Jurnal Antarabangsa (Teknologi Maklumat) 3(2002): 73-87

Conceiver: Not Just Another Program Understanding System

ABDULLAH MOHD ZIN & HANI AHMAD AL-OMARI

ABSTRACT

A number of program understanding systems have been developed over the past decades. Although the development of these systems have solved many issues in program understanding, they are not widely used. In most cases, these systems are only being used by their developers. At UKM, we have embarked into developing a program understanding system that is "usable", called CONCEIVER. The design and implementation of CONCEIVER is described in this paper.

ABSTRAK

Beberapa sistem pemahaman aturcara telah dibangunkan dalam beberapa dekad yang lepas. Walaupun pembangunan sistem ini telah menyelesaikan banyak isu yang berkaitan dengan pemahaman aturcara, kebanyakan sistem ini tidak digunakan dengan meluas. Dalam kebanyakan kes, sistem ini hanya digunakan oleh pembangun sistem tersebut sahaja. Di UKM, kami telah memulakan usaha untuk membangunkan satu sistem pemahaman aturcara yang "boleh diguna", yang dikenali sebagai CONCEIVER. Kertas ini membincangkan rekabentuk dan implementasi CONCEIVER.

INTRODUCTION

Research in the process of understanding a program is motivated by two different reasons. One group of researchers are from the cognitive psychology group and their main interest is to find out exactly what knowledge and understanding expert programmers have that novices don't. Another group of researchers are from the software maintenance group whose main aim is to build up a system which can automatically identify and repair errors in a program.

The first known research in program understanding was made by Halstead (Halstead 1977). In his theory of software science, Halstead argued that algorithms have measurable characteristics similar to physical laws. He further argued that useful measures could be derived from a simple count of distinct operators and operands.

Most of researchers working in this area, however, disagree that the Halstead model is an accurate representation of mental processes for an experienced programmer (Curtis et al. 1984). They do not believe that experienced programmers work at the level of operators and operands when they are developing a program. They argue that programmer's thoughts represent chunks of operators and operands at least at the level of an expression and often at a level where numerous expressions have been compiled to form a function.

Many mental models have been produced to represent the mental processes of a programmer. These models fall under one of two categories. The first category views the understanding of programs as a successive modification of a hypothesis in the programmer's mind. The second category views program understanding as identifying stereotyped fragments of code and relating them to each other.

In the 1970s and 1980s, most of program understanding systems were developed based on the first category of mental models. Some of the systems in this category are Intelligent Program Analysis (Ruth 1974), PROUST (Johnson and Soloway 1985) and PUDSY (Lukey 1980). However, these systems were oriented toward specific domains because these systems need descriptions of the goals in order to operate. Due to this reason, most of these systems were developed in a learning environment, for example to support students in the process of understanding and debugging programs.

By the late 1980s and 1990s, the direction of research shifted towards the second category of mental models. Systems developed based on this category have been demonstrated to have a more general understanding capability and thus can be used to software maintainer to understand real life software. Some of the systems which are based on this mental model are Recognizer (Wills 1987), BAL/SRW (Kozaczynski 1991) and Program Analysis Tool (Harandi and Ning 1990).

Although the development of these systems have solved many issues in program recognition and knowledge representations, they are still in the experimental stage (O'Hare and Troan 1994). Unlike other programming tools, these systems are still not being used widely. In most cases, these systems are only being used by the developers of the systems. In order to address this problem, we have embarked on developing a program understanding system which would be "usable", called CONCEIVER

Characteristics of A Usable Program Understanding System

Following McCall et al. (1977), we define "usability" as the effort required by a user to learn, operate, prepare input and interpret the output of a program. The more effort required the less "usable" the program is.

There are two different types of users for a program understanding system. The first type of user is the end-user, who uses the system in order to understand a particular piece of code. The second type of user is the knowledge engineer, who is responsible for codifying stereotyped fragments of code into the knowledge base. In the area of program understanding, the stereotyped fragments of code is normally called as "plan" and the knowledge base that contains these plans is called the "plan base". The relationship between the system, knowledge engineer and end-user is shown

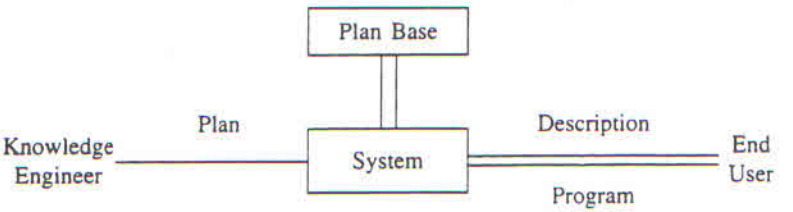

FIGURE 1. Users of the system

in Figure 1.

Most program understanding systems have provided support for both types of user.

A. CODE RECOGNITION

The first characteristic of a "usable" program understanding system is that it must provide support for recognizing program code. The system will process the program code that is submitted by the end-user and generate documentation that describes the functionality of the program. Some systems have even provided support for recognizing at least some parts of the program if full recognition is not possible, recognizing overlapped implementation and recognizing non-localized code.

B. PLAN BASE MANAGEMENT

For knowledge engineers, most of the systems have provided support to enable knowledge to be entered into the plan base. A knowledge engineer writes plans based on the specifications of the programs and submits them to the system. The system will check the correctness of the plan. If the plan is correct, the system will generate a conceptual representation of the plan and store it in the plan base.

Ideally, a program understanding system should be a general-purpose system. Thus, the system must be equipped with enough plans in its plan base so that it is capable of recognizing all types of programs. However, this is an

impossible hope as new programs are written every day to solve new problems. Even if this is possible, the plan base will be very large and thus reduce the efficiency of the system.

A more "usable" and "practical" program understanding system must operate within a specific domain. The knowledge engineer should be informed about the domain so that he/she can equip the system with necessary plans in the plan base. Thus, for a program understanding system to be usable, it must provide all necessary support for the knowledge engineer to enter, update and organize plans in the plan base according to the domain in which it is suppose to operate. Although this is recognized as an important feature of a program understanding system, so far it has not been incorporated in any of the available systems (Wills 1987). Without this feature, the acquisition of new plans will be difficult and hence limit the usability of the system.

C. MINIMUM AMOUNT OF KNOWLEDGE

The real strength of a knowledge-based system is its ability to deduce information based on a minimum amount of knowledge. A program understanding system, for example, needs to identify program code that may occur in many different forms. However, it is not possible for the system to store all of these different forms in the plan base. Thus, the system must b able to understand a program if one form of the program code is available in the plan base. In order to enable the system to understand a program code that appears in a different form, some methods of transformation need to be done.

D. PROGRAMMING LANGUAGE INDEPENDENT

Programs can be written using many different programming languages. Thus, another important characteristics of a "usable" program understanding system is programming language independence, that is, the system should be capable of recognizing codes regardless of the programming language that is used for writing the program.

KNOWLEDGE REPRESENTATION

The first step in designing a "usable" program understanding system is the design of a conceptual model to represent programs and algorithmic cliches in the knowledge base. The design of the plan base must be done properly so as to enable the system to possess the characteristics that have been explained earlier. To support programming language independence, the plan must be represented by a language independent formalism. To allow for transformation, this formalism must be flexible and easily manipulated.

Rich (1981) has proposed the concept of plan calculus as a formal representation of program and algorithmic cliches. Software plans are hierarchical descriptions of computations. Each level of a plan is defined by a set of parts, connections and constraints. In the plan calculus, the parts are operations, tests or data. Connections are expressed by control and data flow. Constraints in the plan calculus include the preconditions and post-conditions of operations and test, and the invariants of data representation.

The plan formalism adopted in this work is greatly influenced by the plan formalism used by Kozacynski (1992) with some extensions and modifications. The plan formalism adopted by Kozacynski is greatly influenced by the plan calculus. However, Kozacynski's formalism is easier to use, highly compact and easy to expand. In this formalism, a plan is divided into three parts: *header*, *components* and *constraints*. The *header* contains the plan name and its parameters. The *components* part contains the body of the plan. The *constraints* part contains the data and control flow relations between the plan's components.

One main weakness of Kozacynski's formalism is that the expression of plans at the statement level is language dependent. Introducing a new programming language implies the addition of hundreds of new plans to represent the knowledge about statements in that particular language. In our formalism, this weakness is eliminated by introducing the concept of a basic plan. Basic plans are plans that deal directly with the parse tree of the program. Thus, basic plans can be viewed as a mapping interface between a programming language and the bottom levels of the plan base. Some of the basic plans defined in CONCEIVER are shown in Table 1.

TABLE 1. Some Basic Plans

Plan Number	Plan Name	Function - To recognize
1	Plus_Operator	'+' operator
2	Minus_Operator	'-' operator
3	Multiply_Operator	'*' operator
4	Divide_Operator	'/' operator
5	Equal_Operator	'=' operator
6	Greater_Operator	'>' operator
7	Less_Operator	'<' operator
8	And_Operator	'AND' operator
9	Int_Divide_Operator	'DIV' operator
10	Greater_Or_Equal Operator	'>=' operator

An example of a plan is shown in Figure 2. A plan must start with a unique number. Following the plan number is the plan name. A set of plans that are supposed to perform the same computation must use the same name. After the plan name, the plan parameters (if any) should be expressed. The

declaration of parameters is done in the data structure part. Notice that more than one data type can be included which implies that the parameter can be used for more than one data type. The body of the plan comes after the reserved word "Components".

```
101 Assign_a_Constant (Var1, $Const)
Data_Structure
     Var1: Integer, Real, Char;
     $Const: Integer, Real, Char
Components
     Operation1: Variable(Var1)
     Operation2: Assign_Operator
     Operation3: Constant($Const)
Constraint
     Root(Operation2)
     LeftChild(Operation1)
     RightChild(Operation3)
```

FIGURE 2. Assign_A_Constant plan

SYSTEM ARCHITECTURE

The system architecture for CONCEIVER is shown in Figure 3. The whole system consists of five components: a user interface, infrastructure tools, an understanding inference, a document generator and a plan base. Infrastructure tools consists of one or more parsers and a transformer.

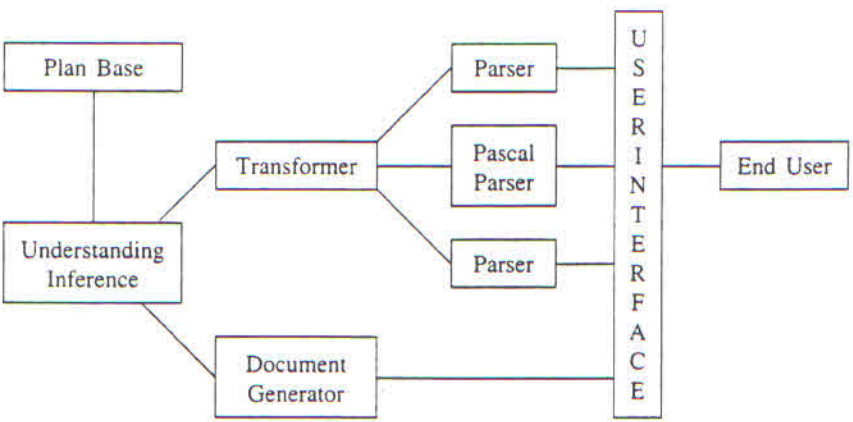

FIGURE 3. The Architecture of CONCEIVER: End-user perspective

A. PARSER

The main role of a parser is to translate a source program into a language-independent representation that is used by the plan base. The translation

process can be done by providing the appropriate translator. For example, to understand a Pascal program, a Pascal translator need to be provided.

Besides checking the syntactic correctness of the program, the translator performs two transformation tasks: expression simplification and programming constructs normalization.

As we have mentioned earlier, program code can be represented in many different forms. One of the reasons for the existence of many different forms is because an expression can be expressed in many different ways. Expressions (9 * 2), (9 + 9), (20 - 2), (90/5) are all equivalent. Since it is not possible to store all of these similar expressions in the plan base, there is a need for these operations to be reduced to a normal form. The normalization of expressions involves four types of activities: evaluating constant expressions, simplification of algebraic expressions, term rearrangement and reducing Boolean operators.

Another reason for the existence of these different forms is the variety of similar programming constructs that are provided by programming languages. Pascal, for example, provides two constructs for selection: the if-then-else statement and the CASE statements, and three constructs for loops: the for statement, the while statement and the repeat statement. To reduce the need for storing too many plans in the plan base, all of these constructs are also normalized to the normal form.

The parser produces three outputs: a parse tree, a flow graph and a data flow analysis. A parse tree is normally used as an intermediate representation of program code. Compilers normally generate the object code by first generating the parse three. Since a tree structure can easily be restructured, it is the most suitable representation for producing optimised object code. A variation of a parse tree is called a syntax tree. A syntax tree is a tree in which each leaf represents an operand and each interior node represents an operator. A parse tree generator in CONCEIVER produces the syntax tree for each of the program statements. The output from this parse tree generation is a forest of syntax trees.

A flow graph that is normally produced by a compiler assumes that each basic block of the program is considered to be a node. A basic block is a sequence of consecutive statements in which flow of control enter at the beginning and leaves at the end without halting or the possibility of branching in between. For our purpose, each node of the flow graph is a program statement instead of a program basic block since the use of a basic block is not suitable for recognizing non-localized code. Each node in the flow graph is connected to the parse tree that i represents. For example, consider the program in Figure 4(a). During the parsing of this program, some normalization has been performed on its construct, particularly on the FOR loop. The normalized form is shown in Figure 4(b).

(a)
```
Program Factorial(Input,Output);
const
    N = 10;
var
    i, fact : integer;
begin
    fact := 1;
    for i := 2 to N do
            fact := fact * i;
end.
```

(b)
```
Program Factorial(Input,Output);
const
    N = 10;
var
    i, fact : integer;
begin
    fact := 1;
    i := 2;
loop:   if (i <= N) do begin
        fact := fact * i;
        i := i + 1;
        goto loop;
    end;
end.
```

FIGURE 4. (a) A simple program to compute factorial
(b) The normalized program

The flow graph for the program is shown in Figure 5.

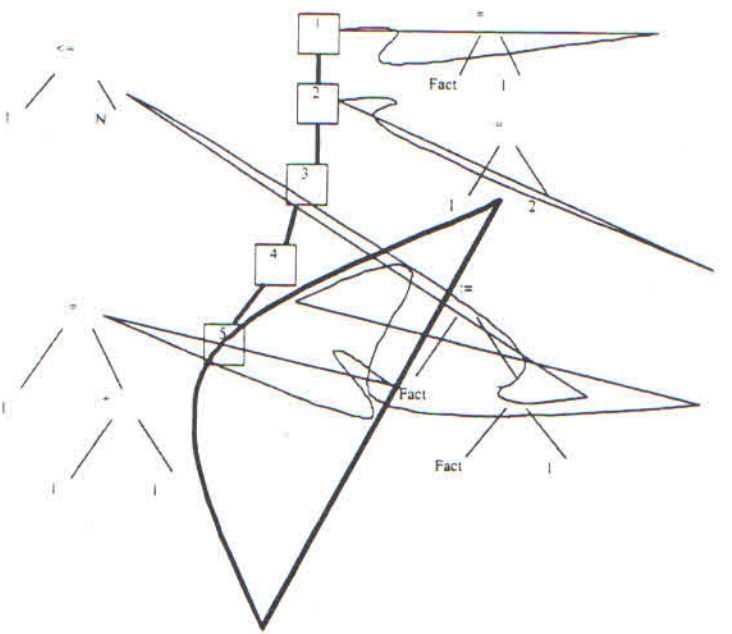

FIGURE 5. Flow graph for factorial

Each node in the flow graph is associated with two pieces of information: variables generated by the node and variables that are used in the node. Figure 6 shows the data flow information which is associated with the flow graph shown in Figure 5.

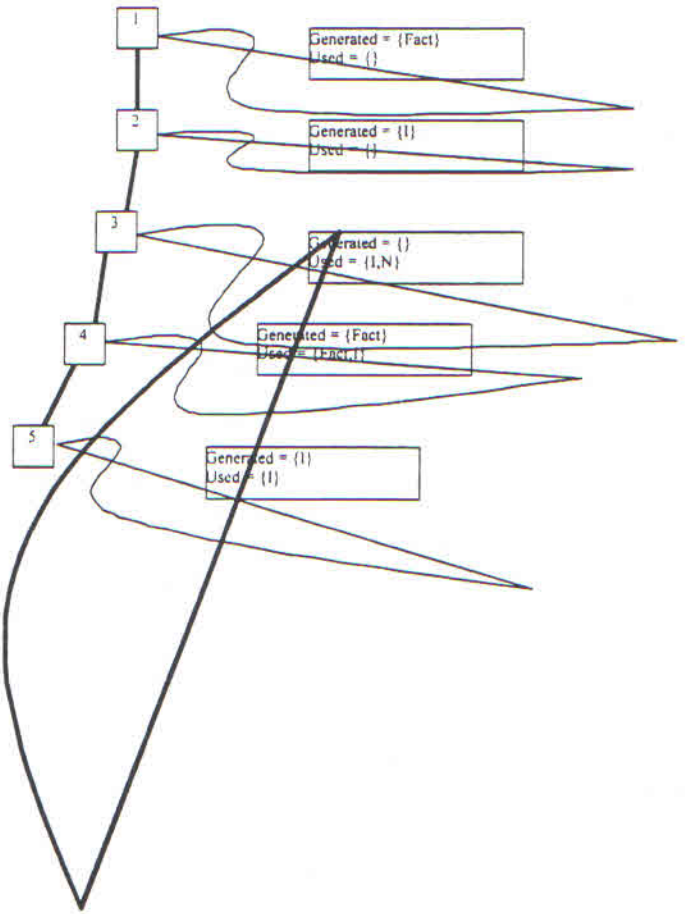

FIGURE 6. Data flow information associated with flow graph in Figure 5

B. TRANSFORMER

After the program has been translated into a language-independent representation, it may need to undergo some additional transformation in order to make it easier for the process of understanding the program. These transformation does not affect the semantics of the program.

One area where transformation is especially needed is the case of assignment statements, which may be written in many different forms. The role of transformer in this case is to simplify the statement without affecting the semantics of the program. In some cases, some of the statements need to be deleted. For example, if a program contains statements such as

```
X : = 6;
Y : = X + 2;
```

the transformer will change them into a single statement

```
Y : = 8;
```

The first statement is deleted from the program (unless X is used elsewhere in the program).

C. UNDERSTANDING INFERENCE

The understanding inference performs the process of unifying the program code against the plan base. The unification process starts at the low level of the program code. First operators and operands are recognized. The operators and operands are combined to form a statement. After recognizing an individual statement, CONCEIVER will attempt to find all valid chunks and relates them together by unifying the code against plans in the plan base. The output from the understanding inference is called a recognition hierarchy.

D. DOCUMENTATION GENERATOR

After the program code is understood, the documentation generator will be invoked. The documentation generator uses the understanding hierarchy to generate natural language description about the function of the program.

KNOWLEDGE ACQUISITION AND ORGANIZATION

The most important contribution of CONCEIVER and the one that distinguishes it from other program understanding systems is the plan editor that provides facilities for a knowledge engineer to acquire and organize plans in the plan base. By using the plan editor, a knowledge engineer could acquire plans, either via the plan preparation tool or via the automatic acquisition of a plan. The structure of the plan editor is shown in Figure 7.

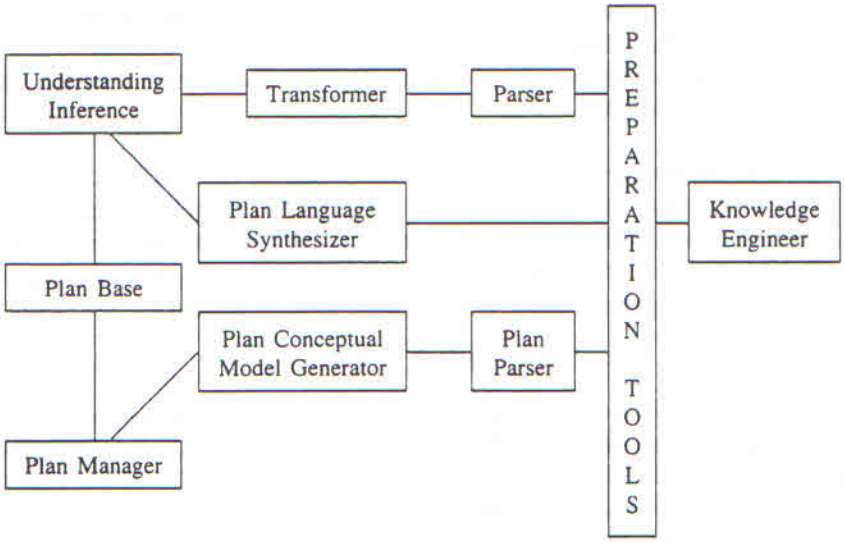

FIGURE 7. The architecture of the plan editor

A. PLAN PREPARATION TOOL

The plan preparation tool allows a knowledge engineer to input plans into the plan base. It consists of four components: Data Structure Editor, Basic Plan Editor, Plan Editing Pads and the Implementation Viewer.

The data structure part of a plan contains the variables used in the plan. Each of these variables is tied to one or more data types. In order to ensure language independent of the plan formalism, data types can be defined by the user. The data types provided by a language are fed to the plan editor by placing them in a file. For example, Pascal provides eleven data types: Array, Boolean, Char, Integer, Real, Record, File, Set, Sub-range, String and Pointer. These language-defined data types are placed in a file named 'PASCAL.TYP'.

Basic plans are plans that cannot be broken into smaller plans. Their components can be obtained directly from the parse tree. Basic plans main role is to act as the interface between program codes and plans.

Plan editing pads are collections of editors which can be used to write, to browse and edit plans. Each part of the plan is contained in a separate editing pad. The plan has been divided into six parts:

- The plan's initial information editing pad contains three editing controls that are used to specify the plan number, name and formal parameter.
- The plan's data structure editing pad contains an editor for the plan writer to specify the plan's data structure.

- The plan's component editing pad allows the plan writer to specify the plan's components.
- The plan's constraint editing pad allows the plan writer to specify the plan's constraints.

The implementation viewer allows plans to be viewed from different perspectives. For example, plans can be browsed in the order they were originally stored, and they can be viewed based on their implementations. Since a plan can have more than one implementation, this perspective gives a good cross-reference view of the content of the plan base.

B. PLAN EDITOR SUPPORT TOOLS

Apart from editing, writing plan also involves other activities such as parsing and generating the plan conceptual representation. To enable these activities to be carried out, the plan editor is supported by three other tools: the plan parser, the plan conceptual representation generator and the plan manager.

The plan parser verifies the syntactic correctness of plans. The techniques that is used to build the parser is similar to the techniques used to construct any programming language parser.

The plan conceptual representation generator is implemented as an extension to the parser. The objective of this generator is to generate the representation that will be used during the process of unification.

Adding, deleting and modifying plans are major activities in the process of program understanding. Most program understanding systems collect and verify their plan in one-batch. Thus, in case a plan need to be changed, it will affect all plan in the hierarchy. Simple alteration of a plan might require a week of intensive work. The plan base manager is a tool that can restructure the hierarchy of the plan base if any plan in the plan base is changed. For example, if a plan is deleted from the plan base, all links to the plan will be destroyed.

C. AUTOMATIC ACQUISITION OF PLANS

The plan preparation tool has provided a mechanism that enables a knowledge engineer to enter and organize plans into the plan base. However, writing plans is a tedious task. A knowledge engineer has to be familiar with the language that is used in representing a plan. He must also find suitable constraints to relate a component of a plan to other components.

CONCEIVER's plan editor provides another mechanism to help a knowledge engineer to acquire plans. Instead of writing a plan in the plan language, a knowledge engineer writes a plan as program code. The CONCEIVER's Automatic Acquisition of Plans mechanism will then convert the program code into plans. The availability of this mechanism helps reduce the time that is needed to construct a plan.

The process of acquiring a plan from source code starts by obtaining the recognition hierarchy of the code. After that, a special tool called the Plan Language Synthesizer constructs the plan from the recognition hierarchy. To ensure the correctness of the plan, the constructed plan will then be presented to the knowledge engineer to be finalized before it is stored in the plan base. The knowledge engineer can edit the plan by using the plan preparation tool.

Since a plan consists of three parts, the generation of a plan by the Plan Language Synthesizer is done in three stages. The first stage is to synthesize the data structure part by obtaining the data types for each of the variables in the program. The second stage is to synthesize the components part. The final stage is to obtain the plan's constraint part by finding all possible constraints for the components of the plan.

User Interface

CONCEIVER'S user interface was implemented as a Multiple Document Interface (MDI) as shown in Figure 8. The user can select the command by using the pull-down menu. However, to make it easier for users of the system, commonly used commands are provided by using buttons. Some of the buttons are described in this section.

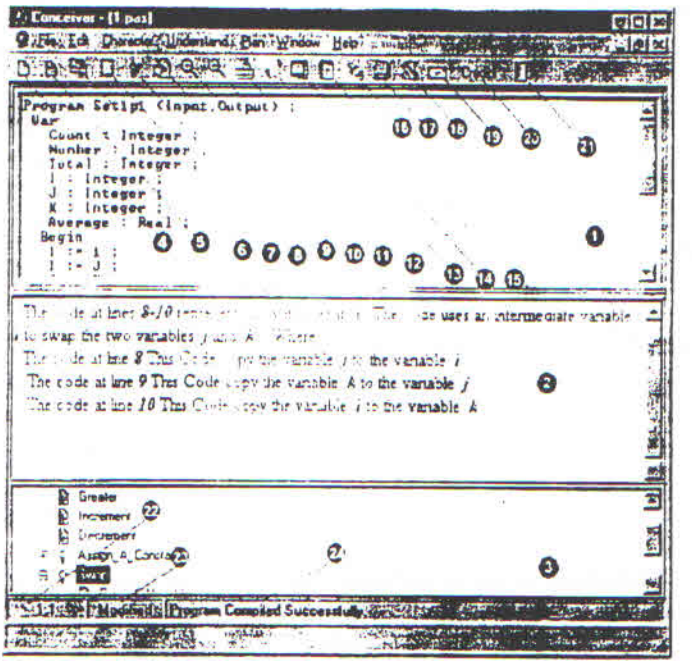

FIGURE 8. The main window of CONCEIVER

Each edited program has two editors, one for original program (labelled 1) and the second for the transformed program (not shown in Figure 6). These two editors are overlapped. The user can switch from the original program to the transformed program by clicking button labelled 13. The document viewer (labelled 2) is used to display the generated documentation about the program. The area that is labelled 3 shows the hierarchy viewer that enable users to view the hierarchy of the recognized plans.

Buttons 4, 5, and 6 are used to create a new file, open an existing file and save a file, respectively. Button 7 activates the translator. Since transformation is not compulsory, no special button is allocated for it. The user can invoke the transformer by choosing the option *Invoke Transformer* for the *Understand* menu.

Button 8 activates the understanding inference. This inference starts at the leaves of the parse tree of each flow graph node.

Some of the buttons are provided for the knowledge engineer to input plans into the plan base. The plan editor, basic plan editor, data structure editor and the implementation viewer can be activated by clicking buttons 14, 15, 16 and 17 respectively. Button 18 activates the plan automatic acquisition system.

Three status panels are shown at the bottom of the main window (labelled 22,23,24). These panels are used to display the cursor position, the file statues and compilation errors.

CONCLUSIONS

The main aim of CONCEIVER is to be a "usable" program understanding system. We have identified that such a system must possess the following characteristics: providing support for recognition of program code, providing support for entering new plans into the plan base, ability to identify program code that may occur in many different forms and ability to recognize code regardless of the programming language used for implementation.

In this paper, we have described the design and implementation of CONCEIVER and have given the justification to support the claim that CONCIEVER does possess the listed characteristics and thus may be considered to be a "usable" program understanding system.

Apart from its recognizing capabilities, another important contribution of CONCEIVER is the plan editor, which allows a knowledge engineer to acquire and organize plans in the plan base. The availability of the automatic plan recognition tool is very important to enable CONCEIVER to be used to recognize programs based on different specifications. This tool allows a user to add plans to the knowledge base simply by giving the programs, and thus makes it very easy for him/her to add plans to suit different specifications being tested.

A few experiments of using CONCEIVER to recognize real programs have been conducted with promising results (Abdullah et al 2003). However, further work is currently being undertaken to enhance the capability of CONCEIVER and to correct some errors that are still present in the system.

REFERENCES

Aho, A. V., Sethi, R. & Ullman, J. D. 1986. *Compilers principles, techniques, and tools*. USA: Addison- Wesley .

Curtis, B., Fonnan, I., Brooks, R., Soloway, E. & Ehrlich, K. 1984. Psychological perspectives for software science. *Information processing and management*, 20:81-96.

Halested, M. E. 1977. *Elements of software science*. New Yark: Elsevier .

Harandi, M. T. & Ning, J. Q. 1990. Knowledge based program analysis. *Software* 7(1):74-81.

Johnson, W. L. & Soloway, E. 1985. PROUST: knowledge based program understanding. *IEEE transactions on software engineering* 11(3).

Kozaczynski, W., Liongosari, E. S. & Ning, J. Q. 1991. BAL/SRW: Assembler re- engineering workbench. *Information and software technology*, 33(9): 675-684.

Kozaczynski, W., Ning, J. & Engberts, A. 1992. Program concept recognition and transfonnation. *IEEE transactions on software engineering*. 18(12).

McCall, J. P. Richards and G. Walters. 1977. Factors in Software Quality (3 Volumes). NTIS AD-A049-015,015,055.

Rich, C. 1981. A formal representation for plans in the Programmer's Apprentice. *Proc. Of The tth International Joint Conference On Artificial Intelligence*: 1044-1052.

Ruth, G. R. 1974, *Analysis of algorithm implementations*, Technical Report #130, MIT, Project Mac, MA.

Wills, L. M. 1987. *Automated program recognition*. Technical Report #904, MIT, Artificial Intelligence lab.

Abdullah Mohd Zin, Hani Mohd Ahmad al-Omari and Syed Ahmad al-Junid. 2003. *An Experiment of Using a Program Understanding System In Learning Environment*. Technical Report, Computer Science Dept, Universiti Kebangsaan Malaysia.

MAKLUMAT PENGARANG

Abdullah Mohd Zin & Hani Ahmad Al-Omari
Programming Research Group
Faculty of Information Science and Technology
Universiti Kebangsaan Malaysia
43600 Bangi, Selangor, Malaysia
amz@ftsm.ukm.my