A Program Recognition and Auto-Testing Approach

Wen C. Pai
Dept. of Business Mathematics
Soochow University
Taipei 100, Taiwan
E-mail: wencpai@ms1.hinet.net

Chin-Ang Wu
Computer Center
Chengshiu Institute of Technology
Kaohsiung County 833, Taiwan
E-mail: cwu@cc.csit.edu.tw

Abstract

The goals of the software testing are to assess and improve the quality of the software. An important problem in software testing is to determine whether a program has been tested enough with a testing criterion. To raise a technology to reconstruct the program structure and generating test data automatically will help software developers to improve software quality efficiently. Program recognition and transformation is a technology that can help maintainers to recover the programs' structure and consequently make software testing properly. In this paper, a methodology to follow the logic of a program and transform to the original program graph is proposed. An approach to derive testing paths automatically for a program to test every blocks of the program is provided. A real example is presented to illustrate and prove that the methodology is practicable. The proposed methodology allows developers to recover the programs' design and makes software maintenance properly.

Keywords: Software quality, Software testing, Program transformation, Program recognition, Reverse engineering

1. Introduction

Software testing is under heavy pressure to carry out the higher quality software as quickly as possible. The major effort in software engineering is spent after development on maintaining the systems to remove existing errors and to adapt them to changed requirements. As needs change, software must be amended, or maintained, to adapt to the new environment. Without an adequate understanding of a program's meaning, it is difficult to maintain it effectively. Maintainers often spend considerable energy trying to recover the design information before making changed. If there is no information about original design, the software becomes obsolete, and the enormous resources invested in its construction are lost.

Software testing is labor intensive and costly in software development. In a typical programming project, over 50% of the total cost are expended in testing the program or system. Testing consumes the majority of the software developers' effort of all the phases of system development.

Although a number of technologies or CASE tools are developed to help the developers to test program. However, these are almost giving effort in finding syntax-type error or program tracing. The static testing technologies are still the main testing approach in the real information development world. These approaches inspect the program by reading the code line by line, but not walking test cases through the program. To raise a technology to reconstruct the program structure and generating test data automatically will help software developers to improve software quality efficiently.

Program understanding and transformation is a technology that can be applied at least three areas in software engineering [3]. 1) Automatic programming is concerned with automated generation of a program from a description of the problem. 2) Program modification is used to change the behavior of a program such as functional enhancement. 3) Reverse engineering applies transformations from code to specification direction.

A lot of researches of program understanding and transformation are proposed. The PAT system, proposed by Harandi and Ning [2], uses interval logic to express semantic information such as control flow dependencies among sub-concepts in order to facilitate computation and reasoning of abstract concepts. Rich and Wills [4] built a prototype to find all occurrences of a given set of clichés in a program automatically, and build a hierarchical description of the program in terms of the clichés it finds. The transformation-based maintenance model, or TMM, developed by G. Arango, I. Baxter, P Freeman and C. Pidgeon [5], which use design histories of the code such as program specifications and the set of design decisions used to implement the program. They assume the design information is availability and accuracy. However, such design histories of the code is often rarely complete and reliable.

In this paper, a methodology to follow the logic of a program and transform to the original program graph is proposed. The proposed methodology is a reasonable and useful process that will allow maintainers to recover the programs' design and will make software maintenance properly.

Section 2 defines a number of program transformation rules. The program transformation algorithm is raised in Section 3. Section 4 gives a real example to illustrate the transformation process. Section 5 presents the conclusion and the future works we intend to finish.

2. Program recognition and transformation rules

Program graph is a useful approach to represent the logical control flow of a program. The maintainers can understand a program's flow by analyzing the program graph. The program graph can help maintainers to know the structure of a program, to test the program, and to derive testing paths

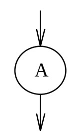
In general sense, the transformation of a program is viewed as a process of rewriting one program into another by repeated application of a set of transformation rules. Since a program is a combination of statements (or instructions), we can decompose a program into eight typical statement types, and define some transformation rules based on each statement type. Base on the transformation rules, a program will be analyzed and transformed to the program graph. The program graph then used to understand and modify the program.

There are eight typical statement types in a program: (1). Sequence statements, such as READ, WRITE, DEFINE a variable, OPEN a file... etc., (2). While-loop statements, (3). For-loop statements, (4). If-then-end statements, (5). If-then-else-end statements, (6). Repeat-loop statements, (7). Switch-case-with-default statements, and (8). Switch-case-without-default statements. Each statement type, or statement type set of the Sequence statements, essentially corresponding to a block in the program. In the paper we will derive testing paths automatically for each of the statement types to test every blocks of the program.

We will raise eight statement-statement flow transformation rules in the following. Although the eight statements types presented in the paper may be not in general condition, however, the other structured language can be considered in the similar approach. These rules will be used in the next section to transform a program. The approach of testing paths generating will also consider in the next section.

Rule 2.1 Sequence statements
transformation rule

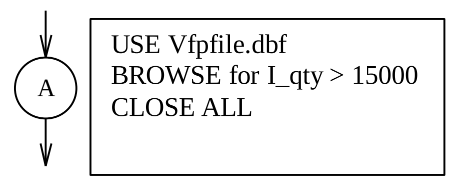
The statement flow of sequence statements is



For example, when transform a Microsoft FoxPro program as:

```
USE Vfpfile.dbf
BROWSE for I_qty > 15000
CLOSE ALL
```

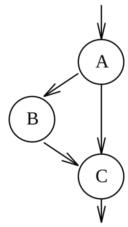
The transformed program flow based on the rule 2.1 is



From the flow, maintainers can maintain the program according to the program flow instead of considering the original program meaning, which will lead to maintain more efficiency.

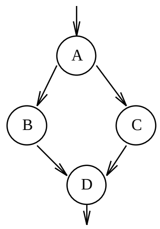
Rule 2.2 If-Then-End statements
transformation rule

The statement flow of If-Then-End statements is



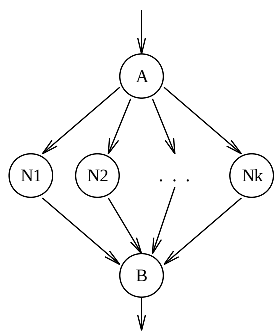
Rule 2.3 If-Then-Else-End statements transformation rule

The statement flow of If-Then-Else-End statements is



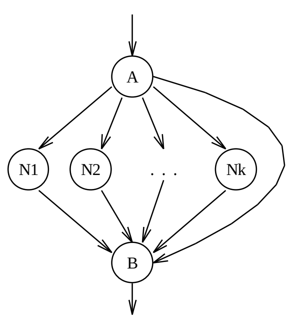
Rule 2.4 Switch-Case-With-Default statements transformation rule

The statement flow of Switch-Case-With-Default statements is



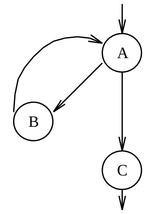
Rule 2.5 Switch-Case-Without-Default statements transformation rule

The statement flow of Switch-Case-Without-Default statements is



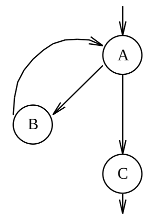
Rule 2.6 For-loop statements transformation rule

The statement flow of For-loop statements is



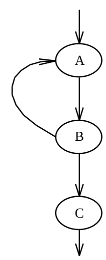
Rule 2.7 While-loop statements transformation rule

The statement flow of While-loop statements is



Rule 2.8 Repeat-loop statements transformation rule

The statement flow of Repeat-loop statements is



Syntactically, a program is a combination of statements. We can transform a whole program by first transforming each statement, and then combining the statement flow to a whole program graph. Maintainers to understand, audit, and modify the program can use the combined program graph. This will make maintenance works more efficiency. The program transformation rule is given in the Theorem 2.1.

Theorem 2.1 **Program transformation rule**

$P=\{S_1, S_2, \dots, S_n\}$ is a program with statements $S_1, S_2, \dots, S_n$ sequence. $F_1, F_2, \dots, F_n$ are the corresponding statement flows of $S_1, S_2, \dots, S_n$ transformed with definition 2.1 to 2.8.

Set G is the program graph of P

$\Rightarrow$ $G= F_1+ F_2+ \dots,+ F_n$ is a combination of F1, F2, …, Fn

Proof:

Set $F_1, F_2, \dots, F_n$ are the corresponding statement flows of S_1,

$S_2, \ldots, S_n$ transformed with definition 2.1 to 2.8 as following:

$$\mathcal{F}_1 : S_1 \rightarrow F_1$$
$$\mathcal{F}_2 : S_2 \rightarrow F_2$$
$$\vdots$$
$$\mathcal{F}_n : S_n \rightarrow F_n$$

Define $\mathcal{F}$ **:** $P=\{S_1, S_2, \ldots, S_n\} \rightarrow G$

1)

If $\exists\, S_i$

$\ni S_i \rightarrow F_i' + F_{i+1}'$, where $F_i' + F_{i+1}' \neq F_i$ such that

$\mathcal{F}$ **:** $P=\{S_1, S_2, \ldots, S_i, \ldots, S_n\} \rightarrow G= F_1 + F_2 + \ldots, + F_i' + F_{i+1}' + \ldots, + F_n$

Since $\mathcal{F}_i : S_i \rightarrow F_i$,

Based on definition 2.1 to 2.8, it is a contradiction!

2)

If $G= F_1 + F_2 + \ldots + F_i' + \ldots, + F_n$ such that

$\exists\, S_i$ and S_{i+1}

$\ni S_i + S_{i+1} \rightarrow F_i'$, where $F_i' \neq F_i + F_{i+1}$

$\Rightarrow$ It is trivial that S_i and S_{i+1} are sequence statements

$\Rightarrow$ S_i and S_{i+1} are in the same block

$\Rightarrow$ S_i and S_{i+1} can be combined to one statement block S_i'

Then $\mathcal{F}$ **:** $P=\{S_1, S_2, \ldots, S_i, S_{i+1}, \ldots, S_n\} \rightarrow G= F_1 + F_2 + \ldots + F_i' + \ldots, + F_n$

$\Leftrightarrow$ $\mathcal{F}$ **:** $P=\{S_1, S_2, \ldots, S_i', \ldots, S_n\} \rightarrow G= F_1 + F_2 + \ldots + F_i' + \ldots, + F_n$

i.e., the program graph $G=F_1 + F_2 + \ldots, + F_n$ is a combination of F1, F2, …, Fn, and the proof is completed.

Based on the transformation rules 2.1 to 2.8 and theorem 2.1, we can decompose a program into a series of statements and transform them to a series of statement flows. The program graph of the whole program is a combination of these statement flows, then the program graph can be used to understand the program. The process of the transformation and combination will be illustrated with a real example in the next section.

3. An algorithm

In a software testing job, a number of testing paths are derived after function requirements be defined and reviewed. A testing path is derived according to the program flow, and software testers must decide what test data will be used. These jobs are processed by reviewing the program flows. If the program flow and testing paths can be provided automatically, it will help testers to test software more efficiency. In this section, a program transformation algorithm is proposed according to the transformation rules presented in the previous section.

To give the algorithm of the program transformation, we must build an instruction table, which lists the transformation rules between statement and statement flow according to definition 2.1 to 2.8. Based on the instruction table, we transform each instruction of the program to the corresponding flow. The program graph is a combination of these flows after the program is completely scanned.

The algorithm of program transformation is giving in the following.

```
algorithm
PROGRAM_TRANSFORMATION
begin
   get PROGRAM
   set START_NODE
      set NEW_NODE
      move POINTER to NEW_NODE
      while not END_OF_PROGRAM
         read next INSTRUCTION
         search INSTRUCTION_TABLE
         if INSTRUCTION = SEQUENCE_STATEMENT
            skip
         else  /* the other statement types */
            set NEW_NODE (or NODES)
      /* according to the instruction table */
            move POINTER to NEW_NODE
      /* according to the instruction table */
         end {if}
      end {while}
       set END_NODE
end { PROGRAM_TRANSFORMATION }
```

In the next section, we will illustrate the transformation approach with a program written with FoxPro language.

4. An example

The real example giving in the following is a program of a MEMBER MANAGEMENT INFORMATION SYSTEM and is written with Microsoft FoxPro language.

We scan the program and transform to

program graph with those rules illustrated in Section 2. The transformed program graph of the program is showed in the Figure 1 and the steps of building program flow are showed in Table 1.

```
1  ********** A FoxPro Program *****
2  ********************************
3  SET TALK OFF
4  PRIVATE LOP
5  STORE "F" TO LOP
6   DO WHILE LOP="F"
7  ******** CHOOSING FILE ****************
8  STORE " " TO ANS
9  CLEAR
10   @  13,30 SAY "Query..." FONT "Times New Roman",
14
11   @  17,30 SAY " Choosing file and press ENTER "
FONT "Times New Roman", 14;
12      GET ANS FONT "Times New Roman", 14
13      READ
14  FL=DBF()
15  ************ INITIALIZE **********
16  STORE "N" TO ANS
17  DO WHILE ANS<>"Y" .and. ANS<>"y"
18     STORE "  " TO YYUP
19     STORE "  " TO MMUP
20     STORE "  " TO DDUP
21     STORE "  " TO YYLOW
22     STORE "  " TO MMLOW
23     STORE "  " TO DDLOW
24  *******************
25  CLEAR
26   @   7,15 SAY "Records Setting …" FONT "Times New
Roman" , 14
27   @   9,15 SAY "Date From Year:" FONT "Times New
Roman" , 14;
28      GET YYLOW FONT "Times New Roman", 14
29      READ
30  @  12,25 SAY "Month:" FONT "Times New Roman" ,
14;
31      GET MMLOW FONT "Times New Roman", 14
32      READ
33  @  15,25 SAY "Day:" FONT "Times New Roman" , 14;
34      GET DDLOW FONT "Times New Roman", 14
35      READ
36  @  18,15 SAY "Until Year:" FONT "Times New
Roman" , 14;
37      GET YYUP FONT "Times New Roman", 14
38      READ
39  @  21,25 SAY "Month :" FONT "Times New Roman" ,
14;
40      GET MMUP FONT "Times New Roman", 14
41      READ
42  @  24,25 SAY "Day :" FONT "Times New Roman" ,
14;
43      GET DDUP FONT "Times New Roman", 14
44      READ
45  @  27,21 SAY "Are You Sure(Y/N)?" FONT "Times
New Roman" , 14;
46      GET ANS FONT "Times New Roman", 14
47      READ
48  ENDDO
49  ********************
50  STORE "N" TO ANS
51  DO WHILE ANS<>"Y" .and. ANS<>"y"
52     STORE "  " TO HB
53     STORE " " TO DV
54  *******************
55  CLEAR
56   @   7,15 SAY "Unit Code..." FONT "Times New
Roman" , 14
57   @   9,15 SAY "Hombu..." FONT "Times New Roman" ,
14;
58      GET HB FONT "Times New Roman", 14
59      READ
60   @  12,15 SAY "Division..." FONT "Times New
Roman" , 14;
61      GET DV FONT "Times New Roman", 14
62      READ
63   @  27,21 SAY "Are You Sure(Y/N)?" FONT "Times
New Roman" , 14;
64      GET ANS FONT "Times New Roman", 14
65      READ
66  ENDDO
67  *******************************
68  CLEAR
69   @ 12,40 SAY "Wait ..." FONT "Times New Roman" ,
14
70  ******************
71  SET TALK OFF
72  STORE "      " TO DUP
73  STORE "      " TO DLOWUP
74  DLOW=YYLOW+MMLOW+DDLOW
75  DUP=YYUP+MMUP+DDUP
76  ******************
77  USE \CSPS\NSFUYO.DBF
78   DELETE ALL
79   PACK
80  ********* Append ***********
81  APPEND FROM &FL FOR fuyodate>=DLOW .AND.
fuyodate<=DUP
82  ************ Start to query
*******************************
83  *************************
84  CLEAR
85   SUM FOR NSFUYO.HOMBU=HB AND
NSFUYO.DIVISION=DV TO TEST
86   @   7,40 SAY TEST PICTURE "$$,###,###,##9" ;
87             FONT "Times New Roman" , 14
88  *************************
89  ********* Continue or not ****
90  STORE " " TO ANS
91   @ 15,40 SAY "Continue (Y/N)?" FONT "Times New
Roman" , 14;
92     GET ANS FONT "Times New Roman" , 14
93     READ
94       IF ANS<>"Y" .AND. ANS<>"y"
95         STORE "T" TO LOP
96         SET TALK ON
97       ENDIF
98  ENDDO
99  **************************
100 CLOSE ALL
101 CLEAR ALL
102 RETURN
```

In Figure 1, the program graph is a combination of two While-statement flows and one If-then-end-statement flow, which satisfying the structure of the original program. With the program graph, two testing path {<1,2,3,4,5,6,7,8,9,10,11,12, 2,13>, <1,2,3,4,5,6,7,8,9,10,12,2,13>} by testing each edge are derived. This can help software testers to maintain the program more efficiently.

5. Conclusions and future works

To avoid software resource waste, software maintainers need an adequate understanding of a program's information. Usually, it is difficult to make changes for program in the absence of program structures. An experienced programmer can reconstruct program's design by recognizing data structures and algorithms. However, programmers tend to heavy rely on their experience as much as possible. We need more technologies to recognize program's design and help maintainers to modify software.

This paper presents eight typical structured statements, and proposes a number of transformation rules to reconstruct program graph. Besides, we also present a real example to illustrate and prove the methodology is practicable. The proposed methodology allows maintainers to recover the programs' structure and makes software maintenance properly.

The maintainers are under pressure to carry out the software modification as quickly as possible. The automated recognition of programs can greatly help the understanding of software and support software maintenance. The methodology proposed in this paper can help us to recognize programs automatically; this will be the next work we intend to finish.

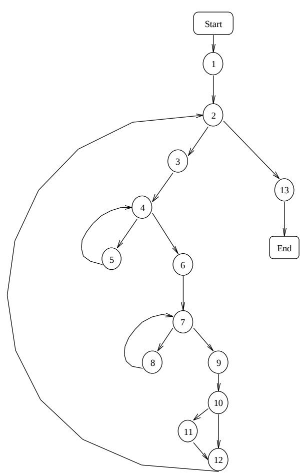


Figure 1. Program graph of the example

Line number	Non-sequence instruction	Node(s) to be set	Pointer
initial		Start node	Start node
1-5		Node 1	Node 1
6	Do while	Node 2, node3	Node 3
7-16		Skip	Node 3
17	Do while	Node 4, node 5	Node 5
18-47		Skip	Node 5
48	Enddo	Node 6	Node 6
49-50		Skip	Node 6
51	Do while	Node 7, node 8	Node 8
52-65		Skip	Node 8
66	Enddo	Node 9	Node 9
67-93		Skip	Node 9
94	If	Node 10	Node 10
95-96		Skip	Node 10
97	Endif	Node 11	Node 11
98	Enddo	Node 12	Node 12
99-102		Skip	Node 12
end		End node	End node

Table 1. The steps to build program flow

References

[1] I.D. Baxter and M. Mehlich, "Reverse engineering is reverse forward engineering," Science of Computer Programming, Vol.36, pp.131-147, 2000

[2] M.T. Harandi and J.Q. Ning, "Knowledge-based program analysis," IEEE Software, Jan., pp.74-81, 1990

[3] V. Kozaczynski, J. Ning and A. Engberts, "Program concept recognition and transformation," IEEE Tran. On S.E., Vol. 18, No.12, pp.1065-1074, 1992

[4] C. Rich and L.M. Wills, "Recognizing a program's design: a graph-parsing approach," IEEE Software, Jan., pp.82-89, 1990

[5] G. Arango, I. Baxter, P. Freeman and, C. Pidgeon, "TMM: Software maintenance by transformation," IEEE Software, May, pp.27-38, 1986

[6] I.D. Baxter, "Design maintenance systems," Comm. of the ACM, Vol.35, No.4, pp.73-89, 1992

[7] B. Biggerstaff, "Design recovery for maintenance and reuse," IEEE Computer, July, 1989

[8] S.H. Edwards, "Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential," Software Testing, Verification and Reliability, Vol. 10, pp. 249-262, 2000

[9] A. Zeller, R. Hildebrandt, "Simplifying and isolating failure-inducing input," IEEE tran. On SE, Vol.28, No. 2, pp.183-200, 2002