

## Short Paper

---

### Software Testing Methodology with the Control Flow Analysis

WEN-CHANG PAI, CHI-MING CHUNG<sup>1</sup>, CHING-TANG HSIEH<sup>2</sup>  
CHUN-CHIA WANG<sup>3</sup> AND YING-HONG WANG<sup>4</sup>

*Department of Business Mathematics  
Soochow University  
Taipei, 100 Taiwan*

*E-mail: wencpai@ms1.hinet.net*

<sup>1</sup>*The Open University of Kaohsiung  
Kaohsiung, 812 Taiwan*

*E-mail: cmchung@ms2.ouk.edu.tw*

<sup>2</sup>*Department of Electrical Engineering*

<sup>4</sup>*Department of Information Engineering*

*Tamkang University*

*Taipei, 251 Taiwan*

*E-mail: {hsieh@ee; inhon@mail}.tku.edu.tw*

<sup>3</sup>*Department of Information Management*

*Northern Taiwan Institute of Science and Technology*

*Taipei, 112 Taiwan*

*E-mail: gcwang@mail.ntist.edu.tw*

Software quality is primarily determined by the quality of the software development process. The goals of software testing are to assess and improve the quality of software. Software testing has proven to be difficult in the absence of design information. Without an adequate understanding of a program's structure, it is difficult to test it properly. Program recognition is a technology that can help testers to recover a program's design and, consequently, make software testing effective. Syntactically, a program is a sequence of statements. If the flow of the program can be recovered and used to analyze the testing paths automatically, then generating test data based on adequate testing criteria will help testers to understand the program structure and efficiently improve the software quality.

This paper provides a method for analyzing the control-flow of a program and obtaining to the original program structure. An approach to analyzing the testing paths automatically to test every branch of a program is provided. The proposed method defines a number of command types and test data generating rules. An algorithm to scan program and analysis testing paths is also provided. This will allow testers to recover a program's design and understand the software structure to assist software maintenance.

**Keywords:** software quality, software testing, program recognition, automatic testing, reverse engineering

---

Received September 3, 2003; revised February 26, 2004; accepted April 28, 2004.  
Communicated by H. Y. Mark Liao.

## 1. INTRODUCTION

Software quality is an important part of software development. Software quality assurance (SQA) is a measurement mechanism that is applied throughout the software engineering process. Software testing is the most important phase in software quality assurance. The major effort in software engineering is spent after development on maintaining systems in order to remove existing errors and to adapt the systems to changed requirements. Software testers often spend considerable energy trying to recover the design information before testing a program. Without an adequate understanding of a program's structure, it is difficult to maintain it effectively.

Program recognition and transformation is a technology that can be applied in at least three areas of 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 through functional enhancement. 3) Reverse engineering applies transformations from code to specifications.

A lot of researches on program recognition and transformation have been conducted. 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 about abstract concepts. Rich and Wills [4] built a prototype to find all the 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], uses the design histories of the code, such as the program specifications and the set of design decisions used to implement the program. They assume that the design information is available and accurate. However, such design histories of code are rarely complete and reliable. Software visualization technologies are widely used in the area of software maintenance, reverse engineering, and reengineering [6]. Software visualization is used to map from program to graphical representations. Many researchers believe in the value of software visualization for understanding software, better, reducing the amount of information to what is needed to perform, and making browsing large information systems easier.

On the other hand, generating test data automatically based on a program's structure can help testers maintain the software more efficiently. Edwards proposed a black-box testing strategy based on specification-based test data adequacy criteria [10]. He generated a flowgraph from a component's specification and applied analogues of traditional graph coverage techniques to generate black-box test sets. Offutt provided a model for developing test inputs from state-based specifications and formal criteria for test data selection [7]. They presented technologies for generating tests at several levels of abstraction for specification purposes. The levels included transition predicates, transitions, pairs of transitions, and sequences of transitions. Four different test criteria were defined in their paper. A state-based specification was viewed as a directed graph and tested based on the four different test criteria.

In this paper, a methodology for analyzing the logic of a program and obtaining the original program structure is proposed. A program essentially can be decomposed into a number of typical statements. We define a number of command types and test data gen-

erating rules based on the control-flow of the program. An algorithm for scanning the program and analyzing its testing paths is also provided. This will allow testers to recover the program's design, understand the software structure, and assist software maintenance.

In the next section, some programming terminologies are reviewed. Section 3 describes the methodological approach. A number of program transformation and test data generating rules are defined in section 4. Section 5 presents the algorithm for path analysis and test data generation. Section 6 gives an example of path analysis to illustrate the transformation process. The last section presents conclusions.

## 2. AN OVERVIEW OF PROGRAM TESTING

In a program  $P$ , we say a statement block  $B = (s_1, s_2, \dots, s_n)$  is a maximal subset of contiguous statements of  $P$  such that statement  $s_i$  is the unique successor of  $s_{i-1}$  and  $s_{i-1}$  is the unique predecessor of  $s_i$ , for all  $i = 1, 2, \dots, n$ . A program graph  $G = \{\text{nodes, edges}\}$  contains a set of nodes and edges, where a node  $n_i$  represents a block of statements of  $P$  such that there is a unique edge from  $n_i$  to  $n_{i+1}$ ,  $i = 1, 2, \dots, n - 1$ . Program graphing is a useful approach to representing the logical control-flow of a program. Maintainers can understand a program's flow by analyzing the program graph. The program graph can help the maintainers understand the structure of the program, test the program, and modify the program. A node without a predecessor is a start node. A node without a successor is an exit node. A path includes a finite contiguous sequence of nodes  $(n_1, n_2, \dots, n_k)$ ,  $k \geq 2$ , and every edge from  $n_i$  to  $n_{i+1}$  for  $i = 1, 2, \dots, k - 1$ . A complete path is a path whose initial node is the start node of  $G$  and whose final node is an exit node of  $G$ .

Testing strategies include White-box testing and Black-box testing [8]. Black-box testing generates test cases without considering the internal structure of the program, only with instances when the program does not execute as indicated in its functional specifications. In White-box testing, the structure of the program is examined, and test cases are derived from the program's logic. A set of testing paths is said to satisfy the all-paths criterion if these testing paths include every complete path of the program graph. The ideal White-Box testing test case set is one that exhaustively executes all the control-flow paths in a program, which is virtually impossible. There are a number of criteria for determining the coverage of a program's logic. A well known logic coverage criterion is decision coverage or branch coverage [9]. Examples of branch or decision statements are IF statements, WHILE statements, and SWITCH statement. This criterion states that each branch direction must be traversed at least once.

In software testing technologies, boundary testing (or boundary value analysis; BVA) is a good test case design. It selects test data at the boundary of the input domain. Experience shows that test data selected with BVA have a higher payoff than other data. More errors tend to occur at the boundaries of the input domain. The guidelines for boundary-value analysis are as follows [16]:

- If an input specifies a range of valid values, write test cases for the ends of the range and invalid-input test cases for conditions just beyond the ends.

- If an input specifies a number of valid values, write test cases for the minimum and maximum number of values and one beneath and beyond these values.
- Use the above guidelines for each output condition.

In the general sense, the transformation of a program is viewed as a process of re-writing one program to obtain another through repeated application of a set of transformation rules. Since a program is a combination of statements (or commands), we categorize a program's statements according to three typical command types: (1) sequential commands, (2) conditional commands, and (3) loop commands. These three command types include eight typical statements in a modern structured language. The eight statements are: (1) sequence statements, such as READ, WRITE, 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. Based on the three commands types, the control-flow of a program can be analyzed and test data generated. The method generates test data automatically and helps testers to verify the software properly. We will describe the method in the next section.

### 3. AUTOMATICALLY GENERATING TEST DATA

Our proposed method involves program recognition and transferring to program structure. With the three command types, we analyze the program graph and generate test data based on the control-flow of the program. The results of path analysis and test data generation help testers maintain the software. The steps involved in the method are described in the following.

#### 1) Choosing Adequacy Testing Criteria

To generate test data, choosing an adequate testing criterion is the most important prerequisite. A number of testing path selection criteria have been proposed. In practice, All-Paths testing is impossible since some infeasible paths exist. Since most failures occur in the branches of a program, the branch coverage criterion is adopted in this paper to generate test data for testing a program.

#### 2) Defining Command Types

Since a program is a combination of statements, we can decompose a program into a set of statements. We define three typical statement types in a program: (1) sequential commands, (2) conditional commands, and (3) loop commands. Each command type essentially corresponding to a block or some blocks in a program. In this paper, we will generate test data for each of the command types to enter every branch of a program.

#### 3) Scanning the Program and Generating Test Data

With the test data generating rules defined in step 2, the program is scanned. The test data are generated from the input domain, which is derived through boundary value analysis with the Branch coverage criterion.

#### 4) Analyzing Test Paths and Testing

Finally, testers analyze the testing paths and test the program with the results obtained in step 3.

### 4. TEST DATA GENERATING RULES

In software testing, test data are employed to enter the statement type and to find the errors in a program. Since a program is a combination of statements, we can decompose it into a set of statements. The decomposed statements are of three command types: (1) sequential commands, (2) conditional commands, and (3) loop commands. In this section, we will define test data generating rules for each of the command types.

#### Type 1. Sequential Command Set

The Sequential commands, such as OPEN, READ, WRITE, and CLOSE statements, are usually written in the form

$\langle C_1; C_2 \rangle$ ,

where command  $C_1$  is executed before command  $C_2$ . Since a number of contiguous sequential commands correspond to one block in a program, they are in the same node of the program graph. Fig. 1 shows an example of the flow. With the branch coverage criterion, it is not necessary to generate test data for the node.

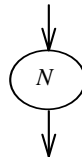


Fig. 1. The flow of sequential commands.

#### Rule 4.1

The set  $S_1, S_2, \dots, S_n$  is a contiguous sequence of statements of a program  $P$ , such that a corresponding block  $B$  exists in  $P$ . A node  $N$  also exists in the program graph  $G$ .

- 1)  $B$  corresponds to  $N$ .
- 2) Test data are not needed to generate this command type based on BVA.

#### Type 2. Conditional Command Set

A conditional command, such as an If-Then-End statement, If-Then-Else-End statement, Switch-Case-With-Default statement, or Switch-Case-Without-Default statement, has a number of subcommands, from which exactly one is chosen to be executed. Conditional commands typically have the form

```

If  $CON_1$  then  $B_1$ 
Else if  $CON_2$  then  $B_2$ 
...
else if  $CON_n$  then  $B_n$ 
end if

```

Each of  $B_1, B_2, \dots, B_n$  contains a number of subcommands. Whether  $B_i$  is chosen to be executed depends on whether  $CON_i$  is true. Fig. 2 shows the structure of the type. Through the branch coverage criterion and boundary value analysis, test data are generated from the boundary value of  $CON_i$ .

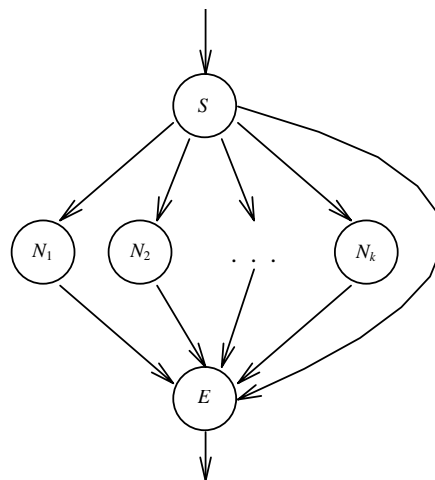


Fig. 2. The flow of conditional commands.

#### Rule 4.2

Set  $S'$  is a set of Switch statements of a program  $P$ , such that a number of corresponding blocks  $B_1, B_2, \dots, B_k$  exist in  $S'$ . A number of nodes  $S, N_1, N_2, \dots, N_k$ , and  $E$  also exist in the program graph  $G$ .

- 1) SWITCH statement  $S'$  corresponds to node  $S$ .
- 2)  $B_1, B_2, \dots, B_k$  correspond to  $N_1, N_2, \dots, N_k$ , respectively.
- 3) The END\_OF\_SWITCH statement corresponds to  $E$ .
- 4) The test data are the bounded values of the conditions in the SWITCH statement.

#### Type 3. Loop Command Set

Loop commands, such as For-loop statements, While-loop statements, and Repeat-loop statements, have a number of subcommands that are executed repeatedly until some conditions are true. Loop commands typically have the form

```

While CON do
  B
End while

```

Block  $B$  contains a number of subcommands. Whether  $B$  is chosen to be executed depends on whether CON is true. Another Loop command form is

```

Repeat
  B
Until CON

```

Here,  $B$  is chosen to be executed repeatedly until CON is true. Fig. 3 shows the typical flow of Loop commands. Through the branch coverage criterion and boundary value analysis, test data are generated from the boundary value of CON.

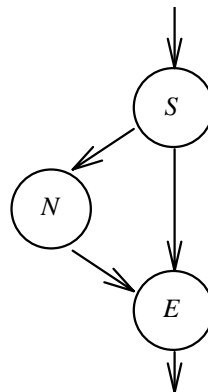


Fig. 3. The flow of loop commands.

#### Rule 4.3

Set  $S'$  is a Loop statement  $s$  of a program  $P$ , such that a corresponding block  $B$  exists in  $P$ . A number of nodes,  $S$ ,  $N$ , and  $E$ , also exist in the program graph  $G$ .

- 1) LOOP statement  $S'$  corresponds to node  $S$ .
- 2)  $B$  corresponds to  $N$ .
- 3) The END\_OF\_LOOP statement corresponds to  $E$ .
- 4) The test data are the bounded values of the conditions in the LOOP statement.

Syntactically, a structured program is a combination of statements of the command types mentioned above. A program can be analyzed by treating it as a combination of statements. This combination is divided into two parts, the Link and Nest, which are defined in the following.

**Definition 4.1 Link** Statement type  $T_1$  links statement type  $T_2$  if and only if the combination formed by  $T_1$  and  $T_2$  is  $(T_1, T_2)$ ; that is,  $T_1$  is the predecessor statement type of  $T_2$ , and  $T_2$  is the successor statement type of  $T_1$ , denoted as  $T_1 + T_2$ .

**Definition 4.2 Nest** Statement type  $T_1$  nests statement type  $T_2$  if and only if, in the combination formed by  $T_1$  and  $T_2$ ,  $T_2$  is contained in  $T_1$ ; that is,  $T_1$  contains  $T_2$  wholly, which is denoted as  $T_1 > T_2$ .

Two combination types of two statements,  $T_1$  and  $T_2$ , are  $T_1 + T_2$  and  $T_1 > T_2$ . In  $T_1 + T_2$ ,  $T_2$  is a *node* of  $T_1$ . Similarly,  $T_2$  is a node of  $T_1$  in  $T_1 > T_2$ . Rules 4.1 to 4.3 can be applied under these conditions.

We can transform a program into a program graph. The test data can be generated automatically with the rules mentioned above. This will help testers work more efficiently. The program transformation rule is given in Theorem 4.1.

**Theorem 4.1 Program transformation rule**  $P = \{S_1, S_2, \dots, S_n\}$  is a program with statement sequence  $S_1, S_2, \dots, S_n$ .  $F_1, F_2, \dots, F_n$  are the corresponding command type flows of  $S_1, S_2, \dots, S_n$  transformed with rules 4.1 to 4.3.

Set  $G$  is the program graph of  $P$ .

$\Rightarrow G = F_1 + F_2 + \dots + F_n$  is a combination of  $F_1, F_2, \dots, F_n$ .

**Proof:** Set  $F_1, F_2, \dots, F_n$  comprises the corresponding statement flows of  $S_1, S_2, \dots, S_n$  transformed with rules 4.1 to 4.3 as follows:

$$\begin{aligned} \bar{F}_1 : S_1 &\rightarrow F_1 \\ \bar{F}_2 : S_2 &\rightarrow F_2 \\ &\vdots \\ \bar{F}_n : S_n &\rightarrow F_n \end{aligned}$$

Define  $\bar{F} : P = \{S_1, S_2, \dots, S_n\} \rightarrow G$ .

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

$\bar{F} : P = \{S_1, S_2, \dots, S_i, \dots, S_n\} \rightarrow G = F_1 + F_2 + \dots + F'_i + F'_{i+1} + \dots + F_n$

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

based on rules 4.1 to 4.3, this is a contradiction!

2) If  $G = F_1 + F_2 + \dots + F'_i + \dots + F_n$  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 form one statement block  $S'_i$

$$\begin{aligned} \text{Then } F: P = \{S_1, S_2, \dots, S_i, S_{i+1}, \dots, S_n\} &\rightarrow G = F_1 + F_2 + \dots + F_i' + \dots + F_n \\ \Leftrightarrow F: P = \{S_1, S_2, \dots, S_i', \dots, S_n\} &\rightarrow G = F_1 + F_2 + \dots + F_i' + \dots + F_n; \end{aligned}$$

i.e., the program graph  $G = F_1 + F_2 + \dots + F_n$  is a combination of  $F_1, F_2, \dots, F_n$ , and the proof is completed.

Based on rules 4.1 to 4.3 and Theorem 4.1, we can decompose a program into a series of statements and transform them a series of command type flows. The program graph of the program is the combination of these command type flows, and the program structure can be used to verify the program. The transformation and test data generation processes will be illustrated with an example in the following sections.

## 5. TEST PATH ANALYSIS ALGORITHM

In a software testing job, a number of testing paths are derived after functional requirements are 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, this will help the testers to test the software more efficiently. In this section, a test path analysis algorithm is proposed based on the transformation rules presented in the previous section.

To present the algorithm for the program transformation, we must first build an instruction table, which lists the transformation rules between statement and statement flows according to rules 4.1 to 4.3. Based on the instruction table, we can transform each statement of a program to its corresponding flow. The paths are analyzed and test data are generated after the program has been completely scanned.

The algorithm for program transformation is as follows:

Algorithm PATH\_ANALYSIS

```

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
    switch (INSTRUCTION)
      case "Switch Statement Set"
        set NEW_NODE (or NODES) /* according to the rule 4.2 */
        move POINTER to NEW_NODE /* according to the rule 4.2 */
      case "Loop Statement Set"
        set NEW_NODE (or NODES) /* according to the rule 4.3 */
        move POINTER to NEW_NODE /* according to the rule 4.3 */

```

```

        case "Sequence Statement Set" /* the sequential commands type */
            skip
        end {switch}
    end {while}
    set END_NODE
end {PATH_ANALYSIS}

```

## 6. AN EXAMPLE

A typical program with 11 blocks is considered in the following:

```

BLOCK 1
  WHILE LOP = "F" DO                                'generate BLOCK 2
    BLOCK 3
    WHILE ANS<>"Y" .and. ANS<>"y" DO                'generate BLOCK 4
      BLOCK 5
    END_WHILE
    BLOCK 6
    WHILE ANS<>"Y" .and. ANS<>"y" DO                'generate BLOCK 7
      BLOCK 8
    END_WHILE
    BLOCK 9
    IF ANS<>"Y" .and. ANS<>"y"                        'generate BLOCK 10
      BLOCK 11
    END_IF
  END_WHILE
BLOCK 13

```

The program consists of three Loop commands and one Conditional command. Through the branch coverage criterion and boundary value analysis, we scanned the program with the algorithm provided in section 5. The data structure of the paths and the domains of the input values are shown in Table 1. The program graph is shown in Fig. 4.

From Table 1, we have the following testing paths:

```

<S, 1, 2, 13, E>
<S, 1, 2, 3, 4, 5, 4, 6, 7, 8, 7, 9, 10, 11, 12, 2, 13, E>
<S, 1, 2, 3, 4, 5, 4, 6, 7, 8, 7, 9, 10, 12, 2, 13, E>
<S, 1, 2, 3, 4, 6, 7, 8, 7, 9, 10, 11, 12, 2, 13, E>
<S, 1, 2, 3, 4, 6, 7, 8, 7, 9, 10, 12, 2, 13, E>
<S, 1, 2, 3, 4, 5, 4, 6, 7, 9, 10, 11, 12, 2, 13, E>
<S, 1, 2, 3, 4, 5, 4, 6, 7, 9, 10, 12, 2, 13, E>
<S, 1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 2, 13, E>
<S, 1, 2, 3, 4, 6, 7, 9, 10, 12, 2, 13, E>

```

Table 1. A matrix showing the data structure of paths and test data.

TO FROM	S	1	2	3	4	5	6	7	8	9	10	11	12	13	E
S		*													
1			*												
2				*										*	
3					*										
4						*	*								
5															
6								*							
7									*	*					
8									*						
9											*				
10												*	*		
11													*		
12			*												
13															*
E															

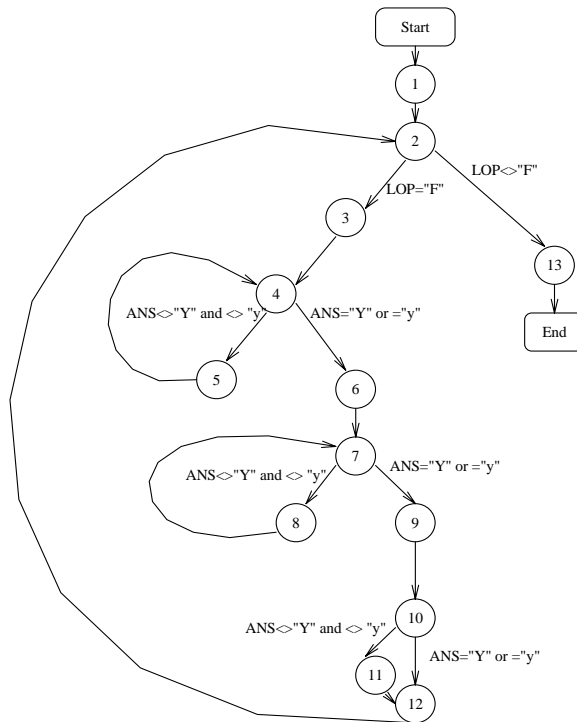


Fig. 4. Program graph of the example.

From testing path  $\langle S, 1, 2, 13, E \rangle$ , for example, the input domain is  $LOP \neq "F"$ , and test data can be any characters except "F". Similarly, From testing path  $\langle S, 1, 2, 3, 4, 6, 7, 9, 10, 12, 2, 13, E \rangle$ , the input domains are  $LOP = "F"$ ,  $ANS = "Y"$ , or  $= "y"$ ,  $ANS = "Y"$  or  $= "y"$ , and  $ANS = "Y"$  or  $= "y"$ . We can test this path by using input  $LOP = "F"$ , and making  $ANS$  equal to "Y" or "y" with three times.

## 7. CONCLUSIONS

Maintainers are always under pressure to perform software modification as quickly as possible. Automatic recognition of programs and generation of test data can greatly help maintainers understand software and perform software maintenance.

This paper has proposed a method for recognizing and generating test data automatically. We have defined a number of command types flows and given some test data generation rules. These have been derived based on a branch coverage testing path selection criterion and boundary value analysis. We have also provided an algorithm and used an example to illustrate the methodology and show that it is practicable. The proposed methodology allows maintainers to recover a program's structure and conduct software maintenance. The method proposed in this paper can help testers recognize and test programs more efficiently.

## REFERENCES

1. I. D. Baxter and M. Mehlich, "Reverse engineering is reverse forward engineering," *Science of Computer Programming*, Vol. 36, 2000, pp. 131-147.
2. M. T. Harandi and J. Q. Ning, "Knowledge-based program analysis," *IEEE Software*, 1990, pp. 74-81.
3. V. Kozaczynski, J. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, Vol. 18, 1992, pp. 1065-1074.
4. C. Rich and L. M. Wills, "Recognizing a program's design: a graph-parsing approach," *IEEE Software*, 1990, pp. 82-89.
5. G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: software maintenance by transformation," *IEEE Software*, 1986, pp. 27-38.
6. R. Koschke, "Software visualization in software maintenance, reverse engineering, and reengineering: a research survey," *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 15, 2003, pp. 87-109.
7. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, Vol. 13, 2003, pp. 25-53.
8. W. E. Howden, *Introduction to Software Validation, Software Testing and Validation Techniques*, 2nd ed., IEEE Computer Society Press, 1981.
9. G. J. Myers, *The Art of Software Testing*, Wiley and Sons, New York, 1979.
10. S. H. Edwards, "Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential," *Software Testing, Verification and Reliability*, Vol. 10, 2000, pp. 249-262.

11. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, Vol. 28, 2002, pp. 183-200.
12. D. A. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, New York, 1990.
13. S. Zweben, W. Heym, and J. Kimmich, "Systematic testing of data abstractions based on software specifications," *Software Testing, Verification and Reliability*, Vol. 1, 1992, pp. 39-55.
14. T. J. Biggerstaff, "Design recovery for maintenance and reuse," *IEEE Computer*, Vol. 7, 1989, pp. 36-49.
15. I. D. Baxter, "Design maintenance systems," *Communications of the ACM*, Vol. 35, 1992, pp. 73-89.
16. E. Kit, *Software Testing in the Real World: Improving the Process*, Addison-Wesley, New York, 1995
17. P. Stocks and D. Carrington, "A framework for specification-based testing," *IEEE Transaction on Software Engineering*, Vol. 22, 1996, pp. 777-793.
18. E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transaction on Software Engineering*, Vol. 20, 1994, pp. 353-363.
19. G. Laycock, "Formal specification and testing: A case study," *Software Testing, Verification and Reliability*, Vol. 2, 1992, pp. 7-23.
20. S. Yau and P. C. Grabow, "A model for representing program using hierarchical graphs," *IEEE Transactions on Software Engineering*, Vol. 7, 1981, pp. 556-574.
21. S. Letovsky and E. Soloway, "Delocalized plans and program comprehension," *IEEE Software*, Vol. 3, 1986, pp. 41-49.
22. J. Y. Cai, "Optimal stopping of multi-project software testing in the context of software cybernetics," *Science in China Series F-Information Sciences*, Vol. 46, 2003, pp. 335-354.
23. Y. S. Dai, M. Xie, K. L. Poh, and B. Yang, "Optimal testing-resource allocation with genetic algorithm for modular software systems," *Journal of Systems and Software*, Vol. 66, 2003, pp. 47-55.
24. N. Morali and R. Soyer, "Optimal stopping in software testing," *Naval Research Logistics*, Vol. 50, 2003, pp. 88-104.
25. Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, Vol. 51, 2002, pp. 420-426.

**Wen-Chang Pai (白文章)** is a Professor in the Department of Business Mathematics at Soochow University in Taipei, Taiwan. He received his B.S. degree in Business Mathematics from Soochow University and received his M.S. degree in Industrial Management from National Taiwan University of Science and Technology. He received his Ph.D. degree in Information Engineering in 1993 from Tamkang University, Taipei, Taiwan. His research interests include software testing, software quality, management information systems, and data mining.

**Chi-Ming Chung (莊淇銘)** was born in 1956. He received the B.S. degree in Environment Engineering from National Cheng Kung University, and the M.S. degree in Computer Sciences from the University and Ph.D. degree in Computer Sciences from the University of Southwest Louisiana, in 1992 and 1996, respectively. From 1996 to 2001, he was an Associate Professor and Professor in the Department of Information Engineering of Tamkang University. From 2000 to 2001, he was the President at Kainan University. Beginning 2002, he is the President at the Open University of Kaohsiung. His current research interests are knowledge management, creativeness development, effective learning, and software engineering.

**Ching-Tang Hsieh (謝景棠)** is an Associate Professor of Electrical Engineering at Tamkang University, Taiwan, R.O.C. He received the B.S. degree in Electronics Engineering in 1976 from Tamkang University and the M.S. and Ph.D. degree in 1985 and 1988, respectively, from Tokyo Institute of Technology, Japan. From 1990 to 1996, he acted as the Chairman of the Department of Electrical Engineering. His current research interests include speech analysis and synthesis, speech recognition, image inpainting and mosaic, object detection and tracking, fingerprint identification and classification, and watermarking.

**Chun-Chia Wang (王俊嘉)** was born in 1966. He received his M.S. and Ph.D. degrees in Computer Science from TamKang University in 1994 and 1997, respectively. He is now an Associate Professor in Department of Information Management at Kuang Wu Institute of Technology. He has also been a Chairman of Department of Information Management at Northern Taiwan Institute of Science and Technology in 2000. His research interests include software engineering, object-oriented technology, multimedia, and e-commerce.

**Ying-Hong Wang (王英宏)** received the B.S. degree in Information Engineering from Chung Yuan Christian University, Taiwan, in 1986 and the M.S. and Ph.D. degrees in Information Engineering from the TamKang University, in 1992 and 1996, respectively. From 1988 to 1990, he worked in the Product Development Division of Institute of Information Industry (III). From 1992 to 1996, he was a lecturer in the Department of Information Engineering of TamKang University. Beginning Fall 1996, he is an Associate Professor in the Department of Information Engineering of TamKang University. He has over 100 technological papers published on International journals and International conference proceedings, he also join many International activities been program committee, workshop chair, session chair and so on. He had been invited as Visiting Researcher in The University of Aizu, Japan, from January to March 2002. His current research interests are software engineering, multimedia database system, wireless multimedia, and mobile agent.