Volume 4, Number 2 April 1990

EDITORIAL BOARD

COLUMN EDITORS

QUALITY DATA PROCESSING (Copyright © 1990) is published quarterly by Quality Assurance Institute (QAI) with editorial and executive offices at Suite 350, 7575 Dr. Phillips Blvd., Orlando, FL 32819 [(407) 363-1111].

QAI members receive four issues per year at no additional cost, but may order additional subscriptions at $43 each; subscription price to nonmembers is $50; larger quantities available at reduced rates.

Individual copies of back issues are available at $10 each; write or call QAI.

Reprints of 500 or more of articles appearing in the journal are available at low cost. Requests for reprints or permission to reproduce may be made by writing or phoning QAI.

Articles, letters to the editor, advertising, and suggestions are welcome. Address correspondence to QAI offices.

TABLE OF CONTENTS

STRUCTURAL SOFTWARE TESTING: THE COMPLEXITY-BASED APPROACH

Eldon Y. Li, California Polytechnic State University

INTRODUCTION

Software testing techniques are traditionally classified into "functional" and "structural" techniques based on their methods of deriving test cases [Adrion, Branstad and Cherniavsky, 1982]. The functional testing techniques derive the test cases from the requirements definition or the external (design) specification, while the structural techniques derive them from the program logic in the source code or internal design specification. The former techniques focus on the functions of the program/system being tested, the latter, on the structure. Therefore, they are also known respectively as the "black-box" and the "white-box" techniques [Myers, 1979, pp. 8-9]. Among the existing structural techniques, the complexity-based coverage technique developed by McCabe [1976] was deemed superior [Li, 1988]. Yet, there are still weaknesses to be rectified. This paper reviews the complexity-based test technique, and discusses the strengths and weaknesses of the technique.

COMPLEXITY-BASED TESTING TECHNIQUE

In his landmark paper, Thomas J. McCabe [1976] proposed the use of the "cyclomatic number" [Berge, 1973] (also called "cycle rank" [Harary, 1969] or "nullity" [Deo, 1974]) in the graph-theory literature to measure the control-flow complexity of a program. Such a measure is known as the "cyclomatic complexity metric." In the same paper, he further proposed two other metrics. One measures the "unstructuredness" of a program, and the other indicates the number of independent paths actually executed by a program running on a test data set. The former is called the "essential complexity metric" while the latter is the "actual complexity metric." The cyclomatic complexity metric V(G) of a program was defined as one plus the number of conditions in the program [McCabe, 1976; Li, 1987]. The essential complexity metric was defined as the cyclomatic complexity minus the number of proper subgraphs with unique entry and exit nodes. McCabe [1976] demonstrated that every structured program can be reduced to a program of unit cyclomatic complexity; i.e., the "essential complexity" equals to one. He further proved that the cyclomatic complexity of an unstructured program is at least three. This is, any program whose cyclomatic complexity equals two is, or can be modified into, a structured program.

According to McCabe, the cyclomatic metric determines the minimal set of required test paths. Each test path represents a test case. The program under test must have a single entry and a single exit. The actual complexity should be maintained as close to the cyclomatic complexity as possible. This latter objective may be attained by finding test data which covers more paths, or by restructuring the program into a program with less complexity (i.e., eliminating the unnecessary decision conditions in a program) and reducing portions of the program to in-line code. To maintain the testability of a program, McCabe suggested that the cyclomatic number of a program should have an upper bound of 10 [McCabe, 1976, p. 314]. Otherwise, the number of possible paths of the program may easily become

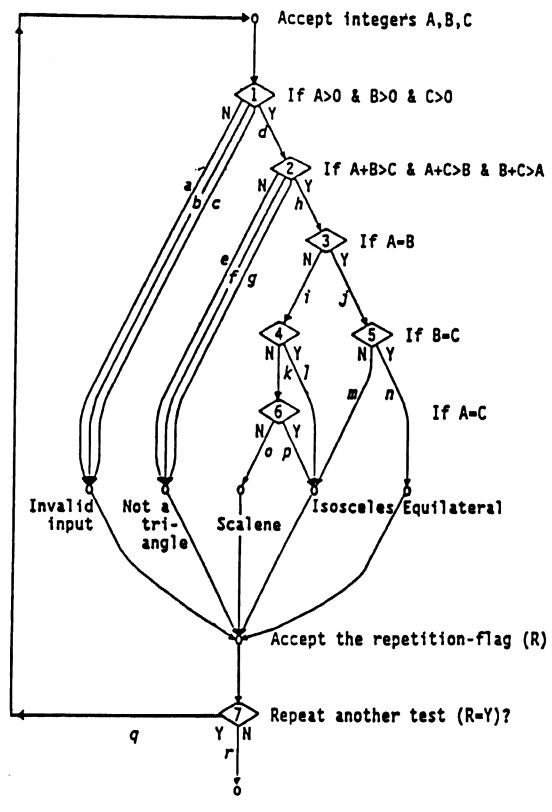


Figure 1: A Control-Flow Graph for the Triangle Program

Continued

unmanageable. Later, in 1983, he demonstrated a five-step structured process for deriving the test cases directly from the control-flow graph of the intended program [McCabe, 1983, pp. 29-30]. If the cyclomatic complexity of a program is N, the process will generate N distinct independent paths which traverse every edge in the program graph. This structured process will be demonstrated with several modifications made by the author, using the control-flow graph adapted from Li [1988] as shown in Figure 1. This graph, representing the triangle program adapted from Myers [1979, p. 1], was drawn by McCabe's [1983] convention which uses multiple branches to represent the true/false outcomes of a decision with multiple conditions. For example, decision 1 in Figure 1 has one "Y" (true) branch and three "N" (false) branches. The former branch represents the outcome "TTT," while the latter three represent the outcomes "F**," "TF*," and "TTF." The "*" sign indicates that the outcome of a condition can be either true or false because it does not affect the outcome of the entire decision. Each decision branch in Figure 1 is labeled with an alphabet, starting at "a" and then "b" through "z." Each decision node is labeled with an integer starting at "1" and then "2" through "0." All outcome branches of each decision should be labeled consecutively. If the upper bound of the cyclomatic number was closely observed, a program should not have more than 9 decision nodes or 18 decision branches. Therefore, this labeling convention should provide enough labels for all the decision nodes and branches in a program. In fact, it provides 10 labels for the decision nodes and 26 for the decision branches.

STRUCTURED PROCESS FOR COMPLEXITY-BASED TESTING

Step 1

Pick a functional "baseline" path through the program which represents a legitimate function and not just an error exit. The key is to pick a path that performs the major full function provided in the program and intersects a maximal number of decisions in the graph, as opposed to an error path that results in an error message or recovery procedure. For example, path 1d2h3i4k6p7r is a possible baseline. Note that our path expression is somewhat different than that of McCabe [1983] in which the decision number does not appear.

Step 2

Identify the second path by locating the first decision on the baseline and flipping its outcome while simultaneously holding the maximum number of the original baseline decisions unchanged. The word "flip" means to change the decision outcome (or branch) from one value to another, i.e., to take on another decision branch. If the decision has multiple conditions, each condition should be flipped one at a time. This process is likely to produce a second path which is minimally different from the baseline path. The result yields three paths: ~1a7r, ~1b7r, and ~1c7r. We use the symbol "~" to indicate that the decision behind the symbol has been flipped.

Step 3

Set back the first decision to its original value before the flipping, identify the second decision in the baseline path, and flip its outcome while holding all other decisions to their baseline values. This process, likewise, should produce a third path which is minimally different from the baseline path. The result yields another three paths: 1d~2e7r, 1d~2f7r, and 1d~2g7r.

Step 4

Repeat the above procedure until one has gone through every decision on the baseline and has flipped it from the baseline value while holding the other decisions to their original baseline values. After flipping the third decision, we have the path 1d2h~3j5m7r. Flipping the fourth decision yields the path 1d2h3i~417r; the sixth decision yields the path 1d2h3i4k~6o7r; the seventh decision yields 1d2h3i4k6p(~7q1d2h3i4k6p)7r. The parenthesized segment on the last path represents the boundary and the interior decisions of the loop. Note that McCabe did not provide any guideline for selecting the path inside the loop. Yet, he did indicate one should avoid picking an error path that results in an error message or recovery procedure unless there is no other choice, and that the baseline path plus the paths resulting from flipping all the decisions in a program graph should equal the cyclomatic number. Based on these rules, the author decided that:

1) the baseline path ~7q1d2h3i4k6p should be taken when the loop decision 7 was flipped,
2) if the path encounters a new loop inside a loop, all new decisions (including the new loop decision) inside the new loop should be flipped, and
3) any decision which has been flipped completely should not be flipped again; otherwise, the total number of paths generated will be larger than the cyclomatic number.

Step 5

Repeat the above procedure for any unflipped decision which is not on the baseline. Once all the decisions have been flipped, the process is then completed. In this case, the fifth decision encountered in Step 4 must be flipped. Flipping the fifth decision yields the path 1d2h~3j~5n7r.

In total, this process generated the 12 test paths listed below. These 12 paths cover all possible combinations of condition outcomes in each decision, as well as all point of entry in a program, therefore meeting the criteria for the "multiple-condition coverage" [Myers, 1979, p. 42].

1) 1d2h3i4k6p7r (baseline)
2) ~1a7r
3) ~1b7r
4) ~1c7r
5) 1d~2e7r
6) 1d~2f7r
7) 1d~2g7r
8) 1d2h~3j5m7r

Continued

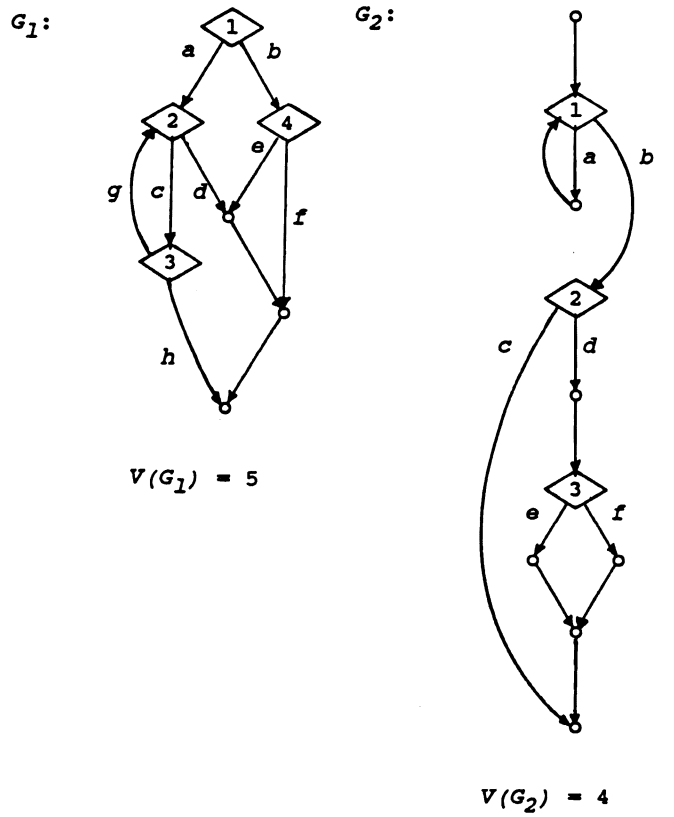


Figure 2: The Control-Flow Graphs for Two Other Examples

9) 1d2h3i~4l7r
10) 1d2h3i4k~6o7r
11) 1d2h3i4k6p(~7q1d2h3i4k6p)7r
12) 1d2h~3j~5n7r

SOME OTHER EXAMPLES

Two other examples will be demonstrated below. The first example is based on the graph G_1 of Figure 2. This graph was adapted from McCabe [1983, pp. 23-24]. Applying the above process to this graph, one may identify the following five paths:

1) 1a2c(3g)2d (baseline)
2) ~1b4e
3) 1a~2d
4) 1a2c~3h
5) ~1b~4f

Alternatively, another five paths may look like:

1) 1a2d (baseline)
2) ~1b4e
3) 1a~2c3h
4) ~1b~4f
5) 1a~2c(~3g)2d

Note that in both cases, decision 2 appeared twice in one of the five generated paths. Only the first decision 2 was flipped in each case.

Similarly, one may find four paths for the second graph, G_2, in Figure 2 as follows. This graph was adapted from Schneidewind [1979, p. 990].

1) (1a)1b2d3e (baseline)
2) ~1b2d3e
3) (1a)1b~2c
4) (1a)1b2d~3f

or alternatively,

1) 1b2c (baseline)
2) (~1a)1b2c
3) 1b~2d3e
4) 1b~2d~3f

DISCUSSION

Note that the above process does not yield one unique set of test paths. Different testers might choose different baselines to start the process, therefore producing different sets of test paths. Such differences typically lie in the different combinations of condition outcomes between the decisions in a program. Moreover, to avoid selecting an impossible path (a path that is not executable because some decisions on the path are masked by another decision or by some foregoing (nondecision) program segments), one must annotate the program graph with the program source code or pseudocode. Yet, this requirement is unique to the complexity-based technique; it is common to all other structural testing techniques.

Although the complexity-based testing technique can be applied to a program (or pseudocode) of any size and any programming style, it is more effective if the target program has a cyclomatic number of no more than 10 and is coded in a structured-programming style. By limiting the cyclomatic number of a program at 10, at most 10 test paths will be generated by the complexity-based process, making the generation of test cases and test data more manageable. By coding a program in a structured-programming style, the cyclomatic number will equal the number of independent paths in the program, making the testing more complete. Due to the fact that an unstructured program usually gives a lower cyclomatic number than what it really has, the author would like to caution all prospective users of the complexity-based testing technique to closely observe the structured programming principles, such as: 1) single entry and exit; 2) no GOTO branch; 3) the use of structured constructs; 4) modularization; etc. [Bohm and Jacopini, 1966; Dijkstra, 1968, 1970; Mills, 1972; McCabe, 1976]. Otherwise, the test paths generated by the complexity-based process might not cover all the independent paths of the intended program.

"THE MAJOR STRENGTH OF THE COMPLEXITY-BASED TECHNIQUE LIES IN ITS WELL-STRUCTURED AND EASY-TO-PERFORM PROCESS OF DERIVING TEST PATHS AS DESCRIBED."

STRENGTHS AND WEAKNESSES OF COMPLEXITY-BASED TESTING

The major strength of the complexity-based technique lies in its well-structured and easy-to-perform process of deriving test paths as described above. This

Continued

process can generate a set of test paths/cases which functionally meets the criteria required by the multiple-condition coverage. Yet, complexity-based coverage is superior to the multiple-condition coverage because the former further explores some (but not all) possible combinations of condition outcomes between any pair of serial decisions (one follows the other) while the latter does not.

A weakness of the complexity-based technique is that it does not meet all the criteria for testing a loop required by a method proposed by Howden [1975], known as the "boundary-interior" method. This method requires that every loop in a program be tested with 0 entry (i.e., skip the loop), exactly 1 entry (i.e., no iteration), and 2 or more entries (i.e., 1 or more iterations). The complexity-based technique only covers the first two criteria, but not the last one. This weakness may be attributed to the underlying assumption of the cyclomatic metric in which a loop construct is considered to have the same complexity as an "IF" construct given they both have the same number of conditions in their decisions. Furthermore, the technique does not fully cover the boundary values of each condition in a decision. For example, given the decision "IF A=B," the process will only cover the two outcomes of "A=B" and "A≠B," rather than the three outcomes of "A=B," "A<B," and "A>B." The latter coverage is known as the "boundary-value analysis" [Myers, 1979, pp. 50-55]. As indicated by McCabe, the complexity-based process would only identify the minimal number of independent paths that should be tested; there are often additional paths to test [McCabe, 1976, p. 318]. Often more than one test must be performed on a path [McCabe, 1983, p. 29] to fully cover all the functional requirements. Recently, Li [1988] demonstrated that the test paths/cases derived by the complexity-based process may not perfectly match those derived by a functional technique, therefore suggesting the complexity-based technique be supplemented by the functional techniques such as equivalence partitioning, boundary value analysis, and cause-effect graphing [Myers, 1979, pp. 44-73].

"...BEFORE AN IMPROVED METHOD IS PROPOSED, THE COMPLEXITY-BASED TECHNIQUE IS BY FAR THE MOST EASY TO USE AND EFFECTIVE TECHNIQUE AMONG THE EXISTING STRUCTURAL TESTING TECHNIQUES."

CONCLUSION

The complexity-based testing technique provides a structured process to derive test paths and a measure of testability (or complexity) to enforce structured-programming practices. By following the process provided V (the cyclomatic number) distinct independent paths will be generated. These paths will traverse each and every edge (branch) in the program graph at least once (i.e., each and every statement in the program will be executed at least once). And, each and every condition in a decision will take on its true-false outcomes at least once. The result therefore conforms to the criteria for structural test coverage as recommended by Miller [1977] and Myers [1979].

However, there are several major weaknesses of complexity-based techniques which call for further improvement of this technique. These weaknesses are as follows:

1) There may be several sets of test paths generated by the process.
2) The path-finding process does not explore all the possible combinations of condition outcomes between every pair of serial decisions.
3) It does not meet the "boundary-interior" criteria for loop testing.
4) It does not fully cover the boundary values of each condition in a decision.
5) It only identifies the minimal number of independent paths that should be tested. No guideline has been provided to derive additional test paths to fully cover all the functional requirements.

Yet, given the above weaknesses, and before an improved method is proposed, the complexity-based technique is by far the most easy to use and effective technique among the existing structural testing techniques.

REFERENCES

Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C. - "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, Volume 14, Number 2, June 1982, pp. 159-192.

Berge, C. - *Graphs and Hypergraphs*, North-Holland, Amsterdam, The Netherlands, 1973, pp. 15-17.

Bohm, C., and Jacopini, G. - "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," *Communications of the ACM*, Volume 9, Number 5, May 1966, pp. 366-371.

Deo, N. - *Graph Theory With Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974, pp. 55-58.

Dijkstra, E.W. - "Go To Statement Considered Harmful," *Communications of the ACM*, Volume 11, Number 3, March 1968, pp. 147-148.

Dijkstra, D.W. - "Structured Programming," *Software Engineering Techniques*, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, April 1970, pp. 84-88.

Harary, F. - *Graph Theory,* Addison-Wesley, Reading, MA, 1969, pp. 37-40.

Howden, W.E. - "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers*, Volume C-24, Number 5, May 1975, pp. 554-559.

Li, E.Y. - "Software Testing Techniques for the Information Systems Professional: A Curriculum Perspective," *Proceedings of the International Confer-*

Continued

STARTING A QUALITY ORGANIZATION

Ed Showalter
Manager Quality and Standards, Unisys

Starting Out is Hard to Do

I have read that once started, any task is half done. I don't know the author so I can't give credit. Many times it is true; however, the starting out can strike fear in the hearts of the bravest people.

It is very easy to ignore or procrastinate doing something before you've started. However, when your boss has given you a task or you've accepted a new position you cannot just pull the cover over your head and go back to sleep.

One position that many people have accepted and are struggling with today is starting a **"quality organization."** They are finding that "starting out is hard to do."

Having been in this predicament three times, in different environments, and each time starting a successful organization, I have learned that starting out is actually easier than it is made out to be.

Now before you throw this in the wastebasket and view me as just another snake oil salesman with one of those cure-all schemes, read on a few more paragraphs and just maybe you will gain a little insight that will make your starting out (or continuing on) easier.

Let's Start

Regardless of what your task is, you will first want to determine the part of your company you can influence and where that part of the company reports. As an example, if your boss is the director of the information systems department that reports to the vice president of finance (see Figure 1), then your sphere of influence is the information systems department. Once the reporting structure is understood, there are only two other things you need to know before starting this part of your company toward producing a better product. The rest of this article will deal with these two issues.

I won't be like a *Reader's Digest* sweepstakes letter and make you read this entire article to find the secret information. The two things you will need to know are **"what's broke?"** and **"who cares?"**.

What's Broke?

By this, I mean determining the top few **most critical** quality problems that face your organization. This list should contain no more than three or four issues. Remember, you cannot solve all the problems of the world overnight, and the bigger the list the more difficult it is to remain objective about where you want to go.

Have you planned vacations filled with all of the fun things you like to do, then ended up miserable or totally exhausted because you tried to do far too much? You then found yourself more concerned about finishing the current activity so you could get on to the next one. You

Continued

TESTING *Continued*

ence on Information Systems, 1988.

McCabe, T.J. - "A Complexity Measure," *IEEE Transactions on Software Engineering*, Volume SE-2, Number 4, April 1976, pp. 308-320.

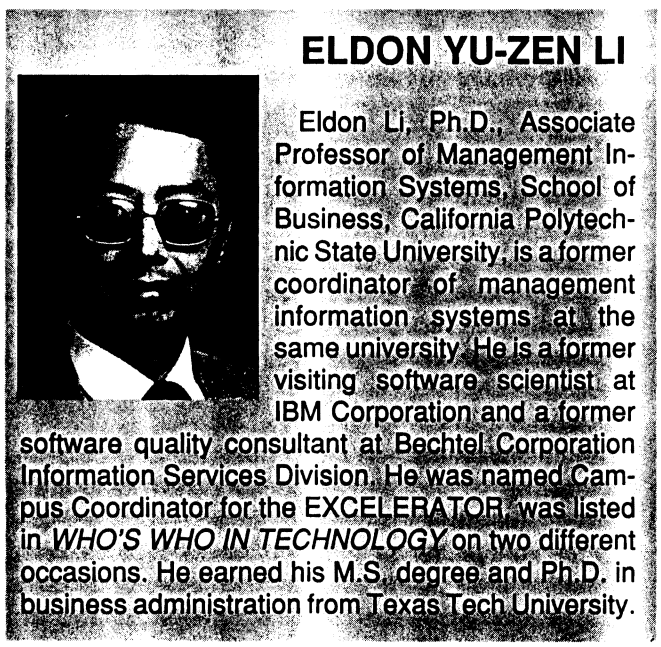
McCabe, T.J. - "A Testing Methodology Using the McCabe Complexity Metric," in T.J. McCabe (ed), *Structured Testing*, IEEE Computer Society Press, Silver Spring, MD, 1983, pp. 19-47.

Miller, E.F., Jr. - "Program Testing: Art Meets Theory," *Computer*, Volume 10, Number 7, July 1977, pp. 42-51.

Mills, H.D. - "Mathematical Foundations for Structured Programming," FSC 72-6012, IBM Federal System Division, Gaithersburg, MD, 1972.

Myers, G.J. - *The Art of Software Testing*, Wiley-Interscience, New York, NY, 1979, pp. vii, 1-11, and 36-76.

Schneidewind, N.F. - "Software Metrics for Aiding Program Development and Debugging," *AFIPS Proceedings of the National Computer Conference,* 1979, pp. 989-994.



ELDON YU-ZEN LI

Eldon Li, Ph.D., Associate Professor of Management Information Systems, School of Business, California Polytechnic State University, is a former coordinator of management information systems at the same university. He is a former visiting software scientist at IBM Corporation and a former software quality consultant at Bechtel Corporation Information Services Division. He was named Campus Coordinator for the EXCELERATOR, was listed in *WHO'S WHO IN TECHNOLOGY* on two different occasions. He earned his M.S. degree and Ph.D. in business administration from Texas Tech University.