Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing

TAKESHI CHUSHO

Abstract—**A new coverage measure is proposed for efficient and effective software testing. The conventional coverage measure for branch testing has such defects as overestimation of software quality and redundant test data selection because all branches are treated equally. These problems can be avoided by paying attention to only those branches essential for path testing. That is, if one branch is executed whenever another particular branch is executed, the former branch is nonessential for path testing. This is because a path covering the latter branch also covers the former branch. Branches other than such *nonessential branches* will be referred to as *essential branches*.**

A testing tool for the new measure is developed in order to discriminate essential branches from nonessential branches and to measure the coverage rate of these essential branches. By using this tool, it is ascertained that the number of essential branches is about 60 percent of all branches.

As a result, the new measure reduces software quality overestimation because the accumulative curve of the new measure to the number of executed test data is closer to linearity than that of the conventional measure. Another advantage is the prevention of redundant test data selection. It results from a 40 percent reduction in the number of branches to be monitored and is confirmed by a reasonable algorithm for test data selection. Furthermore, an efficient algorithm for redundancy elimination of a selected test data set is presented.

Index Terms—**Algorithm, branch testing, control flow graph, coverage measure, path testing, program testing, quality estimation, test data selection.**

I. Introduction

PROGRAM testing constitutes approximately half of total software development costs and is the key to improving software productivity and reliability. Many different software testing tools have already been developed to support the various aspects of software testing [1], [2]. In particular, the most important aspect of software testing is a method for selecting test data [3] because correctness of program logic is a main factor of software reliability, which is a part of software quality. The method is categorized into functional testing and structural testing [4]. The former implies that test data are selected based on the function specification of a program, and the latter implies that test data are selected based on the control structure of a program. It is, however, difficult to support functional testing by using software tools [5] because a formal specification language is required.

On the other hand, there are several methods and tools for structural testing. In particular, great attention has recently been paid to path testing [6]–[8]. Path testing is intended to execute all paths reaching from an entry to an exit on a control flow graph of a program. Practically speaking, a subset of paths is selected, and input data that will cause their paths are found. This technique is called sensitizing the path [20]. Notably, branch testing, a form of simplified path testing, is more practical because exact path testing often requires an enormous amount of test data. A typical branch testing tool measures the ratio of executed branches to all branches in a program. This coverage measure is used to estimate the quality of a tested program with regard to correctness of program logic and to select test data by which unexecuted branches are executed. This technique [9] is used in many tools such as RXVP [10], SADT [11], ATA [12] for Fortran, CIP [13], SMOTL [14] for Cobol, and HITS [15] for microcomputer software.

Conventional branch testing, however, has the following two defects.

1) Redundant test data are apt to be selected when conventional branch testing is used for test data selection since there are many branches, all of which are executed by many test data.

2) Quality is overestimated when conventional branch testing is used for quality estimation since the coverage rate increases rapidly when the first group of test data is executed.

These problems are caused by treating all branches equally and can be avoided by paying attention to only those branches essential for path testing. That is, if one branch is executed whenever another particular branch is executed, the former branch is nonessential for path testing. This is because a path covering the latter branch also covers the former branch. Branches other than such nonessential branches will be referred to as essential branches.

First of all, to present a method for discriminating essential branches from nonessential branches, this paper introduces a directed graph, obtained from a control flow graph of a program by eliminating arcs which correspond to nonessential branches. All arcs of this graph correspond to essential branches and are called *primitive arcs*. The eliminated arcs are called *inheritor arcs* because these arcs can inherit information about path coverage from primitive arcs. This graph is called *an inheritor-reduced graph*. An algorithm transforming a control flow graph into the inheritor-reduced graph is then presented.

Manuscript received July 31, 1984.

The author is with the Systems Development Laboratory, Hitachi Ltd., Ohzenji, Asao-Ku, Kawasaki 215, Japan.

IEEE Log Number 8613876.

Next, a new coverage measure, based on the number of essential branches executed at least once by test runs of a program, is proposed, and a tool for this new measure is developed. Then, through experiments with this tool, it is confirmed that the new measure is more suitable for test data selection than the conventional measure for branch testing since the number of branches to be considered decreases. It is also ascertained that the new measure is suitable for quality estimation. This is because this measure has features similar to path testing features, where the path testing coverage rate is linear to the number of executed test data.

Finally, this paper presents an algorithm for test data selection resulting in reduced redundancy and an algorithm for eliminating redundancy in a selected test data set by paying attention to only essential branches.

II. Conventional Method

A. Branch Testing

In general, program testing is performed by dynamic testing in such a way that a program is executed with various input data and then each result is confirmed. In this method, however, it is impossible to test all possible input data. Therefore, a finite test data set should be selected so as to assure high quality of the tested program under time and cost limitations [3].

Path testing is one technique for this purpose and is intended to execute as many feasible paths from an entry to an exit on a control flow graph of a program as possible. The coverage measure based on this technique is as follows:

$$C_{\text{path}} = \frac{\text{the number of executed paths}}{\text{the number of all feasible paths in a tested program}}.$$

This measure, however, is not practical since the number of feasible paths is enormous in most programs because of iterations. Therefore, for practical purposes, attention is paid to a path component instead of a path. This component, called the *dd* path (decision-to-decision path) [10], is defined as a partial path in a control flow graph such that a) its first constituent arc emanates from either an entry node or a decision box, b) its last constituent arc terminates at either a decision box or an exit node, and c) there is no decision box on the path except those at both ends, where a decision box is a node with two or more exit arcs. The coverage measure based on such dd paths is as follows:

$$C_{\text{dd}} = \frac{\text{the number of executed dd paths}}{\text{the number of all dd paths in a tested program}}.$$

This technique is called branch testing because this measure promotes execution of all branches. The following are the uses of this measure:

1) to detect a lack of test data, and to select additional data so as to reach unexecuted dd paths; and

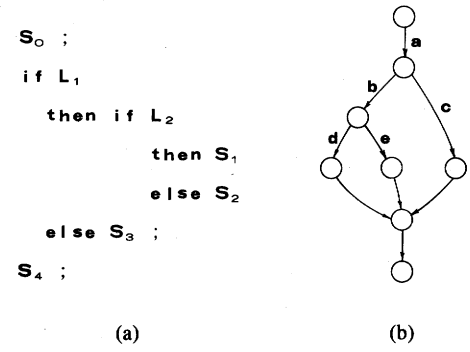


Fig. 1. A program example. (a) Source. (b) The control flow graph.

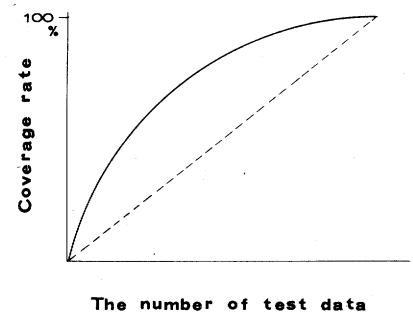


Fig. 2. Accumulative curve of coverage rate to the number of executed test data on branch testing.

2) to estimate the quality of a tested program, assuming that the higher the measure, the higher the quality of the tested program.

B. Problems of the Conventional Method

For a demonstration of the first problem, consider the program in Fig. 1 and the following test cases.

Case 1: Both logical predicates L_1 and L_2 are true.

Case 2: L_1 is true, but L_2 is false.

Case 3: L_1 is false.

There are five dd paths *a*, *b*, *c*, *d*, and *e* in the control flow graph of this program as shown in Fig. 1(a). When Case 1 is first executed, *a*, *b*, and *d* are covered and C_{dd} is 3/5. After Case 2 and 3 are executed sequentially, C_{dd} will become 4/5 and then 5/5, respectively.

However, it is desirable that the coverage rate increase by 1/3 per case when the essential measure C_{path} is used since there are three paths in this program. The difference between C_{dd} and C_{path} in the increase trend is caused by the fact that the nonessential dd paths for path coverage, *a* and *b*, and the essential dd paths, *c*, *d*, and *e*, are treated equally. That is, the degree to which each case contributes to C_{dd} depends on the execution order.

Consequently, when using C_{dd} instead of C_{path}, the quality of the tested program is overestimated, as shown in Fig. 2, in which the bold line is C_{dd} and the broken line is a ratio of executed test data to all test data. That is, when a coverage rate is less than 100 percent, C_{dd} is greater than a ratio of executed test data to all test data.

Next, consider the other program in Fig. 3 and the following test cases to demonstrate the second problem.

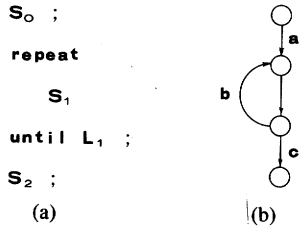


Fig. 3. A program example. (a) Source. (b) The control flow graph.

Case 1: L_1 is true.

Case 2: L_1 is false the first time and true the second time.

Case 1 was first selected so as to include the dd path c. Then Case 2 was selected so as to include b. As a result, Case 1 becomes redundant from the path coverage viewpoint because Case 2 includes all dd paths in the program. As shown in this example, test data selection based on all dd paths has a defect in that redundant test data are apt to be selected. The reason is the same as for the first problem; that is, all dd paths are treated equally, although the dd path b is essential for path coverage but a and c are not.

III. The Primitive Arc Concept

A. Primitive and Inheritor Arcs

In this section, the concepts of primitive and inheritor arcs in a control flow graph are introduced to discriminate the essential branches from nonessential branches described previously.

Definition 1: A program is transformed to a directed graph so that a node will correspond to a basic block,[1] which is a sequence of sequentially executed statements, and so that an arc will correspond to control transfer between basic blocks. Each entry and exit is transformed into individual nodes.[2] This graph is called *a control flow graph* and is denoted by $G(N, A)$ where N is a set of nodes and A is a set of arcs.

In the remainder of this paper, nodes are represented by lower case letters from the end of the alphabet, such as x, y, or z, and arcs from x to y are represented by (x, y) or lower case initial letters of the alphabet, such as a, b, or c.

Definition 2: For each node x, let $\mathrm{IN}(x)$ be the number of arcs entering x and $\mathrm{OUT}(x)$ be the number of arcs exiting from x. A node x with $\mathrm{IN}(x) = 0$ is called *an entry node*, and x with $\mathrm{OUT}(x) = 0$ is called *an exit node*.

Definition 3: For any path from an entry node to an exit node, if the path including an arc a always includes another arc b, b is called *an inheritor* of a and a is called *an ancestor* of b. This is because b inherits information about the execution of a; that is, b is executed whenever a is executed.

Definition 4: An arc which is never an inheritor of another arc is called *a primitive arc*.

Definition 5: A directed graph with no inheritors is called *an inheritor-reduced graph*.

B. Elimination of Inheritors

This section introduces several reduction rules to eliminate inheritors from a directed graph.

Definition 6: Arcs incident to the same node in a path are called *adjacent arcs*.

Theorem 1: If there is an inheritance relation between two arcs which are not adjacent, the inheritor has its adjacent arc as another ancestor.

The proof is shown in a previous paper [16].

Definition 7: For a node x, an arc (x, x) is called *a self-loop*.

Theorem 2: A self-loop is a primitive arc.

The proof is shown in a previous paper [16].

Definition 8: A node y is called *a dominator* of a node x if all paths from an entry node to x include y. A node z is called *an inverse dominator* of x if all paths from x to an exit node include z. Let $\mathrm{DOM}(x)$ and $\mathrm{IDOM}(x)$ be sets of dominators and inverse dominators of x, respectively. An algorithm for obtaining $\mathrm{DOM}(x)$ is detailed in [17]. An algorithm for obtaining $\mathrm{IDOM}(x)$ is conducted from the algorithm for obtaining $\mathrm{DOM}(x)$ by inverting arc directions.

Following the above considerations, the condition for an inheritor is discussed. From Theorems 1 and 2, it suffices to consider whether an arc between different nodes is an inheritor of its adjacent arc or not. The general form of such an arc is shown in Fig. 4, where the broken line implies one or more arcs that may exist.

The condition for a being an inheritor of b, c, d, or e in Fig. 4 will be examined by considering the following four cases.

Case 1: a is an inheritor of b.

A path passing through b necessarily passes through a or c because x is not an exit node. Therefore, a path passing through b necessarily passes through a, only if the following condition holds:

1) there is no c, or

2) there are one or more c's and a path passing through c necessarily returns to x; that is, x is an inverse dominator of the drain node for c.

Case 2: a is an inheritor of c.

The condition of this case is the same as the second condition of Case 1.

Case 3: a is an inheritor of d.

A path passing through d passes through a or e because y is not an entry node. Therefore, a path passing through d necessarily passes through a, only if the following condition holds:

1) there is no e; or

[1]Although the correspondence between a basic block and statements depends on the control statements of each individual programming language, it is not detailed in this paper because there is no relation to the subject of this paper.

[2]Since each entry or exit does not correspond to a node in the definition of a control flow graph in [17], a program with no branches and no loops is transformed into a single node. In this paper, however, each entry or exit corresponds to one node because coverage of paths from an entry to an exit is discussed.

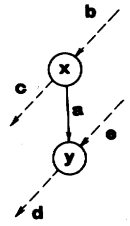


Fig. 4. General form of an arc and its two nodes.

2) there are one or more e's, and a path passing through e has necessarily passed through y previously; that is, y is a dominator of the source node for e.

Case 4: a is an inheritor of e.

The condition of this case is the same as the second condition of Case 3.

The above four conditions give the following reduction rules for the elimination of an inheritor.

Condition 1: For a directed graph $G(N, A)$,

$$x, y \in N \wedge x \neq y \wedge (x, y) \in A.$$

Reduction Rule R1: Under Condition 1, if

$$\mathrm{IN}(x) \neq 0 \wedge \mathrm{OUT}(x) = 1,$$

(x, y) is eliminated from A and x and y are merged into one, as shown in Fig. 5(a).

With respect to arc arrows in Figs. 5 and 6, the bold line is an eliminated arc, the fine line is another arc in existence, and the broken line implies one or more arcs that may exist.

Reduction Rule R2: Under Condition 1, if

$$\mathrm{IN}(y) = 1 \wedge \mathrm{OUT}(y) \neq 0,$$

(x, y) is eliminated from A and x and y are merged into one, as shown in Fig. 5(b).

Reduction Rule R3: Under Condition 1, if

$$\mathrm{OUT}(x) \geq 2$$

and

$$x \in \mathrm{IDOM}(w) \qquad \text{for } \forall\, w \in \{w \mid (x, w) \in A \wedge w \neq y\},$$

(x, y) is eliminated from A and x and y are merged into one, as shown in Fig. 6(a).

Reduction Rule R4: Under Condition 1, if

$$\mathrm{IN}(y) \geq 2$$

and

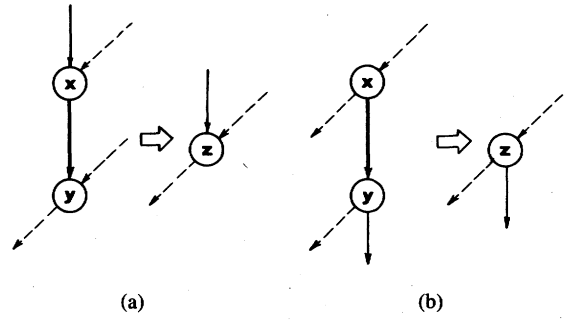
$$y \in \mathrm{DOM}(w) \qquad \text{for } \forall\, w \in \{w \mid (w, y) \in A \wedge w \neq x\},$$

(x, y) is eliminated from A and x and y are merged into one, as shown in Fig. 6(b).

C. Reduction Algorithm

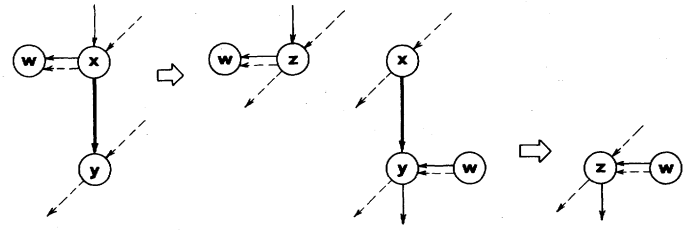
Using these four reduction rules, the algorithm for transforming a directed graph to an inheritor-reduced graph is given as follows.

Algorithm 1: For a given directed graph $G(N, A)$, the following procedure is executed.



(a) (b)

Fig. 5. Applications of the reduction rules. (a) R1. (b) R2.



(a) (b)

Fig. 6. Applications of the reduction rules. (a) R3 (x is an inverse dominator of w). (b) R4 (y is a dominator of w).

1) Apply R1 for any arc which satisfies the condition of R1.

2) Step 1) is repeated until no further suitable arcs are found.

3) Apply R2 for any arc which satisfies the condition of R2.

4) Step 3) is repeated until no further suitable arcs are found.

5) Write an inheritor mark on any arc (x, y) that satisfies the condition of R3, if there is at least one arc without an inheritor mark among input arcs of x or among arcs composing a path from output arcs of x to x, except (x, y) itself.

6) Step 5) is repeated until no further suitable arcs are found.

7) Write an inheritor mark on any arc (x, y) that satisfies the condition of R4, if there is at least one arc without an inheritor mark among the output arcs of y or among arcs composing a path reaching inversely from input arcs of y to y, except (x, y) itself.

8) Step 7) is repeated until no further suitable arcs are found.

9) Eliminate any arc with an inheritor mark, and merge the two nodes on both ends of this arc into one.

10) Step 9) is repeated until no arcs with inheritor marks are found.

The following theorem assures that this algorithm is correct and optimum.

Theorem 3: The directed graph reduced by Algorithm 1 has the following features.

1) A set of paths covering all arcs in the reduced graph also covers all arcs in the original graph.

2) The number of arcs in the reduced graph is least among graphs with the feature of 1).

The proof is shown in a previous paper [16].

Although the application order of R1 and R2 is unimportant, the order of Algorithm 1 is such that R1 is prior to R2. This has the following merits.

1) Each arc in the reduced graph corresponds uniquely to a particular arc in the original graph.

2) Furthermore, each arc in the reduced graph corresponds uniquely to a particular dd path in the original graph since the corresponding arc in the original graph is a branch arc whose source node has two or more exit arcs. Such corresponding arcs are called essential branches, and the other branch arcs are called nonessential branches in the original program.

IV. New Coverage Measure and Its Support Tool

A. New Coverage Measure

In order to improve both effectiveness and efficiency of branch testing, a new coverage measure C_{pr}, instead of C_{dd}, is defined below on the inheritor-reduced graph, which is transformed from a program by Algorithm 1:

$$C_{pr} = \frac{\text{the number of executed arcs}}{\text{the number of all arcs in the inheritor-reduced graph}}.$$

B. A Tool for New Coverage Measure

The tool for measurement of C_{pr}, SCORE (the source-level coverage rate evaluator), was developed and used for comparison between C_{pr} and C_{dd}. SCORE is applicable to Pascal programs and is composed of the following four phases.

Phase P1: A Pascal program is transformed to the control flow graph.

Phase P2: The control flow graph is transformed to the inheritor-reduced graph by Algorithm 1.

Phase P3: An instrument code is embedded into any place in the source program corresponding to any arc in the inheritor-reduced graph.

Phase P4: The coverage rate C_{pr} and unexecuted essential branches are printed out after the code-embedded program is executed.

V. Reduction of Branches To Be Monitored

First, the number of branches to be monitored for C_{pr} is compared to that for C_{dd}. The following three programs written in Pascal are used for this experiment with SCORE.

PL0 Parser: This is the parser for the language PL0, whose source program is shown in Wirth [18, pp. 314–319].

SCORE: This is the tool itself for C_{pr}, whose four phases P1, P2, P3, and P4 are used separately.

PARSE: A structure editor recently developed by our group [19].

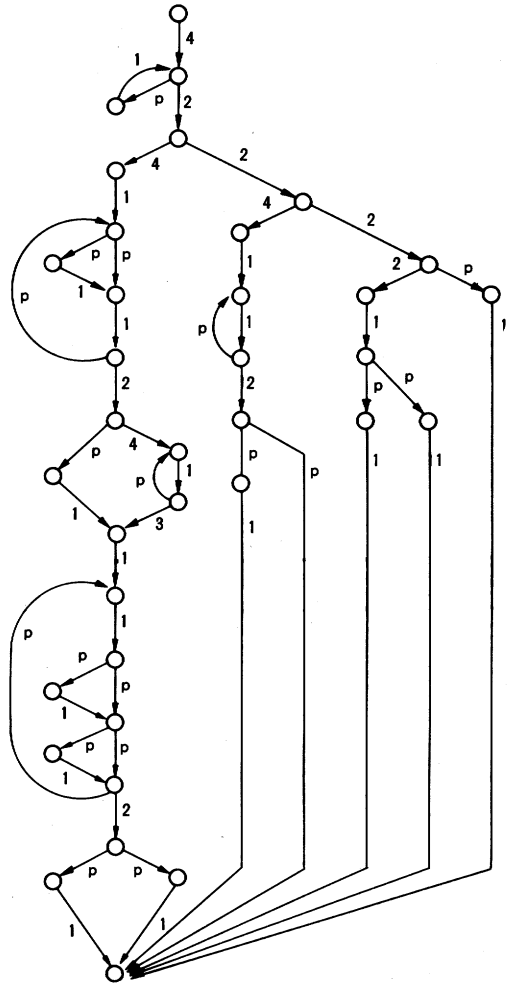


Fig. 7. A control flow graph of the getsym procedure in the PL0 parser, in which an arc with the number n is eliminated by the reduction rule Rn of Algorithm 1.

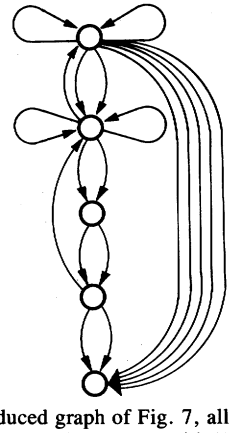


Fig. 8. The inheritor-reduced graph of Fig. 7, all arcs of which are primitive arcs corresponding to arcs with "p" in Fig. 7.

The PL0 parser was selected to assure objectivity, and the others were selected as rather large programs.

For example, consider the getsym procedure in the PL0 parser. This procedure is transformed to the control flow graph as shown in Fig. 7 and then transformed to the inheritor-reduced graph by applying the Algorithm 1 as shown in Fig. 8. The number n in Fig. 7 implies that the

TABLE I
REDUCTION OF BRANCHES TO BE MONITORED WHEN APPLYING ALGORITHM 1 TO PASCAL PROGRAMS

Tested Programs	PL0 Parser	SCORE (P1)	SCORE (P2)	SCORE (P3)	SCORE (P4)	PARSE	Total
1) The number of executable statements	326	1375	1095	516	1052	6663	11 027
2) The number of branches (a denominator of C_{dd})	136	582	533	230	553	2914	4 948
3) The number of eliminated branches:							
a) using the R2 rule	40	87	73	44	52	754	1 050
b) using the R3 rule	5	104	113	18	131	138	509
c) using the R4 rule	10	48	49	11	47	148	313
4) The number of essential branches (a denominator of C_{pr})	81	343	298	157	323	1874	3 076
The ratio of 4)/2)	0.60	0.59	0.56	0.68	0.58	0.64	0.62

arc with the number is eliminated by the reduction rule R*n*. The other arcs with *p* are primitive arcs in Fig. 8.

The entire result is shown in Table I. Item 2) is the number of all branches in a tested program; all these branches are monitored when C_{dd} is applied as a coverage measure for branch testing. Item 3) is the number of branches eliminated as nonessential branches by using the reduction rules R2, R3, and R4 in Algorithm 1. Table I omits the number of arcs eliminated by using the reduction rule R1 because it has no relation to the comparison between C_{pr} and C_{dd}. Item 4) is the number of essential branches that correspond to arcs in the inheritor-reduced graph of a tested program. Only these essential branches are monitored when C_{pr} is applied. Therefore, this experiment demonstrates that the number of branches to be monitored for branch testing is reduced to about 60 percent by using C_{pr} instead of C_{dd}.

As a result, this reduction implies the following advantages when C_{pr} is used for making additional test data so that they will execute unexecuted branches.

Effectiveness: The possibility of selecting redundant test data is less because attention is paid to only essential branches.

Efficiency: It is easier to select additional test data because the number of branches under consideration is lower.

VI. FEATURES OF THE NEW MEASURE FOR QUALITY ASSURANCE

The difference between C_{pr} and C_{dd} is experimentally observed by applying these measures to the PL0 parser mentioned previously.

A. Test Data Selection

The following set of test data is selected for full coverage of all branches in the PL0 parser.

1) A_1–A_{26}*:* These test data cover all arcs of the syntax diagram in [18, pp. 308–310] so that the execution of A_n covers only one arc which is not executed by a set of A_1–A_{n-1}.

2) B_1–B_{25}*:* These test data correspond to 25 invocations of the error-handling procedure ERROR in the PL0 parser.

3) C_1–C_4*:* These test data were added individually so that they would execute four branches not executed by the above test data. These four test data correspond to the following cases.

a) C_1 is the case where the '.' is missing at the end of a program.

b) C_2 is the case where the number of characters for an identifier exceeds ten.

c) C_3 is the case where there is a ':' not followed by a '='.

d) C_4 is the case where two or more different identifiers are used.

Among this set of test data, 1) and C_4 are test data of legal input, and 2) and C_1–C_3 are test data of illegal input.

In this process, the keyword 'to' of the following statement in the main program was replaced with 'downto':

for ch:='*A*' **to** ';' **do** ssym[ch]:=nul;

This is because the character code of '*A*' is greater than the character code of ';' in our computer system. The necessity of this modification was detected by the fact that the body of the **for** statement was not executed by above test data of 1) and 2). This demonstrates the effectiveness of branch testing.

B. Measurement of C_{pr}

First, all the test data were executed in order. As a result, there were eight redundant test data where the accumulative coverage rate did not increase. This is because the types of operators $\{=, \neq, <, >, \leq, \geq\}$, $\{+, -\}$, or $\{*, /\}$ are discriminated at once by using a relative operator **in** in the program, although these discriminations are represented by different arcs that correspond to each operator in the syntax diagram. Thus, these redundant test data were omitted.

Fig. 9 demonstrates the accumulative curves of C_{pr} and C_{dd} by using 47 test data. A tendency to overestimate software quality by using C_{dd} is weakened by using C_{pr} instead because the accumulative curve of C_{pr} is closer to linearity than that of C_{dd}. It is proved conclusively that C_{pr} is more suitable as a measure for software quality estimation than C_{dd}.

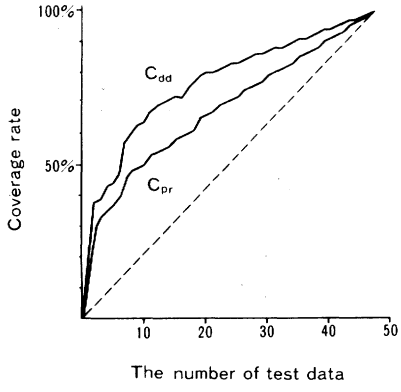


Fig. 9. Accumulative curve of C_{pr} and C_{dd} when testing the PL0 parser with 47 test data.

VII. Application to Test Data Selection

A. Algorithms for Test Data Selection

In order to decrease redundant test data from the path coverage viewpoint mentioned in Section II, a new algorithm for test data selection is proposed on the basis of the following policies.

1) Consideration should be limited to arcs in an inheritor-reduced graph, that is, to essential branches.

2) Furthermore, arcs with less possibility of being included in the execution paths of many test data should be given priority over arcs in an inheritor-reduced graph.

Algorithm 2: The test data for a program are selected according to the following procedure.

1) Transform a control flow graph of the program to the inheritor-reduced graph by using Algorithm 1.

2) Based on this inheritor-reduced graph, select test data including as many of the following arcs as possible among those arcs not yet included in the executed paths of selected test data. The upper items have priority over the lower items:

a) a self-loop;

b) a backward arc [17]; and

c) among sets of arcs from the same source node to the same drain node, an arc chosen from a set which holds the maximum number of arcs.

Although this algorithm is not deterministic, it can be refined so as to be deterministic by neglecting path predicates. However, such automatic path selection is not practical since there is a tendency to select infeasible paths. By using this algorithm, redundant test data decrease considerably.

Next, the effectiveness of Algorithm 2 is discussed. Step 1) is reasonable because it is not necessary to pay attention to inheritors from the path coverage viewpoint. In step 2), a self-loop and a backward arc have high priority because the other arcs in a path that includes these kinds of arcs have a high possibility of also being included in other paths. For example, if there is a path including a self-loop and another path excluding only the self-loop from the former path, all arcs except the self-loop overlap in these two paths. Since a path including a backward arc has a loop, all arcs except the backward arc in this path are apt to overlap with other paths also.

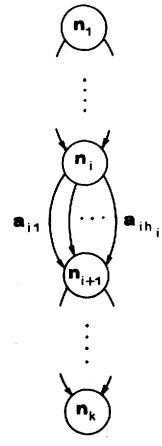


Fig. 10. A chain of nodes.

It is intuitively obvious that item c) of step 2) also has less possibility of overlap and therefore must have the third priority. The following two theorems are given to assist understanding of item c).

Condition 2: For a directed graph $G(N, A)$,

$$N = \{n_1, \cdots, n_k\},$$

$$(n_i, n_{i+1}) \in A \quad \text{for } i = 1, \cdots, k-1,$$

$$(n_i, n_j) \notin A \quad \text{for } j \neq i+1,$$

$$\text{IN}(n_1) = 0,$$

$$\text{OUT}(n_i) = \text{IN}(n_{i+1}) \geq 2 \quad \text{for } i = 1, \cdots, k-1,$$

and

$$\text{OUT}(n_k) = 0.$$

Let h_i be the number of (n_i, n_{i+1}) represented by $a_{i1}, \cdots, a_{ih_i}$, as shown in Fig. 10.

Theorem 4: Under Condition 2, the minimum possible number of paths for covering all arcs is

$$\max_i (h_i).$$

Theorem 5: Under Condition 2, the maximum possible number of paths for covering all arcs is

$$\sum_{i=1}^{k-1} (h_i - 1).$$

Theorem 4 is obvious. The proof of Theorem 5 is described in the Appendix.

B. Optimization of Selected Test Data Set

Software testing is important in the maintenance phase as well as in the development phase because more than 70 percent of total software costs are spent on maintenance. In particular, a test data set is executed more frequently for the regression test in the maintenance phase. Therefore, it is desirable to eliminate redundant test data from the selected test data set. A reduction algorithm based on an inheritor-reduced graph is given as follows.

Algorithm 3: For a given program, its test data set D is reduced as follows.

1) Transform the program to the inheritor-reduced graph $G(N, A)$ by Algorithm 1.

2) For all test data in D, obtain a set of arcs included in the corresponding path p. Let $A(p)$ be the set and P be a set of paths corresponding to D.

3) Obtain a subset of P_s from P as follows:

$$P_s = \left\{ p \,\middle|\, \left\{ L(a_i) \geq 2 \text{ for } \forall\, a_i \in A(p) \right\}, \exists\, p \in P \right\}$$

where $L(a_i)$ is the number of paths including a_i.

4) Eliminate a path p_m satisfying the following condition from P:

$$\min \left\{ L(a_i) \text{ for } \forall\, a_i \in A(p_m) \right\} \geq \min \left\{ L(a_j) \text{ for } \forall\, a_j \in A(p) \right\} \text{ for } \forall\, p \in P_s.$$

In addition, eliminate the test data corresponding to p_m from D.

5) Steps 3) and 4) are repeated until P_s becomes empty.

This algorithm, excluding step 1), can be applied to an original control flow graph as well as to an inheritor-reduced graph. This algorithm, however, is executed more efficiently because treated arcs are limited to primitive arcs.

VIII. Conclusions

A new coverage measure for branch testing was proposed for more effective and efficient software testing. This measure is defined by coverage of only essential branches, whereas the conventional measure is defined by coverage of all branches. The essential branches are defined so that full coverage of all essential branches will imply full coverage of all branches.

The testing tool for this new measure was developed in order to discriminate essential branches from nonessential branches and to measure the coverage rate over these essential branches. By using this tool, it is ascertained that the number of essential branches is about 60 percent of all branches.

As a result, the new measure had the following advantages in comparison to the conventional measure:

1) avoidance of software quality overestimation;

2) prevention of redundant test data selection; and

3) efficient optimization of a selected test data set for redundancy elimination.

The first advantage demonstrated by an experimental fact was that the accumulative curve of C_{pr} is closer to linearity than that of C_{dd}. The second advantage resulted from a 40 percent reduction in the number of branches to be monitored and was confirmed by a reasonable algorithm for test data selection. For the third advantage, the optimization algorithm was presented. Furthermore, any type of tool collecting information about branch coverage can be executed efficiently by monitoring only essential branches because coverage of all branches can be obtained from coverage of essential branches together with the corresponding relation between inheritors and ancestors.

Appendix

This Appendix proves Theorem 5, mentioned in Section VII-A. First, an example of the case in which the number of paths is $\Sigma_{i=1}^{k-1}(h_i - 1)$ is given. Suppose that the path including any a_{ij} $(j = 1)$ is composed of the following arc sequence, and other types of paths do not exist:

$$a_{11}, a_{21}, \cdots, a_{i-11}, a_{ij}, a_{i+11}, \cdots, a_{k-11}.$$

In this case, $(h_i - 1)$ arcs, except a_{i1}, among arcs exiting from the node n_i have a one-to-one correspondence with the set of paths. Then the number of paths becomes $\Sigma_{i=1}^{k-1}(h_i - 1)$.

Next, assume that

$$S = \sum_{i=1}^{k-1} (h_i - 1) + d$$

where S is the number of paths necessary for covering all arcs and $d \geq 1$. Let P be a set of these paths, and let $A(p)$ be a set of arcs composing a path p. Let $L(a_{ij})$ be the number of paths including a_{ij}. Now, if there is a path p satisfying

$$\left\{ L(a_{ij}) > 1 \text{ for } \forall\, a_{ij} \in A(p) \right\}, \exists\, p \in P,$$

p is a redundant path for full coverage of all arcs. Since this contradicts the assumption,

$$\left\{ L(a_{ij}) = 1, \exists\, a_{ij} \in A(p) \right\} \quad \text{for } \forall\, p \in P.$$

This condition implies that there are at least S arcs with $L(a_{ij}) = 1$. On the other hand, if

$$\left\{ L(a_{ij}) > 1, \exists\, a_{ij} \in \left\{ \text{a set of arcs exiting from } n_i \right\} \right\} \quad \text{for } i = 1, \cdots, k-1,$$

the number of arcs with $L(a_{ij}) = 1$ is not more than $\Sigma_{i=1}^{k-1}(h_i - 1)$, and this contradicts the previous result. Then there is a node n_i satisfying

$$L(a_{ij}) = 1 \quad \text{for } \forall\, a_{ij} \in \left\{ \text{a set of arcs exiting from } n_i \right\}.$$

Therefore, for this node,

$$\sum_{j=1}^{h_i} L(a_{ij}) = h_i.$$

On the other hand, by definition of $L(a_{ij})$,

$$\sum_{j=1}^{h_i} L(a_{ij}) = S.$$

Then these two equations conduct

$$h_i = \sum_{i=1}^{k-1} (h_i - 1) + d = (h_1 - 1) + \cdots + (h_{i-1} - 1) + h_i + (h_{i+1} - 1) \cdots \cdot (h_{k-1} - 1) + (d - 1).$$

From the assumption of $d \geq 1$, this equation implies

$$h_j < 1 \quad \text{for } j \neq i$$

and then contradicts Condition 2. □

Acknowledgment

The author wishes to express his gratitude to Dr. J. Kawasaki and Y. Aoyama for providing the opportunity to conduct this study. He is also indebted to T. Watanabe and A. Tanaka for their invaluable technical assistance and K. Horiuchi, who implemented the tool for the new measure.

References

[1] W. E. Howden, "A survey of dynamic analysis methods," in *Tutorial: Software Testing & Validation Techniques*, IEEE Catalog No. EHO 138-8. New York: IEEE, 1978, pp. 184–206.

[2] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[3] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, June 1975.

[4] W. E. Howden, "Applicability of software validation techniques to scientific programs," *ACM Trans. Programming Lang. Syst.*, vol. 2, pp. 307–320, July 1980.

[5] W. R. Elemendorf, "Functional analysis using cause–effect graphs," in *Proc. SHARE XLIII*, New York, 1974.

[6] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–214, Sept. 1976.

[7] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 236–246, May 1980.

[8] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 247–257, May 1980.

[9] E. F. Miller, "Program testing: Art meets theory," *IEEE Computer*, vol. 10, pp. 42–51, July 1977.

[10] J. C. Huang, "Error detection through program testing," in *Current Trends in Programming Methodology*, vol. 2, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 16–43.

[11] U. Voges *et al.*, "SADAT—An automated testing tool," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 286–290, May 1980.

[12] M. A. Holthouse and M. J. Hatch, "Experience with automated testing analysis," *IEEE Computer*, vol. 12, pp. 33–36, Aug. 1979.

[13] A. R. Sorkowitz, "Certification testing: A procedure to improve the quality of software testing," *IEEE Computer*, vol. 12, pp. 20–24, Aug. 1979.

[14] J. Bicevskis *et al.*, "SMOTL—A system to construct samples for data processing program debugging," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 60–66, Jan. 1979.

[15] T. Chusho *et al.*, "HITS: A symbolic testing and debugging system for multilingual microcomputer software," in *Proc. NCC'83*, May 1983, pp. 73–80.

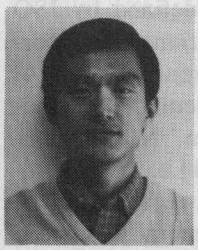
[16] T. Chusho, "Coverage measure for path testing based on the concept of essential branches," *J. Inform. Processing*, vol. 6, pp. 199–205, Feb. 1984.

[17] M. S. Hecht, *Flow Analysis of Computer Programs*. New York: North-Holland, 1978.

[18] N. Wirth, *Algorithms+Data Structures=Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[19] T. Chusho, "A language-adaptive programming environment based on a program analyzer and a structure editor," in *Proc. IFIP'83*, 1983, pp. 621–626.

[20] B. Beizer, *Software Testing Techniques*. Pennsauken, NJ: Van Nostrand Reinhold, 1983.



Takeshi Chusho was born in Marugame, Japan, in 1946. He received the B.S. and M.S. degrees in electronic engineering from Tokyo University, Tokyo, Japan, in 1969 and 1971, respectively, and the Ph.D. degree in computer science from Tokyo University in 1984.

He is a Senior Researcher in the Systems Development Laboratory, Hitachi Ltd., Kawasaki, Japan. Since joining the company in 1971, he has worked on the design and implementation of a structured programming language, compilers, testing tools, a structured editor, and a knowledge information processing language. His current research interests include knowledge engineering and software engineering.

Dr. Chusho received a winning paper award in 1982 from the Information Processing Society of Japan. He is a member of the Editorial Board of the Information Processing Society of Japan and of the Planning Board of the Japan Society for Software Science and Technology. He is a member of the IEEE Computer Society and the Association for Computing Machinery.