Journal of Computer Science 5 (12): 1009-1019, 2009
ISSN 1549-3636
© 2009 Science Publications

Implementation of CONCEIVER++: An Object-Oriented Program Understanding System

[1]Nor Fazlida Mohd Sani, [2]Abdullah Mohd. Zin and [2]Sufian Idris
[1]Faculty of Computer Science and Information Technology,
[2]Faculty of Science and Information Technology,
University Kebangsaan Malaysia, Malaysia

Abstract: Problem statement: Understanding on computer program is a complex cognitive activity. It is ability and also a difficult task especially for novice programmer. The object-oriented languages has widely used in education and industry recently. In programming it is important to have such software which can aid programmers or students to code the program. But, available program understanding systems using the plan based approach usually are developed for non-object-oriented programming languages. Reviewed from the available system also showed that none of the plan formalisms used is for an object-oriented language. Specifically, problem arises when the existing system is not usable for teaching programming purposes. Program understanding system with plan for object-oriented does not exist was the main reason why this research is being carried out. **Approach:** Method used on developed the program understanding system named CONCEIVER++ is Unified Approach (UA). The process involved from UA for developing and testing the system is iterative development and continuous testing. The process must be iterate and reiterate until satisfied with the system. In order to test the quality assurance of the system is by choosing the black box testing strategies. **Results:** The object-oriented program understanding system has been successfully implemented. The implementation is tested with an example of Java programming code. The binary search tree for control flow graph and linked list for plan has been generated. Results of understanding the meaning or semantic of the program codes also has been produced. The black box testing had shows that all statements of line of code of the example program have been recognized and the correctness output has been checked. **Conclusion:** The understanding module of CONCEIVER++, which are code/CFG processor, plan processor and recognition engine has been tested. All line of codes (or nodes) has been recognized and got correct meaning using the developed module.

Key words: Program understanding, implementation, plan base, Control Flow Graph (CFG), meaning, testing

INTRODUCTION

Program understanding is an activity that enables to know the meaning or semantic of programming codes. It is an important activity in maintaining a system, debugging a programming code and as one of activity in reverse engineering process. It also is the intermediate skill to programmers, suggested by Romero[1]. Understanding on computer program is a complex cognitive activity. Therefore, realization of a system is very beneficial to novice and experience programmers. Those who involve in programming activities which is difficult are the programmers. Knowledge and experienced of programmer in programming covers writing capability, reading and understanding of a program code. Understanding of a program code is ability and also a difficult task especially for novice programmer. The important skill for any programmers to be developed is the ability to read an available program code which being coded by other programmers[2].

Research in program understanding is still being study until now and the common approach used for supporting program understanding is with completing the program code through abstraction[1]. The purpose of this study is to present the implementation of object-oriented program understanding system, CONCEIVER++ which is using the abstraction

Corresponding Author: Nor Fazlida Mohd Sani, Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 UPM, Serdang, Selangor, Malaysia,
Tel: +603-89466550 Fax: +603-89466577

approach. In this study will explain on the approach and present the detail processes involved in producing the meaning of statements.

Related works:

Abstraction approach: Abstraction is understanding approach directly on the source code or system that to be comprehend. Intermediate representation is always use with abstraction for the purpose of recognition of programming code. In abstraction approach, plan based also used to save all knowledge's plan for that programs domain. A recognition plan algorithm will be used to match the intermediate parts with plan from the plan based to give understanding or meaning for certain program codes. The advantages of this approach are easier to organize user-defined classes or objects and also to differentiate objects names with methods names. This is important to organize definitions so that the related information can be grouped together in common locations[3]. Furthermore, abstraction can help to reduced complexity and minimizing the numbers of details in certain program codes[4], also helps a lot in the process of understanding purposes. Another concrete reasons using abstraction is the reliability of the understanding or inference result is true based on the source program[5]. Most of the program understanding algorithm with this approach were using library of programming plan with multi-heuristics strategies to find the existence of plans in the source program. This statement has emphasized in former researches such as in[6-9].

Plan base and formalism: Plan base is the important component for any program understanding system, usually with abstraction approach. This is because of the plan base is the library of inference knowledge for each program code that will be identified by the system. The terms 'plan' for program analysis research literature, is used for referring to different subjects such as: (1) Abstract representation for fragment of code; (2) Programming heuristics; (3) Programming abstraction concept; and (4) Knowledge to identify programming concept[10].

According to Wills[9], an experienced programmer is keep on redeveloping lots of hierarchy for program design by recognizing from the data structure and algorithm which is commonly used and typically know how to do the higher level abstraction. The common computerize structure that being used called as cliché. Cliché is a frequently appears pattern in program codes, such as algorithm, data structure or pattern specific domains. Plan is the representation of cliché. The objective of plan recognition is identifying cliché by using the plan. There were three approach of plan recognition, which is top-down, bottom-up and hybrid which combining the top-down and bottom-up approach[11].

Plan that use for the understanding purposes must be formatted into a standard format. The standard form is vital so the derived plan formed in the same format and easily accessed using specified recognition algorithm. Plan formalism is the language design that being used for creating plans. This formalism must be designed to ensure each plan that will be created is in even formed. There are several plan formalisms used by previous researchers. One of is a Plan Calculus in Programmer's Apprentice system is for Lisp language[12]. There was a plan formalism to recognize COBOL programming language by[13]. This formalism is use in development of program understanding system named concept recognizer.

Review from the available system had shows that none of the plan formalisms used is for an object-oriented language. Therefore in this study, based on the Kozaczynski's plan formalism, specific plan formalism for object-oriented language will be developed by relating the results of understanding and debugging. The chosen of Kozaczynski's plan formalism is because of the capability for representing knowledge of programming code with two concepts, language and abstract concepts. For debugging purposes, each common error will be detected driven by the absence of matched plan for certain code statement, results from the recognition or understanding engine. The results of recognition will be displays in details for every node of control flow graph which represent the recognized programming codes. The control flow graph is the intermediate representation of programming codes that is use in CONCEIVER++. The detailed explanation of control flow graph has been presented in Sani *et al*.[14].

Plan base for CONCEIVER++ stored all plans that being retrieved by recognition engine. Plans in CONCEIVER++ contain knowledge for understanding and debugging. All the available system which use plan-based recognition approach, not integrate the debugging knowledge together with understanding knowledge in their plan formalism, except for PAT[10] and GRASPR[12]. Plan for PAT system has a knowledge to identify bugs that relates to the program code. PAT's plan represents the logical of algorithm, while plan developed in CONCEIVER++ represents the common form of statement of code that being used several times. Near-miss cliché recognition in GRASPR involves the use of clichés library for detecting instance of cliché in program using graph parsing algorithm, which differ than in CONCEIVER++ implementation, plan recognition algorithm.

Automated program understanding: Lots of researchers' groups have focused their efforts in developing tool and technique for automating program understanding. Different program understanding systems are tends to apply different representative framework and heuristics in recognition algorithm. Example, Concept Recognizer by Kozaczynski and Ning[8] used top-down library based approach for plan recognition. This system recognizing plan using heuristic approach, specific rules and constrains instruction using representation of component and constraint of plan. Source code will be transformed into abstract syntax tree. The plan recognition algorithm starts by collecting all patterns from the library, then matching all components to the program, come out with a set of potential plans with all components matched. After that, the constraint part of the set of plans will be implemented. Limpiyakorn[15] says that plan representation in Concept Recognizer is simple and unambiguous, also algorithm used is successful to recognize plan in COBOL programs.

Representation of abstraction emitting information that not needed such as syntax tree omits format variations, while control flow graph omitting variation for control statements. Representation replacing codes with abstract model such as event for Quicili[16,17] represents syntax tree entity. Abstraction was needed for recognition because it will simplify searching area for program representation. In addition, abstract representations have multiple use if there any missing information. Syntax tree and control flow graph will retain the same execution.

Learning of programming: The learning of programming aspect is stressed out since research focus is concentrate on learning of programming towards students in university level. In learning of programming, one of the most problem faces by students in writing program codes is programming errors. The students always feel unmotivated on trying to read and understand the meaning of the fragment of program code in order to correct the error. That is why we developed a program understanding system which can help the students. Since the study is on the object-oriented language, some explanation on object-oriented languages will be discussed here with several researches in program understanding that use the object-oriented concept or language. Object orientation is not only programming paradigm. Hoffman[18] has his opinion that it is a design paradigm. Hoffman also said that relationship between components can be designed with scattered, non-procedure. But at the same time, the low level aspects of object-oriented language are the same as the procedural languages.

The use of object-oriented paradigm has showed the achievement's results in software engineering such as maintenance and the usability is easy to achieve. It is relates with encapsulation, inheritance and polymorphism that includes in the object-oriented paradigm. Because of these concepts, object-oriented codes have been used widely and with variety in implementation the same task. Lieberherr and Holland[19] explained that the good object-oriented programming styles and techniques are by writing small candidate functions. Therefore, it produces a system that contains numbers of several small modules.

The available program understanding systems using the plan based approach usually are developed for non-object-oriented programming languages, there are such as Programmer's Apprentice[12], GRASPR[9], PROUST[20], Talus[21], PAT[10], CONCEIVER[22] and BUG-DOCTOR[23]. But the object-oriented languages has widely used in education and industry recently. Obviously in local and foreign countries, this new paradigm programming language has been used in teaching of computer programming and has been proved by Arif[24], Bruhn and Burton[25], Gerailt[26] and Madden and Chambers[27]. Therefore, an object-oriented program understanding system is needed specifically for teaching of programming.

Most of the available programs understanding systems are specially developed for maintaining a system in an organization. Problem arises when the existing system is not usable for teaching programming purposes. By the fact that it can help students on learning programming, reading and understanding certain program codes. Currently, object-oriented programming languages have been widely used to learn programming at many universities, local or abroad. Program understanding system with plan for object-oriented does not exist was the main reason why this research is being carried out. The main purpose of our study is to develop a program understanding system for object-oriented language that is Java using plan base approach. Knowledge relates to programming codes will be parsed and represented or transformed into intermediate representation and then the information about program will be kept as programming plan in the plan base.

Kutti *et al*.[28] have said that the simplicity and versatile nature of Java has made it popular as a general-purpose language within the Computer Science community. Apart from that Java also has a deviation such as the case of declaring reference pointers. The declaration syntax of a reference pointer looks like declaring a normal variable. According to Kutti *et al*.[28] again, to interpret the meaning is depends on the intuition of the programmers. The meaning will become wrong if the programmers are the novices. Even the experienced also will make the same mistake. For learning of programming purposes, the program understanding system can helps the programmers to identified the correct meaning for each statements of object-oriented programming codes especially Java. This is also the importance of developing one program understanding system which aid in the learning process.

Plan formalism for conceiver++: The plan formalism is based on the plan formalism by Al-Omari[22], Kozaczynski *et al.*[13] and Ning[10]. CONCEIVER++ adds the debugging function in the plan formalism and modifies the plan formalism for object-oriented language. One of the program understanding systems that has the debugging facilities is PAT[10]. The difference between PAT and CONCEIVER++ is that the plans inside PAT represent the algorithms while the plan for CONCEIVER++ represents the stereotyped fragments of statement line of code.

The plan formalism developed for CONCEIVER++ is based on the Java programming language syntax. The main role of the understanding engine is to find plans that match the program codes. If it is found, then the explanation will be generated. If it is not found it may due to the presence of errors in the given plan. So, the debugging engine will check the program based on the bugs segment of plan to determine the error, which is the output. The initial design of plan formalism has been discussed in Sani *et al.*[29]. After some refining process on the formalism to represents the knowledge, we found that some of the programming language structure itself is very important to support the recognition process. Therefore, the plan formalism is refined based on the structure of the Java program or cliché. Based on the object-oriented program structure or cliché, the structure of a Java program is commonly consists of name of variable, value of variable, operator, relational operator, modifier, class name, method name and object reference. The result is the plan formalism for Java programming language contains of several segments. Each segment represents certain information for the plan that will use to recognize a statement line of code. The explanation of all segments of plan is as follows:

- Plan number: Each plan is refer through a number
- Plan name: Each plan has a name
- Modifier: Will recognize public, private or protected modifier
- Class name: Name of the class
- Method name: Name of a method in certain class.
- Declarator: Initialization of identifier in certain class
- Relational operator: Operator <, <=, >, >=, == and !=
- Identifier 1, 2, 3: Variable name used in a plan.
- Integer: Integer value of variable
- Constraint:
- Plan: Plan that involved in inferring the plan
- Debugging: Logic error that may exist in plan
- Plan connection: Other plans that connects to this plans
- Meaning: Explanation of the plan

This is the plan formalism that is going to be used to create the programming plans.

MATERIALS AND METHODS

For development and testing of the CONCEIVER++, methodology that will be use is Unified Approach (UA). The process involved from UA for developing and testing the system is iterative development and continuous testing. The process must be iterate and reiterate until satisfied with the system. Since testing often uncovers the weaknesses of the design, usually it will provides additional information that want to use, by repeating the entire process, taking what have been learned and reworking on the design or moving to re-prototyping and retesting. This refining cycle will be continuing through the development process until satisfied with the results[30]. Then finally, the prototype will be transformed into actual system. The process of iterative development and continuous testing is shown in Fig. 1.

In effort to test the quality assurance of the system is by choosing the black box testing strategies. In a black box, the test item is treated as “black”, since its logic is unknown[30]. In this testing, we are trying to put Java program codes and examine the resulting output from the recognition engine of CONCEIVER++. We had chosen this strategy because according to Bahrami[30], black box testing works very nicely in testing objects in an object-oriented environment. The steps involved in doing the test are as below:

- A program code will be taking as source to the system:
  - This source code has to be parsed and transformed to produce the abstract syntax tree (AST)
  - AST will be structured into nodes of line of program codes. This is for the nodes representing the AST will then use for creating the control flow graph (CFG)

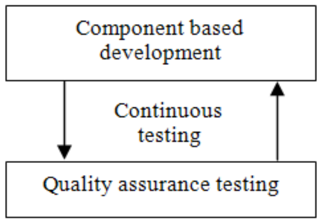


Fig. 1: Iterative development and continuous testing

- CFG will be created for the source code:
    - CFG will show the execution flow of the source code. The CFG is created manually during this step
    - The information of data flow also will be annotated in the CFG
    - Information of each CFG's node will be saved in text file and will be as input to recognition engine of understanding module
- File which kept information of CFG will be generated and all the information will be hold by binary search tree (BST) data structure. While, plan that been saved in the plan base will be accessed and put in the linked list data structure. These two structures will be run and input into the recognition engine
- The BST and linked list will be matched to come out with the understanding result. The result for each node of BST will be shown

Understanding module of Conceiver++: An overall process in the understanding module of CONCEIVER++ is that, programming code written by students has to be parsed and transform using the parser and transformer components. Output from it is in form of Control Flow Graph (CFG) and being kept in one file. The CFG file will read by the code/CFG processor and then all the CFG information will be put in binary search tree structure. In the other side, plan which has been kept in plan base will be access by the plan processor. Plans is read and put into linked list structure. These tree and linked list will be as input to the recognition engine. The recognition engine will match the data from both structure type and will come out with the result of understanding to the user of this system. The detailed model of CONCEIVER++ can be read from Sani *et al.*[31].

This module has been divided into three parts that are code/CFG processor, plan processor and recognition engine. In this research, the user in this module is students or lecturers. Users will write a Java program codes and then insert to the understanding module to be inferred by the system. At the same time, plans will be accessed from plan base for the purpose of inferring the codes. Below discuss on each part of the understanding module. In presenting the results, one example of Java programming code is used to describe the detail processes. This is importance in the implementation and testing process in order to check the accuracy of the system output. The discussion below starts by selecting a select sort program as the input or source program into the module.

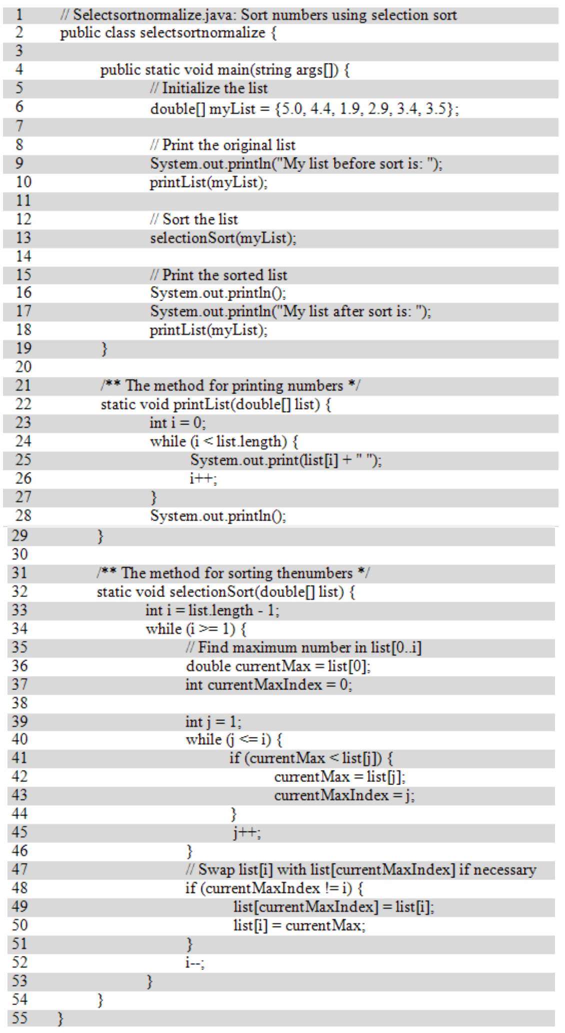
RESULTS AND DISCUSSION

The implementation is discussed and described by using the example shows the execution process of a select sort program that will be understood by students. The original program is parsed and transformed to AST form and then converted into CFG. The resulted CFG form is then will be executed using Understanding Module of CONCEIVER++. The document of understanding for each line of the program code will be shown.

Example of select sort program: The implementation of the select sort program written in object-oriented programming language that is Java. The select sort program covers the fundamental concepts. The concepts are variable initialization, assignment and control statement, methods and array. Figure 2 shows the normalized select sort program. This normalized program has undergone the parsing and transforming process to produce the AST. Normalization is not the main focus of this study, but for this explanation purposes, we just take it as the normalize code is used for implementation purposes. AST is a tree representation of the abstract syntactic structure of program code. It is originates from the parse tree without including the semantic of the program. The AST of each line of code of the select sort program is illustrated in Table 1.

CFG for the select sort program: The AST only show the simplified form of the program after undergoing the parsing and transforming process. The flow of the program is actually shown in the CFG. The CFG that is produced for the select sort program is illustrated in Fig. 3. The Fig. 3 shows the nodes and arrow for the select sort program implementation. One node in the CFG represents one line of program source code.

The CFG starts with the node that is written as Start and then to node number 2. The node number 2 corresponds with line 2 of the select sort program in Fig. 2 and line of code number 2 in Table 1.



```
1     // Selectsortnormalize.java: Sort numbers using selection sort
2     public class selectsortnormalize {
3
4         public static void main(string args[]) {
5             // Initialize the list
6             double[] myList = {5.0, 4.4, 1.9, 2.9, 3.4, 3.5};
7
8             // Print the original list
9             System.out.println("My list before sort is: ");
10            printList(myList);
11
12            // Sort the list
13            selectionSort(myList);
14
15            // Print the sorted list
16            System.out.println();
17            System.out.println("My list after sort is: ");
18            printList(myList);
19        }
20
21        /** The method for printing numbers */
22        static void printList(double[] list) {
23            int i = 0;
24            while (i < list.length) {
25                System.out.print(list[i] + " ");
26                i++;
27            }
28            System.out.println();
29        }
30
31        /** The method for sorting thenumbers */
32        static void selectionSort(double[] list) {
33            int i = list.length - 1;
34            while (i >= 1) {
35                // Find maximum number in list[0..i]
36                double currentMax = list[0];
37                int currentMaxIndex = 0;
38
39                int j = 1;
40                while (j <= i) {
41                    if (currentMax < list[j]) {
42                        currentMax = list[j];
43                        currentMaxIndex = j;
44                    }
45                    j++;
46                }
47                // Swap list[i] with list[currentMaxIndex] if necessary
48                if (currentMaxIndex != i) {
49                    list[currentMaxIndex] = list[i];
50                    list[i] = currentMax;
51                }
52                i--;
53            }
54        }
55    }
```

Fig. 2: The select sort program

The process continues until the last node, which is written as End. Every node carries the information of each line of code. Fig. 3 also shows the flow of data of the node, which is represented as dotted arrow and the variable name represented as dotted box. The flow of data annotated with CFG is the value that will be used in the understanding or recognition process.

Generation of Binary Search Tree (BST) for CFG: The information of each line of code for each node and the flow of data is used as the input to code/CFG processor in the understanding module. After the system has produced the CFG, information for each node is transformed and saved into the BST data structure. The result of CONCEIVER++ for the code/CFG processor part is shown in Fig. 4.

Table 1: AST for the select sort program

Line of code	Abstract Syntax Tree (AST)	Line of Code	Abstract Syntax Tree (AST)
2	Root Modifier: public Class Name: SelectionSort	4	Method Name: main Ident: String Declarator: args
6	Type_specifier: double Declarator: myList	9	Statement Ident: System Ident: out Ident: println String: "My list before sort is: "
10	Statement Ident: printList Ident: myList	13	Statement Ident: selectionSort Ident: myList
16	Statement Ident: System Ident: out Ident: println	17	Statement Ident: System Ident: out Ident: println String: "My list after sort is: "
18	Statement Ident: printList Ident: myList	22	Method Name: printList Type_specifier: double Declarator: list
23	Statement Type_specifier: int Declarator: i Int: 0	24	Ident: I Cmp level op: < Ident: list Ident: length Ident: i
25	Statement Ident: System Ident: out Ident: print Ident: list Ident: i Operator: + String: " "	28	Statement Ident: System Ident: out Ident: println
32	Method Name: selectionSort Type_specifier: double Declarator: list	33	Statement Type_specifier: int Declarator: i Ident: list Ident: length Operator: - Int: 1
34	Ident: i Cmp level op: >= Int: 1 Ident: i	36	Statement Type_specifier: double Declarator: currentMax Ident: list Int: 0
37	Type_specifier: int Declarator: currentMaxIndex Int: 0	39	Statement Type_specifier: int Declarator: j Int: 1 Ident: j
40	Cmp level op: <= Ident: i Ident: j	41	Statement If Ident: currentMax Cmp level op: < Ident: list Ident: j
42	Statement Ident: currentMax Operator: = Ident: list Ident: j	43	Statement Ident: currentMaxIndex Operator: = Ident: j
48	Statement If Ident: currentMaxIndex Ident: i	49	Statement Ident: list Ident: currentMaxIndex Operator: = Ident: list Ident: i
50	Statement Ident: list Ident: i Operator: = Ident: currentMax		

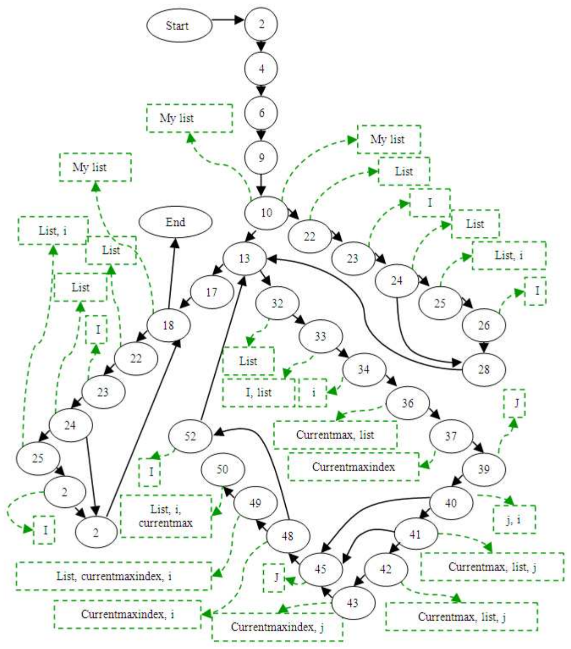
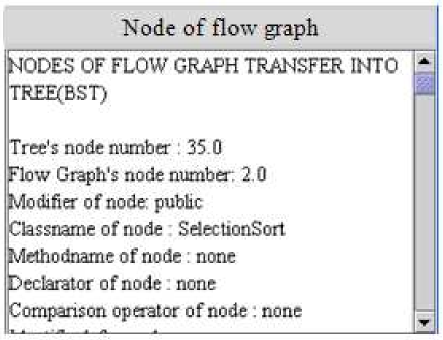


Fig. 3: CFG for the select sort program



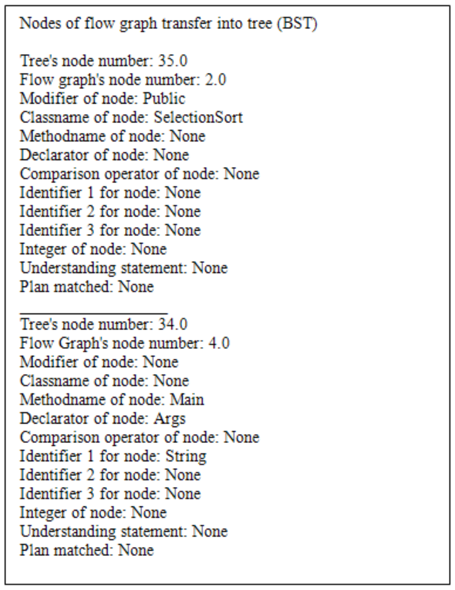
NODES OF FLOW GRAPH TRANSFER INTO TREE(BST)

Tree's node number : 35.0
Flow Graph's node number: 2.0
Modifier of node: public
Classname of node : SelectionSort
Methodname of node : none
Declarator of node : none
Comparison operator of node : none

Fig. 4: Result of code/CFG processor of CONCEIVER++

The detail information of the BST node cannot be seen from Fig. 4 Because of this, Fig. 5 shows the details of two nodes that have been generated. The number of BST node is also displayed.

From Fig. 5, the information of each BST node are tree's node number, flow graph's node number, modifier of node, class name of node, method name of node, declarator of node, comparison operator of node, identifier of node, integer value of node, understanding of node and plan matched. However, not all nodes have all these information. There is no information for understanding and plan matched. These two information will be filled when the CFG are recognized by the understanding processor or recognition engine.



Nodes of flow graph transfer into tree (BST)

Tree's node number: 35.0
Flow graph's node number: 2.0
Modifier of node: Public
Classname of node: SelectionSort
Methodname of node: None
Declarator of node: None
Comparison operator of node: None
Identifier 1 for node: None
Identifier 2 for node: None
Identifier 3 for node: None
Integer of node: None
Understanding statement: None
Plan matched: None

Tree's node number: 34.0
Flow Graph's node number: 4.0
Modifier of node: None
Classname of node: None
Methodname of node: Main
Declarator of node: Args
Comparison operator of node: None
Identifier 1 for node: String
Identifier 2 for node: None
Identifier 3 for node: None
Integer of node: None
Understanding statement: None
Plan matched: None

Fig. 5: Information of two BST nodes

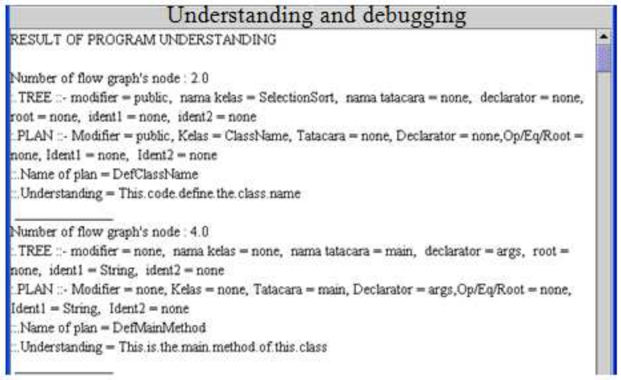
In addition, this information will also be used for higher-level recognition. In Fig. 5, the first CFG node number 2 and tree node number 35, has the modifier information that is public and class name of the node is SelectionSort. The second CFG node number 4 and tree node number 34, has the main method name information, args for declarator and String identifier.

Generation of linked list for programming plan: Plan processor is one of major part in CONCEIVER++. The process involve in this plan processor part is to read the plan in the plan base. Linked list structure is generated and each node in the linked list will contain plan, including the information of the plan. This generated linked list with plan inside is the output from this plan processor part and will be as input for the recognition engine or understanding processor.

All the information is the data about knowledge of language based on the plan formalism that had been mentioned above. Because of the design of the plan formalism is not the focus of this study, researched has been done and the resulted plans are based on discussion has agreed on specifying the information needed for representing the knowledge. For the execution purposes of the plan or plan base processor, data or information for all plans has been kept in a text file (in.dat). All of these plans will be generated into linked list data structure mentioned, will input to the understanding engine. This part or processor is the main knowledge for the recognition process. Table 2 shows a few numbers of plans that contain in the plan base.

Table 2: Plans' data or information that contain in the text file

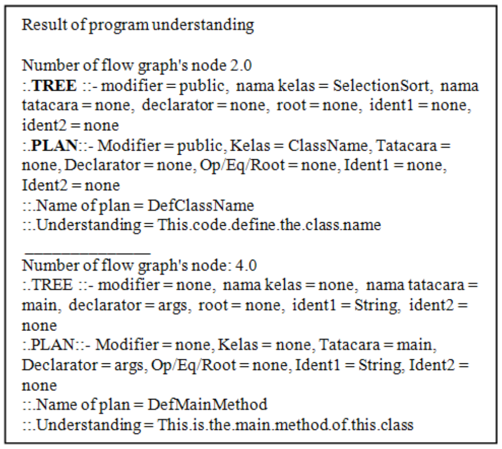
Plan No.	Plan name	Modifier	Class	Method	Declrator	CmpOp	Identifier			Integer	Const	Plan	Meaning
							Ident1	Ident2	Ident3				
101	AssignAValue	None	None	None	none	None	Var1	None	None	Value	None	None	This.code.is.assigning .a.variable
102	AssignAVar	None	None	None	None	None	Var1	Var2	None	None	None	None	This.code.is.assigning. Var2.to.Var1
103	Assign AConstant	None	None	None	None	Final	Var1	None	None	Value	None	None	This.code.is.assigning .a.constant
104	SimpleOut	None	None	None	None	None	System	Out	Print	None	None	None	This.code.print.output. to.the.computer.screen
105	SimpleOut	None	None	None	None	None	System	Out	Println	None	None	None	This.code.print.output.to .the.computer.screen
106	SystemExit	None	None	None	None	None	System	Exit	None	Zero	None	None	This.code.is.a.Java. predefined.class.to.exit. the.system
107	BoolLess	None	None	None	None	Less	Var1	Var2	None	None	None	None	This.code.is.a.Boolean. Expression.which.Var1. is.less.than.Var2
108	BoolLess	None	None	None	None	Less	Var1	None	None	Value	None	None	This.code.is.a.Boolean. Expression.which.Var1 .is.less.than.Value
109	BoolLessEq	None	None	None	None	LessEq	Var1	Var2	None	None	None	None	This.code.is.a.Boolean .Expression.which.Var1. is.less.than.or.equal.to.Var2
110	BoolLessEq	None	None	None	None	LessEq	Var1	None	None	Value	None	None	This.code.is.a.Boolean. Expression.which.Var1.is .less.than.or.equal.to.Value


Understanding and debugging

```
RESULT OF PROGRAM UNDERSTANDING

Number of flow graph's node : 2.0
::TREE ::- modifier = public,  nama kelas = SelectionSort,  nama tatacara = none,  declarator = none,
root = none,  ident1 = none,  ident2 = none
::PLAN ::- Modifier = public, Kelas = ClassName, Tatacara = none, Declarator = none,Op/Eq/Root =
none, Ident1 = none,  Ident2 = none
:::Name of plan = DefClassName
:::Understanding = This.code.define.the.class.name
______________
Number of flow graph's node : 4.0
::TREE ::- modifier = none,  nama kelas = none,  nama tatacara = main,  declarator = args,  root =
none,  ident1 = String,  ident2 = none
::PLAN ::- Modifier = none, Kelas = none, Tatacara = main, Declarator = args,Op/Eq/Root = none,
Ident1 = String,  Ident2 = none
:::Name of plan = DefMainMethod
:::Understanding = This.is.the.main.method.of.this.class
______________
```

Fig. 6: Result of matching or recognition engine for CONCEIVER++


```
Result of program understanding

Number of flow graph's node 2.0
::TREE ::- modifier = public,  nama kelas = SelectionSort,  nama
tatacara = none,  declarator = none,  root = none,  ident1 = none,
ident2 = none
::PLAN::- Modifier = public, Kelas = ClassName, Tatacara =
none, Declarator = none, Op/Eq/Root = none, Ident1 = none,
Ident2 = none
:::Name of plan = DefClassName
:::Understanding = This.code.define.the.class.name
______________
Number of flow graph's node: 4.0
::TREE ::- modifier = none,  nama kelas = none,  nama tatacara =
main,  declarator = args,  root = none,  ident1 = String,  ident2 =
none
::PLAN::- Modifier = none, Kelas = none, Tatacara = main,
Declarator = args, Op/Eq/Root = none, Ident1 = String, Ident2 =
none
:::Name of plan = DefMainMethod
:::Understanding = This.is.the.main.method.of.this.class
```

Fig. 7: Information on understanding result

Result of understanding: The nodes that are stored in the BST data structure and the plans that are stored in the linked list are used as the input to the recognition engine in the understanding module. The recognition process is by matching the BST with the plans in the linked list to produce the document of program understanding.

The process of matching or recognition in the recognition engine is based on the structure of object-oriented programming language. The structure of object-oriented programming language that consists of modifier, class name, method names involve for in class, object name are some of the structures that will be check to identified for recognizing the line of code for the Java programming language.

Some of the result of understanding engine for CONCEIVER++ of the select sort program is shown in Fig. 6. In Fig. 6 shows lists of the plan structure, the structure of the tree and the result of understanding. The information inside the structures is used in the matching process. The result that assists the student is the name of the plan and the explanation of that line of code. Figure 7 is the copy of Fig. 6 that shows the clearer information of the understanding result and for easy explanation purposes. The structures of the plan that are matched with the structures of the tree are modifier, class name (nama kelas), method name (nama tatacara), declarator, root (represent comparison operator) and identifier value.

Table 3: Testing result

CFG node	Plan matched	Meaning of understanding	Correctness output
2.0	DefClassName	This.code.define.the.class.name	√
4.0	DefMainMethod	This.is.the.main.method.of.this.class	√
6.0	DeclareVar	This.code.declare.a.variable	√
9.0	SimpleOut	This.code.print.output.to.the.computer.screen	√
10.0	SendMsgToMethod	This.code.send.mesej.to.method.for.execution	√
22.0	DecMethod	This.code.define.a.method	√
23.0	DeclareVar	This.code.declare.a.variable	√
24.0	BoolLess	This.code.is.a.Boolean.Expression.which.Var1.is.less.than.Var2	√
25.0	SimpleOut	This.code.print.output.to.the.computer.screen	√
26.0	Increment	This.code.increment.value.of.Var.by.1	√
28.0	SimpleOut	This.code.print.output.to.the.computer.screen	√
13.0	SendMsgToMethod	This.code.send.mesej.to.method.for.execution	√
32.0	DecMethod	This.code.define.a.method	√
33.0	DecAssignArr	This.code.assign.array.to.declarator	√
34.0	DecBoolGreaterEq	This.code.is.a.Boolean.Expression.which.Var.is.greater.than.or.equal.to.Value	√
36.0	DecAssignVar	This.code.assign.variable.to.declarator	√
37.0	DeclareVar	This.code.declare.a.variable	√
39.0	DeclareVar	This.code.declare.a.variable	√
40.0	DecBoolLessEq	This.code.is.a.Boolean.Expression.which.Var.is.less.than.or.equal.to.Var1	√
41.0	BoolLess	This.code.is.a.Boolean.Expression.which.Var1.is.less.than.Var2	√
42.0	AssignAVar	This.code.is.assigning.Var2.to.Var1	√
43.0	AssignAVar	This.code.is.assigning.Var2.to.Var1	√
45.0	Increment	This.code.increment.value.of.Var.by.1	√
48.0	BoolNotEq	This.code.is.a.Boolean.Expression.which.Var1.is.not.equal.to.Var2	√
49.0	AssignAVar	This.code.is.assigning.Var2.to.Var1	√
50.0	AssignAVar	This.code.is.assigning.Var2.to.Var1	√
52.0	Decrement	This.code.decrement.value.of.Var.by.1	√
17.0	SimpleOut	This.code.print.output.to.the.computer.screen	√
18.0	SendMsgToMethod	This.code.send.mesej.to.method.for.execution	√
22.0	DecMethod	This.code.define.a.method	√
23.0	DeclareVar	This.code.declare.a.variable	√
24.0	BoolLess	This.code.is.a.Boolean.Expression.which.Var1.is.less.than.Var2	√
25.0	SimpleOut	This.code.print.output.to.the.computer.screen	√
26.0	Increment	This.code.increment.value.of.Var.by.1	√
28.0	SimpleOut	This.code.print.output.to.the.computer.screen	√

√: Indicates that the CFG node and plan is matched correctly and give correct understanding for that particular code

In Fig. 7 for example, the tree node number 2 is recognized from the plan named DefClassName and the line of code explains the definition of the class name. From the Fig. 7, the tree node number 4 is recognized from the plan named DefMainMethod and the meaning of the line of code explains that, it is the main methods of the class.

Result of testing: The results of black box testing which show the understanding module output for each nodes of Control Flow Graph (CFG) has been recognized by the specific plan. Each node which has been identified with the correct plan and gives the correct meaning of the node shows the correctness output of the understanding module that contains three parts as mentioned above.

The result of the generated understanding module for all CFG nodes of the select sort program code is check to make sure that the output is correct. If it is wrong or do not have meaning for certain program code, we check whether there is something wrong on the logic of the program or the plan base is not complete to understand the select sort example code. After refining the development of understanding module and continuing testing, the result of the understanding module is correct for all nodes in the select sort program code. Thus, we can say that the correctness of output for the understanding module of CONCEIVER++ is 100 percents matched for select sort program code. Please refer to Table 3 to prove the result.

CONCLUSION

The implementation of CONCEIVER++ has been discussed in detail in this study. The process of understanding, specifically the understanding module which contains three parts, which are code/CFG processor, plan processor and recognition engine has been tested and explained. Java programming source code, the select sort program is used to show the resulted of correct output for the understanding module by following the black box testing steps as mentioned in the methodology of the study. From this study we have shown that all nodes of the example source code has been recognized and got the correct meaning. For future works we will do evaluation to the system with difference case studies to check the effectiveness of the recognition process to understand the difference style of written programming codes.

ACKNOWLEDGEMENT

The authors acknowledge the financial support (Science Fund) received from the Ministry of Science, Technology and Innovation (MOSTI), Malaysia via University Putra Malaysia.

REFERENCES

1. Romero, P., R. Cox, B. du Boulay and R. Lutz, 2003. A survey of external representation employed in object-oriented programming environments. J. Vis. Languages Comput., 14: 387-419. DOI: 10.1016/S1045-926X(03)00036-3
2. Barr, M., S. Holden, D. Philips and T. Greening, 1999. An exploration of novice programming errors in an object-oriented environment. SIGCSE. Bull., 31: 42-46. DOI: http://doi.acm.org/10.1145/349522.349392
3. Stroustrup, B., 1987. What is object-oriented programming? Lecture Notes Comput. Sci., 276: 51-70. DOI: 10.1007/3-540-47891-4_6
4. Alagar, V.S. and R. Missaoui 1995. Object-Oriented Technology for Database and Software Systems. World Scientific, ISBN: 9810221703, pp: 312.
5. Kozaczynski, W. and J.Q. Ning, 1989. SRE: A knowledge-based environment for large-scale software re-engineering activities. Proceeding of the 11th International Conference on Software Engineering, (SE'89), ACM Press, Pittsburgh, Pennsylvania, United States, pp: 113-122. DOI: http://doi.acm.org/10.1145/74587.74603
6. Quicili, A., Q. Yang and S. Woods, 1998. Applying plan recognition algorithms to program understanding. J. Automat. Software Eng., 5: 347-372. DOI: 10.1023/A:1008608825390
7. Woods, S. and Q. Yang, 1995. Program Understanding as Constraint Satisfaction. Proc. Comput. Aid. Software Eng., 7: 318-327. DOI: 10.1109/CASE.1995.465302
8. Kozaczynski, W. and J.Q. Ning, 1994. Automated program understanding by concept recognition. Automat. Software Eng., 1: 61-78. DOI: 10.1007/BF00871692
9. Wills, L.M., 1993. Flexible control for program recognition. Proceedings of the Working Conference on Reverse Engineering, May 21-23, IEEE Xplore Press, Baltimore, MD., USA., pp: 134-143. DOI: 10.1109/WCRE.1993.287771
10. Ning, J.Q., 1989. A knowledge-based approach to automatic program analysis. Ph.D. Thesis, University of Illinois. http://portal.acm.org/citation.cfm?id=916199
11. Müller, H.A., 1996. Understanding software systems using reverse engineering technologies research and practice. Proceeding of the Tutorial Presented at 18th International Conference on Software Engineering, Berlin, Germany, pp: 1-9. http://www.rigi.cs.uvic.ca/downloads/papers/pdf/ussuret.pdf
12. Rich, C. and L.M. Wills, 1990. Recognizing a program's design: A graph-parsing approach. IEEE. Software, 7: 82-89. DOI: 10.1109/52.43053
13. Kozaczynski, W., J. Ning and A. Engberts, 1992. Program concept recognition and transformation. IEEE. Trans. Software Eng., 18: 1065-1075. DOI: 10.1109/32.184761
14. Sani, N.F.M., A.M. Zin and S. Idris, 2008. Object-oriented codes representation of program understanding system. Proceeding of the International Symposium on Information Technology, Aug. 26-28, IEEE Xplore Press, Kuala Lumpur, Malaysia, pp: 450-454. DOI: 10.1109/ITSIM.2008.4631595
15. Limpiyakorn, Y. and Burnstein, I., 2003. Applying the Signature Concept to Plan-Based Program Understanding. Proceeding of the 19th IEEE International Conference on Software Maintenance, Sept. 22-26, IEEE Computer Society, USA., pp: 325. DOI: http://doi.ieeecomputersociety.org/10.1109/ICSM.2003.1235438.
16. Quicili, A., 1993. A Hybrid Approach to Recognizing Programming Plans. Proceeding of the IEEE 2nd Workshop on Program Comprehension, July 8-9, IEEE Xplore Press, Capri, Italy, pp: 96-103. DOI: 10.1109/WPC.1993.263901
17. Quicili, A., 1994. A memory-based approach to recognizing program plans. Commun.. ACM., 37: 84-93. http://doi.acm.org/10.1145/175290.175301
18. Hoffman, M.A., 2000. Methodology to support the maintenance of object-oriented systems using impact analysis. Ph.D. Thesis, Louisiana State University. http://portal.acm.org/citation.cfm?id=933565

19. Lieberherr, K.J. and I.M. Holland, 1989. Assuring good style for object-oriented programs. IEEE. Software, 6: 38-48. DOI: 10.1109/52.35588
20. Johnson, W.L. and E. Soloway, 1985. PROUST: Knowledge-based program understanding. IEEE. Trans. Software Eng., SE-11: 267-275. http://portal.acm.org/citation.cfm?id=801994
21. Murray, W.R., 2007. Automatic program debugging for intelligent tutoring systems. Comput. Intell., 3: 1-16. DOI: 10.1111/j.1467-8640.1987.tb00169.x
22. Al-Omari, H.M.A., 1999. CONCEIVER: A program understanding system. Ph.D. Thesis, University Kebangsaan Malaysia.
23. Burnstein, I. and F. Saner, 2000. Using fuzzy reasoning to support automated program understanding. Int. J. Software Eng. Knowl. Eng., 10: 115-137. http://md1.csa.com/partners/viewrecord.php?requester=gs&collection=TRD&recid=516084CI
24. Arif, E.M., 2000. A methodology for teaching object-oriented programming concepts in an advanced programming course. SIGCSE. Bull., 32: 30-34. http://doi.acm.org/10.1145/355354.355367
25. Bruhn, R.E. and P.J. Burton, 2003. An approach to teaching java using computers. SIGCSE. Bull., 35: 94-99. http://doi.acm.org/10.1145/960492.960537
26. Gearailt, A., 2002. Using Java to increase Active Learning in Programming Courses. Proceeding of the Inaugural Conference on the Principles and Practice of Programming, June 13-14, ACM Press, pp: 107-112. http://md1.csa.com/partners/viewrecord.php?requester=gs&collection=TRD&recid=20080280018707CI
27. Madden, M. and D. Chambers, 2002. Evaluation of student attitudes to learning the java language. Proceeding of the inaugural conference on the Principles and Practice of Programming, June 13-14, ACM Pres, Dublin, Ireland, pp: 125-130. http://portal.acm.org/citation.cfm?id=638501
28. Kutti, N.S., Z.A. Al-Khanjari, H.A. Ramadhan and J. Fiaidhi, 2005. A note towards reshaping java's features. J. Comput. Sci., 1: 450-453. http://www.scipub.org/scipub/detail_issue.php?V_No=4&j_id=jcs
29. Sani, N.F.M., A.M. Zin, S. Idris and Z. Shukur, 2005. Designing an Understanding and Debugging Tool (UDT) for Object-oriented programming language. WSEAS. Trans. Comput., 4: 137-142. http://portal.acm.org/citation.cfm?id=1363663
30. Bahrami, A., 1999. Object Oriented System Development using the Unified Modeling Language. Boston: McGraw-Hill.
31. Sani, N.F.M., A.M. Zin and S. Idris, 2009. Analysis and design of Object-oriented program understanding system. Int. J. Comput. Sci. Network Secur., 9: 125-134. http://paper.ijcsns.org/07_book/200901/20090118.pdf