Winter 1991

Integration of Abductive and Deductive Inference Diagnosis Model and Its Application in Intelligent Tutoring System

Jingying Zhang
Old Dominion University

INTEGRATION OF ABDUCTIVE AND DEDUCTIVE INFERENCE DIAGNOSIS MODEL AND ITS APPLICATION IN INTELLIGENT TUTORING SYSTEM

by

Jingying Zhang
M.S. December 1981, Beijing University of Aeronautics and Astronautics
Beijing, People's Republic of China
B.S. December 1976, Jiangxi University, Nanchang, Jiangxi, P.R.C.

A Dissertation
Submitted to the Faculty of Old Dominion University
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
Computer Science
Old Dominion University
December, 1991

Approved by:

Stewart N. T. Shen (Director)

ABSTRACT

INTEGRATION OF ABDUCTIVE AND DEDUCTIVE INFERENCE DIAGNOSIS MODEL AND ITS APPLICATION IN INTELLIGENT TUTORING SYSTEM

Jingying Zhang
Old Dominion University, 1991
Director: Dr. Stewart N.T. Shen

This dissertation presents a diagnosis model, Integration of Abductive and Deductive Inference diagnosis model (IADI), in the light of the cognitive processes of human diagnosticians. In contrast with other diagnosis models, that are based on enumerating, tracking and classifying approaches, the IADI diagnosis model relies on different inferences to solve the diagnosis problems. Studies on a human diagnosticians' process show that a diagnosis process actually is a hypothesizing process followed by a verification process. The IADI diagnosis model integrates abduction and deduction to simulate these processes. The abductive inference captures the plausible features of this hypothesizing process while the deductive inference presents the nature of the verification process. The IADI diagnosis model combines the two inference mechanisms with a structure analysis to form the three steps of diagnosis, mistake detection by structure analysis, misconception hypothesizing by abductive inference, and misconception verification by deductive inference. An intelligent tutoring system, "Recursive Programming Tutor" (RPT), has been designed and developed to teach students the basic concepts of recursive programming. The RPT prototype illustrates the basic features of the IADI diagnosis approach, and also shows a hypertext-based tutoring environment and the tutoring strategies, such as concentrating diagnosis on the key steps of problem solving, organizing explanations by design plans and incorporating the process of tutoring into diagnosis.

ACKNOWLEDGEMENTS

I wish to thank my advisor Dr. Stewart Shen for his crucial guidance and encouragement during my study and research.

I wish to thank the other members in my previous and final committees, Dr. Larry Wilson, Dr. Christian Wild, Dr. Ravi Mukkamala, Dr. Robert Lucking and Dr. Alan Mandell for their valuable suggestions and support in this research, and careful reading and helpful advice in reviewing this dissertation.

I would like to express my deep thanks to my parents, Yiaochen Zhang and Yunfang Du, for their lasting guidance, support and understanding.

I also appreciate my husband, Shensheng Zhao, who has endowed most concern and help from many perspectives.

Thanks are also due to other faculty members in Computer Science department: Dr. Michael Overstreet, Dr. Stephen Olariu, Dr. Shunichi Toida, Dr. James Schwing and Dennis Ray for their help.

The friendship and help from my fellow graduate students, Jih-shih Hsu, Myron Xu and Ghassan Issa are also appreciated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Research on intelligent tutoring serves two goals. The first one is to develop systems for automating education, and second one is to explore epistemological issues similar to those studied by psychologists (Anderson 90).

Psychologists, educators, and computer scientists are concerned with the research on intelligent tutoring since 1970s, which has evolved from Computer-Assisted Instruction (CAI). It has become one of the most active fields in Artificial Intelligence (AI) (Barr and Feigenbaum 82) (Clancey 87). Intelligent Tutoring Systems are systems that teach people new knowledge with AI technologies. Why do people precede them with "intelligent"? What does it mean to teach intelligently? Many researchers in this field try to give explanations to this point (Clancey 87) (Siuru 89) (Barr and Feigenbaum 82) (Vanlehn 88) (Sleeman 82). Although there is no acknowledged definition for Intelligent Tutoring Systems, contrasting with CAI, there is one characteristic shared by many ITSs, which is that ITSs refer to a model of the student's current knowledge (Vanlehn 88) (Wallach 87). Based on this model, ITSs can not only transfer the predefined knowledge in selected material, but can also ferret out the student's misunderstandings and adapt the content of instruction to the student's level.

However, understanding students and focusing instructions on their understanding is not easy even for human beings. Psychologists and educationists have been studying

this process since ancient times. The famous Chinese ancient educationist Han Fei Tzu said in the section "The Difficulties of Persuasion" (Han 64):

> On the whole, the difficult thing about persuading others is not that one lacks the knowledge needed to state his case nor the audacity to exercise his abilities to the full. On the whole, the difficult thing about persuasion is to know the mind of the person one is trying to persuade and to be able to fit one's words to it (72-72).

Thus, in addition to the subject matter, there are two difficult tasks for the tutor. First, the tutor must know what the student is really thinking; then he must find an individualized instruction that fits the student's needs.

Humans now use computers to teach students automatically and individually. But these efforts must also confront the above two difficulties. With AI technology, one can build the student module to represent the student's current understanding. Modern computer techniques provide many methods and tools that can be used to create a good tutoring and learning environment that alleviates these difficulties. ITSs are rising as one promising field to solve these two difficult problems.

There are dozens of ITSs scattered throughout the literature. These systems integrate the intelligent tutor with computer-based technologies within the different subjects of expertise. Although researchers in this field made great efforts to put forward a variety of methods, and have built some systems to make ITSs more intelligent, only a few systems provide the applications in the real world. This implies that constructing ITSs is still in a premature stage. Thus it is necessary to build more ITSs for exploration, and also for gathering universal knowledge about how to build practical ITSs (Polson and Richardson 88).

This dissertation presents a new diagnosis model, "Integration of Abductive and Deductive Inference" (IADI), to diagnose the student's current understanding. This mod-

el is applied to student modeling problems of intelligent tutoring systems. A detailed discussion about this diagnosis model is presented in chapter 3 which describes the design and implementation of the intelligent Recursive Programming Tutor (RPT), which teaches students the basic understanding of recursive programming. In this system the pedagogical expertise in recursive programming was employed as domain knowledge, and the case induction method was adopted to tutor students. To successfully achieve this goal, the hypertext based tutor environment was created.

This chapter provides an introduction to this research work. Section 1.1 outlines the existing Intelligent Tutoring Systems (ITSs), section 1.2 reviews the background of the ITS, including the architecture of ITSs and the tutoring environment, section 1.3 briefly introduces the IADI diagnosis model, and section 1.4 provides the dissertation overview.

1.1 Outline of Intelligent Tutoring Systems

Intelligent Tutoring Systems and the traditional Computer-Assisted Instruction systems both have representations of the subject knowledge they teach. But the ITS is distinguished from its predecessor the CAI by the way in which it communicates with students, tries to understand students, and diagnoses the students' misunderstandings. ITSs apply AI knowledge representations and inference mechanisms to present and reason about the subject matters, the student understandings, and pedagogical principles.

When one thinks of an intelligent tutor, one generally imagines an anatomy like in Fig. 1, that provides a convenient classification of the research and direction of development. The domain expert provides the domain knowledge and prepares to reason on that domain. The tutorial expert offers strategies for students to learn the domain knowledge. The diagnosis expert detects erroneous assumptions that the student may have. The student model represents the student's learning background and the current state of knowledge. The instructional environment supports the activities of students and tutors. The three experts, the domain expert, the diagnosis expert and the tutorial expert directly or

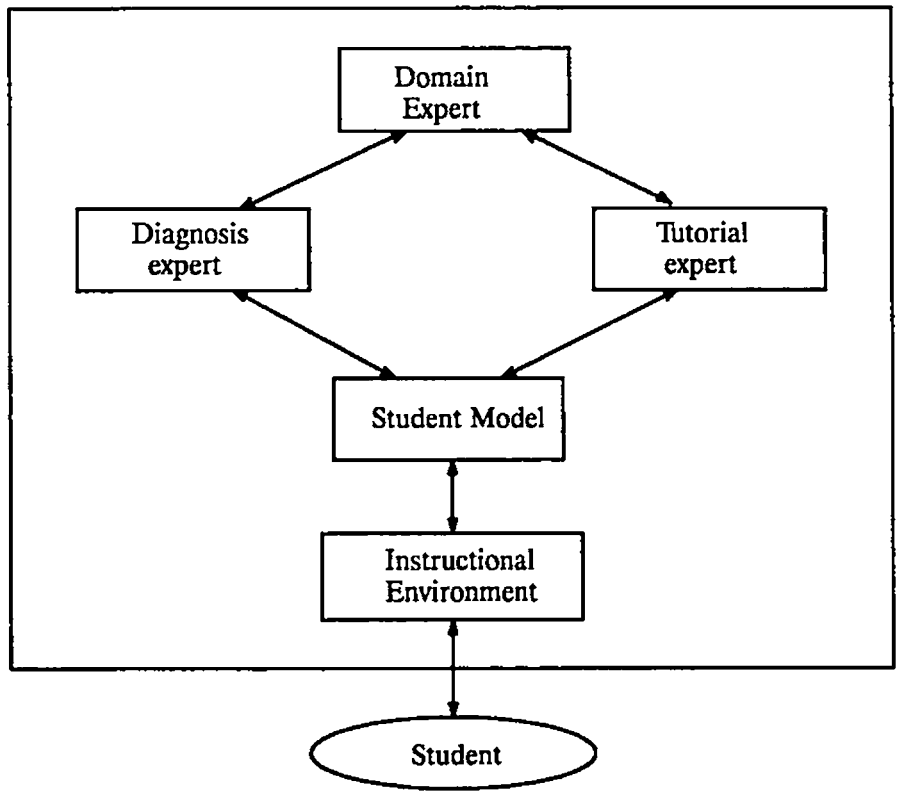

Fig. 1. Components of an ITS

indirectly communicate with the student model in the tutoring process. Through the student model and the diagnostic process, the system can identify what the student does and does not know about the subject matter, and can then focus the tutoring effort on the individual pedagogical needs of a student.

The existing intelligent tutoring systems possess different features. Some of them focus on organization of subject knowledge (Clancey 82); some of them lay the particular emphasis on diagnosing the student's current misunderstanding (Johnson and Soloway 84) (Burton 82) (Bonar 88); some of them are more involved with the instructions of various strategies or pedagogies (Anderson 90) (Woolf and McDonald 84); some of the systems concentrate their efforts on creating tutor environments (Woolf 88) (Hollan 84).

To construct an intelligent tutoring system, one must consider that the crucial work is in building the student model. There are several student model representations in existing ITSs, including production systems (Clancey 82), and procedural networks (Burton 82), frame representations (Carbonell 70) (Laubsch 75), and genetic graphs (Goldstein 82). After a representational scheme has been decided, the student's understanding needs to be detected. That is, for each of the student's behaviors, the system should hypothesize the corresponding explanations. However, sometimes the student's behaviors are incomplete or ambiguous, and this makes the diagnostic process more complicated. Even so growing amounts of research efforts have been thrown into the studies of student modeling problem. Various knowledge representations coordinating distinct diagnostic approaches have emerged in different ITSs.

1.2 Background

The main components of an ITS are the domain expert module, the student modeling module, including the diagnosis process based on a student model, and the tutorial expert module. These deal with the problem-solving expertise, students' knowledge, and tutoring strategies respectively. The tutorial environment, which is also an important part of the ITS, facilitates the interaction between students and the computer tutor (Polson and Richardson 88). This section introduces the previous work according to the architecture of ITSs rather than proceeding chronologically.

1.2.1 Domain knowledge module

The domain expert module provides the knowledge to be imparted to the student, and a standard for evaluating the student's performance. This module is classified in three approaches (Anderson 88). Each of the three approaches moves toward a more cognitively faithful representation of the content expertise.

• **Black box model**

The first one is opaque and is called the black box model. It generates the correct input–output behaviors over a range of tasks in the domain, and therefore can be used as a judge of correctness. The early systems such as SOPHIE (SOPHisticated Instructional Environment) (Brown and Burton 75) and WEST (Burton and Brown 82) perform their calculations as a black box. SOPHIE attempted to teach students to troubleshoot faulty electronic circuits. It only checks the consistency of a student's hypotheses about failed circuit elements. WEST works as a mathematics game. The opaque expert performs an exhaustive search for the possible moves in an electronic game board and determines the optimal move. In the black box approach the internal computations and reasoning processes from a given input to the conclusion are not available to users. Although this approach does not present the reasoning process, it provides the correct output as the information used to recognize the differences between student and expert performances.

• **Glass box model**

The second approach is transparent and is called the glass box model or expert system methodology. Since a major component of an expert system is an articulate, human–like representation of the underlying expertise in the domain, it is natural to use the expert system as the domain expert module of the tutor to avoid the time–consuming knowledge acquisition process. The domain expert component tries to explain and inspect each problem–solving decision in terms that correspond (at some level of abstraction) to those of a human problem–solver. But it only allows for explanations of information processes inherent in the rules of its knowledge base which does not necessarily correspond to the way a human expert reasons. The tutor GUIDON developed by Clancey (Clancey 82) is based on the well–known expert system EMYCIN, whose domain of expertise is the diagnosis of bacterial infections.

• **Cognitive model**

The third approach is the cognitive model which simulates the process of human problem-solving in a human-like manner. This approach simulates not only the domain knowledge, but also the way humans use that knowledge. This model is considered the most effective (Anderson 88). Although its implementation is time-consuming, there have been dramatic improvements over the past 10 years (Wenger 87). It is helpful to consider the types of knowledge to be taught, because that dictates the strategies of instruction. There are three types according to Anderson: procedural knowledge, which is about how to perform a task; declarative knowledge, which is in the form of a set of facts appropriately organized; and causal knowledge, which allows humans to reason about behaviors by using causal understanding.

Procedural knowledge can usually be represented by production rules. This type of knowledge is applied in GEOMETRY Tutor, LISP Tutor (Anderson 90), DEBUGGY systems (Burton 82), and some other systems. One of the major advantages of the production rule is its modularity. Each production rule is an independent piece of knowledge which benefits the instruction. The declarative knowledge in SCHOLAR system (Carbonell 70), which is designed to teach South American geography, is represented by a semantic network. In the network, the nodes stand for geographical objects and concepts, which are organized in a partial hierarchy with relationships represented by links. Some simple inference can be made by propagation of inherited properties via these hierarchical links. However, in the WHY system, which also teaches geography (Stevens, Collins and Goldin 82), the declarative knowledge is represented by a schema consisting of action slots, factor slots, and fillers. The formalisms of causal knowledge are not as mature as rule-based or schema formalisms. This knowledge representation was explored in SOPHIE (Brown and Burton 82) and in de Kleer's work on causal propagation of the behavior of device construction (de Kleer and Brown 83).

Actually the three kinds of knowledge are not isolated. In the cognitive process, humans represent static structure as declarative knowledge, and employ the procedural knowledge to dynamically reason about the behavior by utilizing causal interactions. Anderson's LISP, GEOMETRY and ALGEBRA tutors do the preliminary exploration by combining the different types of knowledge (Anderson 90).

1.2.2 Student modeling module

The knowledge structure that represents the student's current understanding of the subject matter is called the student model. The reasoning process, which detects the student's mistakes by referring to the student model, is called the diagnosis. The student model and diagnosis form the student modeling problem (Vanlehn 88) (Barr and Feigenbaun 82), setting in the student model and diagnosis expert in Fig.1.

Vanlehn expounds the essential problems of student modeling in ITSs (Vanlehn 88). According to his suggestion, the student model can be classified three dimensions. The first dimension is the bandwidth, a measurement of the input of the student activity. The second dimension is the target knowledge type, and the third dimension is the degree of difference between students and experts. These dimensions indicate the structural properties of the student model. These classifications are designed to capture the differences in the student modeling problem, and help the tutoring module decide the different tutor strategies.

How much of the student's activity is available to the diagnostic procedure? The first dimension, the bandwidth, will give a measurement of the input information. Most systems only input the final state to the diagnosis process, such as the student answers to the question of a mathematics subtraction problem in the DEBUGGY system (Burton 82). Some programs can give the intermediate states, such as the SPADE system (Miller 82), which will supply information at several observable stages about what the student is doing. The highest bandwidth is supposed to be able to attain a list of the approximate

mental states as the student solves a problem. A metal model is a coherent collection of knowledge held by a person about some aspect, entity or concept of the world (Gentner and Stevens 83). The LISP tutor (Anderson 90) tries to track the cognitive process as closely as possible to the student's mental states. This tutor uses a menu driven interface to offer a student multiple problem solving paths. Assuming the paths can describe every state of reasoning, the tutor obtains a sequence of mental states.

The second dimension is the target knowledge type which is categorized declarative knowledge and procedural knowledge, including flat and hierarchical. Hierarchical representations allow subgoals; flat ones do not. In the ACM diagnosis system (Langley and Ohlsson 84), the subtraction procedure is a flat representation, but in the BUGGY system it is a hierarchical one (Brown and Burton 78). The inference in a hierarchical representation is more difficult because it takes more steps to know the subgoals, while in the flat representation nothing is hidden. The declarative knowledge representations have been used for meteorology (Stevens, Collins and Goldin 82) and geography (Carbonell 70).

The difference between students and experts is the third dimension. Knowing the differences is a necessary step to the diagnosing process. In most ITSs, the student model can be represented by the domain expert model plus a list of missing conceptions. Such a student model is called an overlay model because the student model is just a proper subset of the domain expert model. The overlay model is the most common type of student model because it is easy to implement (Brown and Burton 78). However, it is of limited value because of the fact that students have misconceptions as well as missing conceptions. More complex student models represent misconceptions as well as the missed conceptions. They employ a library to organize the predefined bugs (the missing conceptions and the misconceptions). The bug library in the DEBUGGY system (Burton 82) is directed by the experimental analysis of several thousand mathematics subtraction tests. In diagnosis the system compares the student's behavior with the expert

module and the bug library to form the student model. It is possible that a student has several bugs and combinations of bugs. So the bug library has to include a large number of compound bugs. There is an alternative to the bug library method which only contains bug parts, and it dynamically constructs compound bugs from a library of bug parts. It does not use the predefined bug library. This method is applied in the ACM system (Langley and Ohlsson 84).

The diagnosis methods vary in different systems. Some of them are listed in table 1. The diagnosis problem will be presented in further detail in the next chapter.

1.2.3 Tutor module

The tutor module communicates with students. A tutor bears responsibility for selecting and sequencing the material to be taught, monitoring and criticizing the student's performance, and providing assistance whenever the student needs it. Usually, a domain expert module is involved with formulating a representation of the material, and selecting and sequencing the particular concepts. The instruction process, which is delivering the curriculum to the student, should be accomplished in the tutor module (Halff 88). In order to deliver the knowledge to the student, and reduce the differences between teacher and student as soon as possible, the tutor must determine the corresponding pedagogical strategies in addition to knowing the student's current understanding. The following are some strategies used in the ITS.

- **Socratic method**

The Socratic method or presentation method is one way to present material by dialogue (Collins and Stevens 82). The dialogue elaborates in different ways for the different instructional objects. Teaching facts and concepts is done by explaining the material. Teaching rules and functional relationships usually involves inducing the student to consider the relevant data and to formulate the rule. Skills for deriving rules are taught

System	Subject Matter	Student Module	Diagnosis Method	Reference
ACM	Subtraction	Library of Bug Parts	Condition Induction	Langley & Ohlsson 84
APROPOS2	Programming in PROLOG	Library of Bug	Plan Recognition	Looi 88
BIP	Programming in BASIC	Overlay	Plan Recognition	Barr 76
Bridge	Programming in PASCAL	Library of Bug	Plan Recognition	Bonar 88
BUGGY	Subtraction	Library of Bug	Decision Tree	Brown & Button 78
DEBUGGY	Subtraction	Library of Bug Parts	Generate & Test	Burton 82
Geometry	Geometry	Library of Bug Parts	Model Tracing	Anderson 90
LISP	Programming in Lisp	Library of Bug Parts	Model Tracing	Anderson 90
GUIDON	Infectious diseases	Overlay	Diagnosis rules	Clancey 82
MENO	Programming in PASCAL	Library of Bug	Plan Recognition	Woolf & McDonald 84
PROUST	Programming in PASCAL	Library of Bug	Plan Recognition	Soloway 83
Scholar	Geography	Overlay	Diagnosis rules	Carbornell 70
SOPHIE	Electronic Trouble-shooting	Overlay	Issue Analyzing	Brown 82
SPADE	Programming in LOGO	Library of Bug Parts	Plan Recognition	Miller 82
Steamer	Steamship Propulsion	Overlay	Issue Analyzing	Hollan 84
WEST	Arithmetic Expressions	Overlay	Issue Analyzing	Brown & Button 82
WHY	Cause of Rainfall	Library of Bug	Diagnosis rules	Stevens 82
WUSOR	Logical Relations	Overlay	Issue Analyzing	Goldstein 82

Table 1. ITSs and Their Diagnosis Methods

as procedures. These procedures are broken down into their components. This method involves questioning the student in a way that will encourage him to reason about what he knows whereupon the system can modify his conceptions. SCHOLAR (Carbonell 70) was the first system that used the Socratic style of tutoring. The original system was developed for teaching South American geography. It first diagnoses the underlying misconceptions in the student's knowledge; it then poses a problem that will guide the student to discover his errors. The WHY system is a follow-up of SCHOLAR (Stevens and Collins 77). It uses a set of tutorial rules to construct an experimental Socratic tutoring system. This method leads the student to find errors or contradictions by entrapping him in the consequences of his own conclusions.

• **Case presentation**

Case presentation is one method that presents the target skill by using worked examples and guided exercises. These skills must be presented to the student in a manner that demonstrates how the expert makes the decision at each step in the case and what strategies he uses. It is also an apprenticeship style of learning. The SOPHIE system is a good example of case presentation (Brown, Burton and de Kleer 82). Faults can be inserted into the system to make a case, and the student is instructed to offer a hypothesis as to what might be wrong. The student then makes a measurement of the circuit to evaluate the hypothesis. Before each measurement the expert explains why it is required. After each measurement, it explains what it can conclude from it. That is, the system tells the student not whether the hypothesis is a correct identification of the fault, but whether it is logically consistent with the information from the measurements. This method makes explicit the strategies a domain expert uses, thereby giving the student an example to follow.

• **Coaching**

This method attempts to maintain control of the tutorial situation in order to protect the student from inappropriate or incorrect learning, and to keep the student from exploring paths that are not instructionally useful. By coordinating the model tracing or issue-based analyzing diagnosis technique (Anderson 90) (Burton and Brown 82), the tutor can trace the student's behavior, and match it to one of the paths that should be the correct or optimal problem-solving solution. When the match fails, the tutor intervenes with advice and suggestions to guide the student to a successful path. There are several coaching systems such as WEST (Brown and Burton 82), WUSOR (Goldstein 82) and LISP tutor (Anderson 89).

1.2.4 Instructional environment

The instructional environment refers to the part of an ITS that specifies or supports the student's activities and provides the tools for convenience learning. The environment should be designed to allow students understand concepts efficiently. The following are systems which have the effective environments.

- **Visualization of multiple concepts**

The Envisioning Machine (Woolf 88) presents a visualization of the concepts of physics. On the screen, the student can grab a ball, drop it or throw it in any direction, and watch the trajectory of the object. At the same time there is a force diagrammatic view of objects in motion. Two vectors in the diagram represent respectively the velocity and acceleration of the object. In the past, acceleration and velocity have been difficult to demonstrate because they only can be illustrated through still pictures. The environment gives the viewer multiple perspectives of concepts: motion trajectory, velocity and acceleration of an object, and parallel displays of physical motion. The student can adjust his conceptions from his observations.

• **Visual reasoning**

An efficacious environment should make explicit or manifest an originally implicit property of the contents, therefore aid the learner to accept and understand knowledge effectively. The GEOMETRY tutor (Anderson 90) is a good example of an environment that provides a new form of visual reasoning for the student, and brings out implicit properties in the task. This system builds a logical chain from the premise to the conclusion, and the system presents the process as a tree on the screen. Every node in the tree is a statement, and a step of the proof derived from inference rules or geometry theorems. The system shows the relationship of the steps and how they are arranged on a path to the proof. The system provides the proof which can be derived from either the bottom upward, using forward inference, or top downward, using backward reasoning.

• **Icon provided**

The STEAMER system provides simulation in a graphical display of a steam plant (Hollan, Hutchins and Weitzman 84). In the graph, it supplies the icons to depict the mechanism of a steam plant. When the simulation (the flow of fluid through the plant) is running, motion in the pipes is animated to indicate the causal connections between different parts of the plant. There are many other kinds of icon displays such as dials that give the pressure at various points in the plant, and curve the graph of pressure as it changes through time, and indicates the rate of change. These displays make visible some aspects of automatic control systems that are difficult to see with traditional gauges. Also these graphic displays can depict a steam plant in the different levels, from the scheme of the entire plant to the separate part of the plant.

• **Dialogue**

Most ITSs use dialogue for the instructional environment since humans naturally use language as the main communication tool. MENO (Woolf and McDonald 84) uses

dialogue in an attempt to understand a student's programming bug and help resolve his misconceptions. The system detects the errors in simple Pascal programs by the Bug Finder, and passes the messages about the location of the error to the tutor. The tutor then analyses the errors by communicating with the student in a dialogue. The student answers the questions asked by the tutor in a simple way, such as a "yes/no" choice of response. The tutor gives the suggestions to correct the program. There are corresponding discourses for each different misconceptions. The LISP tutor (Anderson 90) also uses dialogue, and it applies a menu-driven method. By menu choices, it imposes a structure on what the student is allowed to do. The LISP tutor walks the student through the creation of a LISP function, and corrects the student whenever he deviates from the correct path. SCHOLAR, WHY and GUIDON use dialogue too.

1.3 IADI Diagnosis Model

This dissertation proposes a new diagnosis model, the Integration of Abductive and Deductive Inference diagnosis model (IADI), to diagnose students' mistakes and misconceptions in an intelligent tutoring system. An intelligent tutoring system prototype, the "Recursive Programming Tutor", has been designed and developed to teach students to understand the basic concepts of recursive programming, and to illustrate this diagnosis approach (Shen and Zhang 91) (Shen and Zhang 89b).

The IADI diagnosis model is based on the understanding of the cognitive process of human diagnosticians. Usually a human diagnostician does not make a final decision immediately after seeing a few symptoms. In view of the initial evidence, a doctor can hypothesize about some possible diseases, but he does not make a decision at this moment. He collects more information and uses these information to verify and modify the obtained hypotheses, and then he makes the differential diagnosis. This process may go several cycles.

The IADI diagnosis model mainly uses two inference mechanisms, abduction and

deduction. In the diagnosis process this model also accompanies a structure analysis. Thus the IADI diagnosis approach involves three major processes.

The first one is the structure analysis. It detects a student's mistakes which are present in the program at the start of the IADI diagnosis process.

The second process is the abductive inference which infers a set of plausible candidate hypotheses from the student's demonstrated mistakes. Each hypothesis in this set represents just one candidate for explaining why the corresponding manifestation existed. It is possible that a hypothesis in this set is not the true cause for the student's mistake, and some other hypothesis may be the true cause instead. Therefore, it is necessary to further diagnose and decide which misconceptions in this set are more likely to be the true causes.

The third step, the deductive process, is the misconception verification. It verifies the focused hypotheses in the set of candidates by further checking the related manifestations, and also tries to decide if the unexposed mistakes exist or not. In the IADI diagnostic knowledge base, for each misconception there is a corresponding list which is a menu list to allow the system to interactively investigate the students' possible mistakes.

In the inference processes the plausibility measures are used to evaluate the possibilities of each misconception candidate. Finally a list of misconceptions ranked in decreasing order of plausibilities is given as the diagnostic result.

This model shows a nondeterministic diagnosis. It combines the features of abduction and deduction, detects both mistakes and misconceptions, emphasizes tutoring on students' misconceptions, and incorporates the process of instruction into the diagnosis process.

1.4 Dissertation Overview

This dissertation contains seven chapters. Chapter 2 surveys related work in intelli-

gent tutoring systems. The survey concentrate on the ITSs which teach computer programming and the diagnostic approaches in student modeling problems, and then summarizes three diagnosis models and presents the state of the art in student modeling problems. Chapter 3 presents the outline of the IADI diagnosis model including the characteristics of the proposed diagnosis model, the diagnostic knowledge representations, and the three main processes, structure analysis, abductive inference and deductive inference. Chapter 4 illustrates the application of the IADI diagnosis model in a recursive programming tutor. This chapter gives details on how to organize the diagnosis knowledge, how the three steps of the IADI diagnosis approach work, and how to measure the misconceptions using plausibility values. Chapter 5 describes the tutoring environment of the Recursive Programming Tutor, including the diagnosis environment and instruction environment. It is built on a hypertext model also combining other features. The tutor's performances are evaluated in Chapter 6. The evaluations are made from students' empirical tests. The last chapter summarizes this research work, followed by a brief discussion of future research direction.

CHAPTER TWO

RELATED WORK

Intelligent tutoring systems have been developed for many domains such as arithmetic (Brown 82), algebra (Anderson 90), geography (Carbonell 70), geometry (Anderson 90), indefinite integration (Chan 90), medical diagnosis (Clancey 82), electronic troubleshooting (Brown, Burton and de Kleer 82) and computer programming (Soloway, Rubin, Woolf, Bonar and Johnson 83). Different systems have their own emphases. This chapter surveys the related work in intelligent tutoring systems which teach computer programming, and the work in diagnosis approaches in student modeling problems. Then it summarizes the three different diagnosis models which can be classified and the state of the art.

2.1 ITSs in Programming Tutoring

Programming tutors have been explored for tutoring programming language BASIC (Barr & Beard 76), PASCAL (Johnson 84) (Soloway 83) (Woolf 84) (Bonar 88), LOGO (Miller 82), FORTRAN (Adam 80), LISP (Anderson 90) and PROLOG (Looi 88). The traditional programming tutor, such as BIP (which teaches the BASIC), can not diagnose the students' program when it was developed at beginning in 1975. With the pace of evolving from CAI to ITS, it added the diagnosis capability to improve the system's tutor ability (Barr and Beard 76). The most programming tutors are only able to work on a small problem domain with narrow programming language problem. The PASCAL programming tutors (Johnson 84) (Soloway 83) (Woolf 84) (Bonar 88) intend to work on

a rainfall assignment with tutoring of the knowledge about how to use the loop structure and the related variables. The PROLOG tutor (Looi 88) is targeted for the list reversion and list element number counting only. The LOGO (Miller 82) itself is an elementary programming language. Even so, the efforts provided in these researches made significant studies from various perspectives, especially in the AI technical development and cognitive process.

In this section, several tutoring systems that involve programming teaching will be reviewed. MENO-II, PROUST, MENO Tutor and BRIDGE come from MENO project, which started in the late seventies. This project attempts to build an intelligent tutor for novice Pascal programmers. Its goals were to diagnose nonsyntactic errors in programs, to connect these bugs to underlying misconceptions and to tutor the student with respect to these misconceptions. After the first system MENO-II (Soloway, Rubin, Woolf, Bonar and Johnson 83), developed at University of Massachusetts, the project branched into at least three directions leading to different doctoral dissertations. Johnson and Soloway moved to Yale University and developed PROUST system to study the bug diagnosis from a new angle (Johnson and Soloway 84); Woolf and McDonald developed system MENO Tutor by utilizing a discourse management network (Woolf and McDonald 84); Bonar developed system BRIDGE with an intermediate representation to give students specific mental models to support their problem solving process (Bonar 88).

The LISP tutor, which has been developed over 8 years at Carnage-Mellon University by John Anderson and his colleague, is a good vehicle to teach student LISP programming with cognitive model.

The APROPOS2 was developed in the department of AI, University of Edinburgh. The bug analysis of a PROLOG program in APROPOS2 has been divided at three levels of abstraction, which gives a clear top–down structure.

2.1.1 MENO-II

MENO-II is a diagnostic system that specializes in the analysis of loop structures and related variables in Pascal program (Soloway 83). It consists of two major components: the Bug-Finder and the Tutor. The Bug-Finder parses a student's program into a parse tree that is matched against a simple description of the solution. This is done with the help of specialized knowledge about types of loops and corresponding plans, as well as a library of known bug types. In the bug finding process, there are four stages. In the first stage, the student's program is parsed into a parse tree. The second step is to annotate the parse tree with useful information about the various nodes. During the third stage of the process, the Bug-Finder searches for instances of the various programming plans. Finally, the Bug-Finder searches the Bug Catalogue in the bug library for matches. If a bug is discovered, the tutor then analyzes it by a set of specific inference routines that suggest possible underlying misconceptions.

MENO-II detects the bug inefficiently because it analyzes bugs locally in a context-independent fashion by means of simple template matches, MENO-II can not cope with the complexity of the programming process and with the extraordinary variability in programs.

2.1.2 PROUST

PROUST (PROgram Understander STudents) system tries to identify the nonsyntactic bugs in students' Pascal programs (Johnson and Soloway 84) (Wenger 87). PROUST is an expert at finding bugs in program of the rainfall assignment written by the students. This assignment is to calculate the number of rainy days, find the maximum rainfall on any one day in the period, and average rainfall from an input stream of rainfall values.

Johnson and Soloway believe that diagnostic methods, that look for bugs in programs, merely by inspecting the code can not cope with the variability in novices' programming. They propose an intention-based diagnosis approach which is to construct a

coherent model of what the programmer's intentions were and how they were realized in the program, and to identify errors in these intentions or in their realization based on this intention. To analyze a student's program, PROUST builds goals to understand problem specifications which provide a starting point for identifying the student's intentions, and uses a knowledge base of plans which might be selected by students to realize those goals. For novices, they may not have a clear concept about what kind of plan can be used to reach the goal; they only have an intention to satisfy a goal. Hence, the theme in PROUST is to analyze the intention of a student from the goal that is the problem specification and the solution that is submitted by the student. For example, the main goal of the rainfall assignment is to average the rainfall. PROUST recognizes that an iterative looping plan is required to achieve the subgoal of the main goal. The loop must collect the rainfall values, sum them to calculate the running total and count the number of days. After that, the running total must be divided by the counter to obtain the average. PROUST thus sets up an agenda of goals and attempts to match each of them to the student's code. When all the goals on the agenda have been successfully matched with the student's code, PROUST has understood the student's program because it knows how the student achieved, or failed to achieve, each of the subgoals. Each of the failures is understood by PROUST as a bug. Thus, according to the design stages mentioned above, the diagnosis process works on three layers. The top one consists of the various possible specifications decomposed into goals and subgoals, then the plans that could be selected as implementation methods for each goals or subgoals, and finally the different realizations in which plans can match the code.

PROUST system is an off-line tutor that has access only to a final product or state on which to base its diagnosis of student errors. The complete student programs are submitted to PROUST, which provides a printout of the diagnosis results and corresponding explanations. The designers of PROUST feel that there is a lack of the sophisticated pedagogical expertise; they are remedying it with using the information about

each student's errors to guide the presentation of course material. The real tutor will appear in another system BERTIE (Johnson 86).

2.1.3 MENO Tutor

MENO Tutor tries to capture the discourse strategies observed in human tutors. It works on two domains, rainfall study and Pascal programs. In Pascal program, it just works for Pascal looping concepts (Woolf and McDonald 84).

Besides providing Bug-Finder for detecting the semantic errors, MENO Tutor develops a general framework: Discourse Management Network (DMN), which is a kind of augmented transition network. It also provides a good interface to communicate with students. Fig.2. shows a dialogue in MENO Tutor for a buggy program. The buggy program is for averaging problem, which is in left of the figure.

The most important contribution made by MENO Tutor is the DMN net. The DMN is a set of decision units organized into three planning levels that successively refine the actions of the tutor. The nodes or states correspond to tutorial actions and these nodes are organized hierarchically. The highest level is pedagogic states. In this level a specific tutoring approach will be determined, such as the diagnosis of student's knowledge or introduction of a new topic. At the second level, strategic states, the pedagogy is refined into a strategy, specifying the approach to be used. At the lowest level, a tactic is selected to implement the strategy.

When the system is in the tutoring mode at the pedagogical level, and an incorrect answer from the student has triggered the strategic action, such as teaching her some pieces of data, many actions at the tactical level can then be chosen to accomplish this, such as giving a corrective statement for her answer, making a general statement about some relevant facts, or questioning the causal dependencies underlying her assertion. The path in the hierarchical structures is default path. MENO Tutor provides meta-rules to create preemption path, which occurs in the same level. The preemption path can

Student's Program

```
1. program average (input, output);
2. var Sum, Count, Data: integer;
3.     Avg: real;
4. begin
5.     Sum := 0;
6.     Count:= 0;
7.     read (Data);
8.     while Data < > 99999 do
9.       begin
10.        Sum:= Sum + Data;
11.        Count:= Count + 1;
12.        Data:= Data + 1;
13.      end;
14.    Avg:= Sum/Count;
15. writeln ('The average is ', Avg);
16 end.
```

Tutor: Do you know that Data in line 8 is a control variable for the while loop?
Student: Yes.
Tutor: Does that mean that Data should be
a) Initiated before the loop?
b) Added into Sum within the loop?
c) Modified within the loop?
d) Both a) and c).
Student: c.
Tutor: You seem to understand about the control variable of the while loop. Now let us talk about some other command statements. What do you think read (Data) in line 7 does with the value of Data?
a) Reads all the values of Data at onetime.
b) Reads in a single value of Data.
c) Assigns several new input values to Data.
Student: a.
Tutor: Then do you think that Data:= Data + 1 updates the next of those values?
Student: Yes.
Tutor: There is your misconception. Adding one is a common way to step through a range of values. But it cannot apply here because your read instruction in line 7 retrieves only a single value. You should replace line 12 by a second call to read (Data).

Fig.2 The Buggy Program and A Dialogue On It

obviously provide more in keeping student history or discourse history.

2.1.4 BRIDGE

BRIDGE system intends to build a bridge between novice to experienced programmer. Bridge tries to find natural evolving stages in the development of plans when students design, implement and test their program, and to articulate each stage explicitly by providing intermediate representations. The intermediate representations allow a students to directly represent design and partial complete programs (Bonar 88).

BRIDGE consists three phases. Phase I provides informal English description for

the program. It displays a English Phrase Menu and the submenus for each phrase. The student user chooses the phrases to construct their natural language "program" which is very much like the sudocode. This solution will be shown on the "English Step-by-step Solution" window. The phase II is to refine the informal description of phase I into a series of semi-formal programming plans. In this stage, the English phrases become explicit plans, represented by icon, and displayed in a "Visual Solution" window. Bonar argues that since plans represent high-level programming objects, it is sensible to depict them as icons that suggest their function. The phase III is to translate the plan-based description into actual Pascal code. The student's task is to match each visual plan icon to one or more Pascal statements. After one icon and a Pascal language construct are selected from the "Visual Solution" window and from the pop-up menu separately, the corresponding statement will be shown in a "Pascal Solution" window.

In the phase I and phase II, BRIDGE tries to understand the student's partial work and diagnoses this work. The basic diagnostic strategy is matching student's plan to a particular student model. The English phrases in the solution from phase I are mapped to the catalog of programming plans. In BRIDGE, there are four student models corresponding to four different looping strategies. These models are specified at four different levels. The students' performances will be matched with one of the models, then compared with a list of requirements for a correct solution to the problem. For the incorrect solution, BRIDGE will give hints and suggestions.

BRIDGE shows its method by solving the "Ending Value Averaging" problem, which is a problem of reading data and calculating their average by using loop. BRIDGE was used by approximately 40 students. They suggest that BRIDGE is helpful.

2.1.5 LISP tutor

LISP tutor is built for testing Anderson's cognitive model ACT* (Adaptive Control of Thought), a successor of ACT. ACT* theory has many assumptions, two of them are

very important to our ITS. The first one is that cognitive functions can be represented as sets of production rules. The second one concerns the mechanisms of the learning model. The learning model consists of a set of assumptions about how the student's knowledge state changes in each step during a problem-solving phase. This model is employed in model tracing diagnosis approach in LISP tutor (Anderson, Boyle, Corbett and Lewis 90).

The declarative knowledge and the procedural knowledge are well organized in the tutor system. The declarative knowledge is what is deposited in human memory when someone is told something as in instruction or reading a text. The declarative knowledge is organized in schema-like structures, the PUPS structures (PenUltimate Production System). The procedural knowledge is represented by a set of production rules that define the skill how to solve a problem. The idea is that knowledge is first acquired declaratively through instruction, and that it has to be converted and reorganized into procedures through experience. Only then can it be usefully reflected in behavior. The tutor's task is to help students to acquire the production rules which would be possessed by the competent problem solver.

LISP tutor can help students to write basic LISP code to solve the problems that appear in an introductory LISP textbook. The LISP tutor presents the ideal programmer's knowledge and novice's knowledge in the form of production rules, the ideal rule and buggy rule respectively. The system gives the student the instructions in LISP and tries to bring the student into specific problem solving situation. The tutor provides assistance to student essentially by running the model in synchrony with the student, comparing the student's response at each step to the relevant ideal and bug rules and responding accordingly. This is the model tracing diagnosis technique. A salient feature of LISP tutor is to get the student to mimic the steps of an ideal production model. Every time the student enters one new line of code, the tutor comes back with a response. Once the student makes an error, or the student choses a suboptimal strategy,

the system attempts to diagnose and give a hint as to the correct and optimal solution so that he can change his solution immediately.

The LISP tutor has been in use in an introductory Lisp course at Carnegie-Mellon University since the fall of 1984 (Corbett et al. 1990). It has been tested by the students in classroom and the results are very encouraging. Two groups of 10 students were given the same lectures, but only one group used the tutor.The tutored students spent 30%less time on the problems than those working on their own, but scored 43% better on the test.

2.1.6 APROPOS2

APROPOS2 stands for Analyser of PROlog Programs Of Students. It is a program analyser for a PROLOG programming teaching system. APROPOS2 detects and corrects nonsyntactic errors in student PROLOG programs written to do simple but nontrivial list and number manipulation tasks (such as list reversion and count of the atom number in a list) (Looi 88).

The bug analysis in APROPOS2 has been done at three levels of abstraction: the algorithm level, the predicate definition level and the code level. The analysis at the algorithm level checks the different kinds of design for the task solution. In APROPOS2, for example, three algorithms for list reversion can be recognized. The student's PROLOG program is matched against a library of task algorithm. The analysis at the predicate definition level detects the different types of bugs in the predicate definition of the chosen algorithm, such as missing, extra and incorrect predicate definition. The work at the third level, the implementation level, is checking the code that implements each predicate definition. The bug analysis is done from the top level, the algorithm level to the down level, the code in students' solution. After the analysis, the system will give a report for the result such as which statements are correct, and which might be wrong and the relevant reason.

APROPOS2 uses a heuristic best-first search strategy to map the student's program to a prechosen algorithm, a set of predicate definitions and the code of the program. APROPOS2 combines the dynamic and static analysis as its debugging approach. The static analysis works on the code. The result in this analysis is finding the common bug and suggesting corrections before dynamic analysis is invoked. The dynamic analysis here means to examine the running of the program to get solutions both for student's program and the correct program. The student can compare these solutions and gain some hints.

APROPOS2 has been work on at least 95 students' programs. The result shows that it can correctly detect most bugs, up to 80% (looi 88). Only very few program can it not give the correct answer, since it is possible that there are some disguise in the clauses of programs.

2.2 Diagnosis Approaches

The diagnosis has been one of the major subjects of research in Artificial Intelligence in both the theoretical and the practical area. The diagnostic problem was disposed in many different domains, from trouble shooting in electronic appliances and circuits, to diagnosis of complex mechanical or physical system, to medical diagnosis. The diagnosis process in Intelligent Tutoring refers to collecting the information about the student's activities and inferring his understandings. It is more difficult to deal with than diagnosis in many other expert systems in general, such as device diagnosis, because the object of diagnosis in an ITS is a series of the student's abstract mental states. Since this task often involves the construction of a student model, these activities have also been called student modeling problem. The student modeling problem is raising up to a prominent position. Diagnostic approaches in the existing ITSs are varied with different perspectives in different systems. Plan recognition, model tracing, issue analyzing, condition induction, generate and test, and decision tree have been received much atten-

tion. In this subsection these diagnostic approaches are briefly reviewed. In next subsection, these approaches are further compared and summarized into three different models.

2.2.1 Model tracing

The model tracing approach attempts to track the student's performance across problem at every mental step. The model tracing is based on the psychological assumption that human cognitive behavior can be modeled as a production system. The production rules are viewed as the description of the unit of skill or the prediction of the steps the students will take (Anderson 90).

When a student is working on a problem the tutor generates all the possible next steps, both correct and incorrect. The tutor will display a menu and let student choose one item, which is one step corresponding the student's next action. The tutor assumes that the student use the relative production rule to generate the next particular mental state. After comparing the student's response to the internal expert model and the prestored erroneous actions, the tutor is able to recognize whether the student is on the correct solution path or not, and gives the necessary interpretation for student's activities. The tutor monitors the student's responses essentially on a symbol-by-symbol basis. If the student's half typed code seemly can match one of the templates provided by the system, the system will allow the student to continue without interruption (Corbett, et al. 1990).

The model tracing method intends to create a close correspondence between units of the internal model built by the professional programmer and steps obtained from the student behaviors. This approach has its advantages. Anderson uses this technique in LISP, ALGEBRA and GEOMETRY tutor and call it model tracing to express the fact that the student is made to follow the system's model quite closely and the student will know how to correct the mistake when the student behavior deviates from the ideal path.

Under this model the student never strays far from a correct solution. So, the model tracing approach is good for both diagnosis and guidance.

The model tracing method is based on the goal-restricted production system architecture. The straightforward scheme in LISP tutor restricts the student's activities into a local area and may not fit the need for more complex program analysis. The explanation during the intermediate stage is useful sometimes, especially for reminding the students who are in the early learning stages. But in the programming setting, important errors are usually revealed only after an unsuccessful execution occurs, and only when the student see this result, he can get a deep impression.

2.2.2 Condition induction

The model tracing approach rests on an assumption that for any two consecutive mental states in the student's problem solving there is a rule in its model to connect these two states. ACM system (Automated Cognitive Modeler) uses condition induction approach to construct a rule from one state to the other between two consecutive states instead of storing all the rules in advance (Langley and Ohlsson 84).

The ACM system works on the subtraction domain. The system constructs a set of production rules from the description of the problem states (condition) and the behavior of a student (operator). The operator converts one state to another; the condition in the same rule should be consistent with the applicability of the operator. The conditions and operators are stored in two libraries respectively. For a given problem, there is a "problem behavior graph" (actually it is a search tree) in which each node is a state yielded by one operator. This tree includes all possible states either by correct performances or incorrect performances. ACM intends to decide which path can match student's behavior. Given a student's answers on a set of problems, ACM starts with a set of production rules and then uses a discrimination process to determine which sequences of operators have led to the student answers. For a particular problem, the states lying on the

student's solution path corresponding the positive instances of the rule, and the states are set one step off the path corresponding the negative instances. In order to distinguish the positive instances from the negative ones, therefore lead the path to match the observed behavior, ACM infers additional conditions (beyond the original ones) from a list of tests. ACM provides ten potentially relevant tests such as whether one number was greater than another, whether one row was above another. Once inferred and selected the best matched tests adds them as the appropriate conditions to each of these rules then recombines the more specific rules into a final model. This model simulates the student's subtraction strategy and explains the student's behavior since the rules in this model can reproduce these sequences on the same problems.

2.2.3 Issue analysis

The model tracing is based on the assumption that all of the student's significant mental states are available to the diagnostic program. If the bandwidth is not high enough, the model tracing can not be used. Issue analyzing is based on analyzing a set of issues rather than considering accurate psychological stage of a student.

The on–line coach system WEST (Burton and Brown 82) uses issue analyzing method for elementary mathematics study. It teaches the students arithmetic in an environment of a mathematics game in which the student tries to move to the goal position as fast as possible. The number of moves will be indicated by the value of the arithmetic expression. The plus, minus, times, divide operations, and using parentheses in the expression and other game skills are presented as issues. WEST analyzes these issues embodied in both the students' move and the expert's move. In the game the student organizes three numbers given by the spinners into an arithmetic expression to determine the number of a movement. The expert generates an ordered list of all possible moves in the same circumstances, that is for the same three random numbers and the same starting position. If the student's expression does not produce the expert's optimal move, WEST

starts to diagnose what issues the student is weak to handle. WEST uses issue recognizers to analyze the student's moves and to identify which issues have been used. The issue recognizers also analyzes the expert's moves to select combinations of given numbers and the issues which are better than student's, and to provide a list of issues the student did not apply. From the list, WEST summarizes the issue which are being considered as the weakness of the student. Then the coach provides a prestored text to explain expert's strategies and also gives an example to show the expert's intelligent move.

In the issue analyzing approach, the diagnosis process concentrates on the issue analysis. If a student writes an expression 2*4 + 1, then the system will analyze the times issue and plus issue and check whether they are appropriate in the context of the moves. Issue analyzing approach ignores the choices what the student was trying to do before he decided to make that move, while those choices might be the stages which be fully considered in the model tracing approach. For the issues in WEST, there is no concept of error issues. They are only identified by the differences how often each issue was used properly and how often it was overlooked. The differences are recorded in two counters, used and missed, of each issue. Later on the system can find the student's weak issues based on comparison of the two counters. The diagnosis in WEST is based upon the analysis of the separated issues. It does not deal with the relations of multiple interactive issues.

2.2.4 Plan recognition

Plan recognition uses plans as models to capture the experts' decision and follow the students' attempts in problem-solving. This approach formalizes the human tutor's insight using the plan, the essential ingredients of program design experience. The systems SPADE (Miller 82), BRIDGE (Bonar 88), PROUST (Johnson and Soloway 84) and MACSYMA ADVISOR (Genesereth 82) use the plan recognition diagnosis method.

The early work on plan recognition was done in SPADE (Structured Planning And DEbugging) project which built a programming tutor for interpreting simple picture programs in LOGO and presented a theory of planning and debugging. This theory explores at least three hypotheses. The first one is that the problem-solving behaviors can be described by a series of plans. The second is that bugs can be represented by the consequence of incorrect plans under incomplete knowledge and limited resources. And the third one is that articulating the plans (one's problem solving strategies) facilitates learning. This is the AI contribution to the diagnosis (Miller 82). The plans in SPADE simulate the human tutor's design choices and are classified by a taxonomy of concepts of program design (Fig. 3). SPADE models planning strategies into three categories at the

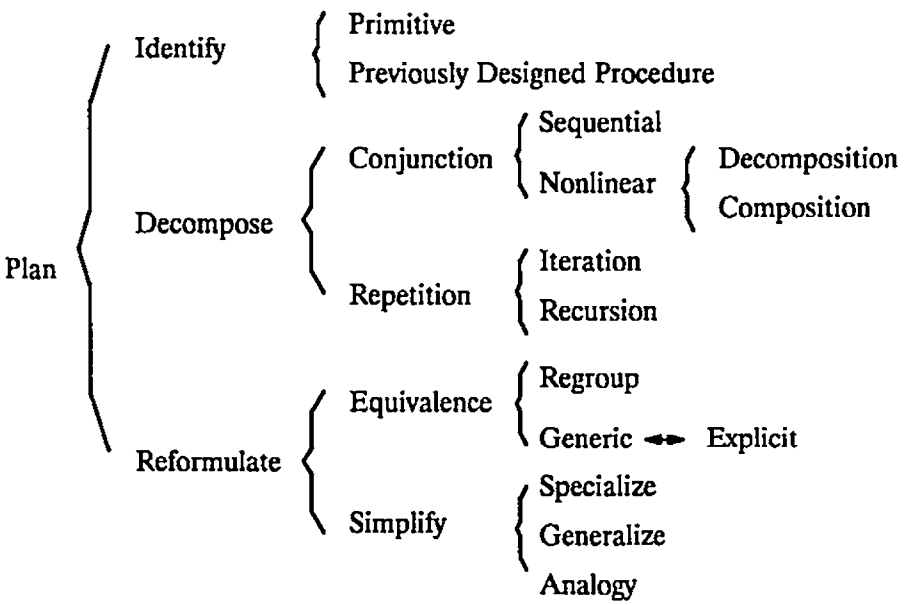

Fig. 3. Taxonomy Of Plans In LOGO

most general level: identification with previously solved problems; decomposition into simpler sub-problems; and reformulation of the problem descriptions. For each strategy, the system provides more details (refer Fig. 3). The design choices are indicated by the

plans in the hierarchical representations. And the hierarchical structure forms a plan tree. The student's developmental plan of the program code are edited by the system according the plan tree. Diagnosis process in this model is depicted by the search in the plan tree. The root node in the tree is the overall goal, nonleaf nodes are the subgoals. The diagnosis task is to infer an incorrect design choice when it is given the student's action as the leave node of the tree. Using plan to make the decision process explicit would encourage a good design strategy. Although SPADE only implemented at the stage of plan-oriented programming editor, its idea of using plans shows basic step towards to building a good tutor.

The plan recognition in PROUST is as an intermediate stage in the diagnosis process. Johnson made the claim: programmers use programming plans not only in understanding programs, but also in writing programs. When programmer write programs, they need to determine what goals must be satisfied, and then select plans which satisfy those goals, although sometimes they choose incorrect plans (Johnson 86). PROUST's main analytic task is to locate in the student's code the plans for each of the goals in the problem specification. PROUST has a list of plans related with the goals they achieve. Plans in the list include the expert plans, buggy plans and some information about misapplication of correct plans. Each plan is represented by a frame which contains a set of slots. The most important part of a plan is described by the Template slot which gives the structure of the plan template. The basic building blocks of plan templates consist of one Pascal statement or a set of Pascal statements because each statement usually can represent a primitive unit of action. The other slots provide various additional facts and assertions about the plan. In the rainfall assignment, for example, accumulating the counter and running the total are necessary to calculate the average. So the program will include corresponding plans, "counter plan" that consists of statements "Count := 0" and "Count := Count + 1", and "running total plan" that consists of statements "Sum := 0" and "Sum := Sum + New" (refer Fig.2 and Fig. 4). Plans are used in PROUST

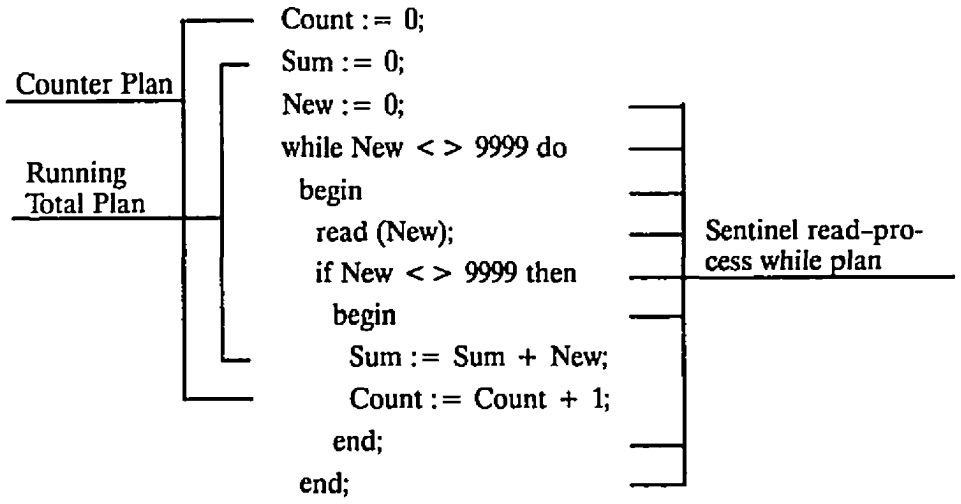

Fig.4 Plans In Program Code

in following way. From program specifications, PROUST selects a goal to be analyzed. It retrieves a set of plans from the plan list, each of which might be used by novices to implement the goal. PROUST then matches each plan against the program as much as possible. The matching process from goal to code can describe the student behaviors, either correct or incorrect. If we consider the plans in SPADE play a role for the program editor to help student to make a good program design, then plans in PROUST would play a role of diagnosis to help tutor to find the error.

The plan in BRIDGE (Bonar 88) is presented in an explicit environment during the period when students solve their problems in three phases as described in section 2.1.4. Bonar asserts that plans play central role while a student constructs his program from informal description of natural language to the formal statements of computer program and the plan be viewed as a mental model and can be presented by visual icons. From macroscopic point of view the plans are used in different way in BRIDGE and PROUST. BRIDGE uses plans to map the student's intended goal into code to help student with writing code while PROUST to map the code back into the student's original goal to infer the student's design process and give the interpretations.

2.2.5 Decision tree

The decision tree technique is working in a deterministic manner and targeted for accurate diagnoses in the problem with low bandwidth. BUGGY system uses this technique to diagnose the student's subtraction errors after the student's answers to a set of test problems are given. It attempts to determine what internalized set of incorrect instructions or rules gives the student's solutions rather than only judge whether the answer is incorrect (Brown and Burton 78).

In the BUGGY system a set of subtraction tests that will be given to students are selected in advance. BUGGY preanalyzes these subtraction test and constructs a decision tree. BUGGY also terms this decision tree as a procedural network that represents the application of the possible skills of subtraction and connects elementary subskills into a network. The top node of the tree corresponds to the first subtraction problem. The answers to the problem, which are made by every possible bug as well as the correct subskills, form the child nodes. Each node is associated with one of the bugs or the correct subskill which produce the same answer. From all of these nodes, the next test problem and corresponding answers under the possible bugs and the correct subskill will form next level of nodes. A diagnostic process based on this decision tree technique contains all the necessary subskills for the global skill, as well as all the possible bugs of each subskill. Each leaf node corresponds to one diagnosis and each diagnosis process may correspond to several paths from a root to a leaf. Thus the decision tree can provide the explanations for the student's incorrect behaviors with a set of exact internalized errors which are translated by the subskills. BUGGY system views the compound bug as one bug. This is too expensive when the combinatorial situations of primitivrebugs aod considered.

2.2.6 Generate and test

It is necessary to consider that the student has several bugs at the same time. In the decision tree method, only the single bugs and possible pairs of bugs are considered. Even then, for 55 bugs, there already have been 55^2 (3025) possible bug pairs. Thus the decision tree will be a huge tree. DEBUGGY system uses generate and test diagnostic method to generate compound bugs dynamically instead prestore all possible bugs and their combinations into a tree. It can deal with up to four or five multiple co-occurring bugs (Burton 82).

DEBUGGY system generates an initial set of bugs by testing the student on the set of given problems. Every bug in this set replaces a subprocedure in the correct model and predicts an answer for each given problem, which will be tested against the student's answer. If any one of those answers can match one of the student's answers, then it explains at least one of the student's wrong answers. Usually, there are several bugs can be generated from the student's answers to the initial given problems. Then the system tries to decide which one will be chosen as the candidate of the compound bug and which one is not necessarily to stay in the set of single bugs. DEBUGGY removes the bugs that are completed subsumed by other single bugs in order to reduce the size of the set of single bugs. DEBUGGY then combines these single bugs to form the compound bugs. The system predicts the answer for those compound bugs and compares them with the student's answers. Finally, it tests all of the bugs and selects the ones that best match the student's answer. Choosing the compound bugs which can explain more answers than its parts can do is also considered. This method needs to test a large sets of data to get the accurate or complete diagnostic result.

DEBUGGY and BUGGY work with a predefined subtraction test and the student's answers to it. IDEBUGGY provides an interactive environment (Burton 82). After obtaining a set of diagnoses from the student's answers, it generates some new subtraction problems by using problem generators. The generators are designed to produce problems that are made to meet certain conditions, such as requiring borrowing, having

zero on the top. These problems will be expected to split the bugs in the current set of diagnoses. Then it will find the more accurate diagnosis as the result. In this process IDEBUGGY yields fewer test problems than DEBUGGY and BUGGY do to achieve the same accuracy.

2.3 Summary on the State of the Art

A considerable amount of research has been devoted to ITS with respectable results. These systems can vary greatly in the type of pedagogical approach they implement and even in the technology they adopt. Most systems built so far just show laboratory experiments primarily intended to demonstrate feasibility; only a few of them have been put into applications like LISP tutor (Anderson 89), GUIDON (Clancey 82), STEAMER (Hollan 84). It is obviously that research on ITS is still very young. It has a long way to go to produce actual intelligent systems for helping people acquire various forms of knowledge. It needs to explore more methods from different prospectives so that people can compare and choose the most reasonable and realistic one.

In ITS research most significant effort has been devoted to the student modeling problem. By virtue of student model and diagnostic process, intelligent tutoring programs can represent what a student does and does not know about the subject matter and can focus teaching and tutoring on the specific needs of an individual student. In this process many fundamental cognitive science issues have to be addressed also, otherwise the teaching and learning can not go further to deeper stage. The student model is very important so that some researchers think that an ITS simply contains a student model and three experts, the domain expert, tutoring expert and diagnostic expert. All of the three experts communicate with student model (Wallach 87).

Although the existing diagnostic approaches in current student modeling problems are based on different formalisms for knowledge representation and different scheme for diagnosis process, they fall roughly into three models according to their diagnosis styles:

the enumerating model, the tracking model and the classifying model. All of these models have both reasonable sides and weak sides.

• **Enumerating model**

The diagnostic techniques such as decision tree (Brown and Burton 78), and generate and test (Burton 82) belong to the enumerating model and they are used for diagnostic problems with low input bandwidth (Vanlehn 88) like a final state or intermediate states. In these approaches the definite answers to a given problem, such as the result of a numerical subtraction, can be obtained from both the student and the domain expert. For every possible incorrect answer given by a student, the system attempts enumerate all possible bugs in an enumerating model. This model can work well when the problem domain is very simple, but it is not suitable for tutoring complex problems that require the diagnosis of abstract mental states. The mistakes in complex problems are not necessary enumerable. They are from students' cognitive activities, and usually they are not definite as the answers for certain simple mathematics problems.

• **Tracking model**

The diagnosis techniques such as model tracing (Anderson 90) and condition induction (Langley and Ohlsson 84) belong to the tracking model. They try to track students' behaviors at every mental step to deal with the diagnosis in more complex problems. This approach has an inherent problem. The human mental activities are not always discernible, and it is very difficult to accurately seize a student's misunderstanding at each step of a problem solving process. In problems at higher levels, obtaining the accurate mental states is simply not feasible, even though it can be very desirable. On the other hand, the tracking model only allows a student to follow the steps given by the system's model and forces the student to solve the problem in the manner that the tutor uses. Therefore the student has no chance to explore and evaluate alternative designs.

• **Classifying model**

The issue analysis (Burton and Brown 82) and plan recognition (Johnson and Soloway 84) can be categorized as classifying models. The techniques under this model characterize significant issues, plans, goals and strategies, and then try to recognize them according to predefined expectations during the student's problem solving process. Based on this classification, the system can interpret the student's behavior most plausibly on the cognitive level. In contrast with the methods under tracking model, these methods encourage student understand the design process on the high level of mental activities, give students the chance not only for response to the states on sentence by sentence basis which is a quit weak view of student problem-solving. Although these approaches under this model seem to be more realistic, there are some problems which need to be solved, such as how to choose the proper issues when there exist complex relationships among issues and how to choose an accurate plan when several plans can be used to serve the same goal.

There exist other views about building and using student models among researchers. They take a sceptical attitude to both the need for detailed student models and the practical possibility of constructing them. The difficulties to build such complex student models and the preconceptions about the potential roles of student model make them doubt whether the cost of building runnable and maintainable student models is worthwhile in terms of the gain in teaching efficiency (Sandberg 87).

Self restates the importance for student models in intelligent tutoring systems and presents the possibility of actually constructing them (Self 88). He states that it is not essential for ITSs to possess precise student models and contain detailed representations of all the components. Self suggests changes in philosophical approaches, such as having the student tell the system what the system needs to know instead guessing, and not diagnosing what the system can not treat. He claims that solutions for some aspects of

the student modelling problem are practically attainable and useful if we back off from the grand vision and adopt more realistic aims.

Based on analyzing the current diagnosis models in ITSs, we can see all of these models are not adequate, they need to be improved. A new diagnostic model, Integration of Abductive and Deductive Inference (IADI) has been explored. This model intends to step forward the human diagnostician's inside, to simulate the human's reasoning ways, therefore avoid disadvantages of existing diagnosis techniques in intelligent tutoring system and synthesize their advantages.

CHAPTER THREE

GENERAL MODEL

OF

INTEGRATION OF

ABDUCTIVE AND DEDUCTIVE INFERENCE DIAGNOSIS

In light of the cognitive processes of human diagnosticians, a new diagnosis model, Integration of Abductive and Deductive Inference (IADI) model, is derived. In contrast with the other three models summarized in last chapter, that based on enumerating, tracking and classifying approaches, the IADI diagnosis model is an inferencing model because it relies on different inferences to solve the diagnosis problem.

Just as Pople indicated in 1973 (Pople 73), "The principal deficiency of existing systems is their reliance on a single form of logical inference -deduction- which, though essential, is inadequate for many types of problem solving activity." Different mechanisms of inference can be complementary to each other when they are combined appropriately, so that the capability to solve a problem can be greatly improved. Thus integrating different types of reasoning is becoming one of the main characteristics of second generation expert systems (Torasso and Console 89). The IADI diagnosis model combines abductive inference and deductive inference to simulate the human diagnostician's cognition in diagnosis problem solving. The IADI diagnosis model as a general diagnosis approach was proposed and applied in a prototype of the recursive programming tutoring system. This chapter describes the basic concept of abductive inference and deductive inference, and presents the outlines of the IADI diagnosis model includ-

ing its basic features, the diagnosis knowledge representation, and three steps in the IADI diagnosis process. The next chapter explains how to apply the IADI diagnosis model to solving a real problem.

3.1 Abductive and Deductive Logic Reasoning

Philosophers have considered abduction as a distinct type of reasoning from the two traditionally recognized types, deduction and induction (Fann 70), (Pople 73), (Charniak and McDermott 85). The various forms of logical inference can be characterized as following different models.

- induction Given { P(a) = q, P(b) = q, ... } (1)
  Infer ($\forall x$) P(x) = q. $x \in$ { a, b, ... }
- deduction Given { ($\forall x$) P(x) $\longrightarrow$ Q(x) , P(a) } (2)
  Infer Q(a).
- abduction Given { ($\forall x$) P(x) $\longrightarrow$ Q(x) , Q(a) } (3)
  Infer P(a) as possible explanation.

Deductive inference is to infer the result for a specific case after being given the general rule (general knowledge) and a specific case (conditional evidence). If the general rule and the conditions are true, then the result, a logical consequence of the given data, is definitely true. In the deductive reasoning process there are two different control strategies: the forward chaining reasoning process and the backward chaining reasoning process. IRIS is one system that works in forward chaining style. It applies the given data to check the conditions of a rule to determine whether the consequence can be drawn or not (Trigoboff and Kulikowski 77). Some systems, such as MYCIN which is based on deductive reasoning, work in a backward chaining fashion. This mechanism builds sub-

goals to prove. It checks the condition part of a rule. This process might go several levels deep. That is, proof for a subgoal may involve several rules. If one of the subgoals turns out to be fail, then the result is false. When deductive reasoning is applied to diagnosis problem solving, the uncertainty measures are usually connected (Shortliffe 76) because diagnosis features require plausibility. Actually this kind of reasoning already deviates from purely deductive reasoning.

Abduction or abductive inference is to infer the best or most plausible explanations for a given set of facts (Pople 73). This reasoning can be characterized as non-monotonic reasoning (Bundy 90) (Geffner 89). One of the important features of abductive reasoning is that abductive reasoning is plausible reasoning. After being given the general rule and a specific case (observed facts) like Q(a) in above (3), abductive inference infers another specific case such as P(a) in (3) as one possible hypothesis or explanation for the given case. The inferred explanation may not be definitely true because we can not conclude that P(a) is certainly true from the given data in the above formula. From other rules, if any, it may infer other hypotheses that would also explain the observed facts. In other words, abductive reasoning is a process which infers a set of the most plausible hypotheses for the given evidences.

Although the term "abduction" was first introduced into the AI literature around 1973, abductive inference has not received much attention from the AI field even though abduction logic reasoning is more suitable for use with many AI systems than other kinds of logic reasoning. Not until very recently did researchers realize that abduction is valuable and can be applied to many areas such as machine vision, natural language processing, legal reasoning and plan interpretation, and especially diagnosis problem solving because human diagnostic inference naturally falls into the category of abduction (Charniak and McDermott 85) (Peng and Reggia 90).

Reggia and his colleagues proposed the Parsimonious Covering Theory which is based on the Set Covering model, for diagnostic problem solving in the abductive expert

system (Reggia et al. 85 a, b, c) (Peng and Reggia 90). Their research is an attempt to build a formal model of abductive inference in a mathematically rigorous fashion for diagnosis problem solving. This model infers the minimal set of hypotheses as the diagnostic result which can cover the given manifestations (symptoms). Although this model captures some desirable features of diagnostic problem solving, it is not necessarily appropriate to have the minimal cover as the diagnostic result. There are some situations where a minimal cover would not be the most plausible explanation for a set of manifestations. For example, there are two sets of plausible explanations in a medical diagnosis, both of them can account for all of the given symptoms. One set contains two common diseases, another contains one rare disease. A physician would rather choose two common diseases as the differential diagnosis than choose the minimal cover which is the rare disease. Hence, having the minimal cover of the given symptoms as the diagnosis result may not be appropriate, especially when we consider the learning process as a cognitive process. On the other hand, this theory treats the hypotheses in the minimal cover with equal weight. It seems that there always are a few, and only a few, hypotheses in the minimal cover which are critical. Human experts usually weight the critical hypothesis more than the less critical ones. Hence, it is worth the study on which one is better, choosing the minimal cover for the given manifestations, or choosing the critical ones as the explanations.

3.2 A View of the IADI Diagnosis Model

Due to the nature of the real human cognitive process in diagnosis problem solving, it is suitable to use abduction as the first reasoning step to hypothesize misconceptions from mistakes, and then use deduction to verify these hypothesized misconceptions. Thus the two processes form an integrated inference process which works at detecting on two levels.

This section represents the basic characteristics of the IADI diagnosis model which

include the two-level detection, the integrated inference mechanism, the rules distinction, the nondeterministic feature, and the three diagnosis steps in the IADI diagnosis process. A detailed description of the three diagnostic steps and their applied knowledge will be given in the following sections. How the IADI diagnosis model works in real systems will be described in next chapter.

3.2.1 Detection at two levels

The purpose in diagnosis problem solving is to find the disorders or mistakes in order to provide a foundation for the system to remedy or correct anomalies in one way or another. In a tutoring system, besides detecting the student's actions to provide a starting point for a tutor to give instruction, the diagnosis process should also be able to detect a student's knowledge of a certain subject. An understanding or a misunderstanding is viewed as a mental activity that is used to interpret people's behaviors. Therefore, knowing the student's knowledge will enable the tutor to give more rational explanations. Thus, it seems to be necessary to diagnose at two levels, action and understanding.

In a tutoring system a mistake is a student's incorrect action which is usually demonstrated when he performs in a problem-solving process, while a misconception is a student's misunderstanding at the conceptual level. Usually the misunderstanding of a concept about the subject material is the reason that students make mistakes. So a mistake is viewed as a defect at the surface level from a cognitive point of view. But a misconception is viewed as a defect at the deep level. Actually, both mistake detection and misconception detection are important. Reporting mistakes can help students to realize what is wrong. Pointing out misconceptions can help students to understand why it is wrong. If only the mistakes are identified without an explanation of the underlying conceptual knowledge, then the mistakes will be repeated under the different circumstances. If only the misconceptions are listed without showing where the corresponding mistakes are and what the mistakes look like, then it seems to be too far and too deep

to be understood by students, especially novice students who are in a course of learning completely new material.

3.2.2 Integrated inferences and applied rules in IADI model

Studies on a human diagnosticians' process show that a diagnosis process actually is a hypothesizing process and followed by a verification process. For example, a human diagnostician only makes conjectures after seeing some symptoms; he does not immediately make the decision for the diagnosis at this time. He needs more information to verify these conjectures. For example, a doctor usually first hypothesizes one or more possible diseases based on initial evidences. Then he collects more information from lab tests, conversation with the patient, investigating the treatment history, and so on. Finally he makes the differential diagnosis. Abductive inference captures the plausible features of this hypothesizing process while deductive inference presents the nature of the verification process. Thus the reasoning process using the combination of abduction and deduction to simulate the hypothesizing and verification processes is much closer to the process of human cognition in the real world. Besides integrating abductive inference and deductive inference, the IADI diagnosis model also incorporates the structure analysis to find the demonstrated mistakes.

Both abduction and deduction need the cause–effect relationship to support the reasoning. In order to represent the causal relationship in a diagnosis problem solving, the rule in the following form is applied in the IADI model:

If the misconception (disorder) exists

then a certain mistake (manifestation) can be caused.

It presents the cause–effect relationships between the antecedent and the consequent in a rule, and supports the abductive reasoning and the deductive reasoning more readily.

Most existing diagnostic systems only use deductive inference. They use rules typically in the form:

If conjunction of mistakes (manifestations) exists

then disorder (misconception) can be decided.

This kind of rules are target for constructing chains of deductive reasoning (Barr and Feigenbaum 82). Such reasoning proceeds from the principle of modus ponens. If there is a rule in the database and its antecedent is satisfied, then the consequent is definitely true. Here the consequent is the diagnostic result. This process may involve several rules in order to prove one result. Although the reasoning process based on such a form is theoretically correct, it is not practical for many types of real world diagnostic problems. In such a rule, the manifestations together form a sufficient condition for the consequent. But in the real world, this condition is difficult to be satisfied because a disorder does not always necessarily cause all of the manifestations simultaneously in most cases. Thus it is unnecessary for all of the manifestations to be able to appear at the same time. Hence it is difficult for some rules to be fired, typically when these rules have several or many manifestations in the conjunction part.

The rules in the first form are easily fired since each rule only needs one manifestation as the fuse; the corresponding misconception can be derived as one candidate hypothesis to explain the existence of the manifestation. The rules in this form naturally support the abductive process to infer multiple plausible hypotheses from the given manifestations because usually there are several rules which can be related to each one of the given manifestations in an abductive knowledge base. The result at this inference step establishes the basis for further selecting the most plausible hypotheses in the next step of the verification process.

The rules in the second form are aimed at heuristic matching while the rules in the first form are aimed at causal reasoning. In a tutoring system the rules in the first form indicate the causal relationships between mistakes made by a student and the misconceptions the student may have. Causal relationships can be effectively used to describe

interconnections of the students' behaviors and their understanding from a surface level to a deep level in a precise way. This definite representation of cause-effect relationships is more important in a tutoring process than in other systems. Based on this clear representation, a tutor can readily conclude the cause from manifestations, and therefore can give instructions in a more reasonable manner. In the verification inference process of the IADI diagnosis model, this rule form provides a causal path for questioning students in order to verify their misconceptions, and instructing on a conception.

3.2.3 Nondeterministic representation of diagnosis

The diagnostic results from the diagnostic process are not "certain" in most cases, especially when incorporated with a tutoring process. In a cognitive process such as learning or tutoring, it is very difficult to indicate one or several misconceptions as the final diagnostic result. The reason is that the student's misconceptions are usually associated. It is difficult to distinguish which particular misconception caused the existing incorrect behaviors because sometimes the effects of misconceptions are intercrossing or overlapping. Sometimes even the students' behaviors themselves are incomplete or ambiguous. This situation happens frequently, especially when someone is in the course of learning new knowledge, such as a novice's programming technique learning. So providing a list of possible misconceptions and arranging them by their plausibilities becomes realistic.

In order to measure the "uncertainty" of the selected hypotheses, different models can be used. Among these models, probability theory is the most classical theory and is relatively mature. Some diagnostic systems even use this model to infer or decide hypotheses, not just to measure them as an attachment. The knowledge in these kinds of models is represented as prior probabilities of disorders and conditional probabilities of disorders for given symptoms (Charniak and McDermott 85) (Pearl 88). This model requires data independence assumptions in a practical system which are not always valid.

Furthermore, this model can work only when all present manifestations are given in advance, and it does not attempt to obtain further information for verifying hypotheses as required in a diagnosis process. In the IADI diagnosis model, uncertainty measure is defined in an intuitive way and can be easily calculated. And this measure is used to help rank the proposed hypotheses as a complementary of abductive and deductive inference processes.

3.2.4 Three steps in IADI diagnosis process

Considering the necessity of detection of both mistakes and misconceptions in a tutoring system, the diagnosis process in the IADI diagnosis model consists of the following three steps:

- Mistake detection by structure analysis;
- Misconception hypothesizing by abductive inference;
- Misconception verification by deductive inference.

The three-step cycle can continue until the mistakes do not occur any more. Fig. 5 shows an overview of the IADI diagnosis process.

Diagnosis in a modeled system requires finding mistakes first, then it can further analyze these mistakes so that misconceptions can be infered and verified. Mistake detection is the first manipulation after a problem is given. In a programming tutoring system the mistake detection process attempts to find program errors (or program bugs) by using the program structure analysis. The program errors will be the input data for the abductive and deductive inferences.

Taking the mistakes obtained from the first step, the diagnosis system infers a set of plausible candidate hypotheses by an abductive inference process, and assigns a plausibility measure for each hypothesis. In a tutoring system these candidate hypotheses are the student's misconceptions for a particular problem. But every hypothesis in this set is just one candidate for explaining why the mistake existed. There might be other hypothe-

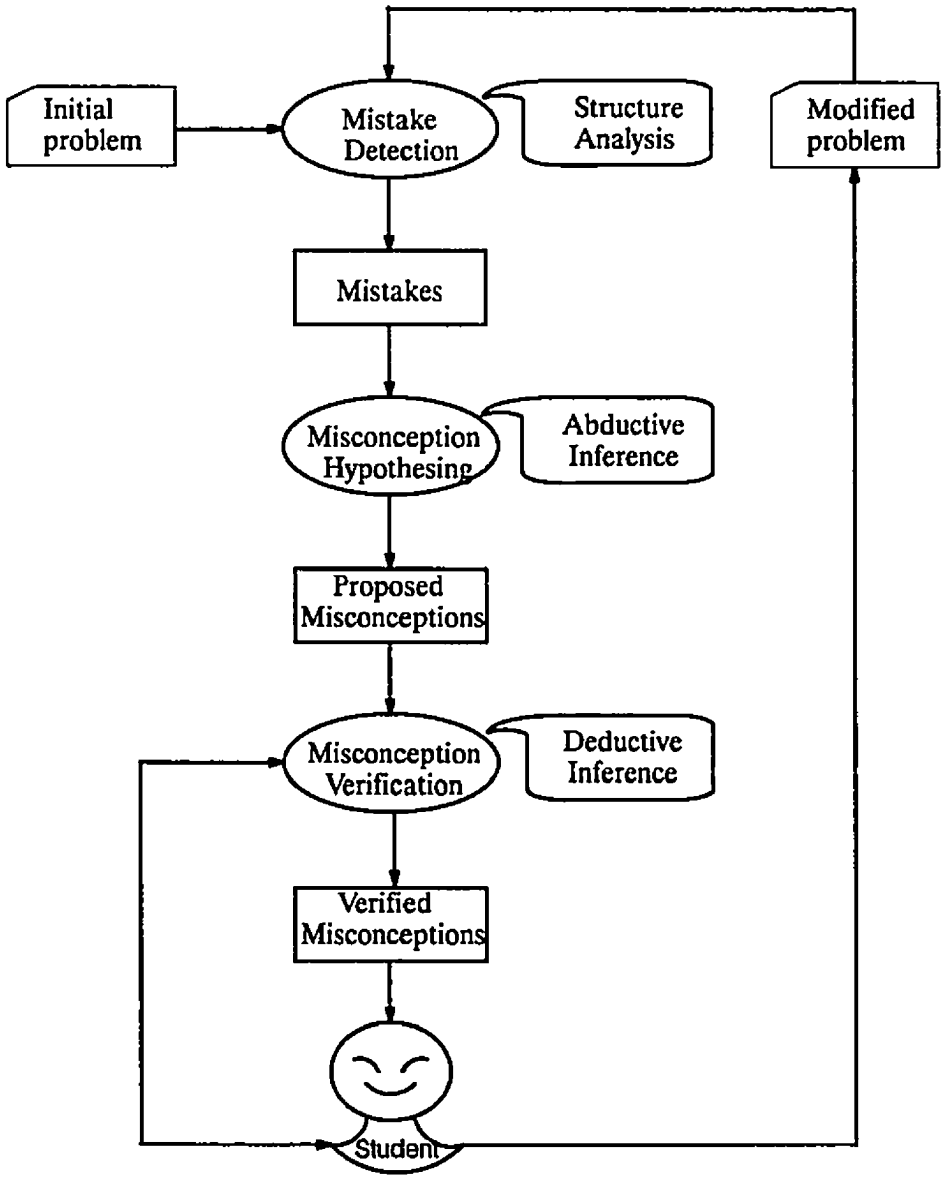

Fig. 5. Overview Of IADI Diagnosis Process

ses that can also account for the mistake. In order to verify which hypotheses are the more precise representations of the misconceptions that truly cause the mistakes made by the student, the system needs a verification process.

In the third step the diagnosis system verifies the proposed candidate hypotheses based on a deductive inference process. The system checks the existence of undemonstrated mistakes and revises the plausibility measure. In this step the system provides an interactive process with the student to get more information. Finally it produces a list of ranked hypotheses as the result of differential diagnosis.

3.3 Knowledge Representation in IADI Diagnosis Model

In the IADI diagnosis model, the knowledge about a diagnosis problem is collected in a diagnosis knowledge base. The IADI diagnostic knowledge base contains structure knowledge, student's mistakes and misconceptions for a particular problem, and checking list.

3.3.1 Structure pattern

A structure is a representation of the basic organization of a substance and the manner of construction. It is the fundamental basis with which to do reasoning (Reiter 87). For example the structure in a computer program is a syntactic structure of a program which can be viewed as a pattern or a template. It consists of specific statements, such as if ... then ... else ...; while ... do ...; case ... which represent the selection structure, repetition structure, and multialternative structure respectively (Leestma and Nyhoff 84). Analysis on the structures can be used to identify the possible mistakes in a program. To avoid an excessively detailed analysis, a structure analysis usually grasps the key words in a structure such as 'if-then-else', 'while-do', and 'case', and just recognizes these key words in a structure analysis.

3.3.2 Hypothesizing rules

A diagnosis process detects the mistakes first by the structure analysis; then it

further analyzes these mistakes, infers and verifies misconceptions. Mistakes and misconceptions are organized in a triple (M, C, R), where

M represents a finite set of all identifiable mistakes. In an ITS, these mistakes are the possible incorrect performances of students. In a programming tutoring system, the mistakes are the manifested program errors or program bugs.

C represents a finite set of all possible candidate hypotheses that are the potential causes of mistakes in M. These candidate hypotheses are the misconceptions that the students may have in the learning process.

R represents a finite set of production rules which indicate the causal relationships between elements in C and M. If there is a rule $c_i \longrightarrow m_j$ in the set R, where $c_i \in C$, $m_j \in M$, that means the misconception c_i can cause the mistake m_j under certain circumstances. This rule is also called a hypothesizing rule or a misconception-mistake rule.

For given sets C and M, the following facts exist:

For one c_i, there might be multiple mistakes. That means one misconception can cause a student to make several mistakes. For example, in recursive programming if the student misunderstands the recursive relationship he might incorrectly give a smaller instance in a recursive procedure call, and he also might miswrite the condition in a condition statement which may cause the recursive procedure to be called. That is, we have

c_i : Confusion on the recursive relationship;

m_j : miswriting the condition in one condition statement;

m_k : misgiving the smaller instance in a recursive procedure call;

and $c_i \longrightarrow m_j$;

$c_i \longrightarrow m_k$.

On the other hand, for one m_j, there might be multiple misconceptions. That

means one mistake can be caused by several misconceptions. For example, besides confusion on the recursive relationship, misunderstanding the termination condition might also make the student miswrite the condition in the conditional statement for a recursive procedure calling statement. Thus we have:

c_l : misunderstanding the termination condition;

and $c_i \longrightarrow m_j$;

$c_l \longrightarrow m_j$.

Consequently we can define following sets:

man(c_i) indicates all possible manifested mistakes which are caused by c_i, and

can(m_j) indicates all possible candidate hypotheses which can cause m_j.

That is

$man(c_i) = \{ m_j \mid (c_i \longrightarrow m_j) \in R \}$

$can(m_j) = \{ c_i \mid (c_i \longrightarrow m_j) \in R \}$ where $c_i \in C$, $m_j \in M$.

An IADI diagnosis problem can be defined as a five tuple

$$\{ M, C, R, M_p, C_p \}$$

where

M, C, R are defined as before, and they are precollected in the IADI knowledge base;

$M_p = \{ m_1, m_2, \ldots m_l \}$, $M_p \subseteq M$, is obtained from the mistake detection on a student's performance;

$C_p \subseteq C$ has different values at different steps:

Before misconception hypothesizing, $C_p = \phi$;

After misconception hypothesizing, $C_p = C_r$ which is the candidate hypotheses set for the intermediary IADI diagnosis;

After the misconception verification, $C_p = C_f$ which is the final IADI diagnosis.

3.3.3 Checking list

For each misconception in C, the system sets up a checking list. The checking lists consist of items. Those items are related with the manifestations which can be caused by this misconception. These items are represented as questions and choices, and they are organized by the cause-effect relationship and design plans. The detailed description is in section 4.5.1.

3.4 Structure Analysis Process

Structure analysis is the initial process for the IADI diagnosis model. It detects a set of mistakes $M_p = \{m_1, m_2, \ldots m_l\}$, $M_p \subseteq M$, and provides the M_p as the input of the abductive inference process.

In programming, although there are many intentions or intermediate processes before a student makes his program and runs it, if possible, usually only the submitted program or the final solution is visible. One can consider the final program code or final execution result as the final solution. So the program error analysis techniques can be basically classified into an analysis of program structures and an analysis of program behaviors. The program structure analysis compares the structures in a student's program and the structures in a instructor's program. The program behavior analysis works on the output of a program or the output of the program's components to detect the program bugs from the incorrect execution results.

In some programming problems, the structures are simple such as the structures in a recursive program. Usually, a recursive program has a succinct program structure, although it could contain things difficult for a novice programmer to understand. Nevertheless, it is this feature of the succinct program structure that makes the structure analysis of a mistake detection feasible. Given a recursive problem, usually there are only a limited number of ways to solve it, therefore a limited number of code patterns for a particular problem solving. In other programming solving problems, it is common

to have a longer and more complex program. Thus, the structure analysis may be much more complicated due to the greater number of possible patterns. In that case the structure analysis has to be supplemented by other methods.

By observing student problem solving behavior, we found that most students begin to learn recursive programming by making an analogy of structures between the given problem and a known program. Research on recursive problem solving of novice programmers shows that 80% of students' programs are accomplished by structural analogies of earlier programs since relatively simple program patterns usually provide these opportunities. Even expert programmers still use the provided templates or a chuck to solve a problem if it is similar to a problem which has been previously solved. Only when the problem becomes complicated, will a more sophisticated analysis be applied. Researchers have found that abstracting the structural features of recursion and simply imitating them is sufficient for beginners to solve routine recursive problems, although its effect decreases when the problem is novel or difficult (Bhuiyan, Greer and McCalla 91) (Wiedenbeck 89). That may be the reason that the LISP tutor basically asks students to use recursion templates to fill in to arrive at a final program when a novice comes to learn recursion (Anderson, Boyle, Corbett and lewis 90).

In the IADI diagnosis model, the structure representation of a modeled system is an abstraction of the key features about a program as well as a syntactic template. In most cases of computer programming, incorrect program structures are the main reason that results in failure of expected functions performed by a program. So the knowledge about program structure is a kind of causal knowledge. It can be viewed as a kind of deep knowledge. This deep knowledge can be used not only for detecting program mistakes, but also for providing the reason for explaining the program errors.

The IADI diagnosis approach first collects possible correct structures and possible incorrect structures in the diagnosis knowledge base. Structure analysis then attempts to find the mismatches between the actual structure and the expected structures. If there is

a mismatch, it is very likely there is a mistake m_i. Then the system includes the m_i in M_p.

3.5 Abductive Inference Process

The IADI diagnosis model uses abductive inference to hypothesize possible misconceptions after a student's demonstrated mistakes are available. This is the first step of diagnosis at the conception level.

The abductive inference process infers a set of plausible candidate hypotheses from the set M_p which is obtained from the mistake detection process, and also from the hypothesizing rules prestored in the diagnosis knowledge base. That is, given an initial IADI diagnostic problem $\{M, C, R, M_p\}$, for each $m_j \in M_p$ (where $1 < j < l$), this process will infer $can(m_j)$ by applying the hypothesizing rules. The $can(m_j)$ is a set of all the possible candidate hypotheses of misconceptions that may cause m_j. The union of $can(m_j)$, $1< j < l$, becomes the selected candidate hypotheses set, called C_s. Every hypothesis in $can(m_j)$ represents just one candidate for explaining why the mistake m_j existed unless $|can(m_j)| = 1$ in which case the $can(m_j)$ is the definite cause for m_j because there is no other alternative misconception for the m_j. If $|can(m_j)| > 1$, it is possible that a hypothesis in $can(m_j)$ is not the true cause the student made the mistake m_j and some other hypothesis in $can(m_j)$, which is generated from other hypothesizing rules, may be the true cause instead, as was the case discussed in subsection 3.3. Hence, there is a need for the verification process to further verify and determine which misconceptions in the set C_s are more likely to be the student's true misconceptions.

In the abductive inference process, the system calculates the Plausibility Measure (PM) for each candidate hypothesis in C_s. The Plausibility Measure $PM(c_i)$ is defined as a numerical value to be assigned to the hypothesis c_i to represent its plausibility for a diagnosis problem. The PM values, calculated in the abductive inference process, are called Initial PM (IPM) values. These values are decided by the number of initially dem-

onstrated mistakes and the information in the knowledge base. The system then yields the initial ranking of the candidate hypotheses in C_s according to the descending order of their IPM values. The ranked set C_s is called C_r.

At the end of the abduction process, the intermediary IADI diagnostic problem $\{M, C, R, M_p, C_r\}$ is produced. The C_r will be sent to the next process, the deductive inference process.

3.6 Deductive Inference Process

Given the intermediary IADI diagnostic problem $\{M, C, R, M_p, C_r\}$, the deductive inference process investigates the hypothesized candidates in C_r, further refines their plausibility measures, and then arranges a final list of the hypotheses as the diagnosis result.

The deductive inference process tries to verify each c_i in C_r in a backward chaining fashion. There is a difference from the traditional deductive inference in a backward chaining control structure. In the traditional deductive inference, the backward reasoning starts with the target that is usually the consequence of a rule, then it retrieves all the rules that can make the target and determines if there is a rule for which the condition, that is the premise in a rule, has been met. If the condition is satisfied, the assumed target can be deduced as true. In the IADI diagnosis model, the inference rules have different forms (refer to section 3.2.2 and 3.3). Hence, even though the backward reasoning has the same fashion from back to front in rules, the meaning is fundamentally different. The deductive backward reasoning in the IADI diagnosis model takes the candidate hypotheses and invokes the related rules to collect their mistakes which are in the consequent parts of those rules. Then it checks whether a student has these mistakes or not, and therefore determines whether the student really has this misconception or not, and then puts the corresponding candidate hypothesis on the right place in the final list of diagnosis results.

In the deductive inference process, the system sets up a checking list for each misconception in C. The checking list contains some items and some questions. Those items and questions are associated with the mistakes that appear on the right hand side of the rules which have same misconception as the left hand side. The checking list is used to allow the system to check on a student's unexposed mistakes to further verify the misconception.

By analyzing the internal relationships between the misconception and its mistakes, we find that there is another relationship besides the cause-effect relationship which is presented in misconception-mistake rules. That is the design plan. When a student is learning a new concept or a new technique, or when he wants to implement a concept in programming, he must follow certain design plans. Plans are the detailed steps to reach a goal. If one design plan deviates from the correct path, it would not be able to implement the correct concept, and a relative mistake is very likely to be presented instead. Obviously, mistakes and design plans are inevitably associated. Solving problems by design plans shows the ability to divide a given problem into smaller ones and synthesize the corresponding solutions into a global solution for the original problem. Therefore it can be viewed as a mental model which can show the coherent knowledge including both structural and functional properties. Combining this mental model into a diagnosis and tutoring process will raise the diagnostic precision and pedagogical effectiveness. Thus the items and questions in a checking list are considered to be organized according to the associated design plans as well as the cause-effect facts. When students interact with a system by using the checking list, the questions and the corresponding explanations displayed in this process will encompass the structure analysis, the cause-effect rationalization, and design plan development. This synthesized knowledge will help detect and verify the corresponding misconception, and also teach how to solve a problem step by step.

According to the student's actions, the system will assess the plausibilities of the

misconceptions and revise the plausibility measure. These PM values, calculated in the deductive process, are called the Final PM (FPM) values. Each FPM(c_i) value is decided in an interactive environment when a student answers the questions or makes a choice in a checking list.

After reranking the hypotheses in C_r according to FPMs, the system concludes the reranked list C_f as the final IADI diagnosis. The IADI approach provides a list of ranked diagnostic results, C_f, in the form of nondeterministic diagnosis.

3.7 Overall IADI Diagnosis Process Description

From the above description, we see that there are three steps in the IADI diagnosis process. In the first step, the mistake detection process, the system detects a set of mistakes, M_p, which are the mistakes initially demonstrated by the student. In the second step, the abductive inference process, the system works on the mistakes in M_p. That is, the system is given M_p, it infers C_s, calculates the IPMs, and ranks C_s into C_r. In the third step, the deductive inference process, it works on the hypotheses in C_r obtained from the abduction process. It uses the checking lists corresponding to the hypotheses in C_r, and collects more information through interactions with the student to refine the plausibility measures, and finally obtains the reranked C_f as the final diagnosis.

To summarize, for a given problem submitted by a student, the overall process for the IADI diagnosis process can be described as follows:

1). For a given problem, detect a set of mistakes $M_p = \{ m_1, m_2, \ldots, m_l \}$

2). For every m_j in M_p ($1<j<l$), infer can(m_j);

3). Calculate the initial plausibility measure IPM for each misconception in C_s;

4). $C_s \longleftarrow \bigcup_{j=1}^{l} \text{can}(m_j)$;

(C_s is the set of selected candidate hypotheses)

5). Rank the hypotheses in C_s according to their IPMs from high to low to obtain C_r;

6). $C_t \longleftarrow \Phi$; (C_t is a temporary set of Hypotheses)

Repeat

7). Get c_i from C_r;

8). Display its checking list $CL(c_i)$;

9). Get the student's choices of items in $CL(c_i)$;

10). Give the explanations to each item selected by the student;

11). Calculate the final plausibility measure FPM for c_i;

12). $C_r \longleftarrow C_r - \{ c_i \}$;

13). $C_t \longleftarrow C_t + \{ c_i \}$;

until $C_r = \Phi$.

13). Rerank the hypotheses in C_t according to their FPMs, to form the final diagnosis C_f..

CHAPTER FOUR

IADI DIAGNOSIS MODEL

IN

RECURSIVE PROGRAMMING TUTOR

The general model of the IADI diagnosis approach can be extended in various ways to accommodate different diagnostic problems. This chapter illustrates the IADI diagnosis model in a Recursive Programming Tutor (RPT). RPT has been designed and developed to teach students to understand the basics of programming in recursion. This system is implemented in C and Sunview languages at the Sun workstation. At present, the RPT is a prototype of the recursive programming tutor. This chapter describes the diagnosis process in RPT. The hypermedia environment of RPT will be described in next chapter.

4.1 RPT Domain

The objective of diagnosis in RPT is to find the run time errors or bugs in a program which contains recursive algorithms, and then to conjecture the students' misconceptions based on the detected program bugs.

4.1.1 Why choose recursion

The concept of recursion is chosen as the tutoring subject because this concept is very useful as is evidenced in expressing various algorithms in computer science. A process is said to be recursive if it partially consists of or is defined in terms of itself

[Wirth 76]. The concept is encountered in many important applications, such as search algorithms, sorting algorithms, and even grammar definitions of the syntax in natural language. The power of recursion is obviously owing to the possibility of defining an infinite process by the finite statements. This feature makes the recursive program code very terse, and also makes it difficult to be understood when a student encounters it for the first time. This problem rises partially from a succinct program structure on the surface but with an underlying a complicated process in the recursive program. Thus, the recursive conception is a challenging subject to teach.

Recursion is also a quite interesting topic to study. Many students claim to experience a significant cognitive change as they gain understanding of the recursive conception (Bhuiyan, Greer and McCalla 91). Usually it is in Pascal that students first encounter the recursion concept, so the recursive programming in Pascal is chosen as the problem domain. Thus we can teach students the recursive concept at the point where they are just beginning to learn it.

Sometimes people use induction to help design recursive algorithms since there are some similarities between them. From a theoretical point of view, we can see that all recursive algorithms can be proved by induction. On the practical point of view, both of them try to reduce a problem to several smaller problems, and generally speaking the methods for reduction are the same (Manber 88). This strategy of reducing a problem into smaller problems is used in the RPT system, but proving a recursive algorithm is not addressed here.

The RPT system provides the mechanism to allow a domain expert to add programs for either problems of instruction or problems of diagnosis into the tutoring system. As a prototype of RPT, the system selects the inorder traversal program as the instruction material, and the issue of inserting nodes into a binary search tree as the representative example to be analyzed in the diagnosis process. This chapter illustrates how the IADI diagnosis model works in the recursive programming tutor for insertion

problem solving. How to give instructions for recursive algorithms will be presented in the next chapter.

4.1.2 Insertion problem description

When we consider which materials or cases to work on, we need to consider whether they are significant in the sense that they evolve gradually from simple to complex in order to allow the student to obtain the knowledge from subskills to the required complete skills. The insertion problem is chosen because it is a typical example used in almost all algorithm introduction text books and it fully illustrates the use of recursion.

The insertion of recursive algorithms involves insert one node at a time into a Binary Search Tree (BST). There are two kinds of cases: one is to insert a node into an empty BST, and the other one is to insert a node into a nonempty BST. The first case, inserting a node into an empty tree, can be easily done by making the new node become the root of the empty tree in a pointer-linked tree structure. In the second case, the algorithm compares the node to be inserted with the one in the root of the BST to decide into which subtree the node should be inserted. If it is less than the root node, then the new node must be inserted into the left subtree; if it is greater, then it must be inserted into the right subtree. That is, the algorithm requires that the properties of a binary search tree are still preserved.

To solve the insertion problem, a correct version is given in Fig. 6. It includes the related specifications, the insertion procedure, and other associated procedures which show how the insertion procedure is called and how the results are printed out.

4.1.3 Analysis on recursive algorithm

The procedure "insert" in Fig. 6 is recursive. A recursive algorithm solves a problem by solving one or more smaller problems of the same type with the same strategies. The smaller problem refers to either a numeric parameter or the size of the data struc-

```
program printBST (input, output);
        type
    nodeptr = ^node;
    node = record
              item : integer;
              Lchild, Rchild : nodeptr
              end;
table = nodeptr;
var T : table;

procedure create (var T : table);
begin
    T = nil
end;

procedure insert (var T : table;  newitem : integer);
{Insert one node into BST at one time}
begin
    if T = nil  then
      begin
        new(T);
        T^. item := newitem;
        T^. Lchild := nil;
        T^. Rchild := nil
      end
    else  if  newitem < T^. item then
              insert(T^. Lchild, newitem)
          else insert(T^. Rchild, newitem)
end;

procedure buildtree(var T : table);
{Insert n nodes to build a BST}
var n, i, k : integer;
begin
    writeln('Please input the number of node ');
    read(n);
    writeln('Please input the node ');
```

```
    for i:= 1 to n do
    begin
      read(k);
      insert(T,k)
    end;
end;

procedure printnode(var T : table);
{Print the nodes in the BST}
begin
    if T <> nil then
      begin
        printnode(T^. Lchild);
        writeln  (T^. item);
        printnode(T^. Rchild);
      end
end;

begin
    create(T);
    bildtree(T);
    printnode(T);
end.
```

Fig. 6 A Correct Version Of Insertion Program

ture that is being manipulated. The size of smaller problems should be decreased at each recursive call. Finally the algorithm reaches a base case. Thus a recursive algorithm must include the base case and the recursive case. The base case, or the degenerate case, is a case in which the problem size is sufficiently small so that the problem can be solved directly. The recursive case, or the general case, is a case in which the solution is expressed in terms of a smaller instance of itself (Helman and Veroff 86). Usually jobs in different cases are different. In the insertion problem, the job in a recursive case is searching for a place; the job in a base case is to insert the new node in the selected

place. In some problems such as the inorder traversal problem, the job in a base case is doing nothing. Even so, remaining a recognition of base case is needed because it ensures that the algorithm will be able to return successfully. The recursive process starts with a general situation and checks whether there is a degenerate case. If that case occurs, the recursive process terminates. Ensuring that a base case can be reached in a finite number of steps is very important because it is expected to terminate a potentially infinite sequence of recursive calls. Otherwise the algorithm will keep issuing recursive calls until all available memory has been used.

4.2 RPT System Outline

The RPT system design emphasizes the diagnosis process. It also provides a hypermedia environment for instruction as well as diagnosis. The system frame is depicted in Fig. 7. The domain knowledge comes from diagnosis experts and tutoring experts. The IADI diagnosis knowledge base contains program structures for a particular problem of programming, and the possible mistakes and misconceptions on that programming problem. The RPT instruction knowledge base contains specific instructions for selected tutoring lessons, and specific representations for instructions on those lessons. The environments for both diagnosis and instruction are created on the hypertext model. Students interact with the environment to communicate with the RPT system. Based on the student's activities, RPT builds a student's model for the system to work on.

Fig. 8 shows the main part of the RPT system, the diagnosis process in the interaction between the student model and the diagnosis knowledge base, which is the representative of the diagnosis expert. How the IADI diagnosis knowledge base is generated and applied, how the three diagnosis subprocesses work together, and what the student model is at corresponding stages will be described in following subsections of this chapter.

4.3 Mistake Detection

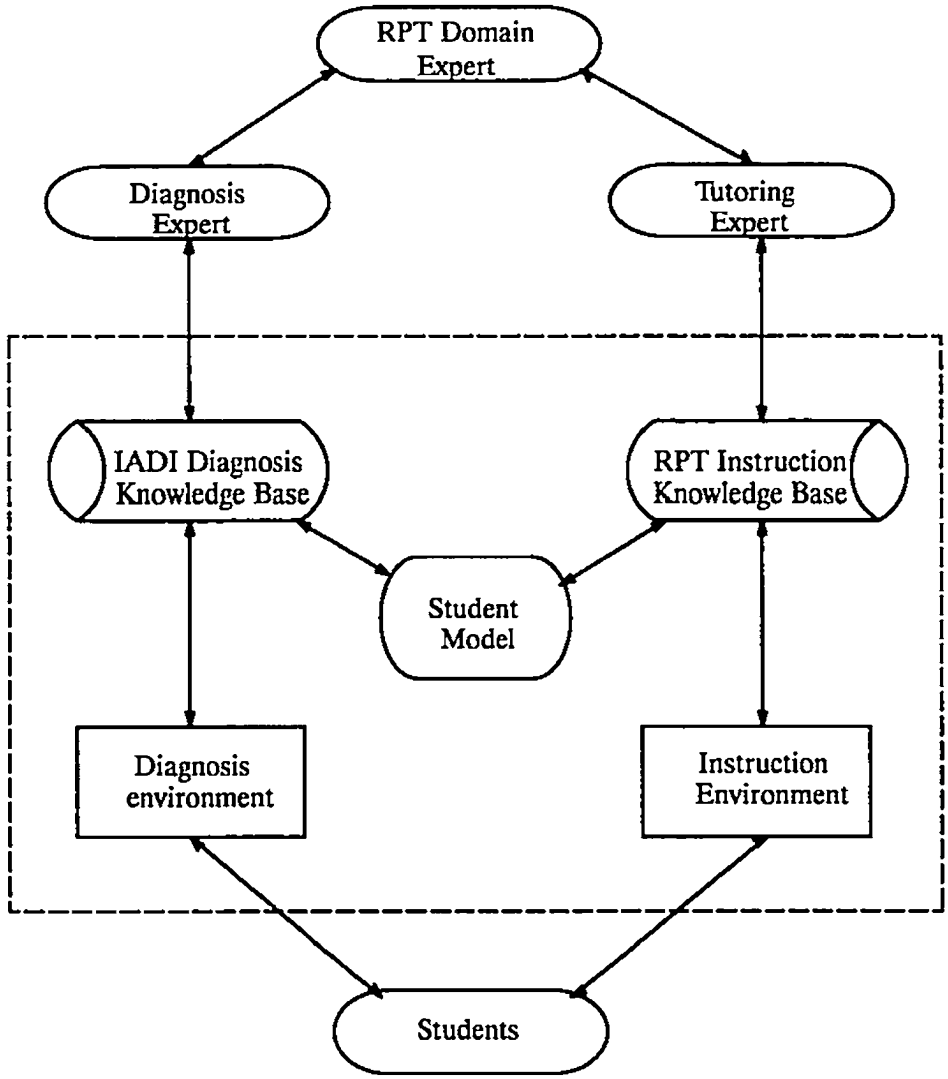

Fig. 7. RPT System Frame

Mistake detection is the first step in the IADI diagnosis process. It finds mistakes from a student's submitted program. This mistake detection process is completed by program structure analysis. The detected mistakes $\{m_1, m_2, \ldots m_l\}$, which is $M_p \subseteq M$, are submitted to the misconception hypothesizing process.

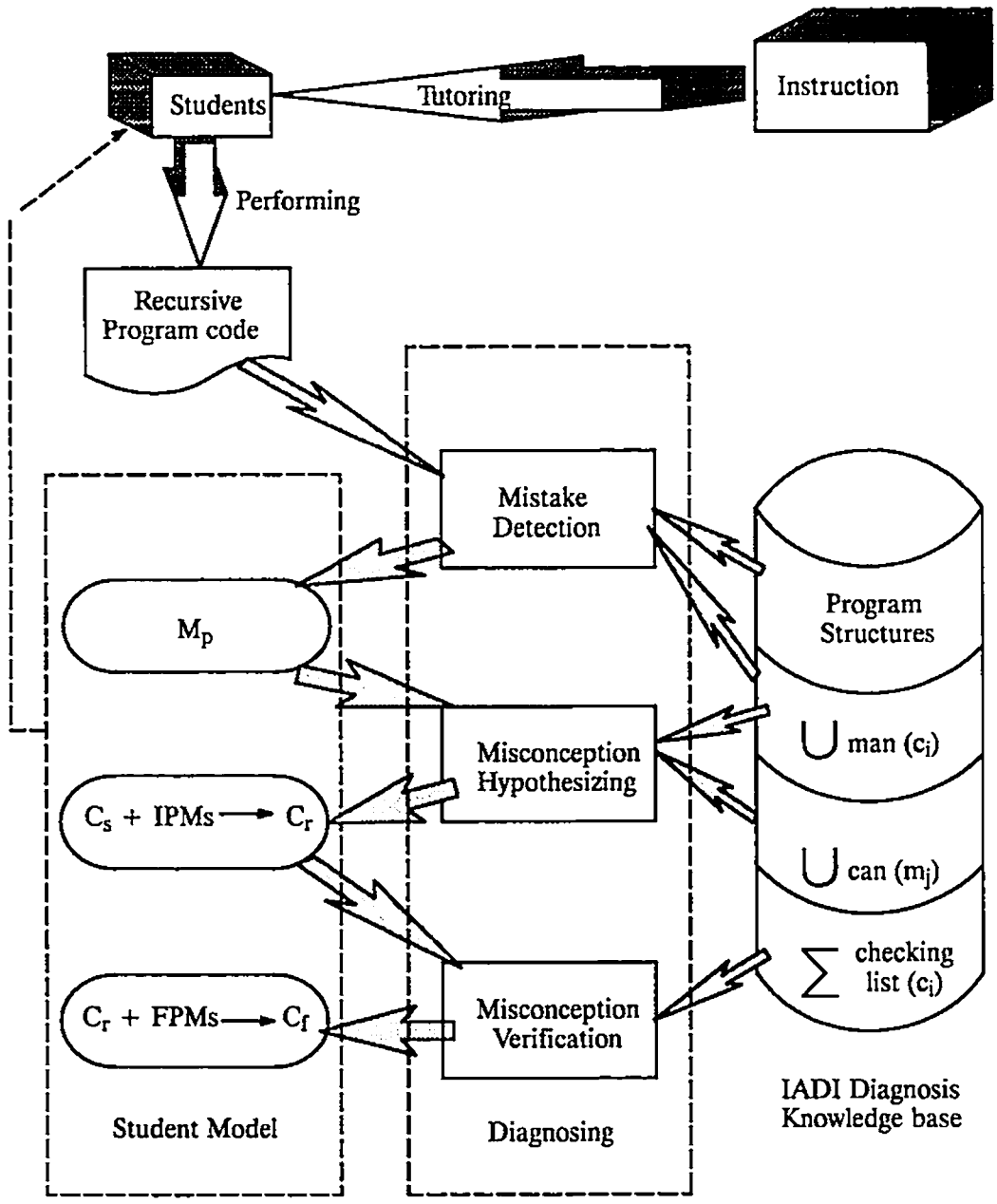

Fig. 8. Diagnosis Process In IADI Model

4.3.1 The organization of program structures

Program structures are organized in the IADI diagnosis knowledge base. There are

two categories of structures which are usually used in diagnosis systems: incorrect structures and correct structures. If a piece of code matches with a correct structure, the system can assume that some requirements have been satisfied and the program is correct on this part. If a piece of code matches with an incorrect structure, the diagnosis system will claim that an error has been identified. But if a diagnosis knowledge base only contains incorrect structures, the system can not find the bug such as a correct structure missing. Obviously missing a correct structure for a necessary function in a problem is also an error. On the other hand, if a diagnosis knowledge base only contains correct structures, the system will claim there is an error when a correct structure is missing or a piece of code does not match with any correct structure. However it can not provide more information about the bugs such as the bug category, bug features, bug location; therefore it does not benefit tutoring greatly.

The IADI diagnosis knowledge base contains both correct and incorrect program structures for a particular problem. Thus it provides a wide range of program structures to be analyzed in proper situations.

Due to the variability in program implementations, the program structures can also be very different. For error recognition purposes, the system needs to have the knowledge of all possible syntactic structures of programs including correct program structure and incorrect program structure. For example, in order to compare two numbers n_1 and n_2, and decide to do t_1 or t_2, the structure patterns can be the following:

if $n_1 < n_2$ then t_1 else t_2

or if $n_1 > n_2$ then t_2 else t_1

or if $n_1 < n_2$ then t_2 else t_1

or if $n_2 < n_1$ then t_1 else t_2

.....

The number of possible combinations of the three syntax elements in one statement: one from n_1 or n_2, sign of inequality and one from t_1 or t_2, is 2^3. Some of them are correct,

but some of them are not. Some students may even use other structures such as while-do to substitute for the if-then selection structure. In some cases the while-do structure is a wrong structure, while in other cases the substitution of a repetition structure for selection structure may be proper. In order to make a correct diagnosis, theoretically the diagnosis knowledge base should contain all of the possible correct alternative structures and all of possible incorrect structures for a particular problem. Practically, however, it is impossible to obtain all of this information even for a trivial problem in programming. It is limited by experts' experience, and also by the obtainability of novices' behaviors. So in the IADI diagnosis knowledge base the correct structures are only their main alternatives, and the incorrect structures are those which frequently appear in students' programs.

4.3.2 Program structure analysis

From the above analysis of the features of recursion, it is easy to see that recognizing the base case and the recursive case should be used as the cornerstone of structure analysis of a program which uses a recursive algorithm. Selecting one of these two cases needs to use a condition, that is called the termination condition. The next problem is deciding what the termination condition in the insertion algorithm is. The insertion algorithm first tries to find a place for a new node to insert. Once the place is located, the node can be inserted. The algorithm searches the location recursively. So the termination condition is actually used to end the searching process. When a tree or a subtree is empty, the condition is reached, that is a place to insert the node is found. Thus the termination condition is when a tree or a subtree is empty.

Usually a selection structure, which has the if-then-else pattern, is applied to branch between the two cases. Although other structures can be used in some problems such as generating permutations problems (Kruse 87), in most problems like recursive algorithms in introductory courses, the repetition structure can not be used for the rec-

ognition purpose because it usually goes into an infinite loop. The repetition structure should not be used in insertion algorithm either. If a repetition structure, such as while-do, needs to be used, it should not contain a recursive call (Dale and Weems 87). But in a novice's program, the loop structure comes into existence when a novice tries to understand and explain recursion in terms of prior knowledge of iteration. Some researchers have investigated and proved that students' knowledge about iteration dominates their knowledge about recursion in the early stages of learning recursion, and students try to draw analogies of recursion to loops or interative structures (Bhuiyan, Greer and McCalla 91). So the repetition structure should be recognized as a incorrect structure. In the mistake detection process, a mistake is announced whenever the repetition structure is found to include a recursive call in the insertion algorithm.

In order to recognize the termination condition in a recursive program, the if-then-else selection structure is usually needed to include the termination condition, base case call and recursive case call. In the selection structure, between the key words if and then, there should be an expression as the termination condition to choose one case to execute. This expression consists of two simple expressions of operands and one of the relational operators such as =, < >, <, >... (Kruse 87). These syntax structures provide the basis for program error detection. In the insertion algorithm described in section 4.1.2, a tree structure is required by the problem nature, and the tree or the subtree is indicated by a pointer. The objects to be compared, or the two operands on the two sides of the operator are the roots of two subtrees. One of them is the empty tree since this expression is used to recognize a termination condition. For an empty tree, the indicated pointer is represented by a nil pointer. Thus, the structure "= nil" or "< > nil" can be used to decide whether a termination condition exists in a student insertion algorithm.

After the base case and the recursive case have been discriminated, the next problem is to decide what should be done in different cases. When a tree is not empty, the

algorithm falls into a recursive case. In a recursive case, the algorithm is trying to locate a place for the new node. The basic strategy is to compare the data in the new node with the data in the root of the current subtree. The comparison result is used to decide in which subtree the further search will be, that is to decide whether a recursive call occurs on a left subtree or on a right subtree. Since the recursive call is a procedure call, that is a call to itself within the procedure, the system can identify the recursive call statement by comparing the procedure call statement and the procedure name in the procedure heading. The recursive call made from within the procedure passes a subtree of the previous tree through the parameter. The parameter should be within a pair of parentheses following the procedure name in the procedure heading. Because the pointer is used to indicate a tree, there must be a '.' mark in a subtree representation. This syntax information can help the structure analysis. Some students do not write subtree representations in parameters directly. They first assign the subtree to a variable in assignment statements before a recursive call statement, and then include the variable in the parameters of the recursive call. In this case the system also uses those syntax structures to recognize whether a subtree has been passed.

Some students do not use the correct action in a recursive case. By comparing the given data and the root of current subtree, they try to find a node in the binary search tree instead of finding a place for inserting the node. In this case they usually use an equality relational operator '=' or an inequality relational operator '< >' instead of use a comparison relational operator '<' or '>' in a selection structure under a nontermination condition (if there is a termination condition to identify this case in the program). For example, they may use 'if newitem = T^.item then,' instead of 'if newitem < T^.item then' as in the version of Fig. 6.

The new data, which is used for comparison when determining the necessity of a further search, can be obtained from the passed parameter. Then the structure analysis can get information about the data from the parameters. But there may be alternative

ways to pass the data. In some programs the new data is not passed by the parameter directly. The new data is contained in a node which is set by another procedure, and it is the entire node that is passed from the parameter. It makes the structure analysis slightly different. This version is showed in Fig. 9.

```
Procedure insert (var T : table; leaf : nodeptr);
begin
    if T = nil
        then T := leaf
        else
          if leaf^. item < T^. item
             then insert (T^. Lchild, leaf)
             else insert (T^. Rchild, leaf)
end;
```

Fig. 9. Another Version Of The Insertion Procedure

The operation in the base case of version 2 is different from the operation in the previous version. The recursive process in the insertion problem is a search process that is looking for a proper position by checking whether the current tree is empty or not. When the empty tree eventually arrives (that is, the base case is reached), the search is terminated and another kind of job should be done. In the base case the new data or the new node will be inserted. Thus the operations in the base case are to apply a new node and assign a value for the data and the pointers for the node as shown in Fig. 6. or to insert the node passed from the parameter as in another version presented in Fig. 9. To determine if these operations are correct or not, the system can check to see the key word like 'new' (for applying the new node) exists or not in version 1, or compare the variable in the base case with the pointer variables in the declaration part to make sure the node is inserted in version 2.

4.3.3 Program structure summary

From the description of the insertion problem, and the analysis of recursive algorithms and insertion program structures in previous sections, the program structures for insertion problem solving recursively can be summarized. The different structures are called different structure elements. The basic structure elements in the IADI diagnosis knowledge base are presented bellow:

1. Data-holding structure elements :

• Record as a formal parameter

The new node to be inserted as an entire record is passed from the formal parameter of the procedure heading.

• Integer as a formal parameter

The new data which will be a part of new node is passed from the formal parameter of the procedure heading.

Recognition of these two different structure elements is used to check whether the correct action is applied in base case.

2. Selection structure elements :

• if ... < > nil then ... else

• if ... = nil then ... else

These structure elements are used to determine whether the termination condition exists.

3. Iteration structure elements :

• while ... do

• for

• repeat

These three structure elements are used to check whether the loop structure is used. If one of them is used, a mistake is indicated since an infinite loop will be caused by the loop structure in the insertion algorithm.

4. Position–checking structure elements

• new (...); if ... < > nil then ... else

• new (...); if ... =nil then ... else

• if ... < > nil then new (...) else

• if ... = nil then new (...) else

The above four structure elements are used to determine whether the action of applying the new node exists and whether it occurs in the right place whenever it is needed.

5. Subset structure element

• Dot '.' in actual parameters of recursive call statements

• Dot '.' in the right hand side of an assignment and the assigned variable appears in recursive call statement

The system determines if one of actual parameters in a recursive call statement is in the form of a subtree by identifying the existence of the syntax representation of a dot '.', either in a recursive call statement or in an assignment statement before the recursive call. If this mark is missing, the recursive call statement does not give the small instance of a subtree to work on and the procedure can not arrive at the ending point.

6. Comparison structure elements

• if ... < ^. then ... else

• if ... > ^. then ... else

• if ^. < ... then ... else

• if ^. > ... then ... else

These four structure elements are used to compare the new data with the data in the root of the current subtree and decide which subtree will be searched further.

7. VPD (Variable Parameter Definition) structure element

• var

It is used to determine if a formal parameter is a variable parameter.

Besides checking the existence of the element structures, the system also checks the relationships or relative operations among the element structures. The procedure in Fig.10 is an erroneous procedure about insertion. The system compares the structures in

```
Procedure insert (var T: table; newitem: integer);
begin
      new(T);
      if T= nil then
       begin
           T^. item := newitem;
           T^. Lchild := nil;
           T^. Rchild := nil;
       end
      else
           if  newitem < T^.item
                then insert (T^. Lchild, newitem);
                else insert (T^. Rchild, newitem);
end;
```

Fig. 10. An Erroneous Procedure.

the procedure with the correct and incorrect structures in the IADI diagnosis knowledge base. Since one of the if .. < > nil then ... structure elements and the if ... =nil then ... structure elements appeared, the systems can determine if the necessary termination condition in this recursive procedure exists. But the operation of applying a new node is misplaced, which should be put under the termination condition. The system checks the relative position of the element structures 'new' and 'if ... = nil then' as well as their existence, and recognizes that the mistake of misplacing the operation of getting a new node had been made.

4.3.4 Mistake types

The program structure analysis serves for mistake detection. The IADI diagnosis process attempts to concentrate its efforts on the mistakes which are related to the crucial concepts. It refers to the major concepts in the key steps of a correct recursive problem solving and other basic programming concepts. It also refers the possible misconceptions about these concepts drawn from the students' experience. These major concepts include (the details will be introduced in next section) :

- There must be a termination condition to stop the recursion.
- The actions in base case.
- The actions in recursive case.
- The smaller instance for each recursive call which represents the recursive relationship between a problem and its subproblems.
- Formal parameter definitions.
- Binary Search Tree definition.
- Necessity of applying a new node to hold the data to be inserted.

From the analysis of program structure elements in the insertion problem, the system can detect the related mistakes. Different mistakes are called different types such as type m_1, type m_2 ... The mistake types in the IADI diagnosis knowledge base form the set M which is defined in section 3.3. The mistake types are listed in table 2.

One example that contains some of these mistake types is shown in Fig. 11. The RPT system interface is shown in Fig.12. The system checks the program made by a student that is in the text window of the right column of a diagnosis interface (the entire interface is described in the next chapter). Initially the tty window, which is on the left column of the interface, is empty. After the command "diagnose" is invoked, the system picks up the recursive procedure from the student's program, and displays it in the tty window. The system further analyzes the structures of this procedure and finds that the expected termination structure elements and the subset structure element do not appear. It identifies the possible mistake types and begins a dialogue with the student in the tty

Types	Mistakes
m_1	Missing the termination condition.
m_2	Misplace the operation of inserting new node.
m_3	Misuse a loop structure.
m_4	Lacking parameter in recursive call statement.
m_5	Did not provide the smaller instance.
m_6	Miswrite the termination condition.
m_7	Missing key word which defines the variable parameter in the formal parameter definition.
m_8	Misorder the data in left subtree and right subtree.
m_9	Not apply a new node to hold the new item.
m_{10}	Apply too many new nodes for inserting one new item.
m_{11}	Try to find nodes in a binary search tree instead finding a place to insert a node.
m_{12}	Create a new node after using it.
m_{13}	Not apply a new node, but try to assign data to it.

Table 2. Mistake Types

window. In this example there are two mistake types that are missing termination conditions and do not provide the smaller instances which are highlighted in Fig. 12. Thus the corresponding mistake types m_1 and m_5 are detected, and the set $M_p = \{m_1, m_5\}$ is formed. The initial IADI diagnosis problem { M, C, R, M_p } will be submitted in the next diagnosis step, the misconception hypothesis process.

```
Procedure insert (var T: table; newitem: integer);
begin
    new(T);
    T^. item := newitem;
    T^. Lchild := nil;
    T^. Rchild := nil;
    insert (T, newitem);
end;
```

Fig. 11. An Erroneous Procedure.

4.4 Misconception Hypothesizing

The misconception hypothesizing process starts from the initial IADI diagnosis problem {M, C, R, M_p}, where M_p is a set of demonstrated mistakes {m_1, m_2 ... m_l}; it infers a set of plausible candidate misconceptions to yield the intermediary IADI diagnosis problem { M, C, R, M_p, C_r }. This misconception hypothesizing is done by abductive inference.

4.4.1 Misconception types

Misconceptions are the reason that students make mistakes under certain circumstances. The misconceptions in the IADI knowledge base are established by deviating from the required concepts identified by experts for the tutoring subjects. The insertion problem solving requires many concepts. Some of them are important, for example, the concept about the binary search tree, the concept about the termination condition when inserting a node into a binary search tree, the concept about the smaller instance to represent the recursive relationship in this problem, and other basic program knowledge such as how to define the variable parameter, when and how to apply a new node ... In the IADI diagnosis knowledge base different misconceptions are called different misconception types such as misconception type c_1, c_2 ... These misconception types for the

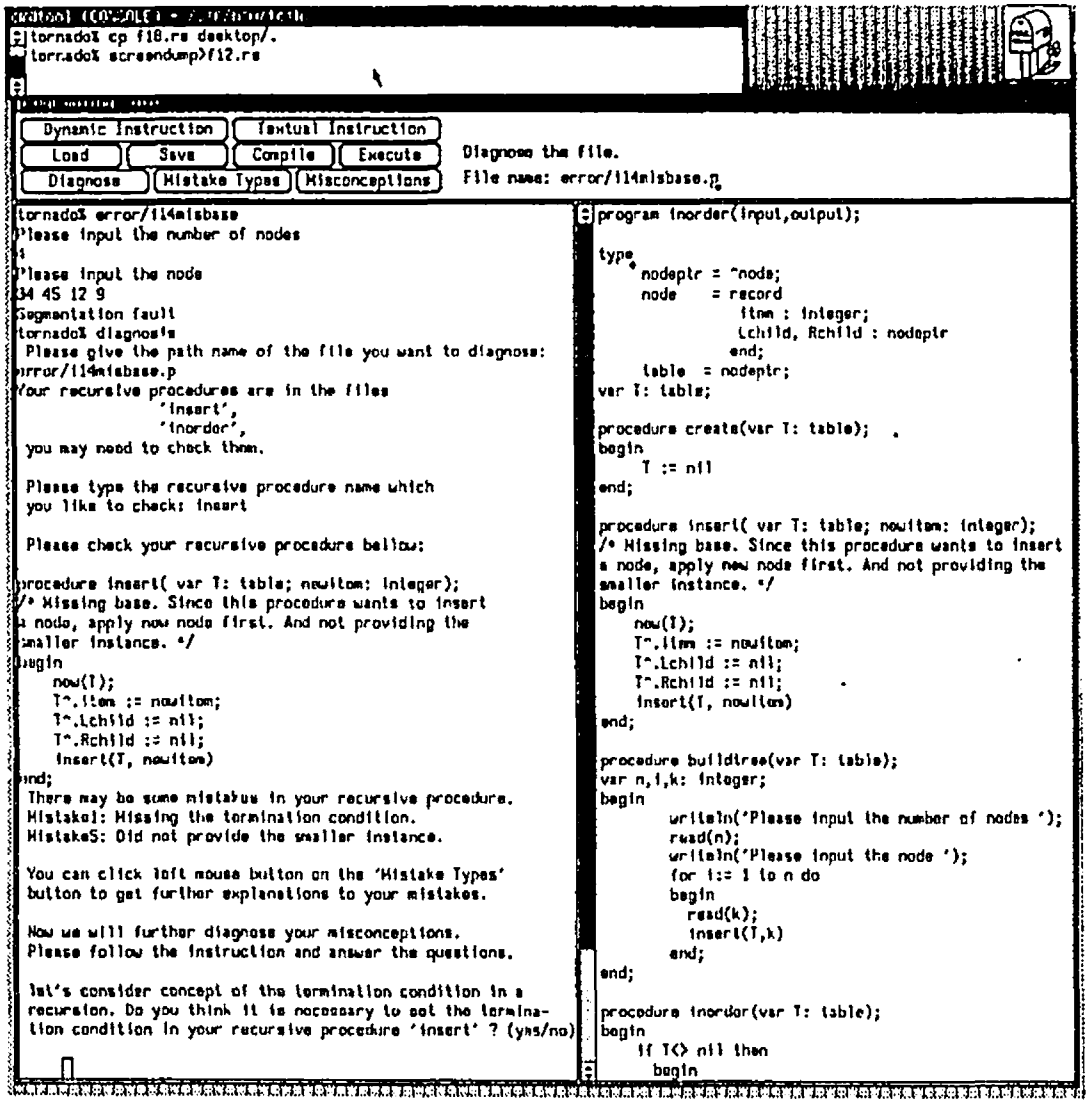

Fig. 12. An Example Of Error Program
and Detected Mistakes.

insertion problem are listed in table 3.

A student may know that there is a need for termination condition in a recursive program when reading a text book or listening to an instructor. But when they begin to program on a recursive algorithm, it is not easy for them to remember this point especially when there is nothing to do in a base case like in the inorder traversal algorithm. Missing the termination condition reflects that the student does not quite understand

Types	Misconceptions
c_1	Not having the correct concept of termination for a recursion.
c_2	Confusion on the recursive relationship.
c_3	Incorrect concept about the definition of a Binary Search Tree.
c_4	Misunderstanding the definition of variable parameter of a procedure.
c_5	Confusion on the concepts of building a Binary Search Tree (BST) and searching a node in a BST.
c_6	No concept how to hold a new item in a tree structure.
c_7	Misunderstanding when to apply new node to hold the new item.

Table 3. Misconception Types

that a recursion needs a condition to suspend issuing recursive calls and that it must return to the previous call after the recursion has ended. Sometimes students do not forget to put the condition, but they may put the wrong condition or put it in a wrong place. In the insertion problem, the algorithm needs to find a place to insert the new node and it proceeds recursively. When an empty subtree is found, the searching process should stop and a new node should be inserted. Some students simply think that since the insertion definitely requires the application of a new node to hold the new data, they apply the new node at the beginning of the procedure, and put the termination condition after applying the new node, such as the procedure in Fig 10. Misplacing the termination condition reflects that the student does not understand exactly why there should be a termination condition, and what should be done when the case occurs. The misconception types c_1 and c_7 in table 3 describe these erroneous concepts.

The strategy for solving a problem in recursion is defining the problem in terms of

a smaller problem of the same type. The smaller problem has the same nature as the original problem, but is of a smaller size. Thus in a recursive case, there must be a relational representation between the each smaller problem and the original problem which represents how the size of the problem is being reduced in a series of recursive calls. This relational representation is called a recursive relationship. This relationship should guarantee that the subproblems generated are closer to the anticipated base case than the original one. The smaller size of original problem is usually given in the actual parameter of a recursive call statement, and it defines a smaller problem that is currently working. That is, the recursive relation is passed by the parameter. For the insertion problem, the smaller problem is in subtree size. Sometimes students do not reduce the size of the problem when they make a recursive call, or they simply forget to put the size which needs to pass to the procedure. The misconception type 2 in table 3 summarizes these errors.

Some students do not have a clear concept of the definition of a Binary Search Tree (BST). They ignore the defining property of a BST that requires that the data value in the BST follow a certain order, that is for any node, the data in it is greater than the data in its left subtree and less than the data in the right subtree (Dale and Weems 86). This misconception about the definition of BST is presented in misconception type 3. Another misconception about the BST is a concern of the concept of building a BST. Some students confuse building a BST by inserting node one by one with searching a node in a BST. This is related to what action should be taken in a BST. The insertion procedure should be targeted for searching for an appropriate place to enter the new data. The purpose is not to look for a node which has a data value equal to the new data. This incorrect concept is addressed in misconception type c_5.

Other misconceptions are related with the basic program knowledge. For example, some students may not have a clear concept about the differences between the variable parameter and the value parameter. Therefore, they do not know that it is necessary to

enter a key word "var" in front of the variable when they want to define a variable parameter to bring the changed value made in the called procedure back to the calling procedure. To insert a node into a tree, a new node must be created first, and then linked the node into the BST. In Pascal the predefined procedure "new" is responsible for creating a new memory cell. Students may not realize that there is a need to allocate the dynamic memory cell. These misconceptions are presented in misconception c_4 and c_6 separately.

4.4.2 Multiple relationships between mistakes and misconceptions

A tutor's main purpose is to help students get rid of confusion at the conception level. However, it is not always easy to identify these misconceptions in a complicated problem solving such as in computer programming, because one misconception can be manifested in several mistakes and several misconceptions may cause one same mistake. The IADI diagnosis model uses the abductive inference process to hypothesize the misconceptions from demonstrated mistakes, as the first step of diagnosis at the conception level.

The abductive inference applies a set of hypothesizing rules. These hypothesizing rules describe the cause and effect relationships between misconceptions and mistakes. From the analysis of section 4.3.4 and section 4.4.1, we can summarize these rules and include them in the IADI diagnosis knowledge base. For each possible misconception there may be several related rules. For example, the rules related with misconception c_1 are:

$$c_1 \longrightarrow m_1,\ c_1 \longrightarrow m_2,\ c_1 \longrightarrow m_3,\ c_1 \longrightarrow m_5,\ c_1 \longrightarrow m_6.$$

That is, not having the correct concept of termination for a recursion could cause five possible mistakes: missing the termination condition; misplacing the operation of inserting the new node; using a loop structure; not providing smaller instances and miswriting the termination condition. But in one student's program, there is usually only one or

several of them appeared under this misconception.

On the other hand, for each possible mistake, there may also be several related rules. The IADI diagnosis model uses the rules to infer the plausible explanations and present their relationship. For example, if there is a mistake m_5: not providing a smaller instance when a recursive call occurs, then the possible causes for it can be drawn from following two rules:

$$c_1 \longrightarrow m_5, \quad c_2 \longrightarrow m_5.$$

They hypothesize that not having a correct concept of termination to a recursion or confusion about the recursive relationship, or both can cause the mistake m_5. We can not eliminate any misconceptions neither favor any particular one at this point. We have to collect them all in a set and make further analysis. Under different circumstances, a student may make different mistakes form one misconception or the student may only demonstrate one mistake, but it is related to different misconceptions. These practical existences form an intricate relationship net between misconceptions and mistakes. The cause and effect representation can make this relationship easier to understand and make the corresponding explanation more eloquent.

4.4.3 Abductive hypothesis

In the misconception hypothesizing process, the system works on the mistakes in M_p which are initially demonstrated in the student program and offered by the mistake detection process.

For each mistake m_i in M_p, the system checks the rules in the IADI knowledge base. The rules in the IADI diagnosis knowledge base are grouped by misconceptions. That is, for each misconception, there is a group of rules that match the same misconception on the left hand side. The system works on the rules in each group. It searches this group to check whether there is a rule which right hand sides matches with the m_i. If so, the misconception on the left hand sides of this rule is marked. Since one m_i can

be caused by several misconceptions, the system searches each group and finds all rules which right hand side matches with m_i. The corresponding misconceptions form a set of candidate hypotheses can(m_i). These misconceptions are considered to be the possible causes of the mistake m_i. After the system hypothesizes all the candidate hypothesis sets for all the mistakes in M_p, it concludes the conjunction of these sets, the union $\bigcup_{j=1}^{t} can(m_j)$, as C_s.

In the erroneous procedure displayed in Fig.11, there are two demonstrated mistake types m_1 and m_5. The system finds the related rules:

$$c_1 \longrightarrow m_1, \quad c_7 \longrightarrow m_1,$$

$$c_1 \longrightarrow m_5, \quad c_2 \longrightarrow m_5$$

and forms can(m_1) = {c_1, c_7} and can(m_5) = {c_1, c_2}. The union of can(m_1) and cam(m_5) is {c1, c_2, c_7}. The misconceptions in the union become the selected candidate hypotheses set C_s. Any c_i in C_s is one possible explanation to one or several mistakes. Then the system hypothesizes the following misconceptions as the candidate hypotheses for that particular problem: not having the correct concept of termination to a recursion; confusion about the recursive relationship and misunderstanding when to apply a new node to hold the new data. These candidate hypotheses are the basis to give the explanations to the demonstrated mistakes, missing the termination condition and not providing the smaller instances. As for how to chose the best explanation, it will be addressed in section 4.5.

4.4.4 Misconception ranking

Each hypothesized misconception in C_s can explain at least one demonstrated mistake. The more a hypothesis can explain, the more plausible the hypothesis is. In order to measure the plausibility of one hypothesized misconception, the IADI diagnosis system builds a measurement, the Plausibility Measure (PM), which is defined in section

3.5. The PM values, calculated in the misconception hypothesizing process, are the Initial Plausibility Measure (IPM). IPM is used to provide the comparisons among the hypothesized misconceptions. IPM(c_i) is calculated by dividing the number of mistakes in man(c_i) which are also in M_p by the number of mistakes only in man(c_i), that is:

$$IPM(c_i) = \frac{| \{m_j \mid m_j \in M_p \text{ and } m_j \in man(c_i)\} |}{| man(c_i) |}$$

For example, if $|man(c_i)| = 4$, and only one of the mistakes in man(c_i) appears in M_p, then IPM for c_i is 1/4.

For every candidate hypothesis c_i in C_s, the system will calculate a value IPM(c_i) for it. Thus, if the majority of mistakes which are related to c_i appear in M_p, then IPM(c_i) has a greater value which shows that this misconception is demonstrated from wider aspects, and therefore it is more likely that the student has this misconception. Conversely, if no mistake in M_p is related with c_i, then its IPM(c_i) value is zero which shows that this misconception does not cause any mistakes in the current programming, and that the student does not appear to have this misconception c_i. According to their IPM values, the system ranks these misconceptions in C_s in descending order of their IPM values, and forms the ranked candidate misconception set C_r.

The IADI diagnosis system sets up counters to record the number of mistakes. For each misconception c_i, which is in the set C, there is a counter ipm(c_i). In the misconception hypothesizing process, the system checks the mistakes in M_p. If the mistake matches the right hand side of a rule, the system finds the corresponding misconception c_i on the left hand side and increases the counter ipm(c_i) by 1 while adding the c_i into C_s. At the end of the hypothesizing process, the counter ipm(c_i) records the number of mistakes which are in M_p and also in man(c_i). There is also a unit for each misconception to record the sum of the related mistakes. Thus the IPM(c_i) can be calculated by dividing ipm(c_i) by man(c_i).

The candidate hypotheses set C_s for erroneous procedures in Fig.11 contains three

misconceptions, c_1, c_2 and c_7. Since there are 6 possible mistakes in man(c_1) and only two of them are demonstrated in M_p, the IPM value for c_1 is 2/6. For the other two misconceptions c_2 and c_7, the system checks the diagnosis knowledge base and gets man(c_2)=3 and man(c_7)=4, and obtains the values IPM(c_2)=1/3 and IPM(c_7)=1/4 respectively. So the misconceptions in C_r are ranked in the order c_1, c_2, c_7.

The IPM measure is just a rough measurement. It is not as strict as the probability theory is, and it is not used as the basis for inferring the hypotheses. In the IADI diagnosis system, inferring the candidate hypotheses relies on the abductive inference. The main purpose of calculating the plausibilities in the hypothesizing process is to provide a ranked list of misconceptions to the verification process. Thus the verification process to the hypothesized misconceptions can work in a more efficient way because this discrimination by descending order provides an opportunity for the user to work on the misconceptions which have high IPM values, and ignore those misconceptions which have very low IPM values or zero value. In another words, this measurement just targets for helping the tutor to select misconceptions which are the individual problems for a particular student, and focus the instruction on these specific concepts instead going through the whole set of misconceptions.

4.5 Misconception Verification

Given the intermediary IADI diagnosis problem {M, C, R, M_p, C_r}, the system begins a misconception verification process. This process is based on deductive inference. It interacts with the student to obtain more information for verifying the hypothesized misconceptions in C_r. During this process, an instruction is also provided which is based on design plans. Finally a list of ranked misconceptions in C_f is provided as the final diagnostic result.

4.5.1 Checking list

The misconception verification process is based on deductive inference in an interactive environment. This environment is shown through a set of checking lists.

In the IADI diagnostic knowledge base, for each c_i in C, there is a group of rules whose left hand sides are c_i. Mistakes on the right hand sides of these rules are considered to be manifestations caused by the c_i. These manifestations compose the set man(c_i). For each c_i, the system establishes a checking list, CL(c_i). The item in the checking list CL(c_i) can be a choice or a question which is related to the mistake in man(c_i).

Fig. 13 shows a checking list for misconception c_2, confusion on the recursive relationship. Since there are three rules in the IADI knowledge base which are related with the misconception c_2,

$$c_2 \longrightarrow m_4,\ c_2 \longrightarrow m_5,\ c_2 \longrightarrow m6.$$

there are three possible mistakes. The checking list CL(c_2) includes these manifestations and organizes them in a way that the items can be related with these manifestations and therefore to the misconception c_2. Since the corresponding mistakes are lacking a parameter in a recursive call statement, not providing the smaller instances and miswriting the termination condition, the problems are listed in the CL(c_2) in the form of a question or multiple choice. After a student gives the answer or makes a choice on an item the system judges the answer or choice, and gives an appropriate explanation or instruction interactively. In the real system, the items in one checking list are separated by the student's answer and the system's explanation and instruction. The corresponding explanations for each item in the checking list CL(c_2) are shown in Fig.14. A real example in the system interface will be shown in the section 4.6.

From above descriptions, we can see that checking lists actually serve two functions. One is for further detection of a student's unexposed mistakes. In order to verify the misconception a student has, the system creates an environment in which all the questions and possible choices are associated with that concept. Compared to the initial-

1. In a recursive relationship, the recursion is expressed in terms of a smaller instance of itself. Did you express this relationship in the manner that the problem is identical in nature but smaller in size? (yes/no)

2. Since we know the recursive relationship will be represented in terms of a smaller instance of itself, and this smaller instance can be a smaller value or a smaller size of the data structure, what is the smaller instance in this insertion problem? Please choose one number from the following choices:
    1. The smaller value in a node for each recursive call;
    2. The empty tree;
    3. The left subtree or the right subtree.

3. When you decide the recursive relationship in a recursive problem, do you need to guarantee that the reduced size will eventually become the degenerate case? (yes/no)

Fig. 13. A Checking List For c_2

ly demonstrated mistakes which are made without any external condition, the student's confusion on a certain concept can be further exposed in this environment. From the student's choices to the items and the student's answers to the related questions, the system can evaluate the degree to which the student suffers from the misconception.

Another function of checking lists is for instruction. When a student demonstrates any manifestations in the lists, the system will give the corresponding explanations to the student based on the cause-effect relationships and also the program design plan which will be addressed in the next subsection. The impromptu instructions help students to get rid of the confusion about the misconception.

Explanation for item 1 (if the student's answer is 'no'):

A recursive algorithm must include a recursive case for which the algorithm is expressed in terms of itself, and in this case the problem size must be diminished at each recursive call. You need to express this relationship in a smaller problem of itself.

Explanation for item 2 (if the student's answer is choice 1):

Only choice number three is correct. You chose number 1 which is not correct. This procedure is to insert data into a binary search tree which is a tree structure, not a single value. So you need to find relationships between the original structure and a smaller structure.

Explanation for item 2 (if the student's answer is choice 2):

Only choice number three is correct. You tried to find an empty tree as the recursive relationship, but that is not correct. Actually, the final target is to find the empty subtree, but it may need several recursive calls to reach it. So you need to represent this relationship between the tree and the left subtree and the right subtree.

Explanation for item 2 (if the student's answer is choice 3):

You have a correct choice.

Explanation for item 3 (if the student's answer is 'no'):

We should ensure that the way that the problem size diminishes makes the degenerate case finally be reached. At this point, the problem is sufficiently small that it can be solved directly and the recursive calls will be ended. Otherwise the process may not stop and goes to stack overflow.

Fig. 14. The Explanations For The Different Choices In Items Of $CL(c_2)$

4.5.2 Programming design plan

From the discussion in section 3.6, we know that the system organizes the items in a checking list according to the design plan as well as the cause-effect relationship. The

design plans describe the sequence of programming steps which directs a program task or subtask to be completed or directs a programming concept to be fully understood by the student if considered from the angle of programming tutoring. For example, when a student wants to find a termination condition for the insertion problem, the termination condition is viewed as a programming concept as well as a program goal (or subgoal) for accomplishing the insertion of a node into a binary search tree. When the student wants to implement the termination condition, he must have a series of design plans in mind. If the student does not follow a correct design plan for the programming problem, there will be a mistake. Therefore the analysis of mistakes and misconceptions is inevitably involved in the analysis of design plans. And the design plan analysis can provide effective information for misconception diagnosis. Under this consideration, when the system collects the items related to one misconception into a checking list, the system arranges these items according to the design plans. Thus, the system connects the design plans to a certain concept while the system analyzes the related misconception by the cause-effect relationship. And the design plans are considered the information at a high level of a conception.

Related with the termination condition concept in the insertion problem, the design plans can be the following:

- Motivate to set up a termination condition;
- Find a case as the condition;
- Check whether the condition can definitely be reached;
- Consider what to do under this condition;
- Deliberate the operations under the termination condition;
- Think over what kind of structure can be used for building the termination condition.

In the system, these plans are posed as questions, such as Why there is a need for a termination condition? What is that condition? What kind of structure should be used

for building the condition? How can you guarantee that the condition will eventually arrive? What operation should be done if the condition is met? After the student gives an answer, the system judges this answer and gives explanations for it. Raising questions to and receiving answers from the student allows the system to be able to obtain more information to verify the misconception acquired from the misconception hypothesizing step. On the other hand, providing solutions to these problems makes the design plan explicit, and it will help students to make a correct design plan to set up the termination condition. Here the solutions provided by the system are the explanations to the student's answers. The explanations combined with the design plans offer the background knowledge for a concept and the programming steps; therefore they can help students to understand the whole programming process.

In the IADI diagnosis process the system arranges these plans into a checking list as items either by questions or by a multiple-choice problem. Fig. 15 shows the design plans for the termination condition concept in the checking list $CL(c_1)$.:

4.5.3 Misconception verifying process

The purpose of the IADI diagnosis process is to find the most likely misconceptions related to mistakes in a student's programming. Since the misconceptions in C_r are only candidates for explaining why the corresponding mistakes exist, it is possible that some misconceptions in C_r are not the reason or main reason for those mistakes. Thus, after the misconceptions are hypothesized from the abductive process, the system still needs a process to verify which misconceptions are the true causes. The misconception verification process is based on deductive inference.

For each candidate misconception c_i in C_r which is obtained from the misconception hypothesizing process, the deductive inference process performs a verification process in the following manner. First the system displays its checking list $CL(c_i)$ to a student. From the previous discussion, we know that a checking list contains information

1. Do you think it is necessary to set the termination condition in your recursive procedure? (yes/no)

2. Since we need the termination condition to end the recursive calls, we should consider a case as the condition. Then what is the condition? Please choose one from the followings:
    1.) When the inserted data is equal to certain value;
    2.) When a Binary Search Tree (BST) becomes empty;
    3.) When the root has a value which is equal to certain value;
    4.) When a tree is not empty.

3. The termination condition is when the BST becomes empty, how can you guarantee the condition will eventually arrive? Please choose one from the following:
    1.) Provide a smaller value through a parameter in the recursive call statement;
    2.) Provide a smaller structure, such as a subtree, through a parameter in the recursive call statement;
    3.) Check whether the subtree in a 'if' statement is empty or not without providing a smaller instance in the recursive call statement.

4. The next problem is to decide what to do when the termination condition is satisfactory. Please choose one number from the following choices:
    1.) Insert the new data;
    2.) Compare the inserted data with the value of the root and then make a recursive call again;
    3.) Do nothing.

5. Under the termination condition, you need to apply a new node and fill it with the necessary data, the problem is where these operations should be entered? Please choose one number from bellow:

1.) At the very beginning of the procedure even before the base case;
2.) When the base case is reached.

6. In this insertion problem, what kind of structure will you use?
   1.) Use a loop structure for control of inserting n nodes in this procedure;
   2.) Use a branch structure to decide to go to the base case or the general case.

Fig. 15. Checking List Arranged By Design Plans

about the manifestations which can be caused by c_i and the design plans which are used to realize the corresponding concept in a program. So the $CL(c_i)$ has a dual function; it can be used to determine if the student has unexposed mistakes and if he follows design plans.

The basic form of an item in a checking list is a question. Following the display of a question, the student gives answers by typing yes or no, or by typing a number to choose an item.

From answers and choices, the system then analyzes the student's understanding to a particular problem and evaluates how much the student suffers from this misconception, and how far he departs from the correct design plans. In this interactive environment the system checks to see if the student is vulnerable to m_k ($m_k \in man(c_i)$) under some circumstances. For each answer and choice, the system gives explanations that are a part of the instruction on the tutoring material.

The system will display all the checking lists for the misconceptions in C_r. The displays of Checking Lists are relayed according to the order of the hypotheses in C_r.

An example of the misconception verifying process will be demonstrated in section

4.6. The interactive process in the RPT diagnosis environment is shown in Fig.23 in the chapter 5.

4.5.4 Misconception reranking

Although each misconception in C_r proposed from an abductive hypothesis may explain several demonstrated mistakes, having each one of them proposed actually only requires one related mistake to be demonstrated. It is very likely that some of them are not the real cause of the mistake m, and that there is another reason which causes the m. To decide which misconceptions in C_r are the student's real problem, the system proceeds with a verification process in the steps described above, and reassesses the plausibilities of the misconceptions.

The system revises the plausibility measure, PM value, according to the student's responses in the misconception verification process. The PM value, calculated in this process, is called the Final Plausibility Measure (FPM). For each misconception c_i in C_r, the system calculates a value $FPM(c_i)$ to evaluate the degree to which a student suffers from c_i. The $FPM(c_i)$ is a value that records the number of wrong answers to the items in $CL(c_i)$. The system weighs the different items with different values depending on the degree of importance and the proximity of a manifestation to the concept. For example, item 1 for motivating the set up of a termination condition in $CL(c_1)$ (refer Fig.15) is more significant to the diagnosing misconception "not having a correct concept of termination to a recursion", than item 3, that checks whether the condition will definitely be reached in $CL(c_1)$, because the necessity of setting up a termination condition is the most important of all the manifestations to diagnose c_1. A wrong answer to item 3 is the mistake m_5. It gives more contributions to diagnosing misconception c_2 than c_1, although this mistake is related to c_1 also. So the system assigns value 3 to item 1 and value 2 to item 3. The basic formula is

$$FPM(c_i) = \sum weight(item); \qquad item \in CL(c_i)$$

The $FPM(c_i)$ is different from the $IPM(c_i)$. The $FPM(c_i)$ value is for the system to weigh a student's understanding of c_i under a certain circumstance in an interactive environment, while the $IPM(c_i)$ value is for the system to measure a student's understanding of c_i in the initial stage without any influence from external effects. An FPM value is based upon considerations of more than one aspect about a concept and trying to get diagnosis solution by verifying the misconception, so it is reasonable to use it to rerank the misconceptions and to submit the list of reranked misconceptions as the final diagnostic result.

The system builds a group of counters to record the total number of the mistakes. For each checking list $CL(c_i)$, there is a counter $fpm(c_i)$. For each incorrect answer, there is a value which is preassigned by the system. When a student answers the questions in the checking list $CL(c_i)$, the system accumulates the value and records the accumulation in the corresponding counter $fpm(c_i)$ if the answer is incorrect. When the system obtains all the answers to items in one checking list, the counter records the value of FPM for the misconception. In order to facilitate the comparison among the FPM values, the system sums the total possible FPM value for each misconception as 10 (that is when a student gets wrong answers to all items in a checking list) although it assigns a different weight to each item in one checking list. Finally the system compares the values in different counters and ranks the misconceptions by their FPM values to yield the final diagnosis list C_f.

There are three possible cases after the verification process:

- If the student's performances testify that all manifestations in a checking list $CL(c_i)$ exist, then the system can decide that the student has the misconception c_i with a very high FPM value and the c_i is the one on the top of the diagnostic result list.
- If the student's performances only partially support the manifestations in a check-

ing list, then it shows that the student has this misconception with a moderate FPM value. In this case, the system still does not eliminate the misconception c_i from the diagnosis result, but lists it in the appropriate place on the diagnosis result list.

• If the student's performances show nothing wrong when he answers the questions in a checking list, then the system will consider the initial appearance of related mistakes as an accidental event and remove the corresponding misconception from C_r.

4.6 Example of IADI Diagnosis Analysis

In this section, an example is given to show how the IADI diagnosis model works on a programming problem-solving.

An erroneous procedure is given in Fig.16. This procedure was written by a student while she was working on her homework for a Pascal programming course. This proce-

```
Procedure insert (var T: table; newitem: integer);
begin
     new(T);
     T^. item := newitem;
     T^. Lchild := nil;
     T^. Rchild := nil;
     if T = nil then
          if newitem < T^.item
               then insert (T^. Lchild, newitem);
               else insert (T^. Rchild, newitem);
end;
```

Fig. 16. An Erroneous Procedure.

dure seems to show that the student has almost all of the concepts which are needed in solving this problem, setting termination condition, giving recursive relationship, using correct formal parameter definition, inserting the node into BST correctly, and applying

the new nodes to hold the new data. However, she miswrote the termination condition, possibly just because she was careless. She also misplaced the operation of inserting a new node. She might have thought that she could apply the node at the beginning of the procedure, since a new node is needed in order to hold the new data every time the procedure requires node insertion. She did not realize that since this is a recursive procedure, this procedure will be invoked and will apply a new node for every recursive call no matter the insertion action happened or not, finally causing stack overflow. If a procedure has this mistake, even if the termination condition is correct, it will never reach the termination condition.

The system checks this procedure by the structure analysis and detects two mistakes: misplacing the operation of inserting a new node and miswriting the termination condition. The system searches the IADI knowledge base and hypothesizes three misconceptions by the abductive inference. There are four rules in the IADI knowledge base, that are associated with these two mistakes,

$$c_1 \longrightarrow m_2,\ c_1 \longrightarrow m_6,\ c_2 \longrightarrow m_6,\ \text{and } c_7 \longrightarrow m_2.$$

After taking the union of can(m_2) and can(m_6), there are three proposed misconceptions c_1, c_2 and c_7. The system ranks them by IPMs and forms the intermediary diagnosis C_r, $\{c_1, c_2, c_7\}$. In the verification process, the system displays three checking lists CL(c_1), CL(c_2) and CL(c_7), one after another. After getting a response from the student, the system will conclude the misconceptions the student may have. One possible response series occurs in the interaction between the system and students, and is listed in Fig. 17. The real system interface will be shown in next chapter.

There may be the following mistakes in your recursive procedure:
Mistake2: Misplace the operation of inserting a new node.
Mistake6: Miswrite the termination condition.

Now we will further diagnose your misconceptions. Please follow the instructions and answer the questions.

Let's consider the concept of the termination condition in a recursion. Do you think it is necessary to set the termination condition in your recursive procedure 'insert' ? (yes/no)

yes

That is correct.

Since we need the termination condition to end the recursive calls, we should consider a case as the condition. Then what is the condition? Please choose one from the following:

1. When the inserted data is equal to a certain value;
2. When a Binary Search Tree (BST) becomes empty;
3. When the root has a value which is equal to a certain value;
4. When a tree is not empty.

Please type a number.

2

That's correct.

The termination condition is when the BST becomes empty; how can you guarantee the condition will eventually arrive? Please choose one from the following:

1. Provide a smaller value through a parameter in the recursive call statement;
2. Provide a smaller structure, such as a subtree, through a parameter in the recursive call statement;
3. Check whether the subtree in a 'if' statement is empty or not without providing a smaller instance in the recursive call statement.

2

That's correct, you need to provide a smaller structure.

The next problem is what you must do when this condition is satisfied.

Please choose one number from the following:

1. Insert the new data;

2. Compare the inserted data with the value of a root and then make a recursive call again.

3. Do nothing.

3

Only the first action is correct; insert the new data. Before the termination condition is satisfied, the algorithm issues the recursive call to find a proper place to insert the new data. Since the base case has been reached, the proper place has been found, and you need to insert the data at this time.

Under the termination condition, you need to apply a new node and fill it with the necessary data. The problem is where these operations should be put? Please choose one number bellow.

1. At the very beginning of the procedure even before the base case.

2. When the base case is reached.

1

That's not correct. If you apply a new node before checking the base case, every recursive call will get one new node and the termination condition is never able to be satisfied. Thus the recursion will not cease until the stacks overflow. So the operation should be given when the base case is reached.

In this recursive procedure, what kind of structure do you have to use?

1. Use a loop structure for control of inserting n nodes in this recursive procedure;

2. Use a branch structure to decide to go to the base case or general case.

Please type one number.

2

That's correct.

Now let's consider the recursive relationship. In the relationship the recursion is expressed in terms of a smaller instance of itself. Did you express this relationship in a manner that expresses that the problem is identical in nature but smaller in size? (yes/no)

yes

That is correct.

Since we know that the recursive relationship will be represented in terms of a smaller instance of itself. And this small instance can be a smaller value or a smaller size of the data structure, what is the smaller instance in this particular procedure?

1. The smaller value in the node for each recursive call;
2. The empty tree;
3. The left subtree and the right subtree.

Please choose one number.

3

That is correct.

When we determine the recursive relationship in a recursion problem, do you think we need to guarantee that the reduced size will eventually become the degenerated case? (yes/no)

yes

Your answer is correct. But in your procedure it seems you did not give the condition to guarantee to reach the degenerated case.

Let's consider when it is appropriate to apply a new node. We need to apply a new node to hold the new item. And this operation can be done before or after we assign the data into the corresponding field. Is that correct? (yes/no)

no

Your answer is correct. We must assign the data into the node after we apply a new node.

We can view the operations of applying a new node and assigning data to it as one group. The next question is when do we use this group to apply a new node?

1. When the termination condition is true;

2. When the nontermination condition is true;

3. At the very beginning of the procedure, that is before any other operations;

4. Others.

Please choose one number.

3

That will cause this operation to be made at every recursive call. Actually it may just apply the infinite nodes for the first data until stacks overflow. The termination condition can never be reached.

Besides the one group under the termination condition, do we need to enter the group under the nontermination condition (that is the recursive case) too? (y/n)

no

That is correct.

You have the following misconception(s):

Misconception7: Misunderstand when to apply a new node to hold the new item.

Misconception1: Not having the correct concept of termination to a recursion.

Fig. 17. A possible Series Of Student's Response To The System

From the series of the student's responses, we can see that the misconception c_2 is not in the final list because the student gives correct answers to all questions in $CL(c_2)$. That means the mistake m_6 is caused by c_1. The system gets two wrong answers from the response to $CL(c_1)$ and one from the responses to $CL(c_7)$. But the $FPM(c_7)$ is 4

which is higher than FPM(c_1), which is 2. So the final diagnostic result is the ranked list $\{c_7, c_1\}$, which is C_f. The part of interaction is shown in Fig. 23.

CHAPTER FIVE

TUTORING ENVIRONMENT

OF

RECURSIVE PROGRAMMING TUTOR

This chapter describes the tutoring environment of RPT. The environment here is used to refer to the part of a system that presents system's diagnosing and tutoring process and supports the student's learning activities. A good tutor should have a good environment for a student to easily carry out actions and to see and understand the results and implication of those actions. The RPT system uses the hypermedia technique to create a graphical environment. The RPT environment includes the instruction part and diagnosis part.

5.1 Objectives For Creating RPT Environment

One fact for students to start to learn recursion is that they try to make an analogy of program structures between the given problem and the sample program, like we discussed before in section 3.4. This arouses us to set an instruction environment where sample programs are given and students can learn from these examples. After they master some basic concepts of recursion, they write their program and enter the diagnosis environment.

The most difficult thing about learning recursive programming is that the recursive statements in a program do not show the procedure step by step explicitly. For example,

the procedure of inorder traversal in a binary search tree can be simply written in recursion as follows:

```
Procedure inorder(T) { T is a binary tree }
begin
   IF T ≠ 0
   THEN  begin
                inorder ( Leftsubtree (T) );
                print    ( Data (T) );
                inorder ( Rightsubtree (T) )
         end
end
```

The student may not be able to see through the process at the beginning. The program is so elegant and succinct, it does not even contain explicit repetitions. It compacts a complicated process into several recursive statements. For program writing, the terse and simple form is good. But for the course of learning, it is better to reverse the procedure, unfold these statements, so that the originally hidden content in the recursive statement becomes visible to the learner. It is also necessary to help students understand how to solve a problem by solving its subproblems. For example, in the recursive program of quick sort, the way to sort a smaller instance and the result of the partially sorted sequence can offer a base for students to connect the similar strategy to the whole problem solving process. Because sorting a smaller instance must be easier to be understood comparing with working on a large instance. And the result for the small instance solving, the partially sorted sequence, is obvious to be able to be seen. So breaking a problem down to the subproblem, solving the subproblems and immediately showing the corresponding result will be conducive to learning. Hence it is necessary for programming tutors to show the execution results of a program at different stages in order for a student to understand the execution process of a computational algorithm. The RPT environment tries to implement these ideas. It creates a graphic illustration plus dia-

logues and menus in the tutoring environment.

The environment for the diagnosis process needs an interactive interface. Because in an interactive environment, it is possible for a student to gradually realize where he made mistake and what is the possible misconception. The explanations for basic mistakes and corresponding misconceptions are needed when the student still is confused. So the environment needs to provide a convenient method to show these explanations when a student asks.

Both instruction environment and diagnosis environment are created based on the hypertext model.

5.2 Hypermedia Environment

The RPT tutoring environment is built in a hypermedia environment. Hypermedia is regarded as a generalization of the hypertext. Hypertext is a model based on the assumption that human idea processing occurs through association. It connects the information in a network and provides the non-linear retrieval. Thus it can more closely model the deep structure of human idea processing (Clarson 88) (Shen and Zhang 89a). Hypermedia is an interconnected net of information in various forms from text to static graphics, animation and other types of media that can be accessed by the computer system (Younggren 88). Among the hypertext and hypermedia systems in existence, KMS (Akscyn 88), NoteCards (Halasz 88), and Intermedia (Garrett 86) are the most famous ones. Conklin gives almost complete historical description of the hypermedia systems (Conklin 87).

Although there is no generally accepted definition for hypermedia, the hypermedia systems can be characterized as follows:

• Network information organization

In hypermedia systems, information is represented by multi-media units, which are called objects. These objects may represent texts, pictures, video images, and voices.

They are connected together by links and form networks.

• On-line nonlinear retrieval

The network structures of hypermedia systems provide a n-dimensional information search space. Users can navigate in any defined path through the links.

• Extraction of common concepts

In a network representation, a common concept can be represented by a single object. This object can be linked to all other related objects. This implementation allows the same concept to be represented only once, therefore eliminates unnecessary redundancy.

• Intelligent environment

Hypermedia systems provide a strong capability to represent knowledge in various media. In this environment, users can execute their tasks in a way closer to human cognitive processes.

Applying a hypermedia model in a tutoring system allows providing the learning environment with graphic illustrations and dynamic representations of knowledge and knowledge relationships. The RPT environment further extends the hypertext model by incorporating dynamic program execution.

5.3 Overall interface

Creating the RPT environment has two purposes. One is to show the critical concepts of the recursive programming in a hypertext model based representation in order to aid the student's comprehension. Another is to reveal the execution of the recursive program in a multi-dimension environment so that the student can see through the problem-solving process. This environment is implemented on the Sun workstations with C and Sunview language under the Unix operation system.

The RPT system provides an overall interface for both instruction environment and diagnosis environment. This interface contains three components framed by a window

showed in Fig. 18. The three components are control panel, tty window and text window.

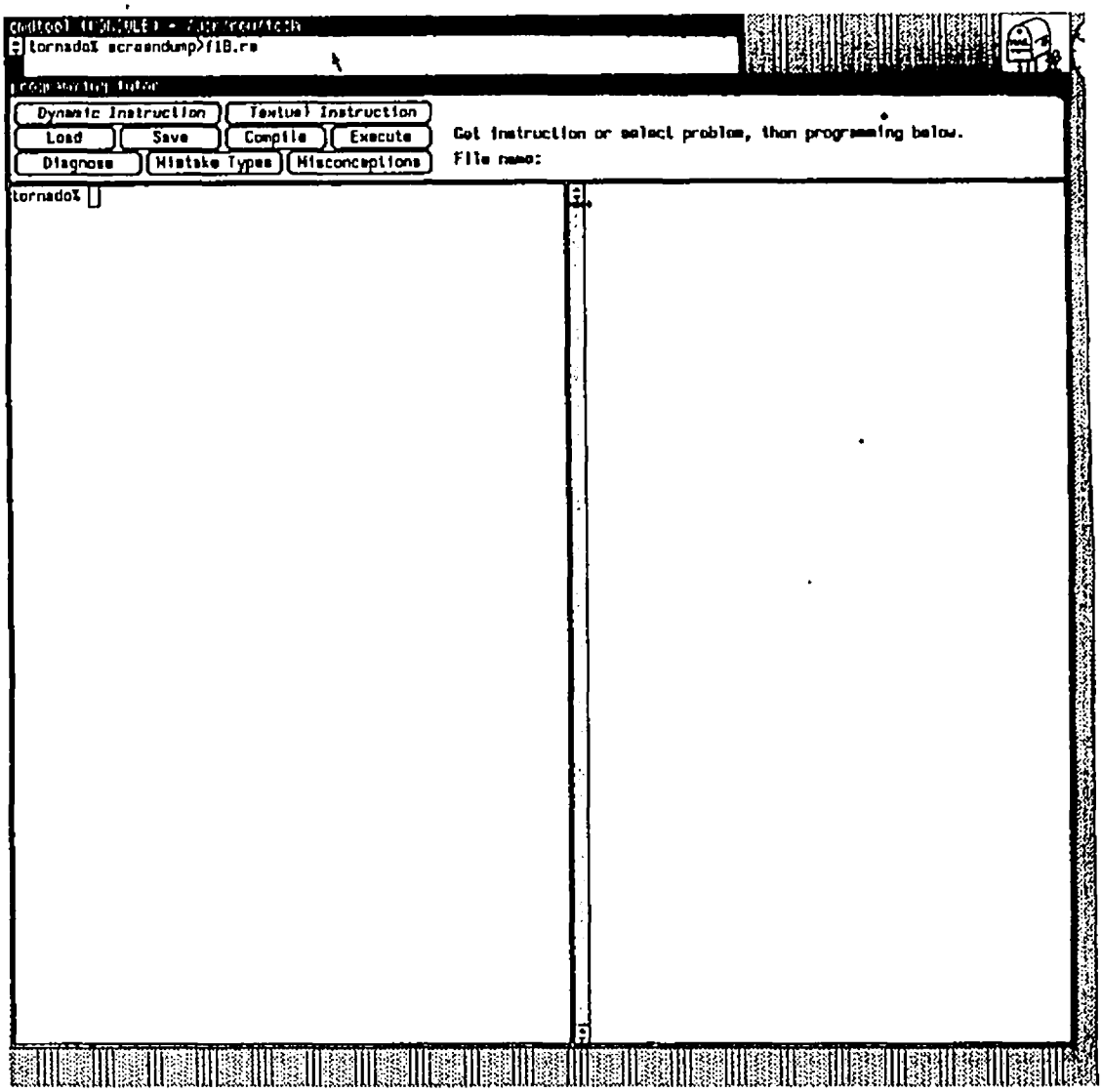

Fig. 18. Overall Interface of RPT

- Text window

  The text window is used for students to write and edit their program.

- tty window

The tty window is a usual command tool shell and is used to emulate a standard terminal. In the programming tutor, the tty window is used to compile and execute the program, and also to interact with student in the diagnosis process.

• Control panel

The control panel contains three groups of buttons. The buttons in the second row are for load, save, compile and execution of a file. The button "Dynamic Instruction" is for entering the instruction environment which consists of another set of windows. The buttons "Textual Instruction", "Diagnose", "Mistake Types" and "Misconception Types" are used in the diagnosis environment. The control panel leaves the space for user to write the file name after the item "File name," and the space in the right part of the panel for communication with the system.

5.4 RPT Instruction Environment

Some systems, such as LISP tutor (Anderson 89), PROUST (Johnson and Soloway 84) and MENO (Woolf and McDonald 84), tutor programming in an environment of dialogue and menu. The natural language is a good tool for communication. But sometimes it is not adequate in certain situations, particularly in the programming tutor. Natural language is usually considered as sentential representation which is sequential. One can not use any sequential representation to effectively impart a complicated and interconnected algorithm represented in a computer program. Actually, it is possible to use computer to create better learning environment. The RPT instruction environment creates a graphical dynamic representation to facilitates the learning. Students can understand many concepts better in a graphic environment since pictures and diagrams can provide more information than verbal descriptions and it has been claimed that one diagram is sometimes worth ten thousand words (Larkin and Simon 87).

Usually a student enters the instruction environment first for getting the basic con-

cept of the recursion. The RPT instruction environment allows a student to select a recursive program to work on. After a student enters the environment by selecting the "Dynamic Instruction" button on the control panel, he can see a window that is the root node of a processing tree. A processing tree describes the execution process of a recursive procedure. It consists of spread nodes and abstract links. A spread node is a basic unit in the instruction environment which represents a status of a program when we consider a set of input data attached to it. An abstract link represents a procedure calling relationship. The processing tree is created to present the different levels of recursion of a recursive program. The spread node at the initial state with the original input is the root node. In addition to the representation of the source code of a recursive program displayed in a spread node, the system provides a graphical representation for the input data in a binary search tree. The system is also be able to automatically generate the corresponding input data for the program at any particular spread node. Each program in a spread node can be executed with the relevant input value at the student's option. These features provide clear visualization of the recursion process thus is very conducive to learning. The following subsections describe the detail structures and functions of the processing tree.

5.4.1 Representation of a spread node

A spread node represents one status of the recursion at a specific level. A spread node contains the following facilities included in a single frame.

• Start panel

A start panel contains selection button, instruction button and message item. The buttons are for a student to select specific control functions to be performed and the message item is for providing messages to the student. The start panel is present only on the head of the root spread node and is not available in other spread node.

- Lesson selection button

  This button is labeled as "Selection of Lesson." Pressing the right mouse button when the cursor points on this button brings a menu to be displayed. This menu is called lessons menu. The menu shows a list of program names for students to choose one of them as the tutoring lesson.

- Instruction button

  This button is labeled as "Instruction." Clicking the left mouse button on this button will cause a text frame to be displayed. The text frame shows a specific instruction text for the selected tutoring lesson. This text includes detailed description of the problem and also the instruction on how to solve the problem.

- Message item

  This item is used for the system to show warning messages and/or specific instructions according to the on-line situation.

- I/O panel

The I/O panel provides the space for displaying the input argument values and the output data for the program at a certain level of recursion. This panel is bellow the start panel.

- Input item

  This item is labeled with "input" in root node and "Input Subtree:" in non root spread node. This item displays the input argument values for the spread node. The input argument value is typed by a student at the input item in root spread node. The values displayed at input items in other spread nodes are generated by the system. Besides the data display, the input item also provides the graphic illustration for the input data (please refer section 5.4.4).

- output item

  This item is labeled with output. This item shows the returning data for the spread node upon the completion of execution.

• Program list panel

The program list panel displays the recursive procedure source code when a student chooses a tutoring lesson from the selection button. This panel is on the underneath part of a spread node.

After a student enters the instruction environment, he gets a basically empty root spread node frame with a message suggesting him to click at the "Selection of programs" button to select a program to work on. After he does so, he gets a list of the available programs. Fig. 19 shows the root spread node. In Figure 19, the student selected the "inorder" program which tutors the subject of solving the inorder traversal problem recursively. The source code of the "inorder" program is shown in the program list panel of the root spread node frame. At this point, the system reminds the student input the argument values at the Input Item of the I/O panel. When a student clicks at the instruction button, the general instruction for this lesson is displayed in the text frame, which is shown in the right of Figure 19. The other spread nodes will be showed in the processing tree described in the next subsection.

5.4.2 Processing tree

Many instructors like to give the classroom teaching of recursive programming in the following way:

(1) Write the program on the blackboard and use some sample input argument value to simulate how the program works.

(2) As the simulation goes on, when the simulated execution comes a recursive call statement, the instructor draws another instance, which is in the smaller size, of the same program on the blackboard. The simulation then continues at the new instance of the program.

(3) When the execution of a certain instance of the program reaches a base case, the recursion stops and the control returns to its parent level at the original call state-

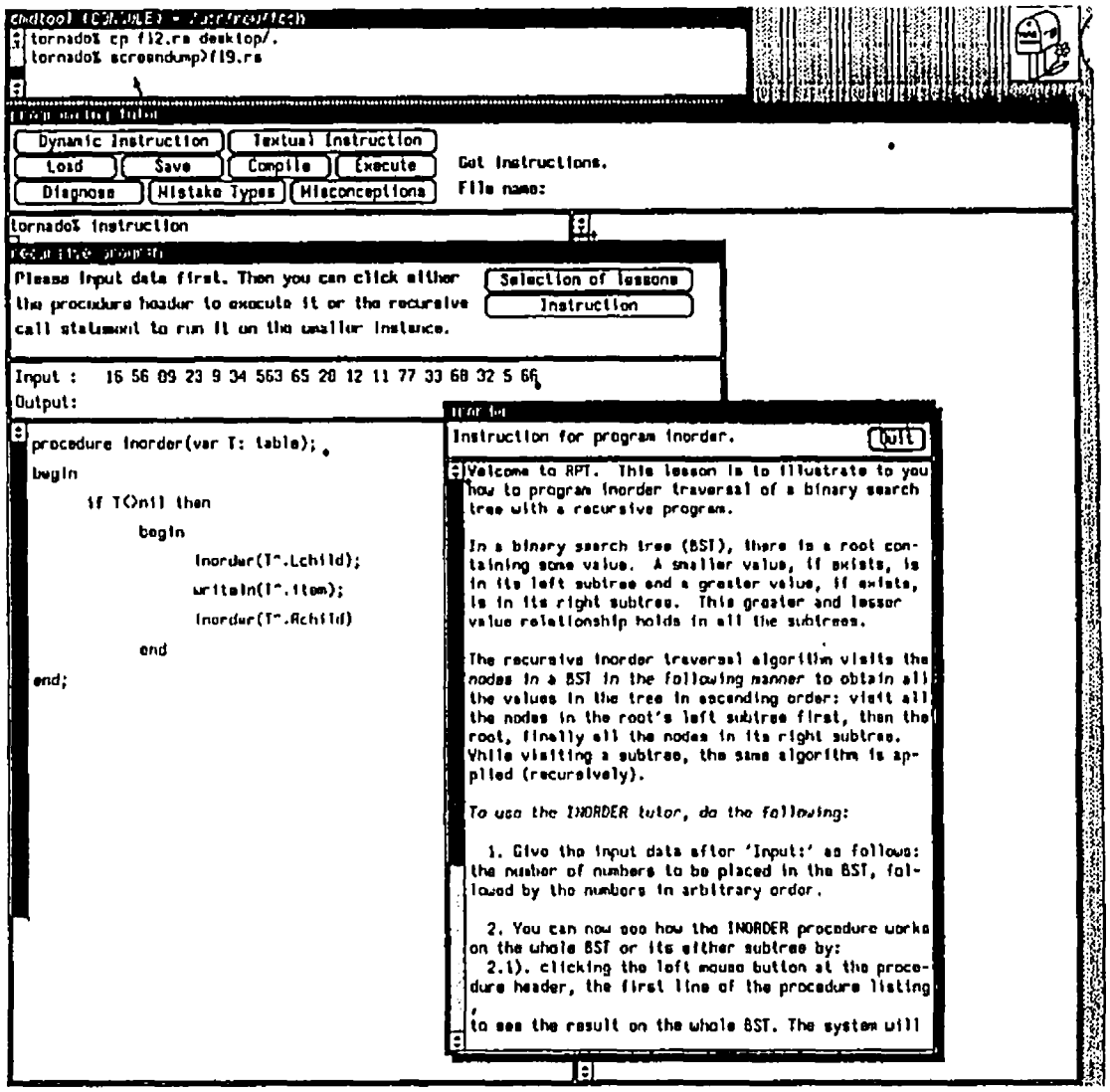

Fig. 19 The Root spread Node

ment with the resulting values. The execution continues from there on at the parent level.

The drawings on the blackboard discussed above naturally amount to a tree like the one in Fig.20. This tree is called the processing tree since it represents a process to reach a solution. The tree shows a process for the inorder traversal procedure with the

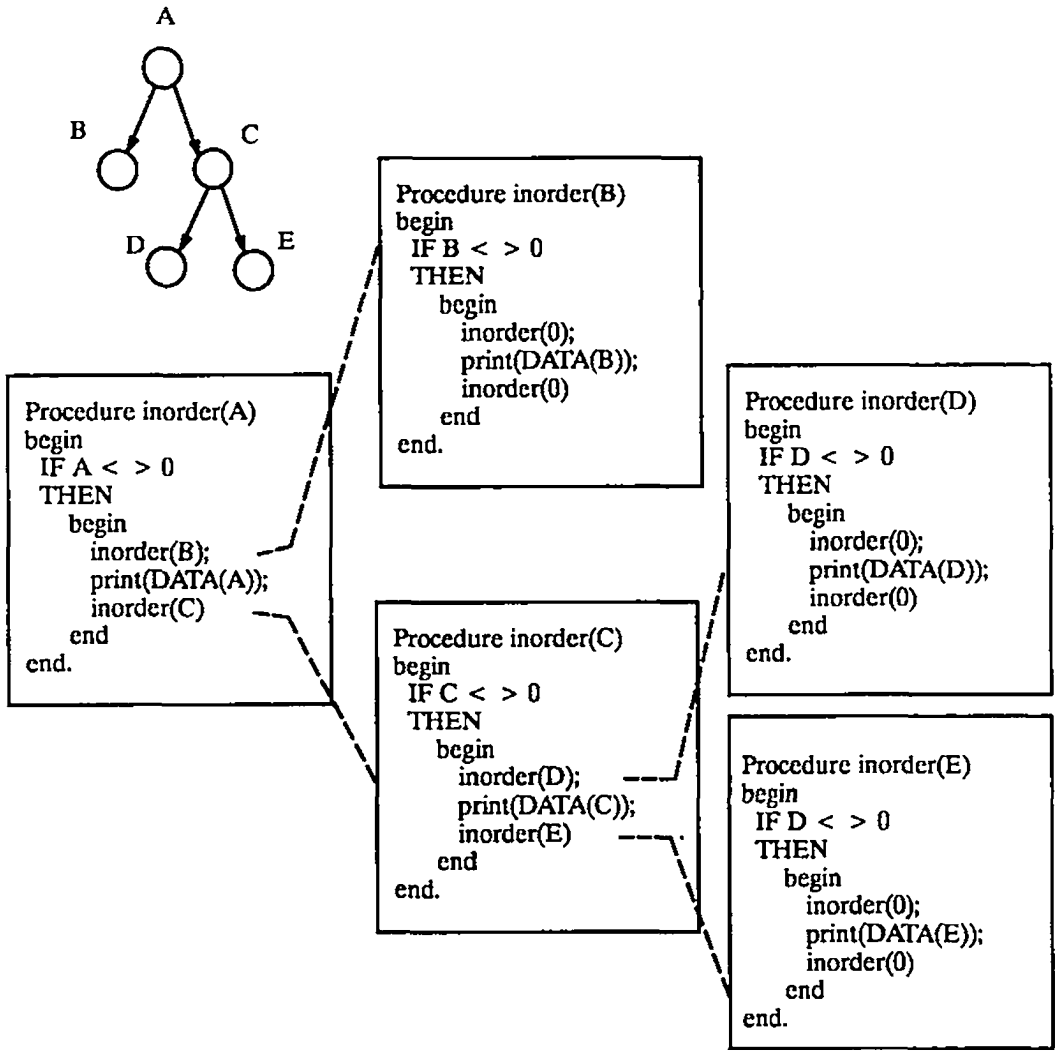

Fig.20 One Demonstration Of Processing Tree
For The Inorder Traversal Problem With Input Binary Search Tree A

input binary tree A. Each "instance" of the program mentioned above amounts to a spread node. The sentences "inorder (B) " and "inorder (C)" in the procedure associate with a spread node respectively by an abstract link, which will further spread these subprocedures on the subtree B and C. In the tutoring environment, the student is allowed to select the actions at his/her own discretion. In other words, the student may

follow the execution in a certain sequence as above, or may selectively do and see what is desired. This environment facilities different learning paths for different students. Thus there are different processing trees for different learning paths.

More formally, a processing tree consists of the spread nodes and the abstract links that are procedure calling relationships among spread nodes. These calling relationships are kept a track internally by the system. The processing tree can describe the process of a recursive program execution at any level. The root spread node is defined at level 1. Every recursive call statement in the list program panel of the current spread node is associated with a spread node at the next level, whose level number is one greater than the level number of its parent. Spread nodes in different levels have identical structures but different input values that are for subproblem solving in different size. A spread node at level i is considered to be at a higher level than another spread node at the level $i+1$. A processing tree may be extended very deep. But, generally speaking, the simulation on primary several levels can give more inspiration to students, so it is most useful and effective for student to master the recursive concept. That means the learning process usually do not necessarily go to the all branches of the processing tree, but stops at certain level.

Fig. 21 shows a processing tree for the inorder traversal procedure in the RPT environment. In any spread node, the student can click the mouse button at a recursive call statement to cause an expansion into a child spread node.

5.4.3 Automatic generation of input argument values

After a student chooses a lesson, he needs to follow the specifications given in the instructions for the lesson and type the input argument values on the input item of I/O panel. In the example of inorder traversal problem showed in Fig. 21, the number 10 at

* As the first step, RPT system just deals with the recursive call that has been explicitly referred to itself.

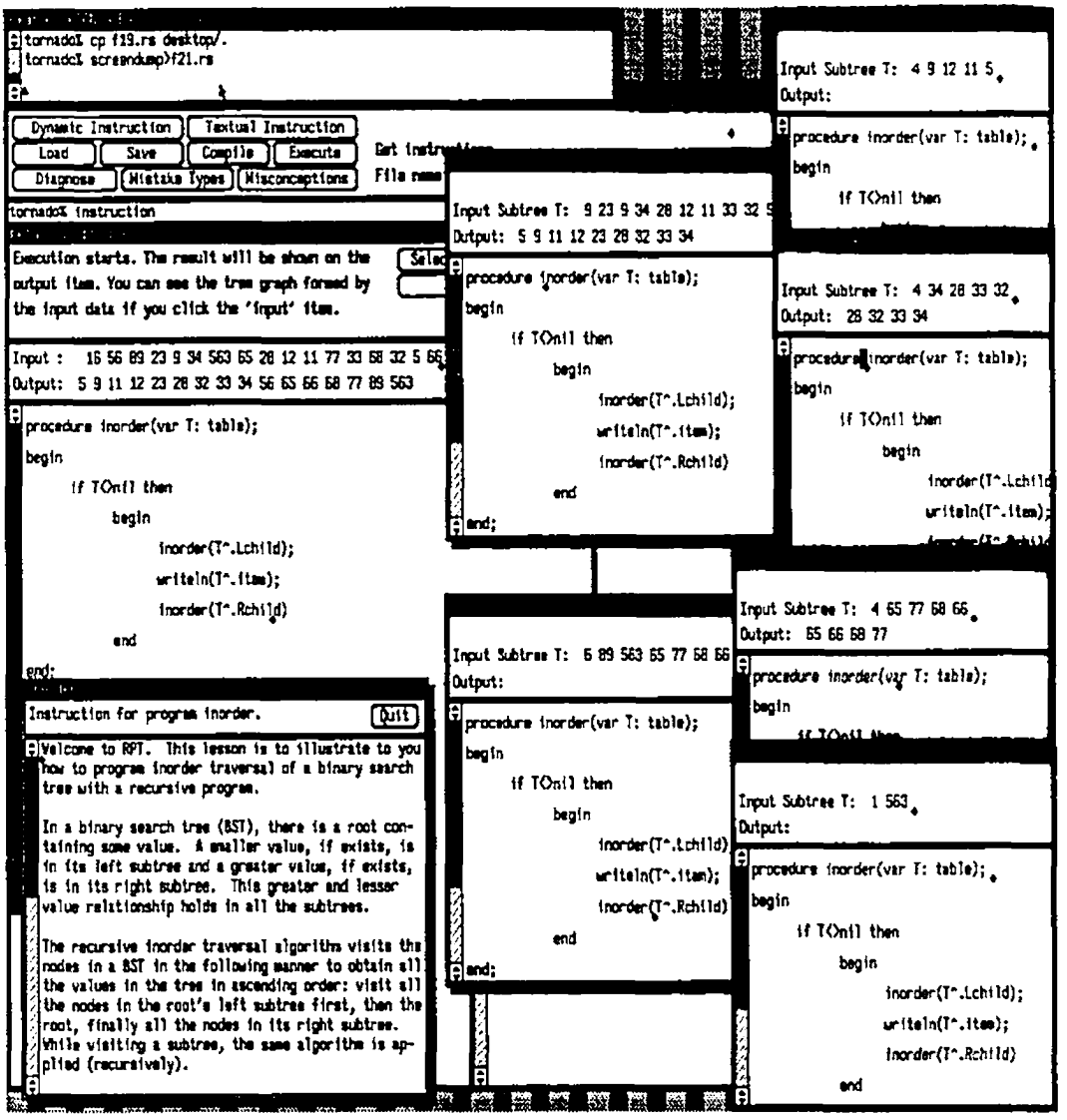

Fig.21. One Processing Tree

the beginning of the input string is the number of nodes in the binary search tree. The other input numbers are the data for nodes of the binary search tree. The input data given by the student goes through another hidden program that generates the interior data structure, the binary search tree, which is in term used by the recursive program "inorder."

After a child spread node is created through the order of a student, clicking the

left mouse button on one recursive call statement, the input argument values for the child spread node are automatically generated by the system and are displayed at the Input Item of the I/O panel in the child spread node. These data are for the smaller size problem solving. In the inorder traversal problem, for example, the input data of a child spread node forms a left (or right) subtree. Under this representation, the student can see the recursive relationships and understand how the recursion works on the smaller instance, therefore it helps students to understand the whole problem solving process. In cases where the current spread node already presents a leaf that the recursive call should not be performed but the students selects such a call, a warning message will be provided. Fig. 21. shows the generated input data in terms of the smaller instance for the corresponding subtree in the input item for each spread node in that processing tree.

5.4.4 Graphical representation of input data

When an instructor demonstrates how to solve the inorder traversal problem, he/ she usually draws a binary search tree for the input data and explains how the algorithm handles the data. The graphical representation of a binary search tree is very helpful for students to understand the algorithm. It is not adequate if the input data is given only in the numerical form. The RPT instruction environment provides the graphical representation for the input data to imitate the human tutors' actions.

For a set of the input data, the system can automatically generate a graph that is a binary search tree matched with the displayed data at the Input Item. This graph is generated and displayed only when a student asks by clicking the left mouse button on the label "Input" of a spread node. The system can generates the graph for every spread node, no matter on what level of a processing tree it is, provided the student clicks the mouse button on the label "Input" in a selected spread node. Thus a student can see a subtree of original binary search tree when he orders the display at a child spread node. Fig. 22 shows the graphical representation of the input data for several different spread

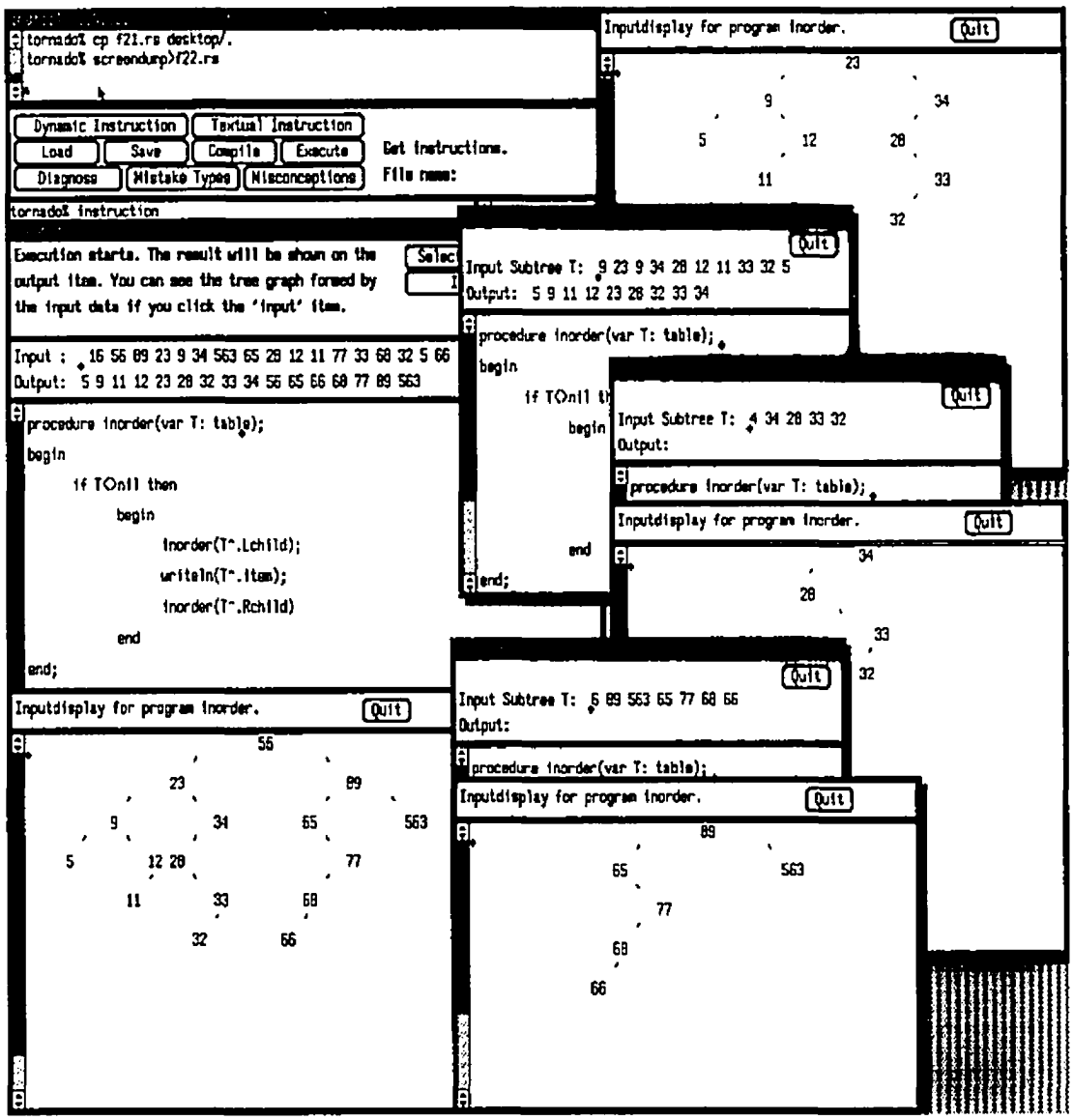

Fig.22 Spread Nodes With The Input Binary Search Tree

nodes. In the learning process a student can see and learn the way how the given instance becomes smaller and smaller and finally arrives an empty tree. This series of binary search trees gives a visualized process to demonstrate how the recursion proceeds on those data.

5.4.5 Execution of the recursion in a spread node

The hypertext systems were traditionally used in organizing and displaying texts and materials. Some hypertext systems even allow procedural attachment to enable the system to perform specific tasks (Conklin 87). For the purpose of recursive programming tutoring, static displays of source programs without animation are hardly conducive to learning. Procedure attachment does not allow the needed animation at arbitrary levels of the recursion. The RPT instruction environment allows the display of the source code and the input argument values at any level. It also allows the selected execution at any level at the direction of the student. The RPT environment has extended the typical hypertext model and it is conducive to learn recursion.

The system provides execution of the recursion in an arbitrary spread node of a processing tree. This is achieved by the student's simply clicking at the selected procedure header section in the spread node. Upon completion of the execution, the results derived from the given input values to the specific spread node are shown to the student in the Output Item of the I/O Panel in the spread node. If a student clicks the left mouse button on the procedure header of the root node, he will get the resulting output for the whole problem. If the student likes to know the execution result of any subset, he just needs to activate the procedure by clicking at the procedure header of corresponding spread node. The result for the subset will be shown on the screen. This capability involves more than just displaying a text as a typical hypertext system does. This system displays the source code but also executes the recursive program to show the dynamic process. In the example shown in Figure 21, the root spread node in the leftmost position and the center spread node illustrate the results of the selective execution ordered by the student who wanted to see the inorder traversal of the initial input binary search tree, and its left subtree and the right subtree of the root. In this case, the corresponding output are "5 9 11 12 23 28 32 33 34 56 65 66 68 77 89 563," "5 9 11 12 23 28 32 33 34"

and "56 65 66 68 77 89 563" for the three binary search trees, respectively.

5.5 RPT Diagnosis Environment

In the overall interface, a student usually enters the instruction environment first. After he got some concepts on the recursion, he can selects one problem to work on and write his program on the text window. If the compilation can be passed but run time errors exist, the student can enter the diagnosis environment to find what are the possible mistakes and misconceptions.

The main feature for the diagnosis environment is that it provides an interactive interface. The diagnosis activities in this environment follows abductive and deductive reasoning path and design plans. The communication between the system and students is in a natural language dialogue style. The system also provides menu, texts and buttons to allow a student to select problem from a list and see the description for the types of mistakes and misconceptions at a student's initiative by open a text window with simply clicking on the corresponding button. The following subsections introduces the each part of the environment.

5.5.1 Program submission

Before diagnosing, a student needs to write the recursive program and eliminate its all syntax errors. The diagnosis environment provides following facilities for submitting a nonsyntax program (Please refer the overall interface in Fig.18):

- Problem selection button

  This button is labeled as "Textual Instruction". Pressing the left mouse button brings a menu to be displayed. This is the problem menu. This menu shows a list of program problems for student to choose to diagnose. Releasing from this button causes a text window to be displayed. The text describes the problem and the related basic concepts.

• Name item

A student should define a file name for a program. The name can be given at the name item which is on the right bottom line of the control panel, labeled as "File name." The student can give either a name as the file name or the path name of a file for system to access.

• Load button

A file will be loaded in the text window if clicking the left mouse button at "Load" button. The file name should be given on the name item in advance. The file being loaded can be any file stored in the user's file system. It is not necessary to be the one being edited in the text window.

• Save button

Clicking the left mouse button at "Save" button causes the file in text window to be saved under the given name in the name item.

• Compile button

Clicking the left mouse button at "Compile" button gives an order for system to generate a command to compile a program. The program is in a file under the name showed in the name item. The generated command is in tty window. It also generates a corresponding executable object program code. The result of compilation showed in the tty window.

• Execution button

The name of executable program code will be showed in tty window and executed by the system if a student clicks the left mouse button on the button "Execution".

• Message item

This item is in the right part of the control panel. The system sends message at this item to communicate with students and give instructions how to use this environment.

5.5.2 Diagnosis interactive process

To enter the diagnosis process, the student needs to click the left mouse button on the "diagnose" button. Then the system starts the diagnosis process. The process is proceeded in tty window.

First, the system picks out the recursive procedures from the being diagnosed program and lists all the recursive procedure names. The student chooses one recursive procedure to diagnose. The system redisplays the code of the recursive procedure in the tty window in order for the student to review the recursive procedure. At the same time, the system is detecting the mistakes of that procedure and lists the possible mistakes in tty window.

The system hypothesizes the possible misconceptions from the obtained mistakes internally. It does not show them to the student since these hypothesized misconceptions needs to be further verified.

Then the system begins a dialogue with the student. For each hypothesized misconception, the system displays its checking list, lists several related topics either by rising questions or by making choices, and analyzes student's answers. These questions and choices are arranged according to the cause-effect relationships between the misconception and mistakes, and also the design plans of the recursive program. For each incorrect answer, the system gives explanations. Fig. 23 shows an interaction while a student is using a checking list. After the system gives explanations, the student may realize what kind of misconception he has. Then he can correct the mistakes or give another version of the design. The student may decide to stop the display of these lists whenever he thinks he already got the enough information to correct his mistake. The system can also give the suggestion of keeping on checking, when there are remained checking lists associated with candidate hypotheses in C_r whose IPM are not very low. So this process is in a mixed-initiative tutoring environment.

Finally, the system concludes the misconceptions based upon the student's answer

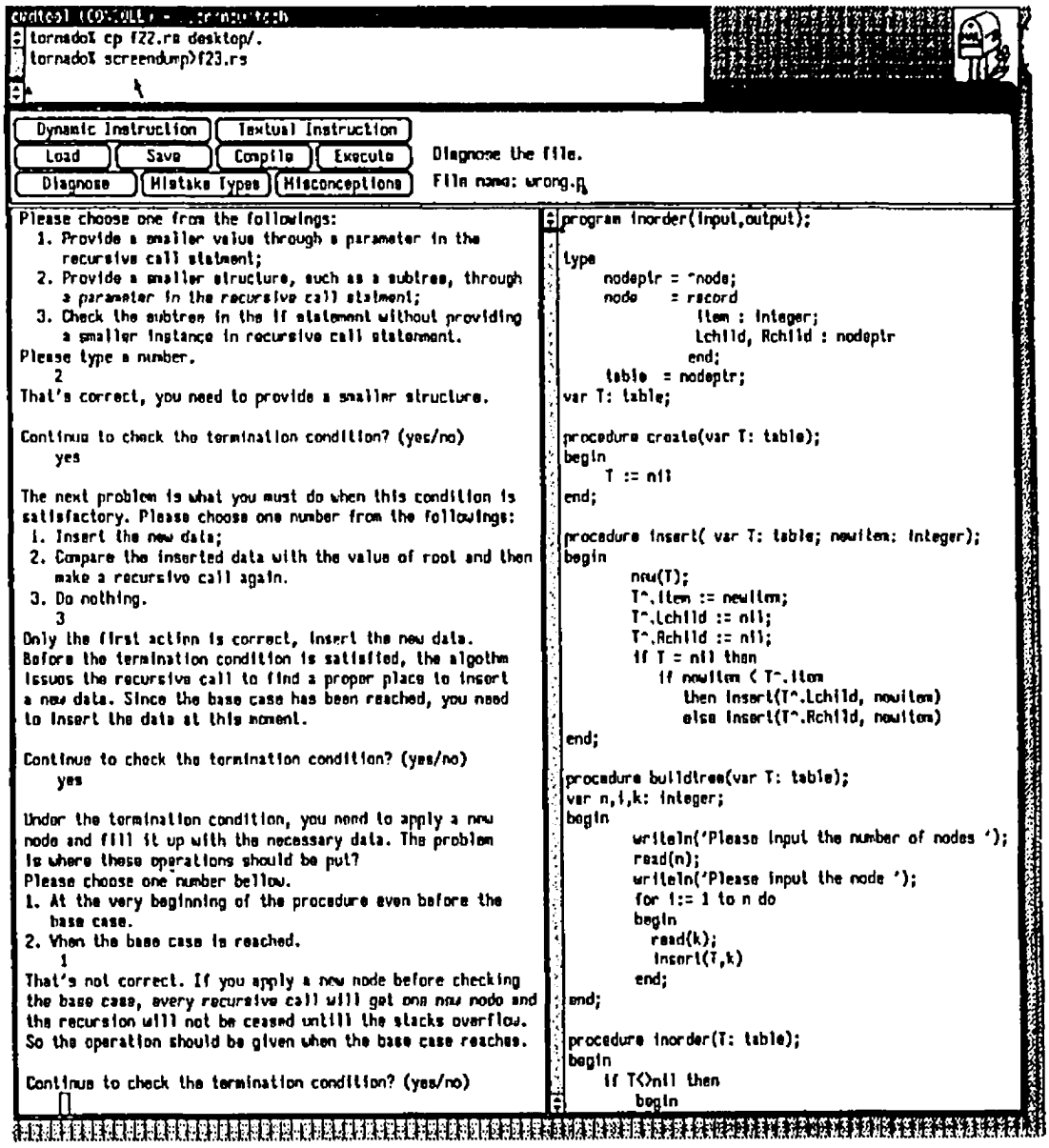

Fig. 23. Part Of A Checking List In Use

to the related questions and choices. The misconception types are listed in the tty window finally.

5.5.3 Types of mistake and misconception checking

When the system finds the mistakes and misconceptions in the diagnosis process, it reports their types and gives very brief explanations in the tty window. For the novice students, they may not fully understand the meaning of error messages. The types needs to be further explained. The system uses one piece of text to explain one mistake type or one misconception type. For each type the student can call the text displayed in a text frame when he needs. There are two buttons facilitating the type checking which are listed bellow:

- Mistake type checking button

  This button is labeled as "Mistake Types." If a student clicks the left mouse button at this button and gives the type number following the prompt in the message item, then the system will open a frame to show the text which gives the corresponding explanation to that mistake type.

- Misconception type checking button

  This button is labeled as "Misconceptions." Using operations on this button in the same way as on the mistake type checking button, a student can see the further explanations to misconceptions in the text frames.

Fig. 24 shows the text frames that give the explanations for misconception type c_1 and type c_7. These types are related with the report in the diagnosis process displayed in tty window.

5.6 File System of RPT

The following subsections introduce the file system of RPT. The RPT system is only a prototype now. But the way the files are organized and the system works provides the system engineers facilities to easily add the subjects to be tutored and the problems to be diagnosed. Whenever an instructor wants to add a recursive program in a different subject, the system engineer can create a group of files and put them into the system.

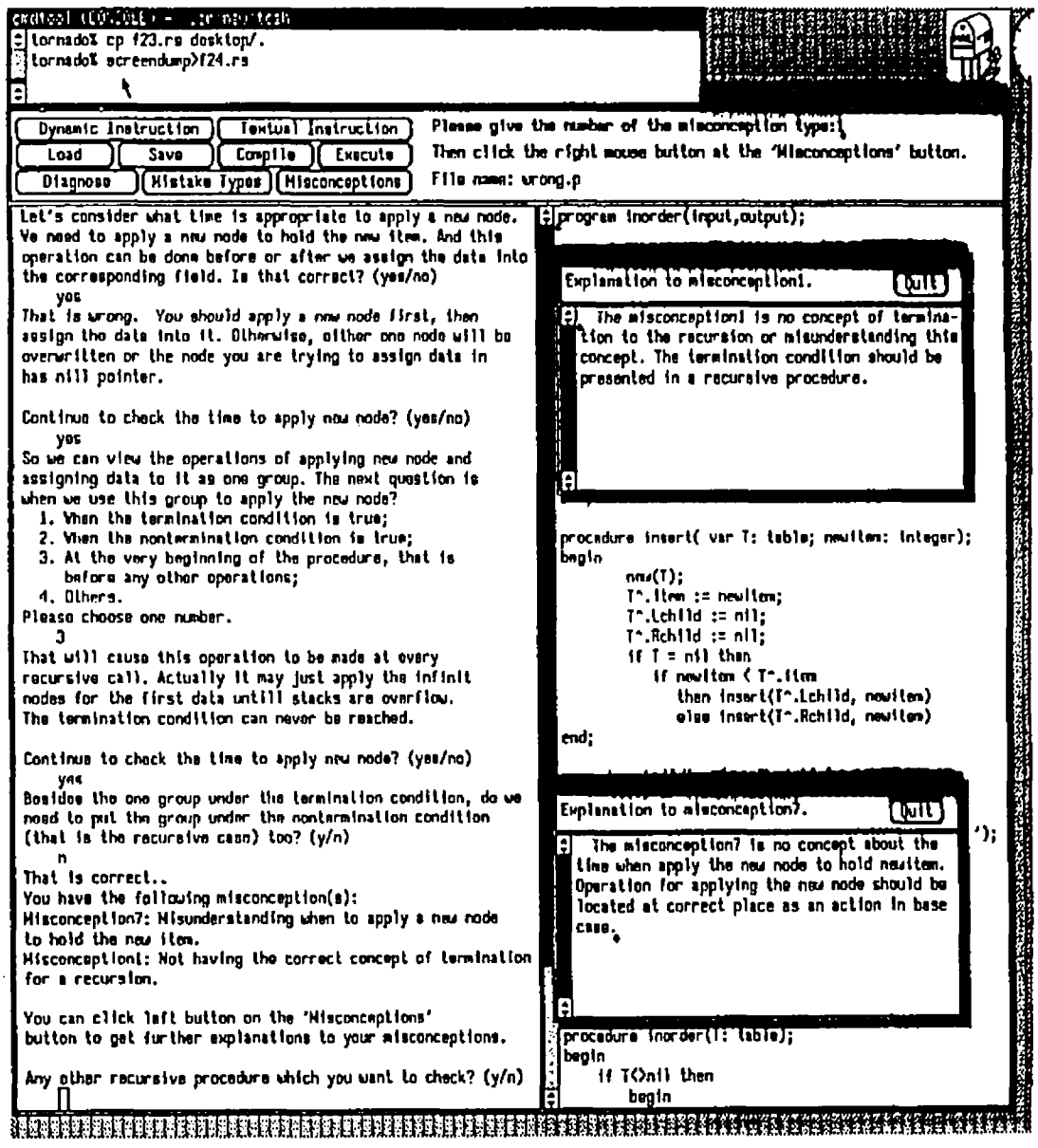

Fig. 24. Types Checking For Misconceptions

5.6.1 File system for instruction

In the RPT system, one lesson is one recursive program. For each program there are a group of related files. Their names are ended with .instruct, .p, .out, .input, and .program respectively. For example, if there is a lesson of inorder traversal program and

the corresponding source code file is inorder.p, then there is a set of files for this lesson, whose names are inorder.instruction, inorder.out, inorder.input and inorder.program.

- .p file

  This file holds the source code of a recursive program for a lesson.

- .instruction file

  This file gives the textual description to introduce the basic concept about one recursive program. For example, for the program of inorder traversal problem solving the text explains the problem in terms of the general problem description. If it is necessary, it will help student review some prerequisite knowledge, such as "What is inorder?"; "What is the differences between inorder and preorder, inorder and postorder?". The text also includes the specification for using the tutoring tool. The text file will be shown in the text window when a student selects the selection button in the start panel of a root node, like the frame in Fig. 19.

- .out file

  This file is an executable file prepared by the system engineer. When a student asks to run the program in .p file, the system automatically generates the corresponding executable program with the file name ending with .out and executes this file.

- .input file

  This file is for the specific display of input data. The system engineer makes this file. For example, for inorder traversal program the system engineer makes this kind of file for students to see the graphical representation of input data.

- .program file

  This file holds the recursive procedure which is picked from the program in file .p by the system. When a student studies on a program, the system will display the procedure in this file on the program list panel. For the inorder traversal procedure, the system provides the functions of execution and the function of display. Both of the functions can work on different instances of this recursive procedure. A student can

see the execution result for the current input when he clicks the left mouse button at the procedure header and see the display of a spread node for next level of recursion when he clicks the button at the recursive call statement in the program list panel. For other programs, the system needs to give different functions.

The first four kinds of files are prepared by the system engineer and the last one is generated by the system. For each new subject, that is a recursive program, the system adds these four files into the system. When the system generates the lesson menu, it makes the menu one item longer since the system finds there is one more .instruction file in the system. Thus the length of menu is exactly according to the number of lessons. This is a flexible way to add new lessons.

5.6.2 File system for diagnosis environment

Files in diagnosis process supports the problem understanding, mistake detecting and misconception recognition. There are following different kinds of files.

- Problem description file

  This file gives the textual description to one recursive program when a student selects a recursive program to work on. This text file has the name ending with .problem. This file briefly introduces the basic concepts about that recursive program and the way to start to solve the problem. It gives the information such as "what is the problem?" "What is the recursive relationships in the problem?" "What is the termination condition of the recursive program?" This file is displayed when a student selects a problem from the problem selection button in the control panel.

- Recursive procedure file

  This file holds the recursive procedure which is picked out from the program in file .p by the system in diagnosis process. This file has the same name as the recursive procedure name. When a student selects a recursive procedure to work on, the system begins a structure analysis on this procedure during the mistake detection process. The

system also displays the procedure in this file in the tty window to let the student review.

- Mistake explanation file

  For each mistake type there is one file to explain the corresponding mistake. It is aimed to help students to understand what kind mistake he has made. This text description is a more detailed description than the brief information in the mistake report. This file is displayed when a student requests from the mistake type button in the control panel.

- Misconception explanation file

  For each misconception type there is one file to explain it. The system uses it to help students to understand what kind of misconceptions he may have. This text is a further explanation to the misconception. This file is displayed when a student requests from the misconceptions button in the control panel.

The system engineer can add more recursive problem, mistake types and misconception types whenever needed. For the different problems he needs to add different problem description file and support files. When the more mistake types and misconception types are found, the corresponding explanation files of mistake and misconceptions need to be added.

5.7 Features of RPT environment

The RPT environment is built with the hypertext model plus the program execution. It provides graphic illustration, multi-dimension display and visualized proceeding to help students to understand a recursive problem solving process.

5.7.1 Graphic illustration

Pictures and diagrams can readily be used in the knowledge representation, and sometimes they can give more information. Graphic representation can immediately

show the objects and their relationships so that people can accept information promptly by using human intelligence. The recursive programming is very difficult to be accepted by novices especially when they first encounter it. It seems that an unknown process is built over the same unknown process, only in multiple different ways. Indeed, comparing with other programming concepts, such as variable definition, input, output, loop structure, it is difficult to master. Using a computer to teach students, especially when it is possible to supply a graphic illustration environment, not just verbal interactions, will provide an opportunity to alleviate the degree of difficulty. The RPT environment allows students to use the processing tree to represent the recursive relationships in nested programs at different levels and to use the graphic representation for a binary search tree to depict the input data. Thus, this environment provide one more dimension in user–interface than other programming tutor system which only provide dialogue and menu. The environment of graphic illustration, in which the student learns the concepts and details about recursive programming, can foster intuition for abstract concepts and will make a lasting impression on the mind of students.

5.7.2 Multi–dimension display

In the RPT environment, travel or search among the spread nodes in a processing tree can be easily carried out by a simple click on the mouse button from any node in the tree. This multi–dimension retrieval process furnishes multiple dimension navigation paths. In the most tutor environments, only the static display is allowed. These environments only give display for the current state step by step in the linear order, such as in the system LISP tutor and GEOMETRY tutor [Anderson 89]. Sometimes the sequential display conforms the course of human knowledge acquisition. But the multi–dimension display is more powerful to catch the brainstorm that just flashed through the student's mind and to follow the instantaneous idea in the cognitive processes. The RPT environment provides the display at multiple level not only for program code but also for the

input and the execution result of corresponding program. And this display can be controlled by students. This mixed-initiative display endows students with more chances for creative thinking.

5.7.3 Visualization of recursion

The most difficult thing for student to learn the recursive programming is that the program is too elegant and succinct to understand the program content. This is due to the feature of the recursion that it defines an infinite process in a terse statement. In the RPT environment the automatically generated spread node can unfold the process of a recursive program. The execution result for the subset of a corresponding input data can be displayed by the student's requiring. Thus, the originally hidden program and the underneath process are visible to the student. When a student learns the case at the spread node, he will see corresponding unfolded program by visually opening the knowledge environment. This gradational display allows students to go through the processing tree from the root to any node in the tree at any level, therefore get corresponding program at an incremental refinement. Different levels of refinement are needed for effectively tutoring different students.

CHAPTER SIX

EMPIRICAL EVALUATION

ON

THE PERFORMANCE OF RPT

Previous chapters have presented the IADI diagnosis model and its applications in a recursive programming tutoring system. This chapter evaluates the performance of RPT with emphasis on the IADI diagnosis model.

The evaluations of ITSs have been underemphasized in the past. Even though there are few of ITS systems intended to do so, there is no standard set of evaluation methods for addressing these problems (Littman and Soloway 88). Because the field of ITSs is too young, building ITSs is still somewhat an art, and there are few ITSs that can be called "finished." Designers of ITSs are currently more concerned with usefully guiding the development of their systems, than with determining if they are effective educational end products.

The evaluations of RPT reported here show an encouraging result from the experience in diagnosis processing of RPT and its running examples. The observations and analyses show that students are learning from the tutor. And this fact is consistent with our subjective efforts and wishes when we set out to construct the tutor with the diagnosis model.

Although empirical tests indicate that an encouraging result has been achieved, there still are some problems. These problems involve transporting the tutor from the research environment to the practical environment. It needs to pay more efforts to solve

these problems. The further considerations and possible works are addressed in the next chapter.

6.1 Bug Collection

The IADI knowledge base currently is for insertion problem which is a problem of inserting nodes into a binary search tree recursively. The possible mistakes in the IADI knowledge base are classified into thirteen different types. This classification is based on the program structures of students' programs for solving this particular problem. These mistake types are summarized from the students' solutions to the insertion problem in their homework and class work for the PASCAL programming class, and also from the sample programs provided by experienced instructors.

At the first stage, the author created bug programs to simulate the detection object based on obtained students' bug programs from helping students to complete their homework. At the first round, eleven different mistake types were summarized. These mistake types were then investigated by several experienced programmers and instructors, and two different mistake types (mistake type 12 and type 13) were added to the knowledge base based on their experience. For each of these mistake types there is one program module in the RPT system to be implemented which is used to analyze and recognize the corresponding mistake.

Then the RPT system is used to analyze the students' programs, and is also expected to receive more information from the real objects of the bug detection. In the 1990 fall semester, the intention of collecting the bug solutions from the students' homework failed because most submitted homework was a correct version of the solutions; the intermediate solution, the bug programs in their previous versions were not turned in. In the 1991 spring semester, students were asked to submit their programs for the insertion problem during class time as class work. The information obtained from their class work is shown in Fig 25.

Total number of programs :	27
Number of analyzed programs :	20
Number of correct programs :	6
Number of bug programs :	14
Number of bugs :	20

Fig. 25. Program Collection

This class work was taken in 30 minutes. The purpose of collecting these programs was to get to know what is the students' thoughts when they were learning the recursion, and what are the possible mistakes they will make at the beginning. There were about 40 students in the class, but only 27 programs were submitted. The 27 programs were made by 24 students. There are three programs in the different version, but by the same students. Among the 27 programs, only 20 programs are analyzed because these 20 programs are syntax error free programs and they are significant for the insertion problem solving. That is, we do not want to analyze on incomplete program which has no meaning to solving the problem, even if there is no syntax error in that program. There are 6 programs which are bug free and give correct solutions. In the other 14 programs there exist 20 mistakes which were going to be analyzed by the RPT system.

6.2 Evaluation of Bug Detection and Bug Collection

The preliminary RPT system ran the 14 students' programs and the result of the sample data is shown in Fig. 26.

The total number of existing bugs in the 14 student's programs is 20. From the results we can see that the preliminary RPT system can recognize the most of the mistakes, (90% in this test). But some mistakes are misreported, which means those base

Number of bugs :	20
No. of bugs recognized correctly :	18
No. of bugs recognized incorrectly :	2
No. of misdetected bugs :	6

Fig. 26. Result Of Running Preliminary RPT System On Sample Data

are classified as the incorrect bug category. For example, the two bugs recognized incorrectly in Fig. 26 were erroneously detected as mistake type 11 when they were actually mistake type 6. There also existed some misdetections under which the correct programs were looked upon as bug programs or the correct parts of the code were reported as mistakes. In three bug programs, a piece of correct code is misdetected as mistake type 5. Also three times the system erroneously reported mistake type 6 in programs which were correct. The misdetections are involved more than three modules in the preliminary detection program. It was necessary to modify these modules in the preliminary diagnosis program in order to eliminate the misdetections.

The reason for the misdetection is that the preliminary system did not contain the case which is more general. For example, for the mistake type 5, "Did not provide the smaller instance", the module to check this mistake only checks whether the smaller instance such as T^.Lchild is presented in the actual parameters. However, some students use the assignment statement like T:= T^.Lchild first, then they do not need to present T^.Lchild in the parameters of the recursive call statement. They can write the recursive call statement as "insert (T, newitem)", instead of "insert (T^.Lchild, newitem)". The same situations are present in other inconsistent cases. Obviously the system should be able to cover the general cases. The preliminary RPT system was remedied after these inconsistent cases were found. Now the RPT system can detect the mistakes

correctly, the cases that misdetect bugs and incorrectly recognize bugs have disappeared from the sample data. Due to the variety of novice programs, it is certain that there will be unexpected cases which the system will not be able to detect. This is one limitation of the system, but it is not a problem of the essential detection approach.

The mistakes that were collected from the students' programs happened to fall into the catalogue of mistakes in the RPT diagnosis knowledge base. The distribution of the twenty mistakes in students' programs after running the revised RPT system is shown in Fig.27.

Mistakes in Diagnosis KB	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}	m_{12}	m_{13}
Mistakes in students' programs	5	1	1	0	0	3	3	0	5	0	0	0	2

Fig. 27. Bug Distribution In Students' Programs

From the bug distribution, we can see the mistakes m_1 "missing termination condition" and the mistake m_9 "not applying a new node to hold the new item" are the most common mistakes made by students since the percentage of these mistakes in the total mistakes reaches up to 25%. Mistakes m6 "miswrite the termination condition" and m7 "missing key word which defines the variable parameter in the formal parameter definition" are easily made by the novice programmers. Both of them take up 15 percent of mistakes in the sample data. Mistakes m_2 "misplace the operation of inserting a new node," m_3 "use a loop structure", and m_{13} "not apply a new node, but try to assign data to it" are made sometimes but not as often. Some mistakes, such as m_4, m_5, m_8, m_{10}, m_{11} and m_{12} were rarely found in the sample data. These mistakes are "lacking parameter in recursive call statement," "did not provide the smaller instance," "misorder the data in left subtree and right subtree," "apply too many new nodes for inserting one new

item," "try to find nodes in a binary search tree instead of finding a place to insert a node" and "create a new node after using it." The percentage may be different in the different group of students' programs. It seems that the mistakes in the IADI knowledge base can cover all of manifestations in the current collected programs.

6.3 Evaluation of Misconception Diagnosis

The RPT system has been run with approximately 40 students in the "Problem Solving and Programming Lab" class of the 1991 spring semester. During class time, the instructor first briefly introduced the RPT system. This introduction includes what is the basic purpose of RPT, how to work in the RPT instruction environment and the diagnosing environment, and also the system's user menu. After that, a homework which includes the insertion problem is assigned to students. Then the students practiced using RPT and tried to start their assignment in class. Students either wrote their program in the text window of the RPT diagnosis environment, or called their prewritten program directly. After the programs passed the Pascal compilation, they were submitted to the RPT diagnostic process.

6.3.1 Comparison in two groups

Most students can not get the correct solution by only running their program once. Usually students detect fewer bugs than actually exist, and they take more time to find them than is necessary for the experienced programmer and computer tutor. When they use the RPT system, students are inspired by the suggestions and hints received when they work on the checking lists in the interactive communication with the RPT system. After about one hour of class time including the instructor's brief introduction and working with RPT, most of students acquire the confidence to complete the assignment. During their homework time, the tutor helps students to complete the homework more quickly than students can by themselves. With the tutor students feel more confident

about trying to find out whether there are bugs, locating the bugs, and then correcting the corresponding code.

The Fig. 28 shows the comparison numbers in cases there students complete the

	Number of students completed the course	Number of students doing the assignment	Number of students finishing the assignment	Percentage of the students completed
Fall 1990	33	27	26	78.8%
Spring 1991	38	36	36	94.7%

Fig. 28. Comparison In Two Groups

assignment in the spring semester and fall semester. The number of students complete the course, rather than the number of registered students, is listed here because it is a more reasonable base to show how many students are studying in the class. These cases are in two groups; students in the fall semester did not use the RPT system, but the students in the spring semester used RPT. The assignment in the spring 1991 is an inorder traversal problem plus the deletion of any node from the built binary search tree, while the assignment in the fall 1990 was just a pure inorder traversal problem. Obviously the assignment in the spring was more difficult than the one in the fall. The result indicates that comparing with the situation in spring semester, more students were willing to and able to complete the similar but even more complicated assignment when they can use the RPT system to help them to understand the basic recursive concepts, diagnose the mistakes in their submitted programs, and clear their misconceptions. Although there may be other facts which can affect the students' attitude to do recursive assignment, we can see that the help from RPT is quite encouraging.

6.3.2 Diagnosis process shown from different versions

Some of the students submitted the intermediate results which become different versions to be analyzed. Tracing the different versions is very helpful in analyzing the system's performance. For example, a student first submitted his program, the first version, of which has the mistake type 9 "not apply new node to hold the new item." The procedure is displayed in Fig. 29. When he entered the diagnosis environment, he got the

```
Procedure insert (var T : table; k : integer);
begin
     if T < > nil then
      begin
          if  T^. item < k  then
                insert (T^. Rchild, k);
          if  T^. item > k then
                insert (T^. Lchild, k);
      end;
end;
```

Fig. 29. A Student's Program

error message after the mistake detection process and began a dialogue with the RPT system. He followed the checking list CL(c_6) which is related to the necessity of applying a new node and the way to use it, and found out the misconception. The questions in the sample are such as

Do you think it's necessary to have the new data saved in a node?

Do you need to apply a new node for holding this new data?

In order to apply a node to hold the new data, what function will you use?

During the interactive conversation the student was enlightened and realized he needed to include the application of the new node in his program. Another version in Fig.30, which is submitted by the student later, testifies to this conjecture.

In the program showed in Fig. 30 the student tried to correct the mistake type 9 in

```
Procedure insert (var T : table; k : integer);
begin
      new(T);
      T^.item := k;
      if (T < > nil) then
       begin
          if  T^. item < k  then
                insert (T^. Rchild, k);
          if  T^. item > k then
                insert (T^. Lchild, k);
       end;
end;
```

Fig. 30. Another Version Of The Program In Fig.29

the version shown in Fig. 29. But this version still was not correct. Another mistake appeared, type 2, "Misplace the operation of inserting a new node." That showed that even the student already knew he needed to apply a new node, but there was another step needs to be considered. This step was to find the correct position in which to insert the new node. Following the checking lists $CL(c_7)$ then, which contains the question like "when you need to apply a new node to hold the new item?" and other related questions, the student realized what was wrong and found the correct place to insert the node. Finally he corrected the mistake and got the correct solution and completed the assignment successfully.

6.3.3 Results shown in finals

The test on the students final examination supports the claim that the RPT system helps students avoid misconceptions in their programming.

In the final examination of the spring 1991 semester, the students were given two problems. One is to write a procedure to calculate the height of any binary search tree

(Helman and Veroff 86) (we call it a height problem here), and another is to find a bug on a given bug problem and correct it. In this final exam this procedure is required to work with other given procedures to solve the problem. The insertion procedure is one of the given procedures which is a basic step in building a binary search tree. One of the correct versions to calculate the height is displayed in Fig. 31. By comparing the procedure in Fig. 31 to the previous correct insertion procedure (one version is in Fig. 6)

```
Procedure height (var m : integer; T: table );
var h1, h2 : integer;
begin
     if T< > nil then
     begin
       m := m+1;
       h1 := m;
       h2 := m;
       height (h1, T^.Lchild);
       height (h2, T^.Rchild);
       if h1>h2 then m := h1;
                else m := h2;
     end;
end;
```

Fig. 31. Procedure To Calculate The Height
Of Binary Search Tree.

we find that some concepts implied in the two procedures are similar. These concepts are about the termination condition in a recursion process, the recursive relationship, and the definition of the variable parameter of a procedure. If we imagine the possible mistakes in the height problem, the related mistake types may be the following:

Mistake m_1 : Missing the termination condition;

Mistake m_3 : Use a loop structure;

Mistake m_4 : Lacking parameter in recursive call statement;

Mistake m_5 : Did not provide the smaller instance;

Mistake m_6 : Miswrite the termination condition;

Mistake m_7 : Missing key word which defines the variable parameter in the formal parameter definition.

Since there are many similar concepts in the two procedures, we can assume that the students tutored by the RPT system with the problem solving of insertion will get the benefits of this and hopefully avoid similar misconceptions and mistakes in the problem solving of height.

We manually analyzed the students' solutions after the final exam, and got the following results shown in Fig.32. In the final examination, only five students gave the

Number of students who took the final : 38;
Number of students who submitted the answer to this problem : 31;
Number of procedures having m1 : 0;
Number of procedures having m3 : 0;
Number of procedures having m4 : 1;
Number of procedures having m5 : 0;
Number of procedures having m6 : 0;
Number of procedures having m7 : 1.

Fig. 32. Finding From Students' Solutions

correct solutions. Most of students can not solve the problem completely at the examination time. One reason is it is a little bit hasty with the 3 hours to solve two problems. The observation shows that the main obstacle is something in the logic of the process to find the height of a binary search tree which is a harder problem than the insertion problem. From the Fig. 32 we can see the results from the submitted solutions. These results indicate that after students used the RPT system to help them get rid of some misconceptions, the most common mistake, type 1, missing termination condition, was

almost eliminated in this final exam. The easily made mistake type 6 is eliminated too, and the another easily made mistake type 7, missing key word which defines the variable parameter in the formal parameter definition were reduced to 1 out of 31 from 3 out of 27 (The one who had made the mistake type 7 in final was within the three who made it in classwork). The mistake type 4, lacking parameter in a recursive call statement, appeared in the final which had not appeared in the class work or homework before, but it was in the diagnosis knowledge base. If we compare this result with the manifestations shown in Fig. 27, we can see that these results are quite encouraging.

CHAPTER SEVEN

CONCLUSIONS

The previous chapters have discussed the Integration of the Abductive and Deductive Inference diagnosis model and its application in a Recursive Programming Tutor. The substantive issues and the state of the art have been presented. This chapter summarizes this research and the contributions of the IADI model to diagnosis problem solving, then proposes some further research issues.

7.1 Summary of IADI Diagnosis Model and Its Significance

The abilities to understand the mental activities of a student, and to detect and correct misconceptions in the student's understanding are the main features that distinguish the ITS from other systems such as expert systems and earlier CAI systems. This motivates many researchers to concentrate their efforts on the student modeling problem and diagnosis problem solving.

There are many different diagnosis methods provided in the existing student modeling problems of intelligent tutoring systems. Table 1 in Chapter 1 listed the major diagnosis techniques in different student models. Chapter 2 described and discussed these different techniques and further classified them into three types of diagnosis models. The enumerating model can work well when the problem size is small, or the combination and the permutation of facts, like the facts in subtraction, are easily obtained. But it does not work well in more complex problems such as the case involving mental states. The tracking model tries to track a student's mental stages at every step in order to establish a complete student's mental model. But the feasibility of grasping every

mental state of a human is questionable, and tightly following steps indicated by the system restricts some possible solutions. The classifying model captures the design plans and the significant issues to build a student model, therefore it is more realistic than the tracking model, and more effective than the enumerating model. But this model does not show how to deal with the complex relationships between basic conceptions and demonstrated mistakes.

The IADI diagnosis model is explored to diagnose students' mistakes and misconceptions in complex problems. It is different from the existing diagnosis models. The IADI diagnosis model combines different types of inferences, the abductive inference and deductive inference. The basic process is divided into three steps, mistake detection, misconception hypothesizing and misconception verification. These three steps are accomplished by structure analysis, abductive inference and deductive inference respectively. The abductive inference is used to catch the plausible features of hypothesizing process while the deductive inference is used to presents the nature of verification process. Thus it becomes an inferencing model, and the different inferences can be supplement with each other. This model provides detection at two levels. It detects mistakes to show what incorrect actions are, and also detects misconceptions to find why the wrong actions occur. Comparing with the detection at only one level, this model can bring potentialities of tutoring into full play. This diagnosis model focuses on the main conceptions in a problem and catches the key steps in problem solving to avoid excessive details. It also attempts to connect the relationships between mistakes and misconceptions in rules for a particular problem, and associate design plans to programming conceptions. Thus it provides rationality to elaborate conception during tutoring. This model simulates human diagnostician's reasoning process in cognitive activities in order to obtain a better solution. This model works in a hypertext concept based tutoring environment. It allows students to master the concept of recursion and the art of recursive programming with relative ease. The general model is outlined in Chapter 3 and the processes are de-

scribed in Chapter 4, with an application in a recursive programming tutor. The working environment is illustrated in Chapter 5. Finally the empirical tests and evaluations are presented in Chapter 6.

The following subsections address the significance of the model by summarizing the features that make the IADI diagnostic model distinguish itself from other models and also summarizing the contributions of the RPT system.

7.1.1 Two-level detection

The IADI diagnosis model provides a two-level detection. The first is mistake detection at the action level. The second is misconception detection at the conceptional level. Diagnosis systems typically only detect mistakes. If a diagnostic process only indicates mistakes, it will not be able to help students to understand the reason, and it may not even be able to convince a student that mistakes were actually made. Obviously, it is not sufficient if the diagnosis process in a tutoring system detects mistakes only on the surface. It can not reach the fullest potential of an intelligent tutoring system. In an intelligent tutoring system, the diagnosis process must carry a step further to find the misconception so that it can provide the basis for a tutor to give possible causal explanations for incorrect actions, and for students to get a deeper comprehension, and therefore can eliminate a whole set of mistakes. Some tutoring systems provide only conceptional explanations to the detected mistakes. These explanations come from experience. The formal relationships between mistakes and misconceptions are not summarized. The IADI diagnosis model summarizes a set of production rules in the IADI diagnosis knowledge base that represent the cause-effect relationships between mistakes and misconceptions which are considered as the most important principle to follow in a diagnosis system (Torasso and Console 89) (White and Frederiksen 90). And this helps the system find misconceptions from mistakes.

7.1.2 Combination of abduction and deduction

Most expert systems, including tutoring systems, for diagnostic problem solving only use deductive inference. Although the reasoning process based on this inference mechanism is essential, it is not adequate by itself in solving many real world diagnostic problems. There are some problems associated with deductive inference systems. For example, many of them have an extensive list of conditions so that it is difficult to fire a rule. And the deductive inference systems lack the representation of cause–effect knowledge in their traditional production rules.

On the other hand, some expert systems for diagnosis use abductive inference only. In such systems it seems that something is lacking when explanations to incorrect actions are needed. It lacks a good organization of the material to be explained. This is a very important point in a tutoring system. By taking into consideration of the complex relationships between mistakes (in other problems they may be called symptoms, bugs, or manifestations) and misconceptions, and exploiting the features of abductive inference and deductive inference, we have combined abduction and deduction into the diagnosis process. This integrated diagnosis process has a hypothesis process and a verification process which are very close to human diagnosticians' process. And the rule form in the IADI knowledge base supports both abduction and deduction. The explanations to each misconception are well organized by the design plans which can be expected to have an effective tutoring result.

7.1.3 Concentrating the diagnosis on key steps of problem solving

The IADI diagnosis model is different from the existing tracking model and enumerating model. The IADI diagnosis model works by taking into consideration mental states, but it does not enumerate all the mistakes derived from various mental states. By studying the differences between experts and novices in different domains, cognitive psychologists have discovered that students usually go through different conceptual stages,

especially in the critical ones of learning a subject [Brown and Burton 87]. Thus teaching is not the pouring of knowledge into an empty vessel, but more a process of reconceptulization of the critical issues. In the instruction stage of the RPT system the tutor tries to grasp critical issues to induce general rules to a student who is learning the recursive programming technique, because clarifying and solving them are the crucial steps to solve the whole problem [Shen, Zhang and Zhao 90]. In the diagnosis process this model selects major mistakes made on key steps in problem solving and fatal misconceptions formed when learning a subject to form the diagnosis knowledge base, which contrasts the tracking model that tries to arrange all mental activities and trace them. And the items in a checking list for verification of a misconception are chosen for presenting the critical issues around the main steps of a design plan to achieve a goal. This disposition is much more efficient and reasonable.

In the internal process of the diagnosis model, the abductive process infers a selected candidate hypothesis set C_s for a student. Every hypothesis in C_s is a probable misconception that the student may have. The deductive process then focuses on the set C_s while the student works with the corresponding checking lists. Thus the efforts of the tutor's instruction is focused on the student's specific misconceptions rather than using some generic instructions. It avoids requiring the tutor to go through the same detail when a different student is being tutored. These are the typical characteristics of an ITS that has a instruction tailored to the individual needs of students.

7.1.4 Incorporating the process of tutoring into diagnosis

The system begins to tutor while it is doing further diagnosis after the initial hypotheses are obtained. This is different from those systems in which the tutor starts to give instructions or explanations only after the diagnosis result is gained [Johnson and Soloway 84] [Woolf and McDonald 84]. In such processes, the scope of diagnosing programming errors is limited to the information that is extractable from the buggy pro-

grams themselves. In the deductive reasoning process of the IADI diagnosis model, the system establishes checking lists to further detect a student's mistakes which are undemonstrative in the original buggy program, and therefore verifies a student's misconceptions and produces a more precise student model. During this process, the system also gives instructions if the student answers questions inappropriately. The verification process and the tutoring process proceed in an interactive environment. During the interaction with a student, the system follows both the cause path and the design plan. The corresponding instructions will help the student to understand the reason why he made a mistake, and to understand the design plan how they can reach a correct concept. Thus the tutoring process is incorporated into the diagnosis process in IADI. In this way, students can receive instructions on particular misconceptions whenever they are told there is something incorrect. These impromptu explanations and instructions can be readily absorbed by students.

7.1.5 Nondeterministic diagnosis

The IADI diagnosis model produces a list of ranked misconceptions as the final result. It is not a deterministic result. In cases where the intersection of man(c_i) and the detected mistakes is a subset of the intersection of man(c_j) and the detected mistakes, c_i may still be a eligible misconception. Thus the method includes both c_i and c_j in C_s as appropriate.

In cognitive activities, excessively or rigorously deterministic mathematical models may not be appropriate. For example, if we have the rules

$$c_1 \longrightarrow m_1,\ c_2 \longrightarrow m_2,\ c_1 \longrightarrow m_2,$$

and we are given $M_p = \{m_1, m_2\}$, then c_1 might be a good explanation for M_p and it also is the minimal cover for M_p in terms of the Set Covering model [Reggia 85]. But we can not rule out c_2 as the possible diagnostic result, because it is possible that the student made m_1 through c_1, and made m_2 through c_2 under a different situation. In the

IADI diagnosis model, both c_1 and c_2 are included in the final diagnostic list and are ordered in appropriate way. Thus, this approach provides more reasonable result.

7.1.6 Hypertext tutoring environment

When we design an ITS, we should consider that the ITS works with a student who does not understand the subject domain very well. There is no doubt that a well-designed environment can enhance the capabilities of an ITS in many ways. The new techniques and research ideas from computer applications and cognitive science are opening up many opportunities for creating a good instructional environment. The RPT system is built in a hypertext based environment. Although the hypermedia technique has been applied to teach English literature, cell biology [Yankelovich 88] and engineering [Bourne 89], it has not been used in programming tutoring. The RPT system uses the hypertext concept accompanying code execution to create a tutoring environment for recursive programming. The significant cases and critical issues in the learning of recursive programming are presented in a multi-dimension graphic environment. This environment can also make the process originally hidden from the succinctly written recursive program visible. It allows students to see through the recursive process at different levels of details and even get the execution result of the procedure at each level. The environment and its other features are described in section 5.7.

7.1.7 Evaluation of RPT

The empirical evaluation of RPT shows an encouraging result. The evaluation data are based on the comparisons in the "Problem Solving and Programming Lab" classes within two semesters. The system has been run with approximately 40 students in the 1991 spring semester. The comparison between the mistakes detected by RPT and the mistakes detected by people shows that mistakes collected in the IADI knowledge base can cover all the mistakes demonstrated from these students' programs. When students

use RPT to solve the insertion problem, the progress steps can be seen from the different versions submitted from students. That is, the misconceptions were clarified by the instructions from the RPT diagnosis and tutoring process. The data of the accomplishment of assignments assigned in the two semesters indicate that students were increasing competent in recursive problem solving since more students (in percentage) in the spring 1991 semester could complete the assignment than students in the fall 1990 semester when not using RPT, even though the assignment was more complicated in the spring 1991 semester. Furthermore, most common mistakes were significantly reduced after the students had used the system. This conclusion became more evident when we compared their class assignments and the final exams where harder problems but with similar concepts were given. On the other hand, other factors that were not indicated in the analysis may also have played some role, even though they were not obvious. The sample size may also be considered quite small.

7.2 Future Research Work

As a first step in proposing and building a diagnosis model, the description about the IADI model is concentrated on the basic principles and the main steps. It definitely has some limitations. From the prototype environment to a real practical environment, it still has a long way to go. In order to improve the performance of the diagnosis model, there is a need from various perspectives. Several research issues are addressed in this section.

7.2.1 The degree of diagnostic details

The deductive reasoning in the IADI model is a process that verifies the hypotheses from the selected candidates. The checking lists are used to list the possible manifestations for every selected hypothesis. From the student's choices the tutor can decide what misconceptions the student may have. The more items a checking list has, the more

precise the student model can be. There is a tacit assumption that tutoring based on fine-grained student models will be more effective than tutoring based on coarse-grained models. No one has attempted to check this assumption. The thing one needs to know is whether the fine-grained modeling is worth the effort, and whether students are willing to answer so many questions in a checking list. Although the items are well organized by the design plans, what degree of detail is suitable for a particular system still needs to be studied.

On the other hand, an acceptable view of diagnosis problem solving behavior is that problem solving is carried out at multiple levels of abstraction. For example, a general misconception can be refined to many specific misconceptions under the general one. Thus, the knowledge structure for cause-effect relationships among the general misconceptions, specific misconceptions, and mistakes may become more complicated if we consider that there is one more dimension added to the cause-effect relationship we described in this model. The question is whether the multiple level, giving fine model description is better, or if the flat structure, giving coarse model description is better. The former may be able to give a more precise diagnostic result. The latter may be more efficient at giving a diagnostic result since fewer relationships need to be taken into consideration. If the former one is chosen, the knowledge structure must be rearranged in the diagnosis knowledge base.

7.2.2 The diversity of mistakes

The IADI diagnosis model is first implemented in a recursive programming tutor. Usually the recursive program is short and its structure is not as varied as in other kinds of programming problems, although the concept about recursion is not easily to be understood. When the complexity of a problem is increased or the size of a solution becomes larger, the mistake detection will become increasingly difficult, because the types of structures in the solutions will be dramatically increased. Especially in the nov-

ice programmers' solutions, bizarre mistakes under various misconceptions will emerge one after another and the unexpected structure types will make detection difficult. Thus, a structure based mistake detection may not be able to deal with it completely. Other methods, such as combining heuristic control strategy, need to be put forward to complement the structure based mistake detection.

7.2.3 Other Applications

Besides the topics mentioned in the last two subsections that need to be studied, the implemented RPT prototype must be further tested in a wider range, such as getting more students to work with it and allowing it to tackle more diagnosis problems. Then we can acquire more empirical data and refine the diagnosis knowledge base and tutoring system, and even replenish the diagnosis model.

There are also many other domains that the IADI diagnosis model can be applied to, such as other programming languages, mathematical subtraction, calculus, medicine, or digital circuit. For each different area, there is a set of tasks to be put into effect, such as bug collection, cause investigation, tutoring subjects induction, and so on. If they are done, they would help us to test more thoroughly the effectiveness of the inferencing-based approach to diagnosing novice solutions. Upon successfully solving new issues in other domains, this diagnosis model will be greatly improved and the generality of this model may be proved.

REFERENCES

Adam, Anne and Laurent, Jean-Pierre. 1980. LAURA: a system to debug student programs. Artificial Intelligence 15: 75-122.

Aho, Alfred V., Hopcroft, John E. and Ullman, Jeffrey D. 1974. The design and Analysis of Computer Algorithms. Menlo Park: Addison-Wesley Publishing Company.

Akscyn, Robert M., McCracken, Donald L. and Yoder, Elise A. 1988. KMS: A Distributed Hypermedia System For Managing Knowledge In Organizations. Communications of ACM July 1988: 820-835.

Anderson, John R. 1983. Retrieval of Information Form Long-term Memory. Science, 220, 1983: 25-30.

Anderson, John R. 1988. The expert Module. In Foundations of Intelligent Tutoring Systems, ed. Polson, Martha C. and Richardson, Jeffrey J., 21-54. Hillsdale: Lawrence Erlbaum Associates Publishers.

Anderson, John R., Boyle, C.Franklin, Corbett, Albert and Lewis, Matthew. 1990. Cognitive Modelling and Intelligent Tutoring. Artificial Intelligence 42: 51-81.

Barr, Avron and Beard, Marian. 1976. An Instructional Interpreter For BASIC. In Computer Science and Education, ed. Colman, R.; and Lorton, R. Jr., (or ACM SIGCSE Bulletin, Vol. 8, no.1) 325-334. Amsterdam: North Holland.

Barr, Avron and Feigenbaum, Edward A. 1982. Application-Oriented AI Research: Education. Chap. in The Handbook of Artificial Intelligence (Vol. II), 225-294. Los Altos: William Kaufmann Inc.

Bhuiyan, Shawkat H., Greer, Jim E. and McCalla, Gordon I. 1991. Caracterizing, Rationalizing, and Reifying Mental Models of Recursion. In Proceedings of The International Conference on the Learning Sciences, in The Institute for the

Learning Sciences, Northwestern University, Evanston, Illinois, USA, August 1991, 120-125.

Bonar, Jeffrey G. and Cunningham, Robert. 1988. Intelligent Tutoring with Intermediate Representations. In Proceedings of International Conference on Intelligent Tutoring Systems, in Montreal, Canada, July 1-3, 1988, 25-32.

Bourne, John R., Cantwell, Jeff, Brodersen, Authur J., Antao, Brian, Koussis, Antonis and Huang, Yen-Chun, 1989. Intelligent Hypertutoring in Engineering. Academic Computing Sept., 1989: 18-38.

Brown, John Seely and Burton, Richard R. 1975. Multiple Representation of Knowledge for Tutorial Reasoning. In Representation and Understanding: Studies in Cognitive Science, ed. Bobrow, Daniel and Collins, Allan, 311-349. New York: Academic Press.

Brown, John Seely and Burton, Richard R., 1978. Diagnostic Models For Procedural Bugs in Basic Mathematical Skills. Cognitive Science 2: 155-192.

Brown, John Seely, Burton, Richard R. and de Kleer, Johan 1982. Pedagogical natural Language and Knowledge Engineering Techniques in SOPHIE I, II and III. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 227-282. London: Academic Press.

Brown, John Seely and Burton, Richard R. 1987. Reactive Learning Environments for Teaching Electronic Troubleshooting. In Advances in Man-Machine Systems Research, ed. Rouse, W.B., 65-98. Greenwich: JAI press,

Bundy, Alan, 1990. Catalogue of Artificial Intelligence Techniques, Third revised edition. New York: Springer-Verlag.

Burton, Richard R., 1982. Diagnosing Bugs in a Simple Procedural Skill. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 157-184. London: Academic Press.

Burton, Richard R. and Brown, John Seely. 1982. "An Investigation of Computer Coaching for Informal learning Activities." In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 79-98. London: Academic Press.

Burton, Richard R. 1988. "The Environment Module of ITS." In Foundations of Intelligent Tutoring System, ed. Polson, Martha C. and Richardson, Jeffrey J., 109-142. Hillsdale: Lawrence Erlbaum Associates Publishers.

Carbonell, Jaime. 1970. AI in CAI : An Artificial Intelligent Approach to Computer Aided Instruction. IEEE Transaction on Man-Machine Systems, 11: 190-202.

Carlson, Patricia Ann. 1988. Hypertext: A Way of Incorporating User Feedback into Online Documentation. In Text, Contexts, and Hypertext, ed. Barrett, Edward, 93-110. Cambridge: The MIT Press.

Chan, Tak-wai and Baskin, Arthur B. 1990. Learning Companion Systems. In Intelligent Tutoring Systems at the Crossroads of Artificial Intelligence and Education, ed. Claude Frasson and Gilles Gauthier, 1-33. Norwood: Ablex Publishing Corporation.

Charniak, Eugene and McDermott, Drew. 1985. Abduction, uncertainty and expert systems. Chapter. In Introduction to Artificial Intelligence, 453-485. Reading: Addison-Wesley.

Clancey, William J. 1982. Tutoring Rules for Guiding a Case method Dialogue. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 201-225. London: Academic Press.

Clancey, William J. 1987. Intelligent Tutoring Systems: A Tutorial Survey. In Current Issue in Expert Systems, ed. Lamsweerde, A.V. and Dufour, P., 39-78. London: Academic Press.

Collins, Allan and Stevens, Albert. 1982. Goals and Strategies of Inquiry Teacher. In Advances in Instructional Psychology, vol. 2, ed. Glaser, R., 65-119. Hillsdale: Lawrence Erlbaum Associates.

Conklin, Jeffery. 1987. Hypertext: An Introduction and Survey. IEEE Computer, Sep. 1987: 17-41.

Corbett, Albert T., Anderson, John R. and Patterson, Eric G. 1990. Student Modeling Tutoring Flexibility in the LISP Intelligent Tutoring System. In Intelligent Tutoring Systems at the Crossroads of Artificial Intelligence and Education, ed. Claude Frasson and Gilles Gauthier, 83-106. Norwood: Ablex Publishing Corporation.

Date, Nell and Weems, Chip. 1987. Recursion. Chapter in Introductio to PASCAL and Structured Design, 722-753. Lexington, Massachusetts Toroto: D. C. Heath Company.

de Kleer, J. and Brown, John Seely. 1983. Assumptions and ambiguities in Mechanistic Mental Methods. In Mental Models, ed. Gentner, Dedre. and Stevens, Albert L., 155-190. Hillsdale: Lawrence Erlbaum Associates.

Fann, K.T. 1970. Peirce's Theory of Abduction. The Hague: Martinus Nijhoff.

Garrett, Nancy, Smith, Karen E. and Myrowitz, Norman. 1986. Intermedia : Issues, Strategies, and Tactics in the Design of a Hypermedia Document System. In Proceedings of the Conference on Computer-Supported Cooperative Work, Austin, Texas, Dec., 1986, 163-174.

Geffner, H. 1989. Default Reasoning, Minimality and Coherence. In proceedings of the first international conference on Principles of Knowledge Representation and Reasoning, Toronto, Ontario, Canada, 1989, 137-148.

Genesereth, Michael R. 1982. The Role of Plans In Intelligent Teaching Systems. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 137-155. London: Academic Press.

Gentner, Dedre and Stevens, Albert L. Eds. 1983. Mental Models. Hillsdale: Lawrence Erlbaum Associates.

Goldstein, Ira P. 1982. The Genetic Graph: A Representation for the Evolution of Procedural Knowledge. In Intelligent Tutoring Systems. ed. Sleeman, D.H. and Brown, J.S., 51-77. London: Academic Press.

Halasz, Frank G. 1988. Reflections on Notecards: Seven Issues For the Next Generation Of Hypermedia Systems. Communications of ACM, July 1988: 836-852.

Halff, Henry M. 1988. Curriculum and Instruction in Automated Tutors. In Foundations of Intelligent Tutoring System, ed. Polson, Martha C. and Richardson Jeffrey J., 79-108. Hillsdale: Lawrence Erlbaum Associates.

Han, Fei Tzu. 1964. "The Difficulties of Persuasion." In Basic Writings, translated by Burton Watson, 73-79. New York and London: Columbia University Press.

Helman, Paul and Veroff, Robert. 1986. Intermediate Problem Solving and Data Structures. Walls and Mirrors. Menlo Park: The Benjamin/Cummings publishing Company, INC.

Hollan, James, Hutchins, Edwin and Weitzman, Louis. 1984. STEAMER : An Interactive Inspectable Simulation-based Training System. Artificial Intelligence Magazine. Vol.5, no.2, 1984: 15-27.

Horowitx, Ellis and Sahni, Sartaj 1978. Fundamentals of Computer Algorithms. Rockville: Computer Science Press.

Johnson, Lewis; and Soloway, Elliot. 1984. Intention-based Diagnosis of Programming Errors. In Proceedings of the National Conference on Artificial Intelligence. Austin, Texas, 162-168.

Johnson, Lewis. 1986. Intention-based Diagnosis of Novice Programming Errors. Research Notes in Artificial Intelligence, 6, Los Altos: Morgan Kaufmann Publishers, Inc.

Kruse, Robert L. 1987. Generating Permutations. Section in Data Structures and Program Design, 278–282. Englewood Cliffs: Prentice-Hall, Inc.

Langley, Pat and Ohlsson, Stellan. 1984. Automated Cognitive Modelling. In Proceedings of American Association of AI, 193–197. Los Altos: Morgan Kaufmann Publisher, Inc.

Larkin, Jill H. and Simon, H.erbert A. 1987. Why a Diagram is (Sometimes) Worth Ten Thousand Words. Cognitive Science, 11, 1987: 65–100.

Laubsch, J. H. 1975. Some Thoughts About Representing Knowledge In Instructional Systems. In Proceedings of the Fourth International Joint Conference on AI, Tsibili, USSR, 122–125.

Leestma, Sanford and Nyhoff, Larry. 1984. PASCAL Programming and Problem Solving. New York: Macmillan Publishing Company, and London: Collier Macmillan Publishers.

Littman, David and Soloway, Elliot. 1988. Evaluating ITSs: The Cognitive Science Perspective. In Foundations of Intelligent Tutoring Systems, ed. Polson, Martha C. and Richardson, Jeffrey J., 209–242. Hillsdale: Lawrence Erlbaum Associates.

Looi, Chee-Kit. 1988. APROPOS2: A Program Analyser For A PROLOG Intelligent Teaching System. In Proceedings of Intelligent Tutoring Systems, Montreal, June 1–3, 1988, ACM, SIGART, SIGCUE, 379–386.

Manber, udi, 1988. Using Induction to Design Algorithms. Communications of the ACM, November 1988, Vol. 31, no. 11: 1300–1313.

Miller, Mark L. 1982. A Structured Planning and Debugging Environment for Elementary Programming. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 119–135. London: Academic Press.

Pearl, Judea, 1988. Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference. San Mateo: Morgan Kaufmann Publisher, Inc.

Peng, Yun and Reggia, James A. 1990. Abductive Inference Modls for Diagnostic Problem-Solving. Springer Series of Symboloc Computation - Artificial Intelligence, ed. Loveland, D.W. New York: Springer-Verlag.

Polson, Martha C. and Richardson, Jeffrey J. 1988. Foundations of Intelligent Tutoring Systems. Hillsdale: Lawrence Erlbaum Associates.

Pople, Harry. 1973. On the Mechanization of Abductive Logic. In Proceedings of the International Joint Conference on Artificial Intelligence Conference, 1973, 147-152.

Reggia, James A., Nau, Dana S., Peng, Yun and Perricone, Barry 1985a. A Theoretical Foundation For Abductive Expert Systems. In Approximate Reasoning in Expert Systems, ed. Gupta, Madan M., Kandel, Abraham, Bandler, Wyllis and Kiszka, Jerzy B., 459-472. North Holland: Elsevier Science Publishers B.V.

Reggia, James A., Perricone, Barry, Nau, Dana S. and Peng, Yun. 1985b. Answer Justification in Diagnostic Expert Systems, part I : Abductive Inference and Its Justification. IEEE Transactions on Biomedical Engineering, vol BME-32, No.4, April 1985: 263-267.

Reggia, James A., Perricone, Barry, Nau, Dana S. and Peng, Yun. 1985c. Answer Justification in Diagnostic Expert Systems, part II : Supporting Plausible Justifications. IEEE Transactions on Biomedical Engineering, vol BME-32, No.4, April 1985: 268-272.

Reiter, Ray. 1987. A Theory of Diagnosis From First Principles. Artificial Intelligence, vol. 32 (1), 1987, 57-95.

Reiser, Brian J., Anderson, John R. and Earrll, Robert G. 1985. Dynamic Student modeling in an Intelligent Tutor for Lisp Programming. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence Conference, los Angeles, 8-14.

Sandberg, J.A.C. 1987. The Third International Conference on Artificial Intelligence and Education. AICOM. 0, 51-53.

Self, John A., 1988, Bypassing The Intractable Problem of Student Modelling. In Proceedings of International Conference on Intelligent Tutoring Systems, July 1-3, 1988 in Montreal, Canada, 18-24.

Shen, Stewart N.T. and Zhang, Jingying. 1989a. A Knowledge-Oriented Hypermedia System. In BIGRE 63-64, May 1989, Workshop on Object-Oriented Document Manipulation, Rennes, France, 29-31 May 1989, 307-316.

Shen, Stewart N.T. and Zhang, Jingying. 1989b. Recursive Programming Tutoring system. In Proceedings of The Sixth IASTED International Symposium: Expert Systems Theory and Applications, Los Angeles, CA, 14-16 Dec. 1989, 60-63.

Shen, Stewart N.T., Zhang, Jingying and Zhao, Shensheng. 1990. A Tutoring System for Critical Thinking. In Proceedings of The Eighth Annual Conf. on Technology and Innovations in Training and Education Conference, at Colorado Springs, Colorado, 12-16 March, 1990, 448-458.

Shen, Stewart N.T. and Zhang, Jingying. 1991. Integration of Abductive and Deductive Inference Diagnosis Methodology In Intelligent Tutoring. In Proceedings of The International Conf. on Artificial Intelligence and Simulation, held in New Orlean, Louisiana 1-5 April 1991, edited by R. J. Uttamsingh and A. M. Wildberger, 127-132. San Diego: A Publication of The Society for Computer Simulation.

Shortliffe, E. H. 1976. Computer-based Medical Consultations: MYCIN. New York: American Elsevier.

Siuru, William D. 1989. Educational Applications - Challenger: A Domain Independent Intelligent Tutoring System. IEEE Expert summer 1989: 77-79.

Sleeman, D and Brown, J.S. 1982, Introduction: Intelligent Tutoring Systems. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S., 1-11. London: Academic Press.

Soloway, Elliot M., Rubin, Eric, Woolf, Beverly P., Bonar, Jeffrey and Johnson, W.Lewis. 1983. MENO-II: an AI-based Programming Tutor. Journal of Computer-based Instruction, Vol.10, no. 1: 20-34.

Stevens, Albert and Collins, Allan. 1977. The Goal Structure of a Socratic Tutor. In Proceedings of the National ACM Conference, Seattle, Washington, 256-263. New York: Association for Computing Machinery.

Stevens, Albert, Collins, Allan and Goldin, Sarah E. 1982. Misconceptions in Student Understanding. In Intelligent Tutoring Systems, ed. Sleeman, D.H. and Brown, J.S. 13-24. London: Academic Press.

Torasso, Pietro and Console, Luca, 1989. Second Generation Expert Systems. Section in Diagnostic Problem Solving, Combining Heuristic, Approximate and Causal Reasoning, 13-24. New York: Van Nostrand Reinhold.

Trigoboff, M. and Kulikowski, C. 1977. IRIS: A System for the Propagation of Inferences in a Semantic Net. In Proceedings of the International Joint Conference on Artificial Intelligence Conference, 1977, 274-280.

Vanlehn, Kurt. 1988. Student Modeling. In Foundations of Intelligent Tutoring System, ed. Polson, Martha C. and Richardson, Jeffrey J., 55-78. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Wallach, Bret. 1987. Development Strategies for ICAI on Small Computers. In Artificial Intelligence and Instruction, ed. Kearsley, Greg, 305-322. Menlo Park, California: Addison-Wesley Publishing Company.

Wenger, Etienne. 1987. Artificial Intelligence and Tutoring Systems. Computational and Cognitive Approaches To The Communication Of Knowledge. Los Altos: Morgan Kaufmann Publishers, Inc.

White, Barbara Y. and Frederiksen, John R. 1990. Causal Model Progressions as a Foundation For Intelligent Learning Environments. Artificial Intelligence 42: 99-157.

Wiedenbeck Susan. 1989. Learning Iteration and Recursion From Examples. International Journal Man-Machine Studies 30, 1-22.

Wirth, Niklaus, 1976. Recursive Algorithms. Chapter in Algorithms + Data Structures = Programs. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Woolf, Beverly P. and McDonald, David D. 1984. Building a Computer Tutor: Design Issues. IEEE Computer, vol.17, no.9: 61-73.

Woolf, Beverly P. 1988. Intelligent Tutoring System: A Survey. In Exploring Artificial Intelligence, ed. Shrobe, H.E. and the American Association for AI, 1-45. San Mateo, California: Morgan Kaufmann Publishers, Inc.

Yankelovich, Nicole, Haan, Bernard J. and Drucker, Steven M. 1988. Connections in Context: the Intermedia System. In Proceedings of the 21 Annual Hawaii International Conference on System Sciences, vol II 1988, 715-724.

Younggren, Geri. 1988. Using an Object-Oriented Programming Language to create Audience-Driven Hypermedia Environment. In Text, Contexts, and Hypertext, ed. Barrett, Edward, 77-92. Cambridge, Massachusetts: The MIT Press.