ISSN:0975-9646

Program Recognition System for C

A Novel Take on Use of Plans and Clichés for Program Understanding

Manasi Deshmukh , Rohan Ingale , Rajat Doshi , Priyanka Sathe
Department of Computer Engineering
SKNCOE, University of Pune, Pune, India.

***Abstract*—This paper presents an approach for deriving an English language description of a C program directly from the source code. Two levels of translation are presented: cliché extraction, to identify commonly used programming constructs, and concept abstraction, to deduce the purpose of the program. Concept abstraction can serve as a basis for intelligent query support for providing relevant documentation. In this paper, we compare prominent works on program understanding systems, and propose an efficient method for plan representation and storage of plans in plan library, as an alternate approach to program recognition using flow graph parsing.**

Keywords—clichés, documentation, plans, program understanding

I. INTRODUCTION

Program recognition falls under both Software Engineering and Artificial Intelligence. In order to understand a program, a programmer attempts to recognize familiar parts, called clichés, and hierarchically builds an understanding of the entire program based on these parts. For example, a programmer may recognize that a bubble sort program is being used to order integers. The data structure used to store these integer elements may be recognized as having been implemented as an array of entries. This method to understand programs, called 'analysis by inspection', has been developed by Rich. 'Program Recognition' encompasses identification of commonly used algorithmic fragments, called clichés, and data structures in a program. Such a program recognition system is schematically shown in Fig. 1. Here, we present a system for C which performs program recognition automatically. The system takes a source code as input and generates a hierarchical description of clichés and data structures of which the code is constructed. Such a description may be useful in activities such as debugging, modifying, maintaining and documenting the program. Aside from its various practical applications, program recognition is a worthwhile concept to study from a theoretical standpoint in Artificial Intelligence. It can help us model how programmers understand programs based on their accumulated experience in programming. It is also a problem in which the representation of knowledge is the key to the efficiency and simplicity of the techniques used to solve the problem.

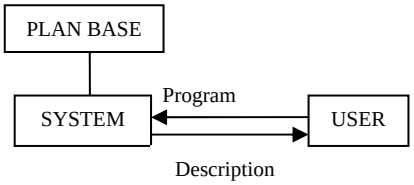

Fig. 1. A simple program understanding system

II. PROGRAM UNDERSTANDING SYSTEMS

Work on the development of program understanding systems had begun since the early 1970s. One of the first systems to tackle the problem of program recognition was developed in the early 1970s by Gregory Ruth. Some of the program understanding systems that were developed in the 1980s are PUDSY, LAURA, PROUST and TALUS. The ones developed in 1990s are Recognizer, PAT, BAL/SRW and DUDU. The latest systems developed in 2000s are Conceiver and Conceiver++. We will discuss and compare these systems below.

A. Ruth's System

One of the first systems to highlight the area of program understanding was developed by Gregory Ruth in the early 1970s. This system took the program code and a task description of the code as its input. It then tried to deduce the algorithm implemented in the code. This was achieved by matching the input code against several implementation patterns, which the system stored as a library. These implementation patterns were in the form of a set of characteristics about the code.

B. PUDSY

PUDSY was developed at the University of SUSSEX in 1980s. Apart from the source code, PUDSY took, as input, information about the purpose of the code that it was analyzing, in the form of program specification, which described the effects of the code. This system did not use this description for searching clichés. Rather, it analyzed the program and then compared the results of the analysis to the specification. Differences resulting from such a comparison were pointed out as bugs. PUDSY was used to analyze Pascal programs. It first used heuristics to segment a code into chunks, which were manageable units of code, such as loops. It then described the flow of information between these chunks by producing assertions about the values of the output variables of each chunk. These assertions were produced by recognizing familiar patterns of statements, called schemas, in the chunk. Each schema was related to a set of assertions describing their effects on the involved variables. For unrecognized chunks, assertions were produced by symbolic evaluation.

C. LAURA

Anne Adam and Jean-Pierre Laurent developed LAURA at the University Of Caen, France in 1980s. LAURA received information about the program to be analyzed in the form of a model code, which correctly performed the task that the code was intended to do. It then compared the graphs of these two codes and treated mismatches as bugs.

As nodes of the graphs were really statements of the code, the graph matching was essentially statement-to-statement matching. LAURA differed from PUDSY in that it represented source codes as graphs. This representation allowed LAURA to abstract the syntactic features of the programming language. Plans were not used during the process of understanding. Instead, programs were transformed to make them more reliable for comparison.

D. ***PROUST***

PROUST was a program for debugging Pascal programs written by novice programmers Johnson and Soloway in 1985 []. PROUST required a program and a non-algorithmic description of the program requirements as input. It produced the most likely mapping between requirements and the program as output. It used plans to represent stereotypical code fragments. Plans were represented as templates that consisted of three items: a Pascal statement, a sub-goal to be implemented as a set of statements, and a reference to a component of another plan. Plans could also contain slots indicating assertions about the plan. The Prior-Goals slot indicated other goals that had to be met before this plan could be activated, whereas the Posterior-Goals slot indicated goals to be added to the agenda after the plan is matched. Thus, these slots were used to control the goal-processing order when PROUST was executed.

E. ***TALUS***

TALUS was developed at University of Texas in 1986. It was designed to analyze programs, written in LISP, involving recursive definition of data structures. The main purpose of this system was to support automatic debugging of programs. TALUS was required to first recognize the input program before it could associate it with known reference functions. It also required a precise description of the problem to be recognized. TALUS separated knowledge representation into three levels: tasks, algorithms and functions. Tasks were basic programming assignments, generally given to students. TALUS had 18 tasks, each at a comparable level of abstraction to 'write a function returning a list of all the atoms in a tree'. TALUS assumed that the tasks were known prior to their execution. TALUS performed four steps to analyze students' programs: code simplification, algorithm recognition, bug detection and code correction. Code simplification put the program in If-Normal form and transformed it into a simpler Lisp dialect. In algorithm recognition, TALUS selected the algorithm that matched the input code. Once the algorithm is selected, reference functions were associated with the program's functions. In bug detection, TALUS determined whether the student's program and the reference program were equivalent, using symbolic evaluation. If no equivalence existed, TALUS would try to infer which bugs were present in the student's program. Symbolic evaluation considered the program in If-Normal form. The system derived conditions necessary to reach a leaf for the student's function and the reference function, and grouped them into cases. It then took the cases derived from the reference functions, and applied them to the input function, and vice versa.

F. ***Recognizer***

The Recognizer was developed at MIT Artificial Intelligence Laboratory as a part of the Programmer's Apprentice Project. The Recognizer only required the source code in order to recognize the familiar algorithmic clichés, unlike the previously described systems. It performed four main activities. Firstly, it analyzed the source code. Secondly, it converted the program plan into its corresponding flow graph. Thirdly, it parsed the flow graph with a grammar derived from a library of clichés. Lastly, it checked constraints on the matched flow graph. The nodes of the flow graph would represent operations and its edges would represent data flow. The recognizer performed bottom-up parsing of the input code to recognize clichés, and from these built an understanding of the entire program. The cliché library used by Recognizer was originally developed by Charles Rich to support the Programmer's Apprentice. This library provided taxonomy of standard computational fragments and data structures represented as plans. There were two forms of clichés, namely, plans and implementation overlays. Plans were used to represent data structures and algorithms. Plan nodes consisted of primitive forms, which were irreducible, or nodes corresponding to other plans. Implementation overlays represented alternative ways of expressing the same concept, usually from an abstract into a concrete form.

G. ***PAT***

PAT (Program Analysis Tool), like Recognizer, operated on code only. It converted the source code into a set of programming language independent objects, called events, using a parser. Then, using the event base, it recognized higher level events that represented function-like concepts. After higher level events were recognized, they were added to the event set. The process of recognition was repeated until no more higher level events were recognized. The final event set was presented to the user. This set presented the purpose of the program. A deductive-inference-rule engine was the main component of PAT. This engine used a library of program plans, stored as inference-rules in the plan base. These were used to derive new, high level events. A plan parser was used to parse the plans. These plans contained the understanding, paraphrasing, and debugging knowledge. When a new event was generated, it triggered other rules to fire, causing the generation of more events.

H. ***BAL/SRW***

BAL/SRW was an interactive knowledge-based environment which supported the process of recapturing and uncovering Assembly language logic and design of the program in order to reengineer it. BAL/SRW required only the source code as its input. The output from the system could merge with the new specification within a CASE tool in the forward phase of system re-engineering. BAL/SRW first performed a quick analysis for the Assembly language code in order to collect primary information about it. After information collection, the program was parsed in order to build the program knowledge base. The obtained

knowledge base represented the program information in the form of objects. Control flow graph of the program was derived using a control flow generator. The plan representation used by BAL/SRW used templates that were assembly fragments of the source code. The control flow was implicitly specified by the order of the templates.

I. DUDU

DUDU (Debugging Using Device Understanding) was developed for the purpose of debugging in 1991 at Ohio State University. Its representation of clichés was a text-based representation of plans that included goals, components for achieving them and casual links to show how the components achieve the goals. The functional representation was used to specify which parts of the program's clichés are supported by which parts of its plan representation. An important advantage of this representation was that it provided information which could make it easier to tolerate variation in how a function was achieved. As it explicitly described the purpose of each part of a cliché in the context of a larger of correctness, if some part of the cliché did not match the program, the functional representation described the function of that part. It thus made it possible to prove that the mismatch portion of the program still achieved this function.

J. Conceiver

Conceiver is a program understanding system which consists of five components: user interface, parser, understanding inference, document generator and plan base. The parser performs the task of translating the source program into a language independent representation to generate corresponding plans, which are then stored in the plan base. Once the language independent representation is generated, the understanding inference performs recognition. The recognition process starts with recognition of individual statements and then combines all such matches to perform comparison with the plans in the plan base. After plan matches are found, the documentation generator produces documentation by considering the hierarchy of recognized plans.

K. Conceiver++

Conceiver++ is program recognition for source codes written in JAVA. It is a line by line program understanding system which generates a description for each line of input code. The main task of this system is to find plans from the plan base that match the statements in the source code. If a match is found then the corresponding explanation will be generated. Else, the debugger in the system tries to find errors in the code that resulted in no match being found. This system takes a program code as input. The input program is parsed and transformed into an abstract syntax tree. In the abstract syntax tree, each node represents a statement from the code. These nodes are then used for construction of a control flow graph which shows the control and data flow of the source code. The control flow graph is then compared with the plans in the plan base to generate the description.

III. OUR APPROACH

We have attempted to simplify the process of program understanding by using simpler and easy-to-use representation of plans. The structure or the underlying framework of a C program can be recognized by removing unnecessary programming details and retaining only its clichés. An outline of the input source code, thus obtained, can then be adorned with necessary and sufficient information to understand the purpose of the code. Let us know take a look at the different phases an input program will undergo to finally generate an English language description of itself. This is schematically shown in Fig. 2.

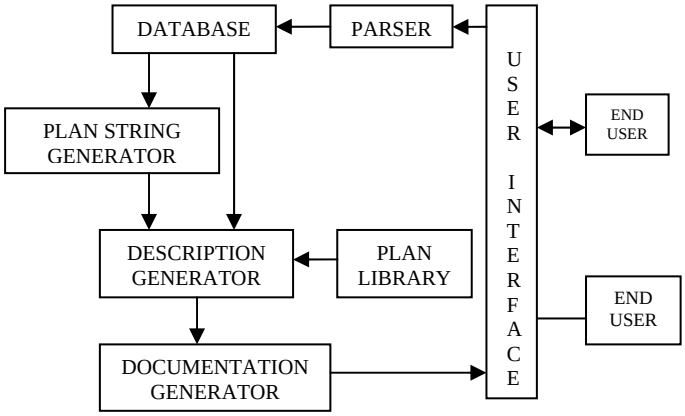

Fig. 2. System Architecture

A. Plan String Generation

The system takes only the source program as input. This program is parsed to generate tables in Microsoft Access 2007. These tables are as shown in Example (1). Information about clichés, i.e. all programming constructs, user defined functions and identifiers, gets stored in these tables. In the next step, this tabular information is processed to derive the plan strings.

Plan string is basically an outline of the input program in terms of the clichés used. These clichés include functions, loops (for, while, do while) and branch statements (if, if-else, switch). To generate a proper outline of the input program, relative positioning of constructs is represented by their 'degree of nesting'. The 'degree of nesting' is a natural number which reflects the extent to which a block or a construct is contained within the outermost block or construct.

B. Flexible Plan

The system's parser identifies all the operations, expressions and assignment statements in the input source code and stores them in a separate table. All variable dependencies in these statements are eliminated. These generalized statements are called plans. The plan library already stores known operations in the same generalized format. Plans obtained from the input source code are then searched for in the plan library with the help of the code's plan string. As the length of operation statements is variable, so will be the length of corresponding plans. To meet this variability, we make use of MongoDB for the storage of plans. MongoDB is an open-source document database which allows the construction of dynamic

schemas and thus helps in the storage of flexible length plans.

C. *Plan Matching*

The generalized statements obtained after processing the input code are then compared with the plans present in the plan library. The number of comparisons is limited by selecting only those plans from the plan library whose plan string matches the plan string of the input code. Once such plans are selected, comparison is performed and the description corresponding to the matched plan is used as the functionality description for that particular block of code.

IV. ILLUSTRATIVE EXAMPLE

A. *Plan String Generation*

Here we explain how the system produces a description of a C code for selection sort.

Example 1:

```
//Selection Sort
#include<stdio.h>
int main()
{
    int s,i,j,temp,a[20];
    printf("Enter total elements: ");
    scanf("%d",&s);
    printf("Enter %d elements: ",s);
    for(i=0;i<s;i++)
        scanf("%d",&a[i]);
  for(i=0;i<s;i++)
  {
      for(j=i+1;j<s;j++)
      {
          if(a[i]>a[j])
          {
              temp=a[i];
              a[i]=a[j];
              a[j]=temp;
          }
      }
  }
  printf("After sorting is: ");
  for(i=0;i<s;i++)
      printf(" %d",a[i]);
  return 0;
}
```

TABLE I. ASSIGNMENT TABLE

Assignment Table		
Statement ID	***Line Number***	***Statement***
0	20	temp = a[i]
1	21	a[i] = a[j]
2	22	a[j] = temp

TABLE II. FUNCTION TABLE

Function Table					
Function ID	***Function Name***	***Return Type***	***No. of Parameters***	***Start Line No.***	***End Line No.***
0	main	int	0	2	32

TABLE III. IDENTIFIER DECLARATION TABLE

Identifier Declaration Table						
Var ID	***Name***	***Initial Value***	***Scope***	***Type***	***Identifier Pointer***	***Line Number***
0	s	0	Main	int	-1	5
1	i	0	Main	int	-1	5
2	j	0	Main	int	-1	5
3	temp	0	Main	int	-1	5
4	a[20]	0	Main	int	-1	5

TABLE IV. INCLUDE TABLE

Include Table		
Include ID	***File Name***	***Line No.***
0	stdio	1

TABLE V. LOOP TABLE

Loop Table			
Loop ID	***Name***	***Start Line No.***	***End Line No.***
0	For	11	24
1	For	14	25
2	For	16	24
3	For	28	31

TABLE VI. PLAN STRING TABLE

Plan String Table	
Plan No.	***Plan String***
0	main/for/for//for///if/for

The number of slashes in the plan string denotes the degree of nesting of the construct that follows.

B. *Plan Matching*

After the plan string is produced, we consider the operations that are taking place within the recognized constructs. The generalized operation statements are compared with those stored in the following table which forms the core of the plan library.

TABLE VII. PLAN MATCHER TABLE

Plan Matcher Table		
Plan String	***Match***	***Description***
main/for/for//for///if/for	scan| loop|C1|0|C1|<|T1|C1|I1| loop|C2|C1+1|C2|<|T1|I1| if|AR[C1]|>|AR[C2]| swap|AR[C1]|AR[C2]| print	Selection Sort

The program's plan string is compared with those listed in the table. The result of this comparison may give more than one perfect matches. For all such matches, we check whether all the generalized operation statements are present in the 'Match' field, in the same order. When such a match if found, its corresponding description is adorned with English language phrases and is output to the user. In case no match is found, the user is prompted with appropriate message to either make changes in his code or to verify the code for its syntax.

In the above table, C1, C2, T1 and AR denote the generalized variables; scan, loop, if, swap, print denote the generalized operations; I1 is used to denote increment by 1; 0 and C1+1 following C1 and C2 respectively, denote the initial value of variables C1 and C2; C1<T1 and C2<T1 are loop conditions; separators denote the flexible storage in MongoDB which allows the 'Match' field to be as long as required.

Operations such as swap and scan are recognized using another table that stored commonly used operations in the same generalized format as used in the above table. If an operation is new to the system, the user will be prompted to add it to the library along with its descriptive notation. This notation is used to replace the entire corresponding operation into a word or two, thus trimming the program.

ACKNOWLEDGMENT

We extend our sincere thanks and deep gratitude to our project guide Prof. Mrs. Vaishali S. Deshmukh for her immensely valuable advice and guidance. We sincerely thank our Head of Department, Prof. Parikshit N. Mahalle, for his precious advice in the early stages of project selection. Lastly, we wish to thank our parents, families and friends for their patience and support.

REFERENCES

[1] G. R. Ruth, "Analysis of Algorithm Implementations," Technical Report 130, MIT Project Mac, 1974.

[2] R. C. Waters, "Automated Analysis of the Logical Structure of Programs," Technical Report 492, MIT Artificial Intelligence Lab., December 1978.

[3] H. E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding," Technical Report 503, MIT Artificial Intelligence Lab., April 1979.

[4] R. C. Waters, "A Method for Analyzing Loop Programs," IEEE Transactions on Software Engineering, vol. 5, no. 3, pp. 237-247, May 1979.

[5] F. J. Lukey, "Understanding and Debugging Programs," International Journal of Man-Machine Studies, vol. 12, pp. 189-202, 1980.

[6] Adam and J. Laurent, "LAURA: A System to Debug Student Programs,"
Arificial Intelligence, vol. 15, pp. 75-122, 1980.

[7] C. Rich, "Inspection Methods in Programming," Technical Report 604, MIT Artificial Intelligence Lab., June 1981.

[8] C. A. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," in Proc. 7th International Joint Conference on Artificial Intelligence, Vancouver, British Columbia, Canada, August 1981, pp.1044-1052. Reprinted in M. Brodie, J. Mylopoulus, and J. Schmidt, editors, "On Conceptual Modelling," Springer Verlag, 1984 and in C. Rich and R. C. Waters, editors, "Readings in Artificial Intelligence and Software Engineering," Morgan Kaufmann, 1986.

[9] R. C. Waters, "The Programmer's Apprentice: Knowledge-Based Program Editing," IEEE Transactions on Software Engineering, vol. 8, no. 1, January 1982.
.

[10] D. Scott Cyphers, "Programming Clichés and Cliché Extraction,"
Working Paper 223, MIT Artificial Intelligence Lab., February 1982.

[11] D. C. Brotsky, "An Algorithm for Parsing Flow Graphs," Technical
Report 704, MIT Artificial Intelligence Lab., March 1984.

[12] W. Lewis Johnson and Elliot Soloway, "PROUST: Knowledge-Based Program Understanding," IEEE 7th Conference on Software Engineering, Orlando, Florida, 1984, pp. 369-386.

[13] W. Lewis Johnson and E. Soloway, "PROUST: Knowlegde-Based Program Understanding," IEEE Transactions on Software Engineering, vol. 11, no. 3, 1985.

[14] R. C. Waters, "KBEmacs: A Step Toward the Programmer's Apprentice," Technical Report 753, MIT Artificial Intelligence Lab., May 1985.

[15] C. Rich, "The Layered Architecture of a System for Reasoning about Programs," in Proc. 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA, August 1985, pp. 540-546.

[16] R. C. Waters, "The Programmer's Apprentice: A session with
KBEmacs," IEEE Transactions on Software Engineering, vol. 11, no.
11, pp. 1296-1320, November 1985.

[17] W. R. Murray, "Heuristic and Formal Methods in Automatic Program Debugging," in Proc. 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA, August 1985, pp. 15-19.

[18] W. R. Murray, "Automatic Program Debugging for Intelligent Tutoring Systems," Technical Report 27, University of Texas at Austin, Computer Science Dept., June 1986.

[19] L. M. Wills, "Automated Program Recignition," Technical Report 904, MIT Artificial Intelligence Lab., February 1987.

[20] C. Rich, "Inspection Methods in Programming: Cliches and Plans,"
Memo 1005, MIT Artificial Intelligence Lab., December 1987.

[21] C. Rich and R. C. Waters, "The Programmer's Apprentice: A Research Overview," IEEE Computer, vol. 21, no. 11, pp. 10-25, November 1988. Also published as Memo 1004, MIT Artificial Intelligence Lab., November 1987.

[22] A. V. Aho, R. Sethi, and J. D. Ullman, "Compiler Principles, Techniques, and Tools," Addison-Wesley, 1987.

[23] L. M. Wills, "Automated Program Recognition: A feasibility demonstration," Artificial Intelligence, vol. 45, no. 1-2, pp. 113-172,
1990.

[24] C. Rich and L. Wills, "Recognizing a Program's Design: A Graph- Parsing Approach," IEEE Software, vol. 7, no. 1, pp. 82-89, January
1990.

[25] M. T. Harandi and J. Q. Ning, "Knowledge-based Program Analysis,"
IEEE Software, January 1990.

[26] D. Allemang, "Understanding Programs as Devices," Ph.D. Dissertation,
Ohio State University, 1990.

[27] W. Kozaczynski, E. Liongosari, and J. Ning, "BAL/SRW: An Assembler Re-engineering Workbench," Information and Software Technology, September 1991.

[28] A. Engberts, W. Kozaczynski, and J. Q. Ning, "Concept Recognition Based Program Transformation," Conference on Software Maintenance, October 1991.

[29] L. M. Wills, "Automated Program Recognition by Graph Parsing,"
Technical Report 1358, MIT Artificial Intelligence Lab., July 1992.

[30] W. Kozaczynski, J. Ning, and A. Engberts, "Program Concept Recognition and Transformation," IEEE Transactions on Software Engineering, vol. 18, no. 12, 1992.

[31] Abdullah Mohd. Zin, and Hani Ahmed Al-Omari, "Implementation of Conceiver: Not Just Another Program Understanding System," Jumal Antarabangsa (Teknologi Maklumat), vol. 3, pp. 73-87, 2002.

[32] Nor Fazlida Mohd. Sani, and Abdullah Mohd. Zin, "Implementation of Conceiver++ : An Object Oriented Program Understanding System," Journal of Computer Science, vol. 5, no. 12, pp. 1009-1019, 2009