Invited Paper

Coverage Measure for Path Testing Based on the Concept of Essential Branches*

Takaeshi Chusho**

A new coverage rate based on essential branches (full coverage implies coverage of all branches) is proposed for efficient and effective software testing. The conventional coverage measure for branch testing has defects such as overestimation of software quality and redundant test data selection, because all branches are treated equally. In order to solve these problems, concepts of essential branches and nonessential branches for path testing are introduced. Essential branches and nonessential ones are called **primitive** and **inheritor arcs**, respectively, in a control flow graph of a tested program.

A reduction alogorithm for transforming a control flow graph to a directed graph with only primitive arcs is presented and its correctness is proved. Furthermore, it is experimentally and theoretically ascertained that the coverage measure on this **inheritor-reduced graph** is nearly linear to the number of test cases and therefore suitable for software quality assurance.

1. Introduction

Program testing constitutes about half of the software development costs and is the key to improving software productivity and reliability.

A lot of different software testing tools have already been developed which support many aspects of software testing [1], [2], and, in particular, great attention has recently been paid to path testing [3]–[5]. Path testing is intended to execute all paths reaching from an entry to an exit on a control flow graph of a tested program. Notably, branch testing 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 testing sufficiency and to select test data by which unexecuted branches are executed. This technique [6] is used in many tools such as RXVP [7], SADT [8], ATA [9] for Fortran, CIP [10] and SMOTL [11] for Cobol, HITS [12] for microcomputer software.

Conventional branch testing, however, has the following two defects because all branches are treated equally:

(1) Redundant test data is apt to be selected when the conventional branch testing is used for test data selection.

(2) Quality is overestimated when the conventional branch testing is used for quality assurance.

These problems can be avoided by paying attention to only essential branches 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, this paper introduces a directed graph obtained from a control flow graph of a tested program by eliminating arcs, which correspond to nonessential branches. Next, a new coverage measure based on the number of essential branches which is executed at least once by test runs of a program is proposed. Finally, it is experimentally ascertained that this measure is more suitable for quality assurance than a conventional measure based on all branches.

2. Conventional Method

2.1 Branch Testing

Software testing is generally 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 limitation.

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:

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

This measure, however, is not practical since the number of feasible paths is enormous in most programs be-

*This paper is a translation from Japanese of Transactions of IPSJ, Vol. 23, No. 5 (1982), which received an award as the 1982 winning paper of IPSJ.

**Systems Development Laboratory, Hitachi, Ltd., 1099 Ohzenji, Asao-ku, Kawasaki 215, Japan.

cause 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), is defined to be a partial path in a control flow graph such that (a) its first constituent edge emanates from either an entry node or a decision box, (b) its last constituent edge 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:

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

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

(1) It is used to detect a lack of test data, and additional data are selected so as to reach unexecuted dd paths.

(2) It is used as an index of testing sufficiency and it is considered that the higher the measure, the higher the quality of the tested program.

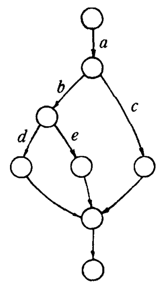
2.2 Problems of the Conventional Method

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.

```
S0 ;
if L1
   then if L2
           then S1
           else S2
   else S3 ;
S4 ;
```

(a)



(b)

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

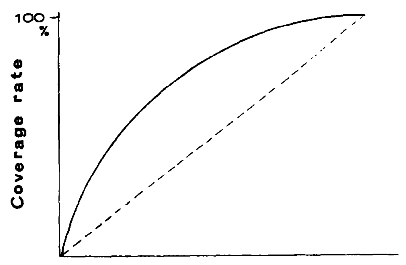


Fig. 2 Coverage rate curve of branch testing.

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(b). 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 5/5, respectively.

However, it is desirable that the coverage rate increases by 1/3 per case when the essential measure C_{path} is used, because 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 degree each case contributes to C_{dd} depends on the execution order, because the nonessential dd paths for path coverage, *a* and *b*, and the essential dd paths, *c*, *d* and *e*, are treated equally. Consequently, when using C_{dd} instead of C_{path}, there is a problem that 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%, C_{dd} is greater than a ratio of executed test data to all test data.

3. The Primitive Arc Concept

3.1 Primitive and Inheritor Arcs

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

Definition 1: A program is so transformed to a directed graph that a node corresponds to a basic block* which is a sequence of statements to be executed sequentially and an arc corresponds to control transfer between basic blocks. Each entry and exit is transformed into individual nodes.** 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

*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 with the subject of this paper.

**Since each entry or exit does not correspond to a node in the definition of a control flow graph in [13], 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.

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**.

3.2 Elimination of Inheritors

This section introduces several reduction rules which 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.

Proof: For two arcs a and b, let a be an inheritor of b, but not adjacent to b. All paths including b are represented by either of the following two sequences of arcs composing a path from an entry node to an exit node:

(1) $e_1 \cdots e_i, a, e_{i+1} \cdots e_j, b, e_{j+1} \cdots e_k$.

(2) $e_1 \cdots e_i, b, e_{i+1} \cdots e_j, a, e_{j+1} \cdots e_k$.

First consider case (1). Since a and b are not adjacent, e_{i+1} exists. Then, assume that a is not an inheritor of e_{i+1}. Because there must be at least one path including e_{i+1} but excluding a, the partial path from an entry node to e_{i+1} of this path can be connected to another partial path from e_{i+1} to an exit node in (1). Since this connected path includes b but excludes a, then a can not be an inheritor of b. This result leads to the contradiction. Consequently, a must be an inheritor of e_{i+1}. The proof of case (2) is the same as for case (1).□

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

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

Proof: It is obvious that a self-loop does not become an inheritor of the adjacent arc. Therefore, it is proved from Theorem 1 that a self-loop is not an inheritor of any arc.□

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 DOM(x) and IDOM(x) be sets of dominators and inverse dominators of x respectively. An algorithm for obtaining DOM(x) is detailed in [13]. An algorithm for obtaining IDOM(x) is led from the algorithm for obtaining DOM(x) by inverting arc directions.

Following the above consideration, 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. 3. The condition for a being an inheritor of b, c, d or e in Fig. 3, 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.

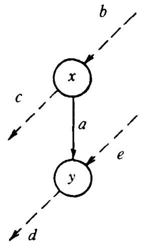


Fig. 3 General form of an arc and its two nodes.

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

(2) there are one or more e's, and a path passing through e necessarily passes 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. 4(a).

With respect to arrows of arcs in the figures, the bold line is an eliminated arc, the fine line is another arc in

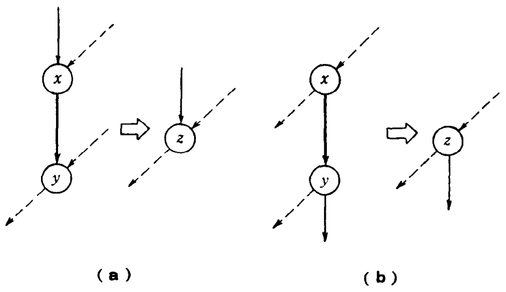


Fig. 4 Applications of the reduction rules. (a) R1. (b) R2.

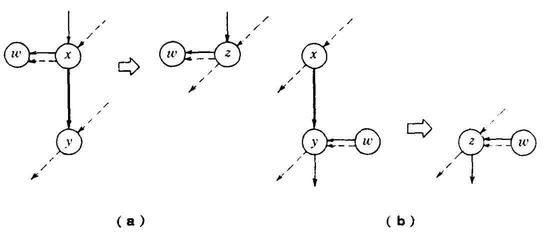


Fig. 5 Applications of the reduction rules. (a) R3 (x is an inverse dominator of w). (b) R4 (y is a dominator of w).

existence, and the broken lines are one or more arcs which 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. 4(b).

Reduction rule R3: Under Condition 1, if

$$\mathrm{OUT}(x)\geq 2 \text{ and}$$
$$x\in \mathrm{IDOM}(w) \quad \text{for } \forall\, w\in\{w|(x, w)\in A \wedge w\neq y\},$$

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

Reduction rule R4: Under Condition 1, if

$$\mathrm{IN}(y)\geq 2 \text{ and}$$
$$y\in \mathrm{DOM}(w) \quad \text{for } \forall\, w\in\{w|(w, y)\in A \wedge w\neq x\},$$

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

3.3 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:

(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) which 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) which 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 an inheritor mark are found.

In the remainder of this section, it will be proved that the directed graph reduced by this algorithm has the following features:

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

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

Several lemmas are introduced.

Lemma 1: By applying R1 or R2, a new path is never created and an old path is never lost.

Proof: It is obvious from Fig. 4.

Lemma 2: By applying R3 or R4, an old path is never lost.

Proof: It is obvious from Fig. 5.

Lemma 3: By applying R1, R2, R3 or R4, a new inheritance relation is never created.

Proof: Let a and b be arcs which are not eliminated by a reduction rule, and suppose that a is not an inheritor of b in the original graph. Then there is at least one path passing through b but not through a. Lemmas 1 and 2 assure that this path is never lost after reduction. Consequently, a does not become an inheritor of b.$\square$

Lemma 4: For any two arcs a and b (not necessary adjacent), if a is an inheritor of b, a can be eliminated by one of four rules.

Proof: It is obvious from Theorem 1.

Lemma 5: Any two arcs in a directed graph obtained by Algorithm 1, do not have an inheritance relation in the original graph.

Proof: Consider Algorithm 1. After step (1), there is no x satisfying the following condition:

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

Next, consider whether the merged node z satisfies the condition of R1 after step (3). Since the condition of R2 is

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

the following conditions on z are obtained:

$$\mathrm{IN}(z)=\mathrm{IN}(x)+(\mathrm{IN}(y)-1)=\mathrm{IN}(x), \text{ and}$$
$$\mathrm{OUT}(z)=(\mathrm{OUT}(x)-1)+\mathrm{OUT}(y)\geq \mathrm{OUT}(x).$$

Therefore, in order that z satisfies the condition of R1, the following condition should be necessary:

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

The existence of such an x contradicts the termination condition of step (2) described previously. Consequently, because step (3) never creates a node which satisfies the condition of R1, there are no nodes satisfying the condition of R1 or R2 after step (4).

By Lemma 4, an inheritor which exists after step (4) and not eliminated at step(10), is only such a one that all ancestors are appended inheritor marks at step (5) or (7). Then, after these ancestors are eliminated at step (10), this inheritance relation is removed. Consequently, all inheritance relations existing at step (1) do not remain after step (10).□

Lemma 6: A directed graph obtained by Algorithm 1 is an inheritor-reduced graph.

Proof: It is obvious from Lemmas 3 and 5.

Lemma 7: For any three arcs *a*, *b* and *c* in a directed graph, if *a* is an inheritor of *b* and *b* is an inheritor of *c*, then *a* is also an inheritor of *c*. (Transitive Law)

Proof: It is obvious by the definition of an inheritor.

Lemma 8: Among ancestors of an inheritor eliminated by Algorithm 1, at least one still remains in the reduced graph.

Proof: It is obvious from Lemma 7.

Theorem 4: 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 minimum among graphs with the feature of (1).

Proof: Item (1) is obvious from Lemma 8. Item (2) is obvious from Lemmas 5 and 3.□

In Algorithm 1, R1 and R2 are applied sequentially because of simplicity and convenience. R3 and R4, however, are not applied sequentially and arcs to be eliminated are marked instead because an inheritor relation is sometimes lost by a new path possibly created after application of R3 or R4.

Although the application order of R1 and R2 is no matter, 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 paticular arc in the original graph.

(2) Furthermore, since the corresponding arc in the original graph is a branch arc whose source node has two or more exit arcs, each arc in the reduced graph also corresponds uniquely to a particular dd path in the original graph.

4. Application to Quality Assurance

4.1 New Coverage Measure

To avoid overestimation of quality assurance of programs as described in Chapter 2, a new coverage measure C_{pr}, instead of C_{dd}, is defined below on the inheritor-reduced graph which is transformed from a tested program by Algorithm 1:

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

4.2 Features of C_{pr}

The control flow graph of Fig. 3.2 in [13] as shown in Fig. 6(a) is used as an example to experimentally observe the difference between C_{pr} and C_{dd}. This graph is transformed into the inheritor-reduced graph of Fig. 6(b) by Algorithm 1. Assume that the test data set of P_1–P_7 as shown in Table 1 is selected to cover all arcs in the control flow graph and executed in this order. It is seen in Fig. 7 that C_{pr} behaves more similarly to linear feature of C_{path} than C_{dd}.

Table 1 also demonstrates the reduction rules applied in Algorithm 1. However, arcs reduced by R1 are excluded since they are not branches and are not related to either C_{pr} or C_{dd}.

Next, the difference between C_{pr} and C_{dd} are theoretically analyzed, for a tested program, let n be the number of test data required to cover all dd paths. In the control flow graph, all arcs except arcs eliminated by R1 are divided into two classes of arcs eliminated by R2, R3 and R4, and other arcs. These arcs correspond to nonessential branches and essential branches respectively. Let F and G be the numbers of essential branches and nonessential branches respectively. Let f_i and g_i be the numbers of essential branches and nonessential branches executed with the i-th test data. Overlap rates α and β are defined as follows:

$$\alpha = \left(\sum_{i=1}^{n} f_i\right) \Big/ \mathrm{F}.$$

$$\beta = \left(\sum_{i=1}^{n} g_i\right) \Big/ \mathrm{G}.$$

The mean values of f_i's and g_i's are defined by $\bar{f}$ and $\bar{g}$ as follows:

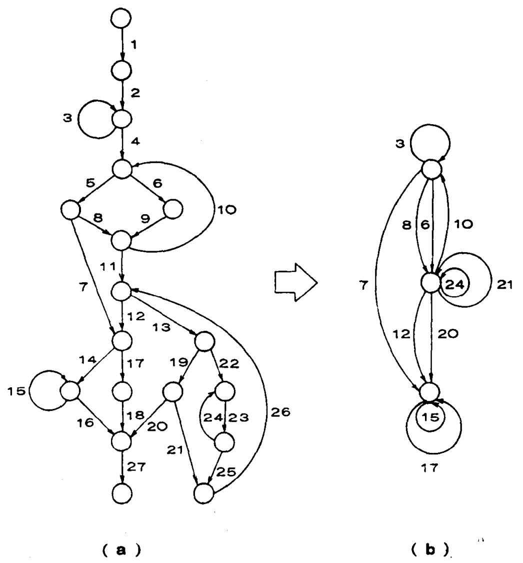


Fig. 6 An example for the comparison between C_{pr} and C_{dd}. (a) A control flow graph. (b) The inheritor-reduced graph.

Table 1 A set of paths for full coverage of all branches and the coverage rate with C_{dd} and C_{pr}.

		Arc No. ($\underline{n}$: Primitive arc)																					C_{dd}	C_{pr}
		1	$\underline{3}$	4	5	$\underline{6}$	$\underline{7}$	$\underline{8}$	$\underline{10}$	11	$\underline{12}$	13	14	$\underline{15}$	16	$\underline{17}$	19	$\underline{20}$	$\underline{21}$	22	$\underline{24}$	25	(%)	(%)
	P1	×		×	×					×	×	×				×				×		×	48	27
	P2	×		×		×				×	×		×		×								62	36
	P3	×		×		×				×	×	×				×	×		×				71	45
Paths	P4	×	×	×	×			×		×		×					×	×					81	64
	P5	×		×	×		×						×	×	×								90	82
	P6	×		×		×			×	×	×		×		×								95	91
	P7	×		×	×			×		×	×	×				×				×	×	×	100	100
Reduction rule		R4	—	R3	R2	—	—	—	—	R4	—	R2	R4	—	R3	—	R2	—	—	R4	—	R3		

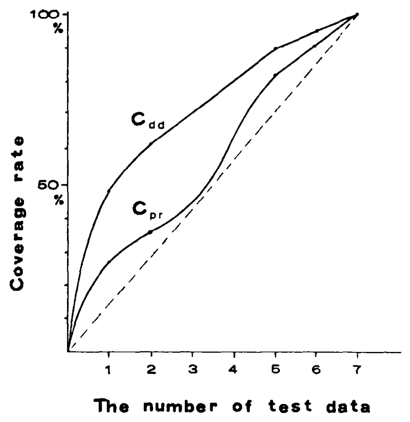


Fig. 7 Curves of C_{pr} and C_{dd}.

$$\bar{f}=\left(\sum_{i=1}^{n} f_i\right)\Big/n=\alpha F/n.$$

$$\bar{g}=\left(\sum_{i=1}^{n} g_i\right)\Big/n=\beta G/n.$$

When one average test data is first executed, C_{dd} and C_{pr} become as follows:

$$C_{dd}(1)=\frac{\bar{f}+\bar{g}}{F+G}=\frac{\alpha F+\beta G}{F+G}\cdot\frac{1}{n}.$$

$$C_{pr}(1)=\frac{\bar{f}}{F}=\frac{\alpha}{n}.$$

Therefore,

$$\frac{C_{dd}}{C_{pr}}=1+\frac{G}{F+G}\left(\frac{\beta}{\alpha}-1\right).$$

Since $\alpha<\beta$ generally,

$$C_{dd}>C_{pr}.$$

This result accords with the previous experimental result and confirms that the difference between C_{pr} and C_{dd} is greater as the ratio of the number of nonessential branches to that of essential branches increases. Applying this analysis to the example of Fig. 6, $\alpha=1.9$, $\beta=4.0$, $F=11$, $G=10$ and then $C_{dd}/C_{pr}=1.52$. This value of C_{pr}/C_{dd} is nearly equal to the experimental value of 1.75 in Fig. 7.

5. Conclusions

A new coverage measure for branch testing was proposed for more effective and efficient software testing. This measure is defined by coverage by 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 may imply full coverage of all branches. These are called primitive arcs in the control flow graph of a program. On the other hand, nonessential branches are called inheritor arcs because they inherit information about path coverage from other arcs.

In this paper, four reduction rules for eliminating inheritors from a directed graph were given. Then, the algorithm for applying these rules to a directed graph was introduced, and it was confirmed that this algorithm transformed a directed graph to an inheritor-reduced graph with no inheritors and that the number of arcs in the transformed graph was minimum among inheritor-reduced graphs transformed from a tested program.

It was demonstrated experimentally and theoretically that the new measure based on coverage of arcs in this inheritor-reduced graph had the following advantage in comparison with the conventional measure:

(1) avoidance of software quality overestimation. Furthermore, this measure has other advantages of:
(2) prevention of redundant test data selection, and
(3) decrease of the number of instrumentation codes which should be embedded into a program for collection of information about execution of branches by a tool for coverage rate evaluation.

Further study is needed for application to test data selection.

Acknowledgement

The author wishes to express his gratitude to Dr. Jun Kawasaki for providing the opportunity to conduct this study. He is also indebted to Tan Watanabe, Toru Kurosaki and Toshihiro Hayashi for their invaluable technical assistance.

References

1. Hoeden, W. E. A Survey of Dynamic Analysis Methods, *Tutorial*: *Software Testing & Validation Techniques*, *IEEE Catalog No. EHO* 138–8, New York, (1978), 184–206.
2. Myers, G. J. The Art of Software Testing, John Wiley & Sons, New York, 1979.
3. Howden, W. E. Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Softw. Eng.*, SE-2, 9 (1976), 208–214.
4. Weyuker, E. J. and Ostrand, T. J. Theories of Program Testing and the Application of Revealing Subdomains, *IEEE Trans. Softw. Eng.*, SE-6, 5 (1980), 236–246.
5. White, L. J. and Cohen, E. I. A Domain Strategy for Computer Program Testing, *IEEE Trans. Softw. Eng.*, SE-6, 5 (1980), 247–257.
6. Miller, E. F. Program Testing: Art Meets Theory, *Computer*, **10**, 7 (1977), 42–51.
7. Huang, J. C. Error Detection Through Program Testing, *Current Trends in Programming Methodology*, Vol. II, (R. T. Yeh, Ed.), Prentice-Hall, New Jersey, (1977), 16–43.
8. Voges, U. et al. SADAT-an Automated Testing Tool, *IEEE Trans. Softw. Eng.*, SE-6, 5 (1980), 286–290.
9. Holthouse, M. L. and Hatch, M. J. Experience with Automated Testing Analysis, *Computer*, **12**, 8 (1979), 33–36.
10. Sorkowitz, A. R. Certification testing: a Procedure to Improve the Quality of Software Testing, *Computer*, **12**, 8(1979), 20–24.
11. Bicevskis, J. et al. SMOTL-a System to Construct Samples for Data Processing Program Debugging, *IEEE Trans. Softw. Eng.*, SE-5, 1 (1979), 60–66.
12. Chusho, T. et al. HITS: a Symbolic Testing and Debugging System for Multilinguel Microcomputer Software, *Proc. NCC' 83*, (1983), 73–80.
13. Hecht, M. S. Flow Analysis of Computer Programs, North-Holland, New York, 1978.

(Received September 14, 1983; revised November 4, 1983)