AN INTELLIGENT TUTORING SYSTEM FOR INTRODUCTORY C LANGUAGE COURSE

J. S. SONG1, S. H. HAHN2*, K. Y. TAK1 and J. H. KIM1
1Department of Computer Science, KAIST, 373-1, KuSong-Dong, Yusong-Gu, Taejon, 305-701, Korea
[Fax: +82-42-869-3510; e-mail: shahn@gori.kaist.ac.kr, jkim@cs.kaist.ac.kr]
2R&D Center, Dacom Co., 34 Kajung-dong, Yusong-Gu, Taejon, 305-350, Korea
[Fax: +82-42-861-8149; e-mail: kyta@halla.dacom.co.kr]

(Received 8 November 1996; accepted 15 January 1997)

Abstract—In this paper, we describe a system called C-Tutor, an intelligent tutoring system (ITS) for novice C programmers. A program analyzer is the most important part of the ITS for programming. Our program analyzer is a compound of a reverse engineering system and a didactic system. Since a novice program usually contains many bugs, information about the intentions of the programmer is inevitable to recognize a buggy program. In our approach, the intentions of a programmer are automatically extracted as a problem description from a sample program by a reverse engineering system called GOES (GOal Extraction System). Based on the problem description, students’ programs are recognized by a didactic system called ExBug (Execution-guided deBugger). As a learning environment, Curriculum Network constructs the knowledge base as genetic graphs to teach programming. C-Tutor is a complete ITS which provides both a program analyzer and a learning environment. Tested with real students’ programs, program analyzer gives acceptable recognition results. Program analyzer and learning environment are closely related so that students can learn C language during programming. New problems can be easily set because GOES automatically generates problem descriptions for program analyzers. This makes C-Tutor a more practical tutoring system for a real C language course. © 1997 Elsevier Science Ltd

INTRODUCTION

An Intelligent Tutoring System (ITS) is a computer-based system intended to provide effective, appropriate, and flexible instruction through the application of artificial intelligence techniques and knowledge representation [1]. Intelligent tutoring systems have been built for many domains. A particularly common domain for ITS research has been the construction of tutors to teach specific programming languages [2], because system designers can act as domain experts as well. There are three modules in an ITS: an expert module, a student model, and a pedagogical module. In addition, an interface module is needed to communicate with students. An expert module has knowledge of domain experts and a student model represents the student’s current state of knowledge, learning capabilities, background, etc. A pedagogical module has knowledge on how to teach. In the programming tutoring field, an expert module should have the ability to understand programs as well as the knowledge on programming language and programming skills. The student model of a programming tutor should represent the misconceptions of students that are found by analyzing their programs, along with their current knowledge of the programming language. According to the status of the student model, the pedagogical module decides whether to teach concepts and skills of programming or to give exercises.

In this paper, we introduce an intelligent tutoring system for novice C programmers, C-Tutor. C-Tutor consists of two subsystems: a program analyzer and a learning environment. The program analyzer of C-Tutor is composed of a reverse engineering system and a didactic system. ExBug (Execution-guided deBugger) [3] is a didactic system which analayses the students’ programs, and GOES (GOal Extraction System) [4] is a reverse engineering system which automatically generates a problem description from a sample program. This sample program is provided by a teacher, and considered as a correct one. It also transforms the input programs into canonical forms in order to absorb syntactic variations. Curriculum Network [5] is the learning environment of C-Tutor. It constructs the knowledge of goals and plans as genetic graphs [6], and teaches concept and skills of programming. To select the topic to teach, it refers to the bugs found by ExBug as well as the current student model.

* To whom all correspondence should be addressed.
RELATED WORKS: ITSS FOR PROGRAMMING

Programming language tutors have typically been problem-solving monitors that critique student solutions to problems [2]. Most of the systems only cover a limited part of the programming course. The programming language used ranges from general purpose languages such as Pascal, Lisp, and Prolog to didactic languages such as Logo and the "Karel the Robot" language. There are systems that are implemented as complete ITSSs, or as program analyzers. A complete ITSS has both the program analyzer and the learning environment in one system. On the contrary, a program analyzer is an environment which can analyze student programs and give feedback on the errors found, but which is not integrated within a real learning environment. However, most of the researchers working in this area intend to integrate their systems as a part of complete ITSSs.

In this paper, we refer to one complete ITS and three program analyzers as related works. Lisp Tutor [7] is an ITS for novice Lisp programmers. It monitors a student's progress, and discovers and instructs about errors. As long as the student follows a path leading to a correct solution, the tutor stays in the background. When it finds that the student is having difficulty coding a problem, Lisp Tutor takes him from the coding mode to planning mode to work through the problem with the student. To analyze students' programs, Lisp Tutor has ideal models along with buggy rules. The former is simulations of the programming knowledge which ideal students use in solving problems, while the latter represents misconceptions which novice programmers often develop during learning. Lisp Tutor follows the student as he types in his code, and tries to figure out what ideal or buggy production rule would have led to that input code. In Lisp Tutor, they designed the curriculum with 18 lessons, and each lesson takes 1 to 4 hours to complete. Each topic involves a small instructional booklet and many problems practicing the skills taught in that lesson. The strength of Lisp Tutor is that it can give immediate feedback. As soon as the student makes a mistake, the tutor responds with an appropriate diagnostic message. In addition, the student can learn programming skills during programming, since most of the time in any lesson is spent going through problems. However, on-line monitoring of the student is possible because Lisp Tutor assigns small problems whose requirements are clear and unambiguous. When the student has a number of active goals, Lisp Tutor cannot analyze the student's program. Another limitation of Lisp Tutor is that the student is forced to solve the problem in a manner that the system can understand. When the student codes in a different way which Lisp Tutor cannot relate to any of the active goals, it immediately rejects the student's action.

Talus [2] is a program analyzer for Lisp language. Talus uses four steps to analyze input programs: program simplification, algorithm recognition, bug detection, and bug recognition. Program simplification transforms the input program into a Lisp core dialect. In algorithm recognition, Talus selects the algorithm matching the input program. Using symbolic evaluation, Talus detects bugs to determine the differences between input and reference functions. Once it identifies bugs in the input program, Talus attempts to correct them using techniques based on theorem proving and heuristic methods. The strength of Talus is that it detects localized mismatches automatically. Talus analyzes a program based on that program's logical structure, and identifies differences between code at the lowest level of the logic tree. Because the differences are extremely localized, this technique would explicitly identify differences between reference function and input program. In addition, Talus can deal with the functions containing functions, which is likely to be encountered in practice. However, Talus can only recognize the input program if its subfunction is allocated exactly as Talus does. Since subfunctions can be allocated in various ways, this can be a serious limitation when dealing with large programs consisting of many subprograms. Another limitation of Talus is that it assumes the task is already known. Furthermore, Talus provides only limited data structure definitions, and has problems with large programs and imperative programming style.

PROUST [8] is an intention-based diagnosis system for novice Pascal programmers. Since it has no learning environment, PROUST also is a program analyzer. PROUST starts to analyze a student's program from the problem description. A problem description is a list of requirements to be solved and typically written by a teacher. PROUST needs the problem description as top-down knowledge. To understand and debug students' programs, the knowledge base of programming plans, goals, and common bugs associated with them are used. PROUST starts to debug a student's program by selecting one goal from the problem description. Then it retrieves a set of plans which implement this goal from the knowledge base, and tries to match the individual plans with the code. Since a plan can have
Subgoals, this process is recursive. PROUST's strengths derive primarily from its knowledge representation. Based on the problem description, PROUST does top-down template matching. This depth-first search technique minimizes the search space required for successful program identification. Since PROUST knows the intentions of a programmer from the problem description, intention-based diagnosis is possible. In addition, by using the analysis by synthesis method, PROUST can simulate the way a programmer programs given problem. PROUST also has some limitations. First, it requires a problem description to analyze a program. However, to write a problem description of a program task is very hard for an ordinary teacher. Second, PROUST has difficulty in coping with syntactic variabilities.

CAMUS [9] analyzes programs using an automated reverse engineering approach. It analyzes the student's program without any foreknowledge of the intended behavior. The result of this analysis is a high-level Program Description of what the student's program does. A correct reference solution for the assignment is analyzed in a similar way. Feedback is generated by comparing both high-level Program Descriptions. In spite of its abstraction skills based on the cognitive model of programming knowledge, the high level description often becomes too complex. Even for programs not exceeding 30 to 40 lines of code, the underlying interpretation process is very complex [10]. This problem is mainly due to the fact that CAMUS does not use the reference program in analyzing the student's program. The reference program is used only to generate the reference Program Description.

C-TUTOR: AN OVERVIEW

C-Tutor is a complete ITS to teach students with C programming language, analyze students' programs and give intention-based diagnosis [8]. In intention-based diagnosis, faults are found in artifacts through a process of understanding the intended structure and function of the artifact, and determining whether or not those intentions were successfully realized. Intention-based diagnosis can identify a wider range of errors than other methods of error diagnosis, because it allows the diagnostician to detect deep faults resulting from design errors. In C-Tutor, the intention of a programmer is provided to the system as a problem description of the program task. This problem description has the information of goals which have to be implemented in the student's program and objects used in the program, and is generated by a reverse engineering system. Using this as top-down knowledge, a didactic system can analyze students' programs more thoroughly and give advice more helpfully. As a learning environment, C-Tutor teaches concepts of C language and skills of programming based on the current status of a student. Figure 1 shows an overview of C-Tutor.
Knowledge representation of C-Tutor

The knowledge base of C-Tutor is represented as frame structure. Goals and plans are the main component of the knowledge base. Plans describe stereotypic action sequences in programs, and goals represent common concepts of programming such as the swapping of two values or finding the maximum value among given data. Figure 2 is an example of goals and plans represented as frame structure.

Goals and plans are related hierarchically. There are three types of goal/plan hierarchy. The major hierarchy is :Instance slots of goal frames. These slots have the names of plans which can be used to implement the goal. In the other hierarchies, goals are included in the plans. Plans can include some goals as their prior goals or as subgoals in the :Template slot. There can exist an exceptional goal for a plan. The plans in the :Instance slot of a goal have OR relations, while the goals included in a plan have AND relations. Thus, to implement a goal, one of its instance plans can be selected. On the other hand, to realize a plan, all of its prior goals, subgoals, and exceptional goals should be implemented. Figure 3 shows an example of goal/plan hierarchy of the Average goal.

This goal/plan hierarchy let C-Tutor manage implementation variations. In the above example, to implement the Average goal, a student can choose one of four plans. Each plan has prior goals and an exceptional goal, and these goals also have alternative choices. Therefore, lots of different styles of programs can be interpreted as implementations of the Average goal.

C-Tutor: running example

There are two interacting cycles in C-Tutor: Student–System and Teacher–Student–System interacting cycle. In the former cycle, referring to the student model in Curriculum Network, the system teaches a programming concept and presents a program task. The generated program is handed to GOES in order to remove syntactic variations. ExBug analyzes the processed program, and gives intention-based diagnosis. Based on this diagnosis, the system decides whether to present another program task or to teach a new concept. The latter cycle includes another external agent, a teacher. ExBug is developed as an assistant of the real C language course. In real classrooms, teachers often present their own problems to students. In this case, a teacher describes the problem in natural language to students, as well as provides a sample program to the system. GOES analyzes this sample program and generates a problem description of it. A student writes a program of that problem, GOES transforms it into a canonical form,

\[
\begin{align*}
\text{:goal-ID} & \quad \text{SentinelControlled-Input-Sequence} \\
\text{:Instance-of} & \quad \text{Read-and-Process} \\
\text{:Form} & \quad \text{(SentinelControlled-Input-Sequence ?New ?Stop)} \\
\text{:Main-variable} & \quad ?\text{New} \\
\text{:ObjectConds} & \quad (?\text{New (:InputObject))?\text{New (:NEQ ?New ?Stop)}) \\
\text{:important-p} & \quad T \\
\text{:Instances} & \quad \text{(Sentinel-ProcessRead-While} \\
& \quad \text{Sentinel-ReadProcess-While} \\
& \quad \text{Sentinel-RBP-DoWhile} \\
& \quad \text{Sentinel-ReadProcess-DoWhile} \\
& \quad \text{Bogus-YesNo-Plan} \\
& \quad \text{Bogus-CounterControlled-Loop)}) \\
\end{align*}
\]

\[
\begin{align*}
\text{:plan-id} & \quad \text{Sentinel-ReadProcess-While} \\
\text{:InstanceOf} & \quad \text{SentinelControlled-Input-Sequence} \\
\text{:Variables} & \quad (?\text{New}) \\
\text{:Constants} & \quad (?\text{Stop ?SeedVal}) \\
\text{:MainCompo} & \quad (:\text{MainLoop :Next}) \\
\text{:Template} & \quad (:\text{Init (:setq ?New ?SeedVal)}) \\
& \quad (:\text{MainLoop (while (:NEQ ?New ?Stop)}) \\
& \quad \quad (:\text{Next (Subgoal (Input ?New))}) \\
& \quad \quad (:\text{Guard (Subgoal (Sentinel-Guard ?New ?Stop ?*)})\})
\end{align*}
\]

Fig. 2. Example of a goal and a plan.
and then ExBug analyzes the program. A student can use the system until there is no error in the program.

**PROGRAM ANALYZER OF C-TUTOR**

Analyzing programs is very difficult because of significant variability. By selecting algorithms, varying control flows, constructing clichés, and choosing keywords, programmers can write a number of correct implementations. If the various types of bugs are considered, the variability of programs is almost infinite even for trivial problems. Johnson [11] made a claim that if a debugging system is to cope with the various types of errors that programmers make, it must understand what the programmer is trying to do. Johnson refers to this as intention-based diagnosis. In our system, the intention of a programmer is provided to the system as a problem description of the program and is extracted from a sample program by a reverse engineering system. Then, a didactic system analyzes students' programs based on the description.

**GOES: a reverse engineering system**

Analyzing a novice program is very difficult because it usually contains many bugs. That is why successful systems have used top-down knowledge which gives information about the programs to be analyzed. There are several ways to provide the system with knowledge of a problem. Some systems have problem libraries, and let the student choose one of them [7]. Some request the external agent to input the problem description in a predetermined form [8]. Others assume that the system already knows about the problem [12–14]. Among the above, the second one has the advantage that teachers can create plenty of problems, but also has the disadvantage that they must write the problem descriptions by themselves. The easiest way for a teacher to write a problem description may be to write a model program of it. In C-Tutor, a problem description is automatically generated from a sample program of a teacher by a reverse engineering system called GOES. Figure 4 is a problem description of the following problem.

**G: Average**

![Diagram of G: Average](image)

Fig. 3. Goal/plan hierarchy of Average goal.
Calculate an average from a series of scores which is entered successively until the sentinel value 999 is entered. Since input data represent scores, they have to lie between 0 and 100. The output values are the number of valid inputs and the average.

GOES generates a problem description in the following process. First, GOES extracts plans from a sample program. Then, goals are extracted from the plans by goal/plan hierarchies. The problem description of a given program is completed by adding information about objects used in the program. ExBug needs a set of test data for dynamic analysis. Sets of test data must be added by a teacher manually.

C programming language provides users with lots of selective alternatives to similar functions for the purpose of convenience in coding. This makes programs differ in shape even though programmers use the same algorithm. In addition, one of the major weaknesses of PROUST and ExBug is that they had difficulty in coping with syntactic variability [10]. To remove these syntactic variations, GOES transforms input programs into canonical forms during the preprocessing phase. Preprocessing is applied not only to the sample program but also to the students’ programs.

Plan extraction is the most important and time-consuming process in program understanding. GOES raises the efficiency of plan matching by using labels which denote the functionality of each programming statement in the template slot of a plan frame. In Fig. 2, :Init, :MainLoop, :Next, and :Guard are labels. Along with labels, GOES uses the information of variables which programmers usually use to group noncontiguous programming statements [15]. GOES extracts candidate plans using only labels and variables. To extract candidate plans more efficiently, introduced are the concepts of necessary and sufficient condition of a plan to the program. To satisfy the necessary condition to a program statement, the plan must have the same label as that of the program statement. The sufficient condition of a plan to a program is that all the label and variable pairs of the plan must exist in the program, and there should be no conflict in variable bindings nor data-flow violation. The necessary condition reduces the search space, while the sufficient condition makes it possible to use the top-down knowledge of the plans. Thus GOES finds its candidate plans in a hybrid way as Quilici did [16]. Candidate plans are the ones which satisfy both the necessary and the sufficient condition. Plans are extracted from the candidate plans by applying exact code matching. Since exact code matching is applied only to the candidate plans, GOES can save lots of computational efforts. Goals are extracted from plans by Goal/Plan hierarchies. A plan is recognized if and only if all of its sub-, prior-, and exceptional-goals are extracted. Once a plan is recognized, its parent goal is extracted. Since this goal can be one of the sub-, prior-, or exceptional goals of any plan, this process also continues recursively. Goals mentioned in the problem description are the highest ones on the hierarchy and the ones of which the result variable is used as an argument of an "Output" goal. In the example of Fig. 3, the Average plan is recognized if and only if its three children goals, Count, Sum, and Guard-Exception are recognized, and these goals are recognized when one of its instance plans is recognized. Since the Average goal is the highest one, it is mentioned in the problem description of Fig. 4. In addition, since the result variable of goal Count should be printed, it is also mentioned in the problem description.

*ExBug: a didactic system*

ExBug is a knowledge-based program analyzer for novice C programs. Program analysis can be categorized as dynamic analysis or static analysis [2]. Dynamic analysis finds bugs by the bug symptoms
generated during program execution, while static analysis detects errors by internal behavior analysis, program verification, or program recognition. Static analysis can detect errors which are difficult to find by dynamic analysis, but a more thorough understanding of a program is needed. ExBug debugs programs using both dynamic and static analysis. ExBug is composed of three subparts: Program Executor, Plan Matcher, and Bug Finder.

Program Executor runs the program with given test data. A simple plan-matching approach shows high accuracy in diagnosing bugs, but there are some serious problems. The knowledge base of stylistically dubious plans makes the system hard to decide which plan was used in the student’s program. Correct but extraordinarily implemented programs cause a significant number of false alarms. In addition, there is not sufficient information for precise matching in a plan level. Execution results generated by the Program Executor are used to help the Plan Matcher to diagnosis the students’ programs more efficiently and precisely. The execution results are used in following ways:

1. Guide to goal selection
   By executing the student program, we can know which goal is implemented correctly and which is not. This information is used as a guideline for the goal selection in the plan-matching process. ExBug matches correctly implemented goals first.

2. Information on variable binding
   Using the execution result, ExBug can assume the role of variables. The information on variable binding is used to reduce the search space of plan matching.

3. Information to select the best interpretation for unmatched components
   Program execution gives the evidence for the matching results, and gives information for the selection of the best interpretation. If an initialization component in a plan is not matched, there will be an unbound variable reference error during the program execution. When a plan component of guarding an object is not matched, ExBug tries to find an occurrence that the invalid data is used in other calculations.

4. Example of a bug
   Execution results can be used to explain the bug according to the cause–effect relationship. An explanation using a bug example is more natural and understandable for novice students.

Plan Matcher analyzes the student’s program by matching it against the relevant programming plans in the knowledge base. The methodology of plan matching is similar to that of PROUST [8], analysis-by-synthesis method. To understand a program, ExBug selects one goal from the goal agenda which is initialized with the information of goals in the problem description. Then it retrieves a set of plans that implement this goal from the knowledge base, and tries to match the individual plan with the code. If a plan has subgoals or prior goals, those goals are also put into the goal agenda. ExBug tries to find a suitable path of interpretation until the goal agenda is empty.

Bug Finder analyzes the bugs in the student’s program using the results of the Plan Matcher and Program Executor. ExBug tries to explain the matching differences by consulting with the execution results. Once the analysis of the program is completed, the bugs are reported to the student. The list of bug descriptions produced by the bug finder is passed to the ExBug’s bug reporting mechanism which orders the bugs according to severity and groups similar bugs together. It will then generate English or Korean text to describe the bugs. In addition, if there is any example of a bug, Exbug tries explain the bug according to the cause–effect relationship.

LEARNING ENVIRONMENT OF C-TUTOR

In C-Tutor, the learning environment is built as Curriculum Network. Curriculum Network is not only the tutoring module of C-Tutor, but also the subject matter and the student model module of C-Tutor. This network is composed of goals, plans, and some exercises. Goals and plans are in the same knowledge base as ExBug and GOES. To construct the curriculum of C language with goals and plans has following advantages.
1. Novice programmers tend to program by syntactic and control-based code segments. However, experts use only plan groupings in both writing and understanding programs [17]. This implies that teaching novices with programming plans helps them to learn the programming skills of experts.

2. The knowledge base of C-Tutor is organized hierarchically. Therefore, the system can use the information of prerequisite goals in tutoring module.

3. Since C-Tutor analyzes students’ programs by intention-based diagnosis, the feedback from the system is also based on the goals and plans. A curriculum based on goals and plans is the best one for this purpose.

As mentioned before, goals and plans are organized as and/or graphs. For every frame of goals or plans, added are a slot for the teaching concept of each goal and plan, and a slot for whether the goal or the plan is studied or not by the student. In addition, each link of the graph is augmented with the relation between frames such as analogy, generalization/specification, deviation. This augmented link is used to teach students plan concept with pre-studied ones.

Some exercising problems are included in Curriculum Network. A problem node is also represented as a frame which has five slots. The :problem-id slot has the name of the problem and the :content slot describes the problem in natural language. The :needGoal slot has the name of prerequisite goals, and the :solved slot has information about whether the student already solved the problem or not, whether he/she programmed the problem correctly. The last slot, :PDF, has a pointer to the problem description for ExBug to analyze a student’s program. Curriculum Network is also used as a student model of C-Tutor. Each node of goals and plans has a slot named :Studied. If a student already studied a goal or a plan, the :Studied slot is marked. Since the knowledge status of a student is described as a subset of the system’s knowledge base, this is an overlay model. The tutoring cycle is composed of the concept study phase, the programming phase, and the review phase. In the concept study phase, the system selects the lowest plan concept unstudied in the knowledge base, and teaches the concept of the plan. Then the system selects a proper exercise which is related to the plan and all of the prerequisite goals are studied. ExBug analyzes the student’s program and gives intention-based diagnosis to the student. If the student failed to program a plan, the system goes into the review phase. In the review phase, the system shows the concept of the plan along with the feedback message from ExBug. After the review phase, the system presents another problem to the student. The programming phase and the review phase are repeated until the student can program the given problem correctly. The relation between goals and plans are used to explain a new concept of a plan or to explain the misconception of students.

**IMPLEMENTATION AND EVALUATION**

Each module of C-Tutor is implemented in Common LISP on a workstation while the interface windows are implemented in C language on an X/Motif environment. An individual test for ExBug has been performed on students’ programs. These programs are collected from the students who attended the KAIST introductory C programming course by modifying the UNIX command they use to invoke a C compiler. Table 1 shows a test result of ExBug on a rainfall problem from PROUST [8].

<table>
<thead>
<tr>
<th>Table 1. Test result of ExBug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of programs:</td>
</tr>
<tr>
<td>Number of successfully analyzed programs:</td>
</tr>
<tr>
<td>Total number of bugs (from 223 programs):</td>
</tr>
<tr>
<td>Bugs recognized correctly:</td>
</tr>
<tr>
<td>Bugs not reported:</td>
</tr>
<tr>
<td>False alarms:</td>
</tr>
<tr>
<td>Number of programs failed to be analyzed:</td>
</tr>
</tbody>
</table>

GOES successfully generated problem descriptions when tested with 41 problems which include 37 exercises and examples of [18], average problem, Rainfall problem [8], sort problem, and rainfall plus sort problem. The sample programs are generated by 12 teaching assistants of the C language course. In the experiment, GOES can interpret any of the implementations as long as they exist in the knowledge base. Rich and Waters [19] pointed out the difficulties of the plan-based program understanding. They are syntactic variabilities, implementation variation, noncontiguously, and overlapping implementation. Preprocessing of GOES can absorb syntactically different programs into a canonical form. Implementa-
tion variabilities can be handled by the goal/plan hierarchy. During the experiment, programs implemented in very different styles produced the same set of goals. Noncontiguousness problem is solved because label and variable pairs efficiently group the noncontiguous plan fragments. Once the plans which satisfy the necessary condition are extracted, GOES matches the plan templates to the program. Therefore, overlapping implementation can be managed easily.

A program analyzer for ITS investigates a student's solution for a certain problem, and provides feedback on this solution. The requirements of a program analyzer for ITS are proposed by Vanneste [9] as follows:

1. Handle **syntactic variation**: students must be allowed to choose their variables and procedures, to use different loop constructs, and so on.
2. Cope with **implementation variation**: different obvious algorithms for the same concept must be recognized.
3. Recognize **erroneous implementations**, and give precise descriptions of the errors found.
4. **Give feedback on the efficiency** of programs.
5. Recognize its own limits: if it has not enough knowledge to analyze a certain program, then it should say so and direct the student to a teacher for further help. Thus, the em *reliability* should be high.
6. Generate **understandable feedback**, based upon the concept known by the students or the concept being taught to students.

In our system, syntactic variation is removed by parsing and transforming. The input C program is parsed into a Lisp-like one. Then syntactic variations are removed by program transforming. The hierarchical structure of goals and plans let the system handle implementation variations. Erroneous programs are efficiently recognized in a top-down approach. By combining dynamic and static analysis, our system can raise reliability and reduce false alarms significantly. Feedback is based on the intention-based diagnosis along with the running example of a bug. Therefore, students can get more useful advises such as the cause of an error, and understand their errors naturally.

Besides the above requirements, the extensibility of the knowledge base and the problem scopes is also an important requirement for the program analyzers of ITS. Since our approach chose the frame structure to represent goals and plans in the knowledge base, it is relatively easy to append new plans and goals. In addition, new problems can be easily set because the problem description is generated from a sample program automatically.

**CONCLUSION**

In this paper, a knowledge-based programming tutoring system for novice C programmers is described. The most important part of a programming tutor is a program analyzer. In C-Tutor, we built a program analyzer which is composed of a reverse engineering system and a didactic system. The reverse engineering system extracts the problem description from a teacher's program. Based on the description, the didactic system can diagnosis students' buggy programs. A knowledge base of C-Tutor is reorganized as genetic graphs of goals and plans by Curriculum Network. This enables C-Tutor to teach students the concept of plans and goals using their relations such as analogy or generation/specification.

C language is one of the most popular programming languages all over the world. However, there has been no ITS for C language until now because programs written in C language are usually too hard to be analyzed automatically. The program analyzer for C-Tutor has successfully analyzed real novices' programs collected during beginner C language courses, and satisfies all the requirements as a program analyzer for an ITS. In addition, it gives intention-based diagnosis to the programmers, which can diagnose the misconceptions of a programmer as well as locate the bugs. The bugs found by program analyzers are used as guidelines to teach the concepts of programming skills and select pre-stored exercises to practice them. Teachers can set new problems easily, since all they have to do is write a sample program of the problem and GOES automatically generates a problem description. This makes C-Tutor more practical.

**REFERENCES**