Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

INTELLITUTOR: integrated intelligent programming environment for novices

H. Ueno, A. Nakajima

Department of Systems Engineering, Tokyo Denki University, Hatoyama-cho, Saitama-ken 350-03, Japan

ABSTRACT

The INTELLITUTOR was designed as an integrated environment for both learning programming and friendly programming aid. The system consists of three software modules, i.e., GUIDE, ALPUS and TUTOR, one document database and four knowledge bases. GUIDE is a guided editor for easy coding, ALPUS is a knowledge-based program understander, and TUTOR is an embedded intelligent tutoring system. ALPUS infers user's intentions from buggy program codes by means of knowledge-based reasoning. INTELLITUTOR attempts to work as a human programming tutor to guide a user in programming, to detect logical errors within a user' program, and to make advices not only for fixing them but also for letting him notice his misunderstandings. The key knowledge used to understand user's programs is the knowledge about algorithms, which is organized in a form of hierarchical data structure called Hierarchical Procedure Graph(HPG). The knowledge about programming techniques and errors are organized around it. Although the experimental system is dealing with PASCAL programs most knowledge maintained within the knowledge base is applicable to other procedure-oriented programming languages as well. The system was implemented in the frame-based knowledge engineering environment ZERO on a UNIX workstation.

INTRODUCTION

An integrated intelligent programming environment must be one of the most attractive and useful applications of this kind of technologies[1]. By an intelligent programming environment we mean that a computer assists users in programming, comprehends user's intentions from buggy program statements, detects logical bugs included, makes advices for fixing them, and so on. These intelligent functions can only be achieved by means of knowledge-based AI technologies.

The key issue in realizing knowledge-based intelligent programming environments is that the system has the ability of program comprehension as mentioned above. We can obtain useful hints from watching the intelligent tasks achieved by human tutors. They can understand semantics of computer programs, detect logical errors, guess programmer's intentions from buggy programs, and make advices to fix these bugs. An intelligent programming environment is required to have these functional abilities. Knowledge about

topics, algorithms, coding techniques, programming languages and errors seem to play an important role in this kind of situation. It is quite reasonable that this kind of programming environments should include an intelligent CAI system as a subsystem in order to realize high performance in guide, help and teach. In other words this kind of systems is useful as an intelligent tutoring system for learning programming. This is one of major reasons we have been developing the INTELLITUTOR system.

While, it should be noted that usual tutoring systems need a long sequence of interactions between a user and a computer in order to create a user model, i.e., a student model, so that teaching session is achieved suitably for him. This interaction is time consuming and uncomfortable, and should be minimized. By an embedded tutoring module within an programming environment this kind of interactions could be minimized, since most information for creating a user model can be obtained through the program comprehension functions. This is an issue of our INTELLITUTOR trial.

The ability of program comprehension is one of the key functions in realizing intelligent programming environments as mentioned above. Several approaches for program comprehensions have been tried by other groups. PROUST[3] is a cognitive model based program understanding system for novice programmers, where programming plans play a key role. For a very simple PASCAL program to solve such as 'rain fall problem' PROUST demonstrated interesting functions. The system detects several logical bugs occurred within a student's program, gives explanation about causes of them and makes advices for fixing them. In LAURA[1] a graph model, called 'program model', is used. LAURA comprehends simple FORTRAN programs by a graph matching method between a teacher's correct program and a student's buggy program. Ideas of cognitive science is not applied in this system. Therefore the ability is limited. LAURA can detects buggy codes. It cannot, however, infer user's intentions.

In ALPUS the knowledge about algorithms plays a key role, and the knowledge about programming techniques supports it. Because of algorithm-based approach ALPUS can deal with algorithmically complex PASCAL programs. In PROUST a program is supposed to be designed as a combination of simple programming plans, i.e., programming parts. While, in ALPUS an algorithm is supposed to give the skeleton of a program, and programming techniques are used to implement it. A plan-based approach is applicable to very introductory programming education. While, an algorithm-based approach is suitable for higher classes in programming education. Another characteristic of ALPUS is that it is an embedded subsystem within an integrated programming environment, while PROUST and LAURA are stand-alone systems.

ALPUS uses four kinds of programming knowledge in comprehension: knowledge about algorithm, knowledge about programming techniques, knowledge about programming language and knowledge about how novices make errors. The knowledge about algorithm is represented in a hierarchical data structure called Hierarchical Procedure Graph(HPG). HPG is an abstract representation of knowledge about an algorithm. Each node of HPG represents each process, i.e., a chunk of operations, of the algorithm by means of a frame-based knowledge representation scheme. Knowledge about programming techniques and errors is attached to associated nodes in a HPG graph. No IF-THEN rules, i.e., shallow knowledge, for detecting bugs are used in ALPUS. Program comprehension is achieved by means of semantic pattern matching method, instead. The HPG-based knowledge structure is interpretation-independent and is shared by multiple purposes in the INTELLITUTOR environment, i.e., for guide of user's program coding, for program comprehensions, and for program teaching. This means that the HPG-based knowledge representation is one of deep knowledge representations, and its method of program comprehension is a kind of model-based reasoning which is much more flexible and powerful than rule-based reasoning used in ordinary knowledge-based systems.

Although the current system deals with the PASCAL language, most of the knowledge is applicable to those of procedure-oriented programming languages such as FORTRAN and C.

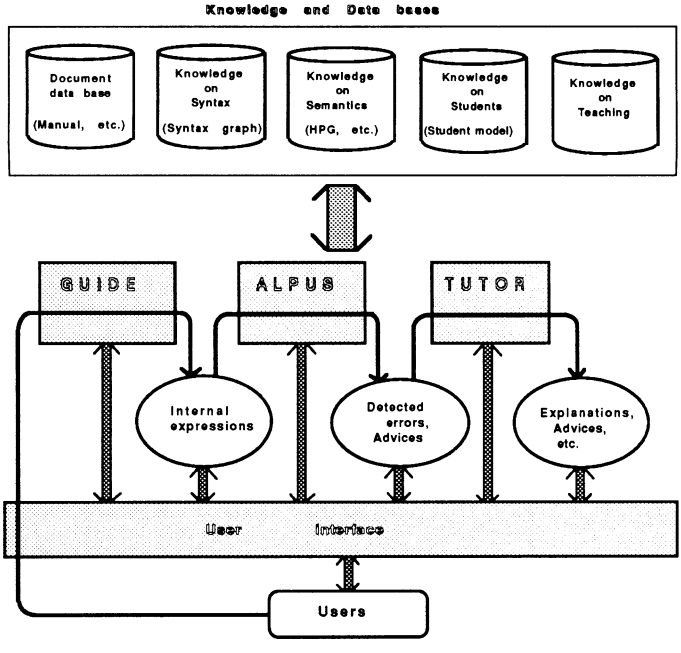


Fig.1 An overall view of the INTELLITUTOR system.

The INTELLITUTOR system was implemented in the frame-based knowledge engineering environment ZERO and working on a UNIX workstation for system evaluation.

In this paper we discuss the concept and outline of the methodology of implementing the INTELLITUTOR system. The discussion is mainly focused on the program comprehension function since the most important characteristic of the system has been brought from this ability.

OUTLINE OF INTELLITUTOR SYSTEM

In interactive programming situations a human tutor watches user's program texts displayed on a screen terminal, guesses his intentions from buggy and/or unskilled program sentences, detects semantic errors. According to this analysis the tutor gives advices for fixing the errors, and teaches him proper programming knowledge. INTELLITUTOR attempts to work like this. The system consists of one document database, four knowledge bases, three major software modules and a user interface as shown in Figure 1.

The knowledge bases consist of the knowledge on syntax which is based on the syntax graph of the PASCAL language, the knowledge on semantics which is represented in the HPG graph formalism, the knowledge on programming techniques which is linked to associated HPG nodes, the knowledge on a user which is represented as a student model and the knowledge on teaching. The document database includes documents of the PASCAL programming manual, the language specification in BNFs and syntax graphs and program examples. These data are organized in a form of hierarchically structured hyper texts and linked together so that suitable information can be retrieved and displayed for a user in both program coding session and tutoring session by request. Attribute informations such as key words and

pointers are maintained by frame systems, while documents themselves are stored within document files for higher flexibility and maintainability.

The software modules, i.e., GUIDE, ALPUS and TUTOR, achieve intelligent functions of the INTELLITUTOR system by means of knowledge-based reasonings respectively. In other words, these modules are inference engines for solving problems. Frame-based reasoning technologies such as inheritance controls, attached procedures and message passings are applied to implement these functions, in addition to symbolic data processing techniques.

GUIDE is a knowledge-based guided editor. This subsystem guides and assists a user who is trying to write a program by using the knowledge on syntax, i.e., syntax graphs, for PASCAL-oriented program editing, and the knowledge on algorithms, i.e., HPG graphs, for algorithm-oriented program editing. The syntax graph-based guide is the first choice. Help functions are available during program editing by request. Source programs entered by the user are converted into an internal representation form which is an object for program comprehension by ALPUS. Text editor is available as well. In this case the guide function does not work.

ALPUS receives a program from TUTOR, comprehends it, detects logical bugs and makes advices for fixing them. In program comprehension ALPUS uses not only correct knowledge about algorithms and associated implementation techniques but also the knowledge on how novice programmers make confusions and mistakes. The second type of knowledge is used to guess user's intentions from buggy program statements in order to detect logical errors and to give advices for fixing. The knowledge for program comprehension is obtained from students by means of cognitive experiments and organized around the HPG-based data structures as discussed in the following chapter. The information obtained via GUIDE and TUTOR sessions is passed to the TUTOR module for creating a student model.

TUTOR may be activated by the system monitor when the ALPUS module terminates in normal situation of use. The TUTOR system receives the information about the student from the GUIDE and ALPUS systems in order to create a student model on current user. Additional information is obtained by interactions. Because major information for creating the student model can be obtained via GUIDE and ALPUS the interactions could be minimized. According to the student model TUTOR can teach him appropriate programming knowledge. Most of the information maintained in the document database, the syntax knowledge base and the semantic knowledge base are available for teaching by the TUTOR system as well.

The user interface module provides a friendly human interface for users. The concept of Direct Manipulation was applied in designing the interface. Therefore, users are allowed to look the contents of data and knowledge, to define and manipulate them, to activate a variety of functions, by means of multi-windows and a mouse pointing device. There is no need to know the complicated data structures and programs in using the INTELLITUTOR system for users, i.e., novice programmers.

GUIDED PROGRAM EDITING

The GUIDE subsystem guides a user who is trying to write a new PASCAL program and modifies old versions of programs. Three guide modes are available: a text editor mode, a syntax editor mode and a HPG-based editor mode. The HELP functions are available for every mode during program editing by user's request. The system does not guide anything in the text editor mode since this is the same as a usual text editor for writing programs.

The syntax editor guides a user by means of the syntax graphs of the PASCAL language. The combination of the multi-window system and the mouse-based pointing device is quite useful in implementing the GUDE system. During program editing the system monitors user's programming tasks and shows informations to be done next. At each place of writing a new statement in programming the editor shows a list of available statements within the *symbols*

window. All he has to do is to choose one by mouse clipping. For example, only the 'program' statement is available at the beginning of PASCAL programming since the first sentence must be the 'program' statement. Therefore, only 'program' is displayed within the window. The contents of the window are updated automatically to show the next candidates just after the clipping operation. The statement template for the clipped one is appended or inserted within the program codes. Next, the place and the type of sentence to be filled out is shown by the system. Figure 2 shows an example of program coding under the guide of this mode. As shown in the text window of the figure the user is filling out the *for* statement which was inserted by mouse clipping before. The system requires him to fill out FOR-LIST at this moment.

Within the *Pascal User Manual* window the paragraph expressing the *for* statement is displayed upon his request, i.e., by pointing the Help function within *System-Menu*. Within the *Backus-Naur Form* window the BNF definition for the *for* statement is shown as well. This information is displayed by pointing Backus-NaurForm within the window. If he need more information it is allowed by continuing clipping. By pointing Quit he can return to the *Text-Window* in order to continue the program editing. Putting an identifier into the statement is done by clipping one from a list of the identifiers within the *Identifier* window. The list of identifiers was created at the time when he defined the variables before. It should be noted that even if there is no chance to occur any syntactic bugs in editing a program in the syntax guide mode, semantic errors would be included when he has misunderstandings about the effects of some PASCAL statements. For example, there are novices who have confusions between WHILE and REPEAT statements. This type of errors could be detected by ALPUS. In case of HPG-based guide the user will be able to write algorithmically proper structure of a program much more smoothly. In this mode the syntax guide is included automatically. User's programs may include still algorithm-oriented bugs since the HPG shows just the skeleton of the algorithm. The program written by the student is converted into the internal representation and passed to the ALPUS system, where logical errors are detected and student's intentions are inferred by knowledge-based program comprehension techniques.

PROGRAM UNDERSTANDING

In this chapter we discuss the concepts and methodologies of program understanding done by ALPUS.

Program Semantics and Programming Knowledge

It may be defined that understanding is the process to reason about the meaning of things. The meaning of things may be represented by means of a data structure, i.e., a cognitive model, of the object. This data structure can be called semantic data structure, or simply semantic structure. Understanding can be achieved by pattern matching of an object to the semantic structure associated with it. The meaning of programs might be divided into four semantic levels of hierarchy as follows[10]: Data solving notion level, Algorithm level, Data processing technique level, and Basic data handling operation level, from the top to the bottom of the hierarchy. The semantics at these levels is programming language independent. The semantics of a programming language comes under the data handling operation level. In this level effects of commands constitute the semantics. For each level of program semantics specific programming knowledge is associated. Current ALPUS focuses on the lower three levels of the programming semantics.

Program understanding can be achieved by knowledge-based reasoning. Programming knowledge is used in each phase of programming tasks by a programmer. In other words, the quantity and quality of programming knowledge result in programmer's ability in program development. Expert programmers seem to have a large amount of programming knowledge, by which they design a data structure, an algorithm and implementations of it, write the program and so on.

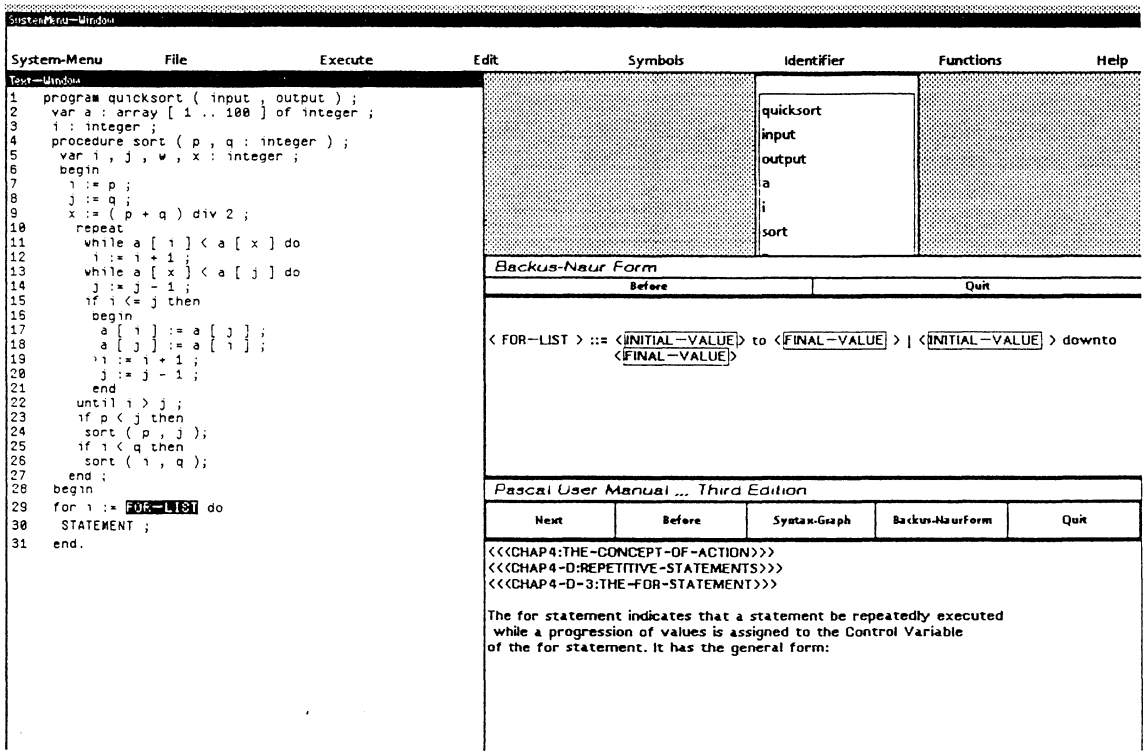


Fig.2 An example of program editing session by GUIDE.

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Programming knowledge is classified into Semantic knowledge, and Syntactic knowledge. The first one is the knowledge about programming methods, and the second one the knowledge about programming languages. These two classes of knowledge should be independent from each other. The knowledge on programming methods seems to be organized in a four-level hierarchical structure just like the semantics of programs. In other words for each level of program semantics specific programming knowledge is associated. That is, Knowledge about data solving notions, Knowledge about algorithms, Knowledge about data processing techniques, and Knowledge about basic data handling operations.

The knowledge about basic data handling operations includes value assignment to a variable, comparison between two data, conditional branching and so on. These operations are grouped in modules as program commands in programming languages. Any complex algorithm can be implemented by proper combinations of the basic data handling commands in writing programs. The knowledge about data processing techniques consists of such as techniques of exchanging values between two variables, counting the number of data, and getting the largest value in an array. Programs written by expert programmers are easier to read because they have enough of this type of knowledge and use them in writing programs. It is very impressive that even if a systematic training on coding techniques has not been given to programmers their knowledge about coding techniques becomes similar through long experiences in programming [3,10]. Coding techniques are referred to as programming techniques as well.

The knowledge about algorithms consists of a set of typical data processing algorithms for basic problems such as an algorithm to compute a product of two arrays and to sort the data stored in an array. Since a unit of knowledge in this level is relatively large, only an abstract framework of an algorithm is stored within the memory, and details are referenced from associated external memories when they are needed. Pointers and related attribute information might be attached for reference. The knowledge about data solving notions is a declarative notion about typical data solving problems such as a notion to compute a product of two arrays, and a notion of sorting a set of data. There exist higher level notions which include lower level notions as subnotions. For example we can define a complex problem solving notion such as a notion of file maintenance which includes the notion of sorting a set of data as a subnotion. It is important that higher level knowledge is more declarative than lower level knowledge, and can be expressed by a set of lower level knowledge. The current ALPUS does not deal with this type of knowledge.

Concepts of Knowledge-based Comprehension

There is a well known concept on the relationship between a program and an algorithm, that is "program = data structure + algorithm". In this scheme the data structure can be defined as a model of the world. The aim of the program is to generate a goal state of the data structure from an initial state by repetitive conversion of the contents of the model. The algorithm gives a proper sequence of conversion operations on the model. While, an actual program consists of a set of variables and a sequence of operation statements. The set of variables is divided into two groups which are variables to describe the world model and additional variables for working memories for local data handlings.

From the discussion above we can easily derive the following assumption, that is, "a program has a specific structure". The program structure is strongly restricted by the associated algorithm. In other words, the proper sequence of data handling operations is strongly restricted by the algorithm. For example, although there are several well known sorting algorithms the Quicksort program has to be programed according to the Quicksort algorithm. Even if a program produces a correct result it is not to be evaluated as a correct program if the sequence of operations is not based on the specific algorithm. A programmer is allowed to write programs in his manner under this restriction. Programs which break one or more of the restrictions must have logical bugs. This is the important policy in designing the ALPUS system which should be called as an algorithm-oriented program understander.

We think that algorithm is not so important when a problem to be solved is very simple. "Rain fall problem" used in PROUST is such an example. Plan-based approach works well in the PROUST system for very introductory programming education. We know that in cases of solving a complex problem an algorithms play an important role. The Quicksort algorithm which we are dealing with as an case study in ALPUS is such an example. Even if a student's program to sort a sequence of values works out well we can detect some logical bugs within the program when it brakes the algorithmic structure of Quicksort in parts. We suppose that this approach is applicable to actual programs written by professional programmers as well. This is the major reason why we have chosen the algorithm oriented approach in program comprehension.

In case of a complicated algorithm such as the Quicksort algorithm the sequence of operations can be divided into several chunks. Each chunk consists of a number of operations and achieves a specific meaningful role in solving the problem. We call this a *process*. One process may be constructed by several sub-processes, each of them be constructed by several sub-subprocesses, and so on. Therefore an algorithm can be represented in a hierarchy of the processes. Each process, i.e., a group of operations, can be implemented by using proper programming techniques and then be written in a specific programming language.

From this scheme we easily understand that each variable has specific role in defining a world model and working memory and each operation also has specific role at the specific position in the sequence of data handling operations, so that the program achieve the objective appropriately. Program comprehension is the cognitive work to identify the roles of the variables, statements and processes by means of knowledge-based guessing. In other words a human tutor tries to create an image of a world model and an algorithmic structure in his mind from student's buggy program statements by means of knowledge-based guessing using a variety of his programming knowledge, which is obtained through his long time teaching experiences and organized systematically in his memory.

Knowledge Used in ALPUS

We have been trying to realize a program understander which works something like a human tutor, as mentioned before. We suppose that a human tutor understand a program by creating an image of the algorithm in his mind from a sequence of program statements written by a student. In this situation the knowledge of algorithms, techniques and errors may work together in creating the image. ALPUS uses four kinds of programming knowledge: the knowledge about algorithm, the knowledge about programming techniques the knowledge about programming language and the knowledge about errors. The knowledge about programming techniques include both the knowledge about data processing techniques and the knowledge about basic data handling operations in this paper.

The knowledge about algorithm plays a key role in ALPUS as discussed above. The rest of the programming knowledge is associated around this knowledge. We have developed the HPG(Hierarchical Procedure Graph) formalism for representing the knowledge about algorithm[7,8]. As mentioned before the algorithm gives a proper sequence of data handling operations on the world model. The operations can be divided into several groups in sequence. A group of operations achieves a meaningful task in solving problem and is called as a "process" in ALPUS. Each process may include several serial or parallel sub-tasks, and can be divided into subprocesses, which can be divided into sub-subprocesses. HPG is a hierarchical data structure to represent the relations of the processes. Each node represents each process. The sequence of processes is linked by directed arrows. An example of HPG for representing the Quicksort algorithm will be shown later. The knowledge about programming techniques are represented in semantic networks and attached to associated process nodes.

The knowledge about errors are linked to associated nodes as well. Logical bugs made by novices are divided into three types, which are algorithm-related bugs, technique-related bugs

and language-related bugs. The algorithm-related bugs are caused by misunderstandings and/or confusions of an algorithm. In programming education the lecture of specific algorithm is given to students prior to programming practice. Misunderstandings would occur in the lecture session. It is not easy to give a correct image about dynamic data handlings to students by means of blackboard-and-chalk method. AUPUS is useful to detect their misunderstandings and to let them aware correct knowledge.

Technique-related bugs may be caused by incomplete knowledge about programming techniques and operation semantics. These knowledge are obtained through repetitive programming practices personally and stored in memory. By repetitive usage of them the knowledge about programming techniques are polished up and standardized in his memory. Careless mistakes of using programming techniques in writing a program might cause bugs as well even if his knowledge is correct. Language-related bugs may be caused by confused memory of language semantics or careless mistakes.

The knowledge about bugs has been extracted by cognitive experiments upon novice programmers and analyzed. Typical bug knowledge have been represented as in buggy patterns, and then attached to related HPG nodes, technique frames or language frames respectively. By analyzing the buggy knowledge we have extracted programmer's intentions. These informations have been linked to associated buggy knowledge. Detection of logical bugs are achieved by means of the buggy patterns. ALPUS can infer programmer's intentions from buggy codes in addition to detecting logical bugs using these informations. Current methods of inferring intentions are rather simple. We are trying to develop more sophisticated methods for inferring intentions such as by means of reconstruction of mental programming actions in programmer's mind based on cognitive psychology.

Representation of Programming Knowledge

In ALPUS the algorithm-oriented programming knowledge plays a key role in understanding a novice's buggy program as discussed before. This knowledge is represented in a hierarchical data structure called HPG. The HPG graph works as a program template both in understanding a program in ALPUS and in writing a program in GUIDE. In this section the outline of the knowledge representation method in HPG is discussed by using an example of the Quicksort algorithm. The Quicksort algorithm was chosen for case study because the algorithm was popular and useful in both programming education and actual applications, and because the program was not too complex and not too simple.

The Quicksort algorithm gives a procedure to sort a random sequence of data which is stored in an array memory by a repetitive division of the sequence. By single division cycle one sequence of data is divided into a smaller value group and a larger value group using a base value. The division cycle is applied repeatedly so that every group member becomes single. The following example shows the first division sicle.

```
Initial state:      [2  6  3  9  4  5  1  7  2  4  8]
                     ^              ^              ^
                  Leftindex      Basevalue     Rightindex

After division:     [2  4  3  2  4  1] [5  7  9  6  8]
```

Figure 3 shows a skeleton of the HPG graph for the Quicksort algorithm. In the HPG formalism a *process* is represented by a rectangle and a sequence of processes is by arrows. A bold rectangle represents an operation process in which a chunk of operation commands is included. A fine rectangle represents an aggregation which is used to assign a name to a chunk of operation processes fo. readability. The aggregation process is a dummy process. A double rectangle represents a repetition control process and a rectangle having cut-marks at three

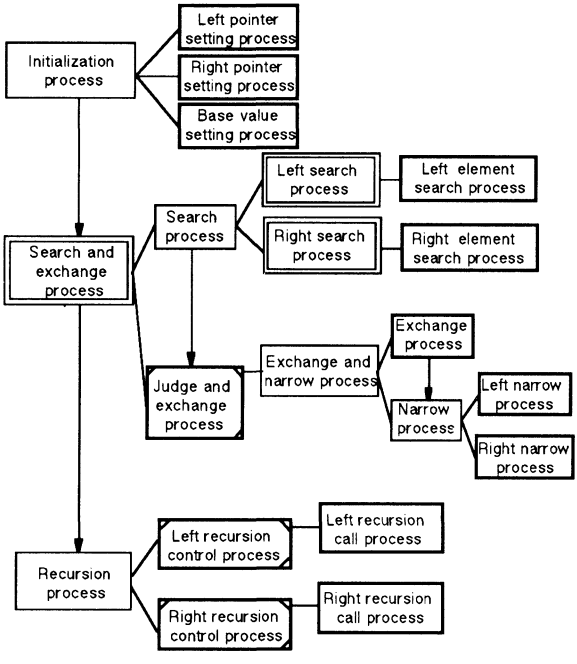


Fig. 3 The HPG graph of the QUICKSORT algorithm.

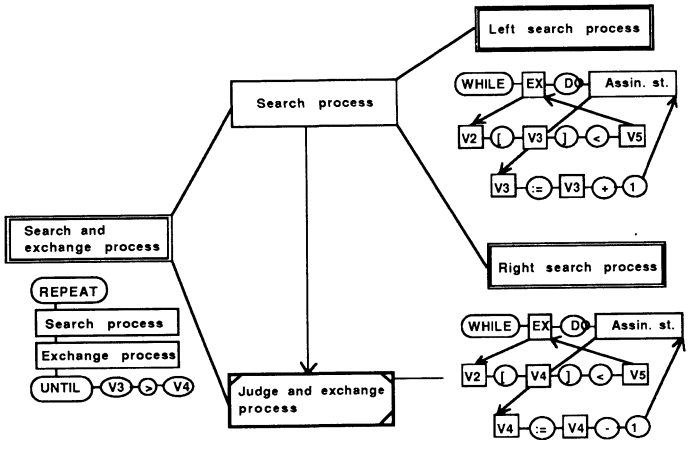


Fig.4 Semantic networks attached to associated

corners represents a conditional judge process. Each process is represented in a frame in ZERO as shown later.

As in the figure the SORT procedure consists of three major sequential processes which are Initialization process, Search-and-exchange process and Recursion-control process, in this sequence. It should be stressed that the data handling operations to achieve the Quicksort algorithm are divided into three major groups and that the operations associated to each processes are not mixed together over different processes, that is, any operations included within the Initialization process must not be located after the first operation associated to the Search-and-exchange process. The Initialization process consists of three parallel subprocesses which are Left-pointer-setting process, Right-pointer-setting process and Base-value-setting process, as shown in the figure. By parallel processes we mean that the order of execution is not restricted. The Search-and-exchange process, which is a repetition control process, has two serial subprocesses which are Search process and Judge-and-exchange process in this order of execution. The Search process is a process to find two elements to be exchanged and has two parallel subprocesses which are Left-search process for finding the first value element smaller than the base value from the left hand side of the array and Right-search process for finding the first larger value element from the right. The Judge-and-exchange process which is a conditional judge process exchanges the two elements which have been found by the Search process and then narrow the width of the sequence by two and check the termination of the process. The subprocesses of this process are structured as in the figure. The Recursion process which is the third major process controls the recursive execution of the Search-and-exchange process and consists of two subprocesses as in the figure.

Each process in the HPG representation can be implemented by means of the knowledge for implementation, i.e., programming techniques, which is represented in a semantic network and attached to the process node as shown in Figure 4. For example, for the Search-and-exchange process repetition control techniques by means of REPEAT or WHILE commands are typically applicable. In case of using the REPEAT command a proper template is given as in the figure where v3(variable-3) and v4 are referenced to decide the termination of repetition as shown. These two variables must appear within the Left-search process and the Right-search process where other control strategies and operations are given as in the templates as in the figure. These attribute informations are represented in a abstract expression so that the knowledge can be used not only for PASCAL but also for other procedural programming languages.

Knowledge Representation in Frames

Programming knowledge discussed in previous sections is represented in frames and organized as the semantic knowledge base as shown in Figure 1. Since the INTELLITUTOR system is implemented in the frame-based knowledge engineering environment ZERO[7], the knowledge base has been realized in the ZERO formalism. In order to make the knowledge base flexible and maintainable the programming knowledge was divided into several sub-knowledge bases which were algorithm-oriented knowledge, i.e., HPGs, technique-oriented knowledge and language-oriented knowledge. Each sub-knowledge base is organized in a ISA hierarchy.

The three sub-knowledge bases are linked together in order to retrieve in both directions. For example, the implementation techniques for the Left-search process which is shown in Figure 4 are stored in the technique-oriented knowledge base and the pointers are attached to both frames. Same method is used to link between techniques and languages. The knowledge about errors are linked to associated frames of these sub-knowledge bases respectively. For example, technique-related error knowledge is attached to specific frame of the technique-oriented knowledge base. Figure 5 shows an example of representing the technique knowledge for Search-Larger-Value. As shown in the figure this technique has one standard pattern, one acceptable pattern and two buggy patterns. The buggy pattern represents the knowledge on bugs.

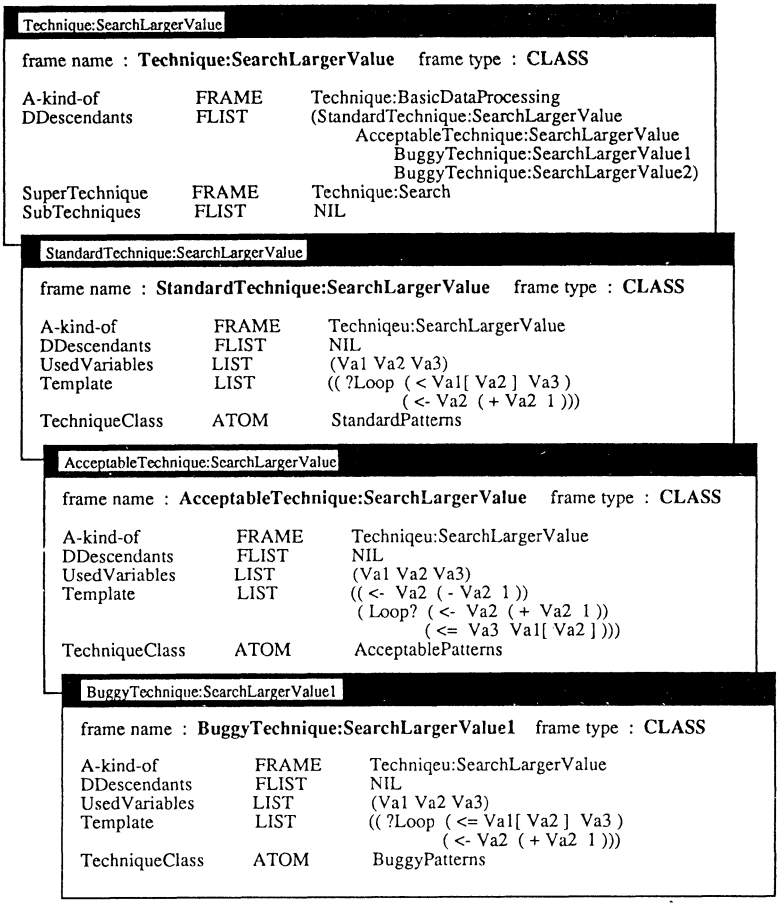


Fig.5 An example of frames for representing the knowledge of programming techniques and buggy techniques.

Method of Program Understanding

The program understanding is achieved by means of a knowledge-based pattern matching. That is, user's program elements are checked against associated program templates represented around the HPG graph. Understanding is achieved by three steps. At first, normalization of program representations is applied to an user's program. Since novices may write a program in a variety of coding styles normalization is needed to minimize the variety. Normalization results in higher flexibility in program comprehension by reasonable number of patterns. Second, identification of the role of each variable is done by means of data-driven reasoning, i.e., bottom-up reasoning. Because variable names that appear in the program would be different from those which are defined in the template a reasoning to identify the role of each major variable is needed. Since ALPUS has the knowledge about the variables,

which includes types of the variables, their roles and their algorithmic locations to be referenced or handled, it is possible to identify the role of each variable by checking these attributes between the data, i.e., variables used in the user's program, and the template, i.e., the knowledge. Prior to variable identifications segmentation of the program, i.e., rough identifications of HPG processes, is done. By this analysis associations between HPG nodes and related program chunks are roughly assigned. The segmentation is not easy when user's program has major algorithm-oriented bugs.

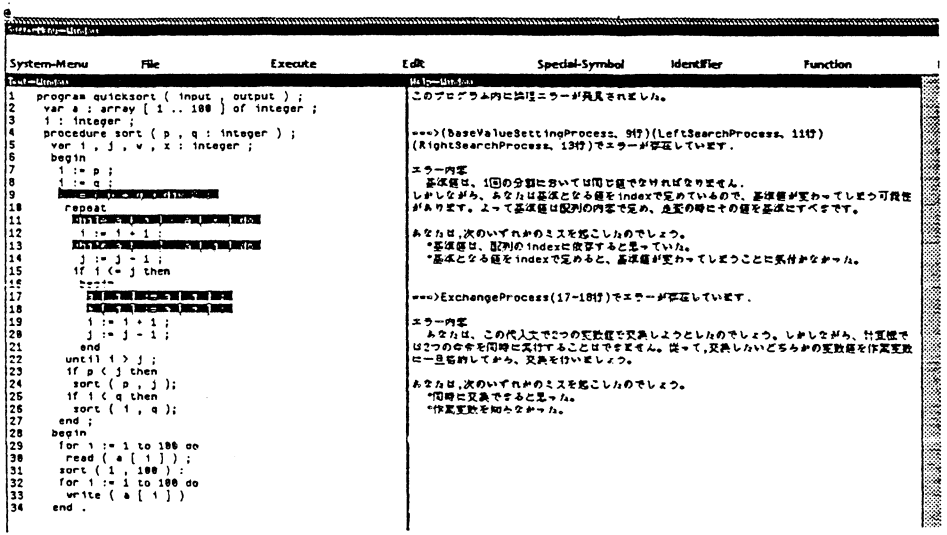
Finally, identification of each process and evaluation of operations used is done by means of model-based reasoning, i.e., top-down reasoning. Since the programming knowledge base includes several patterns for algorithms, techniques, languages and bugs, and associated knowledge is linked together within the knowledge base as discussed in the previous section, this analysis does not require complicated reasoning techniques. Template-based pattern matching is done by way of segment to segment matching. If the user's program segment matches the standard pattern, then the system understands this segment as a correct one. If it matches one of the acceptable patterns, then the system is able to make an advice to the user for writing better programs. If it matches one of the buggy patterns then the system understands this part as a buggy coding and assumes the user's intention using informations attached to the pattern. According to this analysis ALPUS can make an advice for fixing the error and letting him know his confusions, careless mistakes or misunderstandings. Further advices are given by the TUTOR system according to a student model.

Figure 6 shows an example of error detections by the ALPUS system and advices produced by the system for a user. In this case two logical errors have been detected. One is an algorithm-related error which occurs over three statements: in the base value setting process in the initialization process this student assigned the index value to the base value identifier instead of base value itself by confusion or misunderstanding of the Quicksort algorithm. The position, i.e., index, for the base value may be moved during a singe division cycle. Another one is a technique-related error that is the error on value exchange between two variable identifiers. This error might be caused by improper knowledge or careless mistake. The error locations in the source program texts are pointed by coloring as in the figure. The system tells the precise error locations in the algorithmic structure of the program, types of the errors, reasonable error causes, and gives advices for fixing them. These messages are displayed in the right hand side of the screen. If the student wants to edit his program according to the advices by the ALPUS system he can switch the session back to the GUIDE session. Additional advices and expressions are given by the TUTOR system.

OUTLINE OF TUTOR SYSTEM

The TUTOR subsystem receives informations from ALPUS, creates a student model, and teaches a user appropriate programming knowledge based on the student model. By program comprehension functions of ALPUS the TUTOR subsystem is able to obtain a variety of data about the user as initial informations to create the student model, which include what he knows, what he does not know, what he knows improperly, and so on. In usual ITS systems a lot of human-computer interactions are required to create a student model. These interactions are time-consuming and uncomfortable for the user, therefore should be minimized. We think that heavy load of interactions is one of serious problems in ITS systems. This seems to be one of major reasons why such systems have not been widely used in actual teaching domains. In the INTELLITUTOR trials we would like to demonstrate the benefit of the program comprehension functions to minimize such interactions.

TUTOR has a skeleton of structured programming knowledge for building a student model. The ideas of overlay model and buggy model are used for student modelling. Model generation is achieved by marking simple information onto associated links of the skeleton by three steps. Marks include such as "understand" and "do-not-understand" for algorithm-related knowledge, and "know", "do-not-know" and "err" for technique-related knowledge.



===>Error occurs in ExchangeProcess(line17-18)

Content of the error

You might try to exchange two values by these assignment statements. However, it is imposible to execute two statements at once by computer. Therefore, prior to exchnging you have to store one of the velues in a working variable. Then you can do it.

Causes of the error are supposed as follows:

*You supposed to be able to exchange at once.

*You did not know the technique of a working variable.

Fig.6 An example of advices produced by the ALPUS system. Two logical bugs were detected (the second advice was transrated into English).

Transactions on Information and Communications Technologies vol 3, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

At first, according to the informations received from the ALPUS system initial marking is done. Scope of the student model is decided based on the information as well. That is, the skeleton is defined to cover very wide range of programming knowledge for generality. While, the Quicksort program needs a subset of it. By setting the scope the range of knowledge becomes clear, which results in reasonable interactions and teaching. In order to identify reasonable causes of bugs we developed an advanced student modelling method which was re-composition of a mental process of bug generations in programming[11]. The re-composition of the mental process was considered from the point of view of intentions in novice's programming tasks.

Programming tasks seem to be achieved by novices in the following processes: Programming starts when a problem to be solved is given, he creates the procedure or chooses the algorithm to solve it next, then he assigns reasonable programming techniques to implement it, and he writes a program in a programming language. Within these processes he seems to consider *intentions* first, which would be decomposed next. This cycle is repeated so that the top level intention is decomposed to a combination of detailed processes to implement it. By re-composition of mental processes we are able to understand a programmer more precisely as a mental model. In other words, causes of bugs could be identified more precisely such like: a cause must be due to mis-understandings of an algorithm, improper knowledge of programming techniques, simply careless mistakes, or combinations of them. It should be stressed that the errors are caused by both improper intentions and improper implementations of proper intentions.

Next, according to the initial marks additional marking is made by using the knowledge which represents causal relationships among parts of the model. The causalities are represented in such as IF-THEN rules and causal links. If there exists unmarked links which are related to "unknown" or "buggy" knowledge within the scope, interactions are requested by the system to the student at the third step. Then, teaching of programming knowledge is made according to the student model.

DISCUSSIONS

The INTELLITUTOR system was designed as an integrated intelligent programming environment for novices. By integration we mean that editor, semantic checker and ITS modules are combined as a total programming environment. The integration is one of issues in this system. While, in actual use each module is useful as programming and/or learning aids for novices. For example, GUIDE can be used as a comfortable programming aid. This subsystem has the hyper-text data base in which associated informations are linked together. The data base includes hierarchically structured text files of an introductory text book on PASCAL, PASCAL program examples and the PASCAL language specifications. These informations are retrieved and displayed during editing a program by user's request. By combining GUIDE and ALPUS programming one can write a correct program more smoothly.

HPG works very well to comprehend complex programs such as Quicksort programs. Definition of new HPGs for other algorithms is relatively easy using the HPG definition aid of the system. However, appending new knowledge about bugs is time consuming since cognitive experiments are required to obtain such informations from novices. By separating the programming knowledge base to sub-knowledge bases which are the knowledge base about algorithms, about techniques, about data handling operations and about programming languages the maintainability and flexibility of the knowledge was improved.

The HPG-oriented knowledge representation schema seems to be applicable to other domains such as maintenance of programming knowledge, program rewriting between different programming languages, automatic error corrections, program reuse, program generation, Reverse Engineering and so on. We hope that in the future the shared programming knowledge base will be developed and used for a variety of applications.

We have examined the ability of program understanding of ALPUS by experiments on intermediate class students of the Department of Systems Engineering, Tokyo Denki University. The experiment was done in the following way. At first, the outline of Quicksort and Straight-sort algorithms were taught to the students by means of a-blackboard-and-chalk method. After the lecture they wrote PASCAL programs on coding sheets based on their understandings. The student programs were entered to the system by project members and examined by ALPUS.

In case of Quicksort algorithm 54 programs out of 78 were accepted as syntactically correct ones. 37 programs(68.6%) were understood correctly by ALPUS by means of single typical HPG graph. Correctness ratio may grow up to 87.0%(47 programs) by using six HPG sub-graphs, since there are five methods to achieve the division of data array based on the concept of the Quicksort algorithm. In case of Straight-sort algorithm 71 programs out of 80 were syntactically correct. 55 programs(77.5%) out of 71 were understood correctly by means of single HPG graph. Current ALPUS has limitations on the number of knowledge about sub-HPGs and buggy patterns and the ability of standardizations and segmentations. We are trying to extend the flexibility and ability of ALPUS by improving these functions.

ACKNOWLEDGEMENT

This work was supported in part by The Science Research Foundation of Ministry of Education, NTT Software Institute and Research Institute for Technology, Tokyo Denki University. The authors would like to thank project members of INTELLITUTOR at Ueno laboratory for system implementations and discussions.

References

1. Adam, A. and Laurent, J. P., "LAURA: A System to Debug Student Programs", Artificial Intelligence 15, pp.72-122, 1980
2. Ito, H. and Ueno, H., "ZERO: Frame + Prolog", Logic Programming '85, Lecture Notes in Computer Science, Vol. 221, pp.10-17, Springer 1986
3. Jonson, W. E. and Soloway, E., "PROUST: Knowledge-Based Program Understanding", IEEE Trans. on Soft. Eng., Vol.SE-11, No.3, pp.11-19, 1985
4. Ruth, G. R., "Intelligent Program Analysis", Artificial Intelligence 7, pp.431-441, 1976
5. Soloway, E. and Ehrich, K., "Empirical Studies of Programming Knowledge", IEEE Trans. on Soft. Eng., Vol.SE-10, No.5, pp.595-609, 1984
6. Ueno, H., "INTELLITUTOR: A Knowledge Based Intelligent Programming Environment for Novice Programmers", Proc. IEEE COMPCON89 Spring, pp.390-395, 1989
7. Ueno, H., "ALPUS: A Program Understanding System by Means of Algorithm-Based Programming Knowledge", Proc. PRICAI90, pp.693-698, 1990
8. Ueno, H., "An Integrated Knowledge-Based Intelligent Programming Environment for Novice Programmers", Proc. IEEE COMPSAC 91, pp. 124-129, 1991
9. Ueno, H., "A Knowledge-Based Program Understanding for Intelligent Programming Environment for Novice Programmers", Proc. bit'89, pp.42-55, 1989
10. Ueno, H.: Knowledge Based Intelligent Programming Environments - From the Point of View of Program Comprehension, Information Processing, Vol.28, No.10, pp.1280-1296, 1987(in Japanese)
11. Mizoguchi, M.: Learner modelling Method by Means of Recomposition of Intentions for Programs by Intermediate Students, ME thesis, College of Science and Engineering, Tokyo Denki University, 1993