2(_7958),
        [')'],
        end.
s(_8247,_8296,_8402,_8403,_8270,_8723)-->
        h(_8247,_8296,_8402,_8403,_8270,_8723),
        [':'],
        ['-'],
        b(_8247,_8296,_8402,_8403,_8270,_8723),
        ['.'].
h(_8247,_8296,_8402,_8403,_8270,_8723)-->
        pred(_8247),
        ['('],
        arg1(_8402),
        pred1(_8270),
        ['('],
        a1(_8287),
        var1(_8296),
        a2(_8305),
        [')'],
        arg2(_8403),
        [')'],
        {_8723 is _8287+_8305+1}.
b(_8247,_8296,_8402,_8403,_8270,_8723)-->
```

```
        conj1,
        pred(_8247),
        ['('],
        arg1(_8402),
        var1(_8296),
        arg2(_8403),
        [')'].

/*****************************************************/
/*    DCG rules for conjunctive recursion    */
/*****************************************************/

prog(_7866,_7867,_7868,_7869)-->
        cla,
        s(_7866,_7867,_7868,_7869),
        cla.

s(_7911,_7942,_8104,_8105)-->
        h(_7911,_7942,_8104,_8105),
        [':'],
        ['-'],
        b(_7911,_7942,_8104,_8105),
        ['.'].
h(_7911,_7942,_8104,_8105)-->
        pred(_7911),
        ['('],
        arg1(_8104),
        ['('],
        var1(_7942),
        [','],
        var2(_7959),
        [')'],
        arg2(_8105),
        [')'].
b(_7911,_7942,_8104,_8105)-->
        pred(_7911),
        ['('],
        arg1(_8104),
        var1(_7942),
        arg2(_8105),
        [')'],
        [','],
        pred(_8056),
        ['('],
        arg1(_8104),
        var2(_8079),
        arg2(_8105),
        [')'].

/*****************************************************/
/*    DCG rules for numerical recursion      */
/*****************************************************/

prog(_7374,_7375,_7376,_7377)-->
        f(_7374,_7376,_7377),
        s(_7374,_7375,_7376,_7377).

f(_7390,_7435,_7436)-->
        pred(_7390),
        ['('],
```

```
        arg1(_7435),
        fig,
        arg2(_7436),
        [')'],
        end.
s(_7636,_7659,_7796,_7797)-->
        h(_7636,_7659,_7796,_7797),
        [':'],
        ['-'],
        b(_7636,_7659,_7796,_7797),
        ['.'].
h(_7636,_7659,_7796,_7797)-->
        pred(_7636),
        ['('],
        arg1(_7796),
        var1(_7659),
        arg2(_7797),
        [')'].
b(_7636,_7659,_7796,_7797)-->
        conj1,
        var2(_7700),
        equal,
        var1(_7659),
        operator0,
        fig,
        conj3,
        pred(_7636),
        ['('],
        arg1(_7796),
        var2(_7765),
        arg2(_7797),
        [')'],
        conj2.

/*************************************************/
/*    DCG rules for catch all                    */
/*************************************************/

prog(_1805,_1806,_1807,_1808)-->
        f_cla(_1805),
        f(_1805,_1807,_1808).

f(_1832,_1898,_1899)-->
        pred(_1832),
        ['('],
        arg1(_1898),
        var1(_1855),
        arg3(_1864),
        var2(_1873),
        arg2(_1899),
        [')'],
        end.

/*************************************************/
/*    DCG rules for subconcept                   */
/*************************************************/

small_letter-->
        [X],
        {member(X,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,
```

```
                o,p,q,r,s,t,u,v,w,x,y,z])}.
capital_letter-->
        [X],
        {member(X,['A','B','C','D','E','F','G','H','I','J','K','L',
        'M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'])}.
digit-->
        [X],
        {member(X,[0,1,2,3,4,5,6,7,8,9])}.
built_in-->
        builtin.
symbol-->
        [X],
        {member(X,['+','-','*','/','<','>','¥','@','#',
                '$','^','&','~',':',',','?','.','''])}.
fig-->
        [X],
        {integer(X)}.

operator0-->
        ['+'].
operator0-->
        ['-'].
end-->
        ['.'].
end-->
        [':'],
        ['-'],
        ['!'],
        ['.'].
%
char-->
        small_letter.
char-->
        digit.
anychar-->
        small_letter.
anychar-->
        capital_letter.
anychar-->
         digit.
anychar-->
        symbol.
char_list-->
        char.
char_list-->
        char,
        ['_'],
        char_list.
char_list-->
        char,
        char_list.
symbol_list-->
        symbol.
symbol_list-->
        symbol,
        symbol_list.
anychar_list-->
        anychar.
anychar_list-->
        anychar,
```

```
        anychar_list.
anychar_list-->
        anychar,
        ['_'],
        anychar_list.
var0-->
        capital_letter.
var0-->
        ['_'],
        char_list.
var0-->
        ['_'].
var0-->
        capital_letter,
        char_list.
var1(V,X,Y) :-
        var0(X,Y),
        eqdifset(X,Y,V).
var2(V,X,Y) :-
        var0(X,Y),
        eqdifset(X,Y,V).
atom-->
        small_letter.
atom-->
        small_letter,
        char_list.
atom-->
        symbol_list.
atom-->
        [''''],
        anychar_list,
        [''''].
atom-->
        ['('],
        anychar_list,
        [')'].
constant-->
        atom.
constant-->
        fig.
%
%
pred(P,X,Y) :-
        atom(X,Y),
        eqdifset(X,Y,P).
pred1(P,X,Y) :-
        atom(X,Y),
        eqdifset(X,Y,P).
structure(P)-->
        pred(P),
        ['('],
        argument_list(_),
        [')'].
list-->
        ['['],
        [']'].
list-->
        ['['],
        argument_list(_),
        [']'].
```

```
list-->
        ['['],
        argument_list(_),
        ['|'],
        argument,
        [']'].
argument-->
        var0.
argument-->
        list.
argument-->
        constant.
argument-->
        structure(P).
argument_list(1)-->
        argument.
argument_list(A)-->
        argument,
        [','],
        argument_list(B),
        {A is B+1}.
a1(A)-->
        arg1(A).
a2(A)-->
        arg2(A).
arg1(0)-->
        [].
arg1(A)-->
        argument_list(A),
        [','].
arg2(0)-->
        [].
arg2(A)-->
        [','],
        argument_list(A).
arg3(0)-->
        [','].
arg3(A)-->
        [','],
        argument_list(A),
        [','].

term-->
        constant.
term-->
        var0.
term-->
        list.
term-->
        built_in.
term-->
        structure(P).
%
%
goal-->
        structure(P).
goal-->
        byed(_).
goal-->
        ['!'].
```

```
goal-->
        var0,
        [i],
        [s],
        var0,
        operator0,
        fig.
goal-->
        var0,
        [i],
        [s],
        var0,
        operator0,
        var0.
conjunction-->
        goal.
conjunction-->
        goal,
        [','],
        conjunction.
conj1-->
        [].
conj1-->
        conjunction,
        [','].
conj2-->
        [].
conj2-->
        [','],
        conjunction.
conj3-->
        [','].
conj3-->
        [','],
        conjunction,
        [','].
rule(P)-->
        structure(P),
        [':'],
        ['-'],
        conjunction,
        ['.'].
fact(P)-->
        structure(P),
        ['.'].
clause0(P)-->
        rule(P).
clause0(P)-->
        fact(P).
program(P)-->
        clause0(P).
program(P)-->
        clause0(P),
        program(P).
cla(_)-->
        [].
cla(P)-->
        program(P).
f_cla(P)-->
        fact(P).
```

```
f_cla(P)-->
        fact(P),
        f_cla(P).

'fail'-->
        [f],
        [a],
        [i],
        [l].
write-->
        [w],
        [r],
        [i],
        [t],
        [e].
nl-->
        [n],
        [l].
equal-->
        [i],
        [s].
eqdifset(A,A,[]) :-
        !.
eqdifset([A|X],Y,[A|Z]) :-
        eqdifset(X,Y,Z).
```

BIBLIOGRAPHY

[1] Carbonell, J. R.: AI in CAI: An artificial intelligence approach to computer-aided instruction, IEEE Trans. Man-Mach. Syst., Vol.MMS-11, No.4, pp.190-202 (1970).

[2] Barr, A. and Feigenbaum, E. A.: The Handbook of Artificial Intelligence, Vol.II, PITMAN, London, pp.225-235 (1983).

[3] Papert, S.: MINDSTORMS - Children, Computers, and Powerful Ideas, Basic Books, New York (1980).

[4] Shapiro, E. Y.: Algorithmic Program Debugging, MIT Press, London (1982).

[5] Fujii, K. et al.: A Conversion of C-Prolog into the Super Mini-Computer MV/8000II, Proc. of 28th National Conference of Information Processing Society of Japan, 2G-7, Tokyo, [in Japanese] (1984).

[6] Barr, A. and Feigenbaum, E. A.: The Handbook of Artificial Intelligence, Vol.I, PITMAN, London, pp.141-222 (1981).

[7] Goldstein, D.: The genetic graph: a representation for the evolution of procedural knowledge, in Sleeman, D. et al. (ed.), Intelligent Tutoring Systems, Academic Press, London, pp.51-77 (1982).

[8] Brown, J. S. and Burton, R. R.: Diagnostic models for procedural bugs in basic mathematical skills, Cognitive Science 2, pp.155-192 (1978).

[9] Sleeman, D. and Hendley, R. J.: ACE: A system which Analyses Complex Explanations, in Sleeman, D. et al. (ed.), Intelligent Tutoring Systems, Academic Press, London, pp.99-118 (1982).

[10] Clancey, W. J.: Tutoring rules for guiding a case method

dialogue, in Sleeman,D. et al. (ed.), Intelligent Tutoring Systems, Academic Press, London, pp.201-225 (1982).

[11] Clocksin, W. F. and Mellish, C. S.: Programming in Prolog, Springer-Verlag, New York (1981).

[12] Nakamura, Y. et al.: An Interface of the Tutoring System for Programming in Prolog, Proc. of 30th National Conference of Information Processing Society of Japan, 4L-3, Tokyo, [in Japanese] (1985).

[13] Pereira, F. C. N. and Warren, D. H. D.: Definite Clause Grammars for Language Analysis, Artificial Intelligence 13, pp.231-278 (1980).

[14] Goto, S.: A Geometrical Displaying Method for the Prolog Execution, Proc. of WG/SYM of IPSJ, Vol.27, No.6, [in Japanese] (1984).

[15] Numao, M.: Visual Debugging Tools for Prolog, Proc. of WG/SYM of IPSJ, Vol.32, No.1, [in Japanese] (1985).

[16] Kawai, K., Ganke, M. and Toyoda, J.: FLOGS: An Extended PROLOG System for Procedural processing, Trans. of Information Processing Society of Japan, Vol.26, No.1, pp.112-120, [in Japanese] (1985).

[17] Kawai, K., Mizoguchi, R., Kinoh, H., Ganke, M., Kakusho, O. and Toyoda, J.: A Framework for Intelligent CAI Systems based on Logic Programming and Inductive Inference, Trans. of Information Processing Society of Japan, Vol.26, No.6, pp.1089-1096, [in Japanese] (1985).