ISSN 1009 – 3095 *Journal of Zhejiang University SCIENCE* V.3, No.1, P.37 – 41, Jan. – Mar., 2002
http://www.periodicals.com.cn; http://www.zju.edu.cn/depart/zdxb
http://lib.zju.edu.cn/eindex.htm; http://www.zjupress.com; jzu_s@mail.hz.zj.cn

Block-based test data adequacy measurement criteria and test complexity metrics*

CHEN Wei-dong(陈卫东), YANG Jian-jun(杨建军), YE Cheng-qing(叶澄清), PAN Yun-he(潘云鹤)
(*Department of Computer Science and Engineering, Zhejiang University, Hangzhou 310027, China*)

Received Mar.18, 2001; revision accepted July 28, 2001

Abstract: On the basis of software testing tools we developed for programming languages, we firstly present a new control flowgraph model based on block. In view of the notion of block, we extend the traditional program-based software test data adequacy measurement criteria, and empirically analyze the subsume relation between these measurement criteria. Then, we define four test complexity metrics based on block. They are J-complexity 0; J-complexity 1; J-complexity 1 +; J-complexity 2. Finally, we show the Kiviat diagram that makes software quality visible.

Key words: block, node, segment, control flowgraph model, test data adequacy measurement criteria, test complexity metric, Kiviat diagram
Document code: A **CLC number:** TP311.56

INTRODUCTION

Today 85% of the total cost of a typical computing and information management system is spent on software (ISA, 1999). There are many problems with today's software engineering:

1. Software systems have become larger and more complex.

2. There is no general and practical way to prove whether a program is correct (bug-free), so software developers must try to find errors within a program through extensive testing. This is why about 50% of software development cost is spent on testing.

3. About 50% of the code is still untested in a program "fully tested" using functional test methods alone. Structural testing, however, is very difficult to perform without automatic tools.

4. Software systems often need to be changed to meet new requirements or to add new features. Making changes to an existing system is risky it may often introduce inconsistencies in other modules of the system.

5. Software is hard to maintain because the design documents are often not readily accessible or are out of date with respect to the source code.

We developed computer-aided software testing tools for some languages such as Visual Basic and C/C++, respectively. They bring automation to the software development life cycle to make a software system easy to understand, test, modify, and maintain.

Our software testing tool is designed for structural testing. It works on the level of unit testing and integration testing, and is extended to support regression testing. It provides auto-generation of graphs and charts, test coverage analysis automation, quality measurement automation, dynamic tracing automation, and testing execution automation (Sun, 1999).

From Fig.1, we can conclude that the computer-aided software testing tool consists of two modules: Engine and Automation. The former is the kernel of testing tool. The engine reads source code (parsing), generates data files

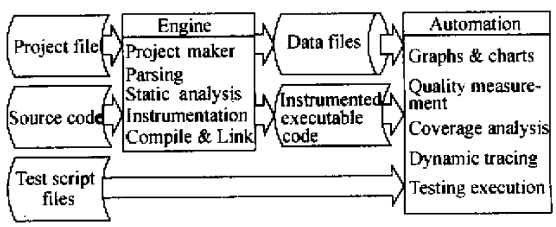

Fig.1 The computer-aided software testing tool's automatic processes

* Preject(No.60073027) supported by the National Natural Science Foundation of China.

(static analysis) and transforms source code (code instrumentation) to meet the test requirements, and supports the consequent automation modules. Here we discuss the foundation of the testing tool rather than its implementation.

As we mentioned above, the tool is designed for structural testing, especially for path testing. Path testing based on the use of the program's control flow as a structural model is the cornerstone of testing (Zheng, 1992). It requires complete knowledge of the program's structure (i.e., source code). In order to describe the program's structure efficiently, we present a new control flowgraph based on block (discussed in section 2 below). It is a new control flow diagram that is structured and colorful, represents the result of path analysis, uses different notations to represent different logic, and is clearly annotated with condition of each decision and the source code location. Furthermore, on the basis of the notion of block, we have empirically extended the traditional program-based test data adequacy measurement criteria and test complexity metrics.

BLOCK-BASED CONTROL FLOWGRAPH

The control flowgraph is a simplified representation of the program's structure (Zhu et al, 1997). Using traditional procedural logic diagrams such as control flowgraphs is inefficient, because those diagrams are often unstructured. Here, however, the control flowgraph we discuss is not what we usually use to represent the program's control flow. There is only one kind of component: block (Yang et al, 2000). For every program, there exist two kinds of blocks: node and segment.

A block is a sequence of program statements. Formally, it is a sequence of statements such that if any one statement of the block is executed, then all statements thereof are executed. Less formally, a block is a piece of straight-line code.

Node

There are three kinds of nodes, including decision, junction and the entry/exit point of a program unit.

1. Decision

A decision is a program point at which the control flow can diverge. Here are some examples in C++ programming language (Table 1).

Table 1 Some examples in C++ programming language

Statement	Example	Decision

IF statement	if (condition)
	  if body
	[else
	  else body]
		if (condition)

SWITCH statement	switch
	(expression) {
	……
	}
		switch
		(expression)

FOR statement	for(expression; expression; expression) for loop-body	for(expression; expression; expression)
WHILE statement	while(condition) while loop-body	while (condition)

DO...WHILE statement	do
	  do...while
	  loop-body
	while(condition)
		while (condition)

2. Junction

A junction is a point in the program where the control flow can merge. In C++ language, examples of junctions are:

1) "DO" in DO...WHILE statement;

2) "ELSE" in IF...ELSE statement;

3) "CASE: " and "DEFAULT: " in SWITCH statement;

4) statement labels.

3. The entry/exit point of a program unit

The entry/exit point of a program unit is defined as the begin/end point of a scope, which is the portion in a program within which a declaration applies. For example, the C++ "{" and "}" are the begin/end point of a scope. In addition, the body's begin/end point in branch statements (such as "if body" in IF statement) is the begin/end of a scope. From the point of view of branch body, although the body's end point is also a node; the begin point is not an isolated node; and it is used together with the decision or junction node mentioned above.

Segment

A segment is a sequence of statements between two consecutive branch points. It has one

entry and one exit. Here the branch points include the above nodes and the position between unconditional jump statements and its next statement. In C++ language, examples of unconditional jump statements are: GOTO, RETURN, BREAK and CONTINUE statements. Then, there are three ways in which we describe the partition of segments:

1. A sequence of statements between two consecutive nodes is a segment. The two nodes do not belong to this segment.

2. A sequence of statements between a node and an unconditional jump statement is a segment.

There are two cases as follows:

1) If a node precedes an unconditional jump statement, we define that the unconditional jump statement belongs to this segment; and it is the last statement of this segment. However, the node does not belong to this segment.

2) If a node follows an unconditional jump statement, we define that the unconditional jump statement and the node do not belong to this segment.

Warning: the unconditional jump statement must belong to the previous segment.

3) A sequence of statements between two consecutive unconditional jump statements is a segment. The first unconditional jump statement does not belong to this segment, but the latter belongs to this segment; and it is the last statement.

Invisible segment

Besides the above segment, there is another special segment-invisible segment, which is designed for recording the paths that have been executed.

For each decision statement, if there is no executable statement associated with the decision statement when its condition is unsatisfied, we define there should be an invisible segment next to the decision statement. For example, any IF statement lacking an ELSE part has an invisible segment by definition.

For each repetition statement, there are two invisible segments. One of them will be executed when the program control reaches the statement but its condition is never satisfied. We call this invisible segment "low-end loop boundary invisible segment". The other will be executed if the condition is satisfied at least once, and the program control exits the repetition body normally when the condition is no longer satisfied. We call this invisible segment "high-end loop boundary invisible segment".

Example

Here is an example of block-based control flowgraph.

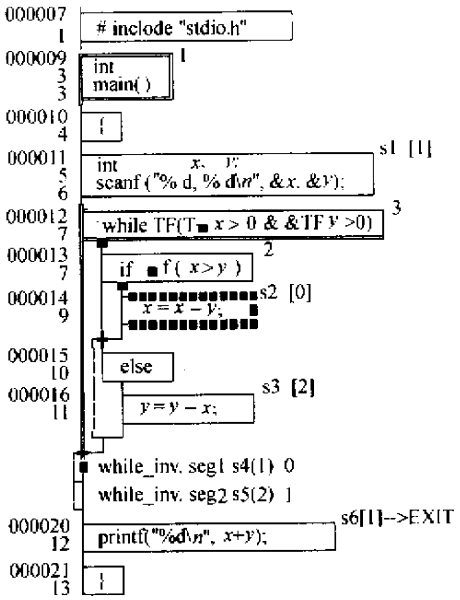

Fig.2 An example of block-based control flowgraph

BLOCK-BASED TEST DATA ADEQUACY MEASUREMENT CRITERIA

An important aspect of testing is the ability to decide when enough testing has been performed. There are many such measurement criteria proposed in research and practice (e.g., statement test coverage and branch test coverage.) (Zheng, 1992). In this section, we describe how we have extended these traditional test data adequacy measurement criteria.

1. SC0

SC0 is the percentage of visible blocks that have been executed. A set of test cases of a program satisfies SC0 if all nodes and visible segments of the program have been executed at least once. It is computed as following:

SC0 = the visible blocks that have been executed

at least once ÷ the sum of visible blocks × 100%.

SC0, the basic block test coverage, is better than the statement test coverage. For example, there is a decision statement in which the condition part and the execution part of a decision statement are written in the same line:

If (condition) statement;

The condition part of line above is never satisfied; the execution part (segment) has never been executed. It is difficult for statement test coverage to report the actual coverage of the example mentioned above, whereas SC0 can clearly point out an unexecuted segment (the execution part of the decision statement).

SC0 covers statement test coverage.

2. SC1

SC1 is the percentage of visible and basic invisible blocks that have been executed. Basic invisible blocks consist of if, do-while, switch and high-end loop boundary invisible blocks. A set of test cases of a program satisfies SC1 if it satisfies SC0 coverage and basic invisible segments of the program have been executed at least once. It is computed as following:

SC1 = the visible and basic invisible blocks that have been executed at least once ÷ the sum of visible and basic invisible blocks × 100%

SC1, the standard block test coverage, is better than the branch test coverage. A test based on execution of each branch does not enforce the execution of each statement. So, the branch test coverage cannot be used to check dead code. For example, the following code contains an unexecuted line (line 3). It is difficult for the branch test coverage to report the unexecuted code, whereas SC1 can clearly point it out.

Statement1;
Goto 50;
Statement 2;
50: Statement 3;

SC1 covers branch test coverage and SC0.

3. SC1 +

SC1 is the percentage of visible and all (basic and low-end loop boundary) invisible blocks that have been executed. A set of test cases of a program satisfies SC1 + if it satisfies SC1 coverage and the entire low end invisible segments of the loops in the program have been executed at least once. It is computed as following:

SC1 + = the visible and all invisible blocks that have been executed at least once ÷ the sum of visible and all invisible blocks × 100%

SC1 + covers SC1.

4. J-coverage

J-coverage is defined as the ratio of the number of executed visible and invisible blocks plus executed outcomes of conditions to the number of all visible and invisible blocks plus all outcomes of conditions in a program or program module. It is computed as following:

J-coverage = the visible , invisible blocks and condition outcomes have been executed at least once ÷ the sum of visible, invisible blocks and condition outcomes × 100%.

J-coverage covers SC1 + . It is the strongest test data adequacy measurement criteria provided by our testing tool.

BLOCK-BASED TEST COMPLEXITY METRICS

We provide two kinds of complexity analyses: Cyclomatic complexity (Zhu et al, 1997) and Test Complexity (Sun, 1999). Cyclomatic complexity may be used to determine the structural complexity of a coded module. The use of this measure is designed to limit the complexity of a module. Test Complexity (J-complexity) focuses on software testing. It provides the precise measurement of the efforts required for achieving levels of test coverage results. J-complexity is useful for professionals in planning the test activities and in measuring the true structural complexity. The higher the test complexity value is, the harder to fully test the code.

From the analyses in section 2 (consider the notion of block), we define four Test Complexity as follows:

1. J-complexity 0 (Basic block test complexity)

J-complexity 0 is the minimum number of instrumentation points required for recording basic blocks test coverage (SC0) data. It is the sum of visible blocks.

2. J-complexity 1 (block test complexity)

J-complexity1 is the minimum number of instrumentation points required for recording block test coverage (SC1) data. It is the sum of visible blocks plus basic invisible blocks (if, do-

while, switch, and high-end loop boundary invisible blocks).

J-complexity1 covers J-complexity 0.

3. J-complexity 1 + (Block test complexity plus)

J-complexity1 + is the minimum number of instrumentation points required for recording all block test coverage (SC1 +) data. It is the sum of visible blocks and all invisible blocks (basic and low-end loop boundary invisible blocks).

J-complexity1 + covers J-complexity1.

4. J-complexity 2 (condition-block test complexity)

J-complexity 2 is the minimum number of instrumentation points required for recording all condition-block test coverage (J-Coverage) data. It is the sum of visible and invisible blocks plus all outcomes of all conditions in all the decision statement.

J-complexity 2 covers J-complexity1 + .

KIVIAT DIAGRAM

Any metrics that rely on having the programmers fill out long questionnaires or manually calculate metric values is doomed. If a metric's calculation is not fully automated, it probably harms productivity and quality more than any benefit we can expect from metrics. So we also provide an automated tool, which automatically generates the above metrics. Use of these metrics will make software quality visible and result in the great improvement of software quality as well as the acceleration of the software development. The Kiviat diagram (K.html) is one of the forms that display the software quality. Fig.3 shows one example.

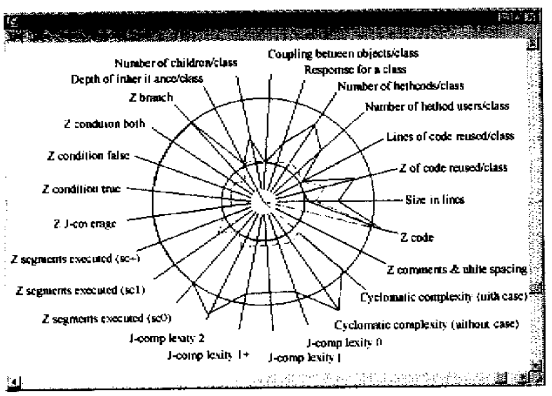

Fig.3 Kiviat diagram

A Kiviat diagram is a pie-like diagram that contains radiantly extending lines each of which represents a metric and two concentric circles that represent the acceptable bounds for the metrics (ISA, 1999). The value for each metric is shown as a point on the corresponding line. If the point falls within the bounds, it is considered acceptable.

CONCLUSIONS

The control flowgraph is a graphical representation of a program's control structure. In this paper, we have presented a new control flowgraph model for many structured programming languages. As a consequence, these test data adequacy measurement criteria and test complexity metrics we describe support many structured languages. So we can easily develop software testing tool for other languages such as Delphi.

On the other hand, although our testing tool allows you to measure and review the quality and reliability of your software system developed using object-oriented techniques, it fully satisfies structured language rather than object-oriented language.

References

International Software Automation Inc, 1999. Panorama VB User's Manual. Computer Department of Zhejiang University.

Sun Ting, 1999. Design and Implementation of the software testing tool Thesis for master degree. Zhejiang University, Hangzhou 72p. (in Chinese).

Yang Jianjun, Chen Weidong, Ye Chengqing, et al., 2000. Design and Implementation of Testing Tools for Context-free Languages. *Journal of Computer Research and Development*. **37**(11): 1375 – 1381 (in Chinese, with Engligh abstract).

Zheng Renjie, 1992. Computer's Software Testing Techniques. Tsinghua University Press, Beijing, 264p. (in Chinese).

Zhu, Hong, Jin Lingzi, 1997. Software Quality Assurance and Testing. Science Press, Beijing, 255p. (in Chinese).