7

Functional Testing and Structural Testing

*T. Chusho**

Abstract

The most important aspect of software testing is a method for selecting test data because the correctness of program logic is a major factor in software reliability, which is a part of software quality. These methods are categorized into functional testing and structural testing. The former implies that test data is selected on the basis of the functional specification of programs, and the latter implies that test data is selected on the basis of the control structure of programs.

The testing methods are twofold. First, a systematic test case generation method for functional testing (AGENT) is developed. AGENT includes a functional diagram (FD) which formally expresses the functional specification of programs by using a state transition model and a boolean function model. AGENT automatically generates test cases from an FD.

Structural testing is more practical and compensates for the inadequacies of functional testing. The conventional coverage measure for branch testing has, because all branches are treated equally, defects such as overestimation of software quality and redundant test data selection. These problems can be avoided by paying attention to only those branches essential for path testing.

In addition, a testing tool for the new measure (SCORE) is developed to differentiate essential branches from non-essential 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% of all branches. As a result, the prevention of redundant test data selection is confirmed by a reasonable algorithm for test data selection based on a 40% reduction in the number of branches to be monitored. Furthermore, an efficient algorithm for redundancy elimination in a selected test data set is presented.

*Systems Development Laboratory, Hitachi Ltd., Kawasaki, Japan

7.1 Introduction

Program testing constitutes approximately half of the overall cost of software development 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 finding a method for selecting test data [3] because the correctness of program logic is a major factor in software reliability, which is a part of software quality.

Such methods are categorized into functional testing and structural testing. The former implies that test data is selected on the basis of the function specification of a program, and the latter implies that test data is selected on the basis of the control structure of a program.

Functional testing is thought to be more basic than structural testing because errors in a program are defined as behaviour discordant with the functional specification of the program. A systematic test case generation method for functional testing called AGENT (automated generation method of test cases) has been developed [4,5]. AGENT consists of the following components:

- a functional diagram (FD) which formally expresses the functional specification of a program;
- a mechanical procedure for generating test cases from the FD.

An FD model is composed of a state transition model and a boolean function model. AGENT automatically generates test cases from an FD. Section 7.2 presents the method used for generating test cases. It explains the FD notation and the generation algorithms and provides a concise outline of the AGENT program.

Structural testing is more practical and compensates for the inadequacies of functional testing. There are several methods and tools for structural testing [6]. In particular, a lot of attention has been paid to branch testing, a form of simplified path testing. A typical branch testing tool measures the ratio of executed branches to all branches in a program. The coverage measure [7,8] is used to estimate the quality of a tested program with regard to the correctness of program logic and to select test data by which normally unexecuted branches are executed.

However, because all branches are treated equally, the conventional coverage measure for branch testing has defects such as overestimation of software quality and redundant test data selection. 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 non-essential for path testing.

This is because a path covering the latter branch also covers the former branch. Branches other than such non-essential branches will be referred to as essential branches.

SCORE (the source-level coverage rate evaluator), a testing tool for the new measure, has been developed in order to differentiate essential branches from non-essential branches and to measure the coverage rate of these essential branches. It has been found that the problems of conventional branch testing are reduced if this tool is used. Section 7.3 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.

7.2 A functional testing tool

7.2.1 *A functional testing method*

Functional testing is thought to be more basic than structural testing because errors in a program are defined as behaviour discordant with the functional specification of the program. One conventional approach to functional testing is to have people read a functional specification and to pick out test cases intuitively. However, high software reliability cannot be achieved by this method.

AGENT, a systematic test case generation method for functional testing, was therefore developed. AGENT is composed of the following components:

- an FD, which formally expresses the functional specification of a program;
- a mechanical procedure for generating test cases from the FD.

An FD model is composed of a state transition model and a boolean function model. A state transition expresses input data sequences and corresponding output data. A boolean function in a state expresses the correspondence between conditions on input and output data. A test case constitutes a sequence of states passed through in testing and a pair of conditions in each state which must be satisfied by the input and output data. AGENT automatically generates test cases from an FD.

AGENT has the following characteristics:

- Description of a function specification with an FD is easier than with a boolean function alone because the order of input data need not to be changed to satisfy the constraints needed for a boolean function.

- Criteria for test case generation are clear, and the number of test cases generated is reasonable.
- Test cases are automatically generated from an FD.

This section presents the method used for generating test cases. It explains FD notation and the generation algorithm, and it provides a concise outline of the AGENT program.

7.2.2 *Function diagram*

Some studies have been carried out on the description of a functional specification as a dependence between input and output data. A cause–effect graph expresses the correspondence between input and output data based on a boolean function and the interdependence between input data owing to constraint conditions.

A functional specification of a program usually consists of a dynamic specification and a static specification. The former expresses the order of input data (or translation order); the latter, a correspondence between input and output conditions. A specification that consists only of a static specification is impractical because of the large number of possible combinations. A dynamic specification must be used to simplify the functional specification. A state transition model is suitable for describing a dynamic specification. In a state transition model, the output data and subsequent state are determined by the input data and the present state. A boolean logic model is suitable for describing a static specification. In a boolean logic model, the output data is determined only by the input data.

7.2.2.1 Function diagram notation

A function diagram is composed of state transitions and boolean functions as follows:

State transition A state transition is described with states and transitions as shown in Figure 7.1.

- A state indicates a place (or time) wherein data is input. An initial state is the starting point of activity and final states are possible end points of activity.
- A transition indicates a change of a state. A state prior to transition is called a tail state and, after transition, a head state. The boxed T symbol over an arrow expresses a boolean function with a decision

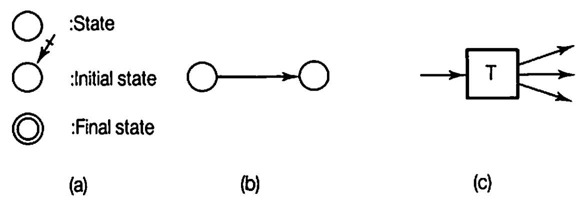


Figure 7.1 *State transition notation in function diagrams: (a) state; (b) transition; (c) decision table or cause–effect graph.*

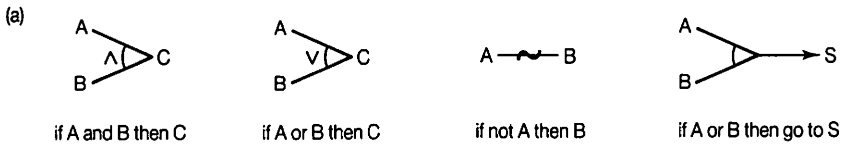
table or a cause–effect graph. If the boolean function is simple, then the input and output condition is written directly on the arrow.

Boolean function A boolean function in each state is described with a cause–effect graph or a decision table as shown in Figure 7.2. Interdependences between input conditions are expressed in terms of constraint conditions.

7.2.2.2 An example of a function diagram

Figure 7.3 is a sample function diagram representing the following specification of a simplified automatic teller machine (ATM). State S3 represents the display ‘insert card’.

(1) The ATM displays ‘key in pass code’ when a cash card is inserted.
(2) The ATM checks the correspondence between the keyed-in pass code and the code on file. If they are the same, the ATM displays



(b)

input	C1	Y	Y
	C2	Y	N
output	E1	*	
	E2		*
state	S1		*
	S2	*	

Y: input condition is true
N: input condition is false

*: true
 :false

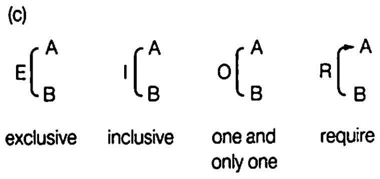
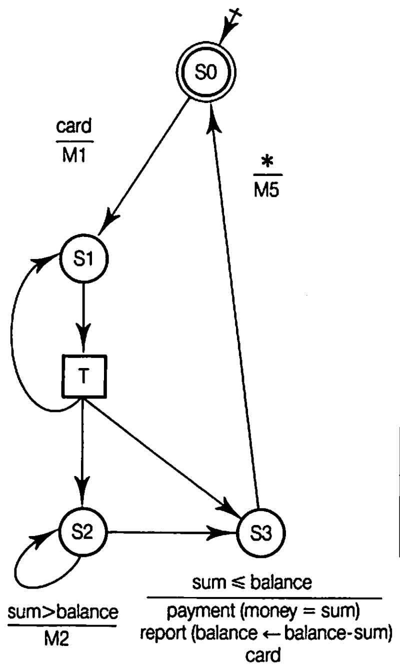


Figure 7.2 *Boolean function notation in functional diagrams: (a) cause–effect graph; (b) decision table; (c) constraint conditions.*


S0
card / M1
* / M5
S1
T
S2
S3
sum>balance / M2
sum ≤ balance / payment (money = sum) report (balance ← balance-sum) card

(a)

messages
M1: key in pass code
M2: key in sum
M3: key in pass code again
M4: stop process
M5: insert card

Decision table: T

input	pass code = registration	Y	N	N
	mistaken = three times	N	Y	N
output	M2	*		
	M3			*
	M4		*	
	cancel card		*	
head state	S1			*
	S2	*		
	S3		*	

(b)

Figure 7.3 *An example of a function diagram (ATM): (a) state transition part; (b) boolean function part.*

'key in sum'; if not, the ATM checks whether the number of times that the correct code has not been entered is equal to three. If so, the ATM displays 'stop process', cancels the card registration, and displays 'insert card' for the next customer. If not, the ATM displays 'key in pass code again'.

(3) When an amount is keyed in, the ATM checks whether it is less than or equal to the balance. If the amount is greater than the balance, the ATM displays 'key in sum' and waits for the amount to be keyed in again. If the amount is less than or equal to the balance, the ATM pays the money requested, reports the balance and displays 'insert card'.

7.2.3 *Method of test case generation*

In generating test cases from an FD, it is important that the number of test cases be practical and that the criteria for test case generation be clear. An FD is composed of state transitions and boolean functions.

Myers [9] described a test case generation method for boolean functions in which test cases are generated from a cause–effect graph. The criterion considered for a state transition is the path testing strategy of programs where a state is substituted for a node and a transition is substituted for a branch.

Using clearly defined criteria, a practical number of test cases are generated from an FD by combining test cases of state transitions (testing paths) with test cases of boolean functions (partial test cases).

7.2.3.1 Test case generation criteria

Partial test cases which are generated from the boolean functions of an FD include true and false cases of input data conditions; this constitutes almost the minimum possible number of test cases. The number of test cases increases linearly with the number of input data conditions.

There are many criteria for state transitions. Among these, pass-all-states (C0 coverage) and get-through-all-transitions (C1 coverage) are well known. In programming, the case when a loop is bypassed is apt to be overlooked and mistakes can be made concerning a loop termination condition. Testing paths should include both a case of bypassing a loop and one of getting through a loop in a state transition.

It is difficult to recognize all the loops of a typical state transition but easy to recognize them in a structured state transition (SST) which is composed of only three kinds of forms (for example, sequence, selection and iteration, as shown in Figure 7.4(a)). Figure 7.5(b) shows an example

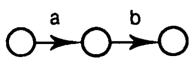
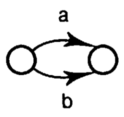
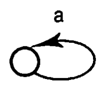
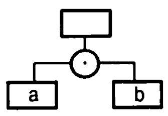
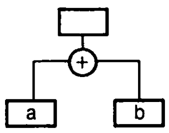
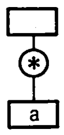
		Sequence	Selection	Iteration
(a)	Transition expression			
(b)	Regular expression	a·b	a+b	a*
(c)	Tree expression			

Figure 7.4 *Notation for an SST.*

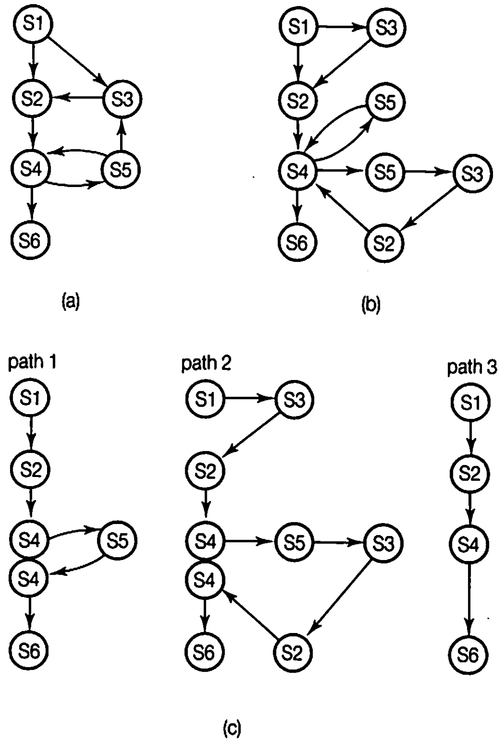


Figure 7.5 *An example of an SST and test paths: (a) state transition; (b) SST; (c) testing paths.*

of an SST corresponding to the state transition in Figure 7.5(a). There are two loops in Figure 7.5(b): (S4, S5, S4) and (S4, S5, S3, S2, S4).

To test path generation, the test cases must pass all transitions of an SST at least once, and they must include the cases of bypassing and getting through a loop. Figure 7.5(c) is an example of testing paths which satisfy this criterion.

7.2.3.2 Test case generation procedure

Test cases are generated from an FD by the following procedure.

(1) Generation of partial test cases. In each state, partial test cases are generated from a cause–effect graph in which causes are input data conditions and effects are output data or head states of the boolean function. If a decision table is used to describe the boolean function, it is translated to the equivalent cause–effect graph. A partial test

case is composed of a combination of cause values (input conditions) and a combination of effect values (output data or states) which corresponds to a combination of cause values. Only one of the effects which correspond to the states should be true in each partial test case.

(2) Testing path generation. A state transition is translated to an SST composed of the three forms in Figure 7.4(a) by means of a translation from a transition matrix to an expression [10] composed of the three operations shown in Figure 7.4(b). A testing path is generated from the initial to the final state with the criteria in the previous section always being satisfied.

(3) Synthesis of test cases. Test cases of an FD are synthesized from the testing paths and partial test cases in each state of the FD. They are made up of sequences of states from the initial to the final states and combinations of input data conditions and combinations of output data in each state. In the synthesis algorithm, a partial test case is assigned to a state of a testing path in which a head state of the partial test case is the same as the next state of the assigned state in the testing path.

7.2.3.3 Synthesis algorithm for test cases

An SST is expressed as described by the tree expression shown in Figure 7.4(c) for test case synthesis. A tree expression of an SST is called a condition structure tree (CST). The CST of the ATM is presented in Figure 7.6.

Test cases are synthesized according to the following steps:

(1) The number of partial test cases which have the same head state is counted in each state, and the sum is set for the corresponding leaf of the CST.

(2) Each node of the CST is retrieved in post order, and the number of test cases in each node is calculated according to the rules shown in Figure 7.7.

(3) Test cases are synthesized in numerical order from 1 to *n* where *n* is the number of test cases in the root node of the CST, while test cases are made up in each node from the children's test cases.

A test case is composed of a sequence of tail states which are extracted at the leaves and combinations of input conditions and output data which are parts of the partial test cases at the leaves.

The numerals on the right-hand side of the CST node box in Figure 7.7 represent the number of test cases of each node. The ATM example has four test cases. Each test case is described in Figure 7.8.

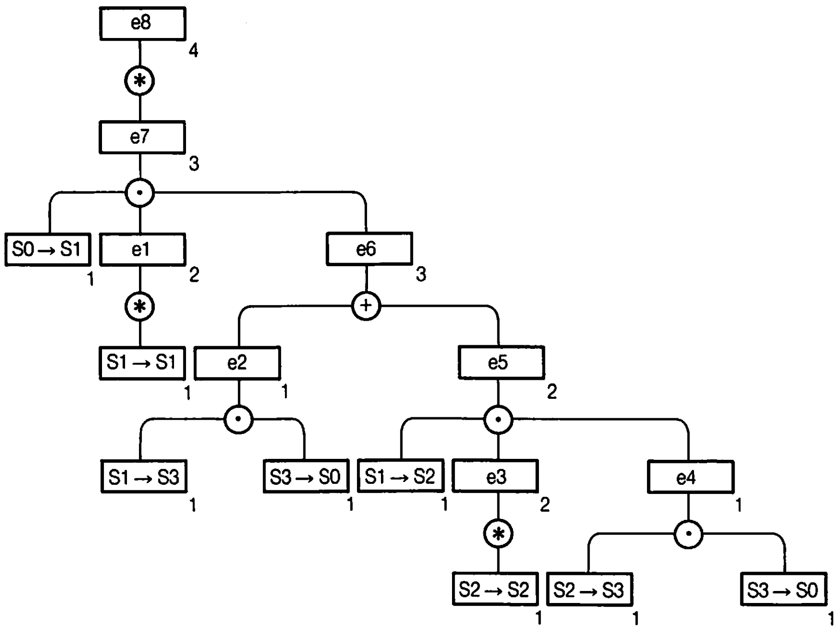


Figure 7.6 *An example of a condition structure tree.*

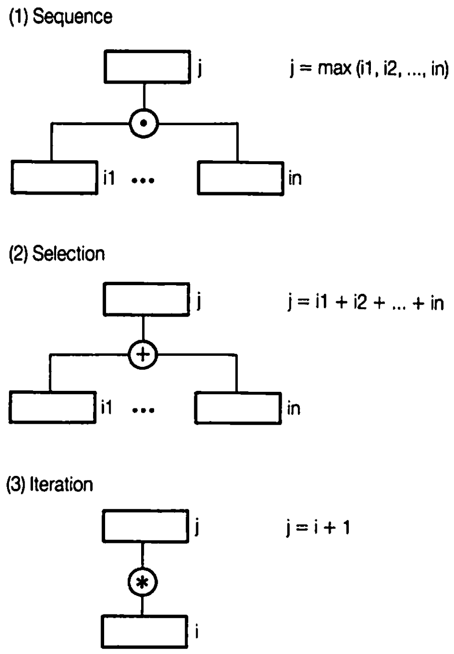


Figure 7.7 *The rule for counting the numbers of test cases.*

Test case 1

(empty)

Test case 2

$$\begin{tikzcd}
S0 \arrow[d, "\text{(insert card)}"] \\
S1 \arrow[d, "\text{not (pass code = registration code) and (mistaken = three times)}"] \\
S3 \arrow[d, "\text{(*-uncondition)}"] \\
S0
\end{tikzcd}$$

Test case 3

$$\begin{tikzcd}
S0 \arrow[d, "\text{(insert card)}"] \\
S1 \arrow[d, "\text{not (pass code = registration code) and not (mistaken = three times)}"] \\
S1 \arrow[d, "\text{(pass code = registration code) and not (mistaken = three times)}"] \\
S2 \arrow[d, "\text{(sum balance)}"] \\
S3 \arrow[d, "\text{(*)}"] \\
S0
\end{tikzcd}$$

Test case 4

$$\begin{tikzcd}
S0 \arrow[d, "\text{(insert card)}"] \\
S1 \arrow[d, "\text{(pass code = registration code) and not (mistaken = three times)}"] \\
S2 \arrow[d, "\text{(sum balance)}"] \\
S2 \arrow[d, "\text{(sum balance)}"] \\
S3 \arrow[d, "\text{(*)}"] \\
S0
\end{tikzcd}$$

Figure 7.8 *An example of test cases.*

7.2.4 *The AGENT program*

The AGENT program, which automatically generates test cases from an FD, was written in PL/I. The functions of the AGENT program and some of our experiences with it are briefly described below.

7.2.4.1 Input and output

The input to the AGENT program is an FD which is written in FD language. An FD is composed of a title statement (TITLE), state statements (STATE), an initial state statement (INITIAL), a final state statement (FINAL) and an end statement (END). In each state statement, a boolean function is described with condition definitions (NODE), boolean expression (RELATE) or decision table definitions (DECISION), and constraint condition definitions (CONST).

The main output of AGENT is a table of test cases. A source list, decision tables and a transition matrix can also be output. In test case tables, a transition is composed of a tail state (TAIL) and a head state (HEAD). Conditions (NODE) in the tail state are divided into those for input data (I) and output data (O). Each test case is expressed in one column of a test case table. A blank indicates that a test case did not get through that transition, an 'O' indicates that a condition is true, and an 'X' indicates that a condition is false.

The first test case in Figure 7.8 did not get through any transition. Since the ATM example has the same initial and final state (S0), the first test case is a case of bypassing the loops at S0. The second test case gets through a sequence of states (S0, S1, S3, S0) where, at each state, input conditions are set and output data is checked. For example, at state S1, the former are to key in an erroneous pass code and to iterate three times to the error condition and the latter are to certify that the message 'key in pass code again' is not output, the message 'stop process' is output, and the card registration is cancelled.

7.2.4.2 Experiences with AGENT

Table 7.1 is a summary of AGENT applications. Numbers 1–5 are experimental specifications for FD descriptions; numbers 6–11 are parts of practical software. The following points have been deduced from the data and user suggestions.

(1) Size of the FD. As shown in Table 7.1, the number of states is less than 15. It is thought that this is due to the conventional notation for describing a functional specification (natural language or state

Table 7.1 *Summary of experience*

Number	n	L	e	e_s	T
1	4	1.5	7	8	4
2	2	4.5	1	1	15
3	1	13.0	1	1	12
4	9	3.4	21	59	25
5	2	4.5	3	3	13
6	6	2.5	11	14	13
7	5	3.2	11	11	12
8	9	4.2	21	51	33
9	11	2.7	15	33	22
10	10	18.5	28	37	131
11	7	18.9	28	104	95

n, the number of states in a state transition diagram;
L, the mean number of input conditions;
e, the number of transitions;
e_s, the number of SST transitions;
T, the number of test cases.

transition matrix etc.). Thus the limitation on the number of states may be changed by using an FD. In our experience, when the number of states is over 20, the FD cannot be readily grasped by human operators.

The mean number of input conditions varies widely depending on the software. In our experience with cause–effect graphs, the number of conditions (input conditions, output data and head states in an FD) is limited to 30–40. There is a problem in the decision table (which is induced to an FD to describe boolean functions) in that the combination of conditions increases exponentially in proportion to the number of input conditions. Thus the number of input conditions at each state should be kept as small as possible.

(2) Remarks on writing an FD. It is noticeably easier to describe the ordered logic of a functional specification in an FD than in a cause–effect graph. However, when states are created without much thought, non-executable test cases are apt to be generated by AGENT. Non-executable test cases are caused by inadequate attention to state setting. Such cases can be solved by state decomposition.

(3) Effective use of the AGENT method. The AGENT method is particularly effective because it allows detection of design errors in a functional specification by translating it to an FD. Even inexperienced personnel are able to generate test cases with AGENT. The following factors enhance the effectiveness of the AGENT program.

- Description of an FD at the design stage. This is effective for early detection of design errors. The FD review ensures that development people will have a common understanding of a functional specification.
- Use of inexperienced personnel for generation and execution of basic test cases in functional test. This allows experienced people to concentrate on cases not covered by the AGENT method or on very special cases which are deduced by experience and human intuition.

7.3 A structural testing tool

7.3.1 *Essential branches for path testing*

There are several methods and tools for structural testing. In particular, a lot of attention has recently been paid to path testing. 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 are selected and input data that will cause them to be executed is found. It should be noted that 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 the correctness of program logic and to select test data by which unexecuted branches are executed. This technique is used in many tools.

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

- Redundant test data is 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.
- 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 result because all branches are treated 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 non-essential for path testing. This is because a path covering the latter branch also covers the former branch. Branches other than such non-essential branches will be referred to as essential branches.

First of all, to present a method for differentiating essential branches from non-essential branches, this section introduces a directed graph, obtained from a control flow graph of a program by eliminating arcs which correspond to non-essential branches. All arcs of this group 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.

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.

Finally, 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 are presented.

7.3.2 *Conventional method*

7.3.2.1 Branch testing

In general, program testing is carried out by dynamic testing in such a way that a program is executed with various input data and then each result is confirmed. It is impossible, however, to test all possible input data with this method. Therefore a finite test data set should be selected so as to ensure a high quality of the tested program under the given time and cost constraints.

Path testing is one technique for this purpose. The aim of this method is 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 focused on a path component instead of a path. This component, called the **dd** path (decision-to-decision path), 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 for those at the two 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:

$$C_1 = \frac{\text{the number of executed dd paths}}{\text{the number of all dd paths in a tested program}}$$

This technique is called branch testing because this measure promotes the execution of all branches. This measure can be used for the following:

- To detect a lack of test data, and to select additional data so as to reach unexecuted dd paths.
- To estimate the quality of a tested program, assuming that the higher the measure, the higher the quality of the tested program.

7.3.2.2 Problems of the conventional method

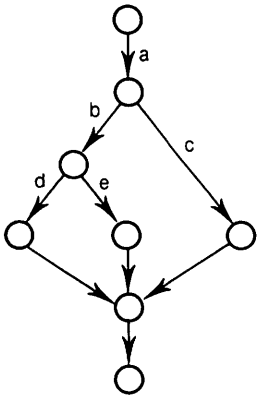
For a demonstration of the first problem, consider the program in Figure 7.9 and the following test cases:

- Case 1: logical predicates L_1 and L_2 are both 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 Figure 7.9. When case 1 is first executed, a, b and d are covered and C_1 is 3/5. After cases 2 and 3 are executed sequentially, C_1 will become 4/5 and then 5/5 respectively.

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

(a)



(b)

Figure 7.9 *A program example: (a) source; (b) the control flow graph.*

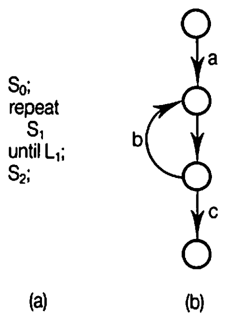


Figure 7.10 *A program example: (a) source; (b) the control flow graph.*

However, it is desirable that the coverage rate increases by 1/3 per case when the measure C_{path} of essential paths is used, since there are three paths in this program. The difference between the ways in which C_1 and C_{path} increase is due to the fact that the non-essential 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_1 depends on the execution order.

Consequently, when C_1 is used instead of C_{path}, the quality of the tested program is overestimated. That is, when a coverage rate is less than 100%, C_1 is greater than the ratio of executed test data to all test data.

Next, consider another program, shown in Figure 7.10, and the following test cases to demonstrate the second problem:

- 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 the defect that redundant test data is 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 paths a and c are not.

7.3.3 *The primitive arc concept*

7.3.3.1 Primitive and inheritor arcs

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

□ **Definition 1:**
A program is transformed to a directed graph in such a way that a node will correspond to a basic block [11] which is a sequence of sequentially executed statements and that an arc will correspond 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 chapter, 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 IN(x) be the number of arcs entering x and OUT(x) be the number of arcs exiting from x. A node x with IN$(x) = 0$ is called an **entry node** and a node x with 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.** ■

7.3.3.2 Elimination of inheritors

This section introduces several reduction rules that can be used 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 given in a previous paper [12].

☐ **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 given in a previous paper [12].

☐ **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 respectively of x. An algorithm for obtaining DOM(x) is detailed in [11]. An algorithm for obtaining IDOM(x) is derived from the algorithm for obtaining DOM(x) by inverting arc directions. ■

The condition for an arc to be an inheritor will now be discussed in view of the above considerations. 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 Figure 7.11, where the broken line implies one or more arcs that may exist.

The condition for a to be an inheritor of b, c, d or e in Figure 7.11 will be examined by considering the following four cases.

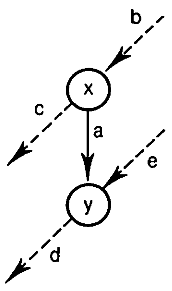


Figure 7.11 *General form of an arc and its two nodes.*

- **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 one of the following conditions holds:
  (1) There is no c.
  (2) There are one or more arcs c, 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 for 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 one of the following conditions holds:
  (1) There is no e.
  (2) There are one or more arcs e, 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 for this case is the same as the second condition of case 3.

The above four conditions lead to 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 node as shown in Figure 7.12(a). ■

With respect to the arc arrows in Figures 7.12 and 7.13, the bold line is an eliminated arc, the fine line is another extant arc, 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 node as shown in Figure 7.12(b). ■

□ **Reduction rule R3:**
Under condition 1, if

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

and

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

(x, y) is eliminated from A, and x and y are merged into one node as shown in Figure 7.13(a). ■

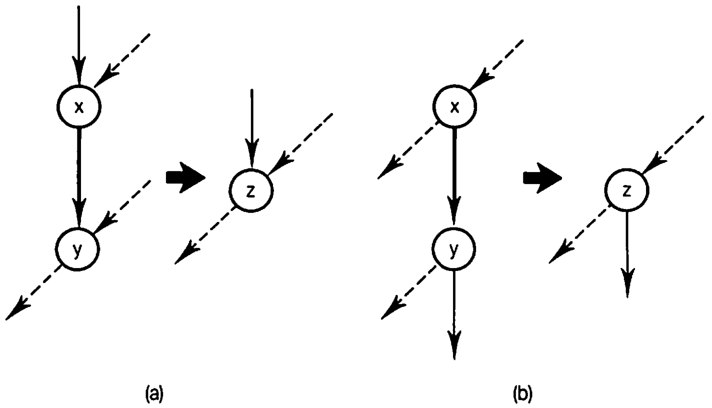


Figure 7.12 *Applications of the reduction rules: (a) R1; (b) R2.*

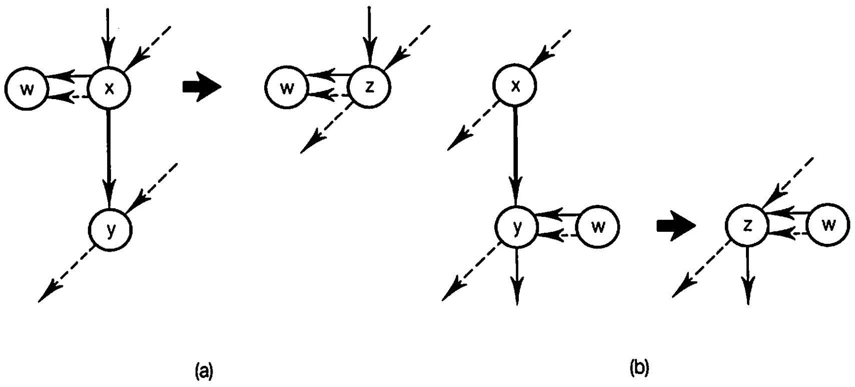


Figure 7.13 *Applications of the reduction rules: (a) R3 (x is an inverse dominator of w); (b) R4 (y is a dominator of w).*

□ **Reduction rule R4:**
Under condition 1, if

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

and

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

(x, y) is eliminated from A, and x and y are merged into one node as shown in Figure 7.13(b). ■

7.3.3.3 Reduction Algorithm

The algorithm for transforming a directed graph to an inheritor-reduced graph can be given in terms of the four reduction rules R1–R4 as follows.

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

(1) R1 is applied for any arc which satisfies the condition of R1.
(2) Step (1) is repeated until no further suitable arcs are found.
(3) R2 is applied for any arc which satisfies the condition of R2.
(4) Step (3) is repeated until no further suitable arcs are found.
(5) An inheritor mark is written on any arc (x, y) that satisfies the condition of R3 if there is at least one arc without an inheritor mark among the input arcs of x or among the arcs forming a path from the output arcs of x to x except (x, y) itself.
(6) Step (5) is repeated until no further suitable arcs are found.
(7) An inheritor mark is written 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 the arcs forming a path reaching inversely from the input arcs of y to y except (x, y) itself.
(8) Step (7) is repeated until no further suitable arcs are found.
(9) Any arc with an inheritor mark is eliminated and the two nodes on both ends of the this arc are merged into one node.
(10) Step (9) is repeated until no arcs with inheritor marks are found. ■

The following theorem assures us that this algorithm is correct and optimal.

□ **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 given in a previous paper [12].

Although the order in which R1 and R2 are applied 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 non-essential branches in the original program.

For example, consider the control flow graph of Figure 3.2 in [11]. This graph is transformed to the inheritor-reduced graph by applying Algorithm 1 as shown in Figure 7.14(a). The numerals in Figure 7.14(a) imply that an arc labelled with the number n is eliminated by the reduction rule Rn. The other arcs labelled with p are primitive arcs in Figure 7.14(b).

7.3.4 *New coverage measure and its support tool*

7.3.4.1 New coverage measure

In order to improve both the effectiveness and the efficiency of branch testing, a new coverage measure C_{pr}, instead of C_1, is defined below on an inheritor-reduced graph into which a program has been transformed by Algorithm 1:

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

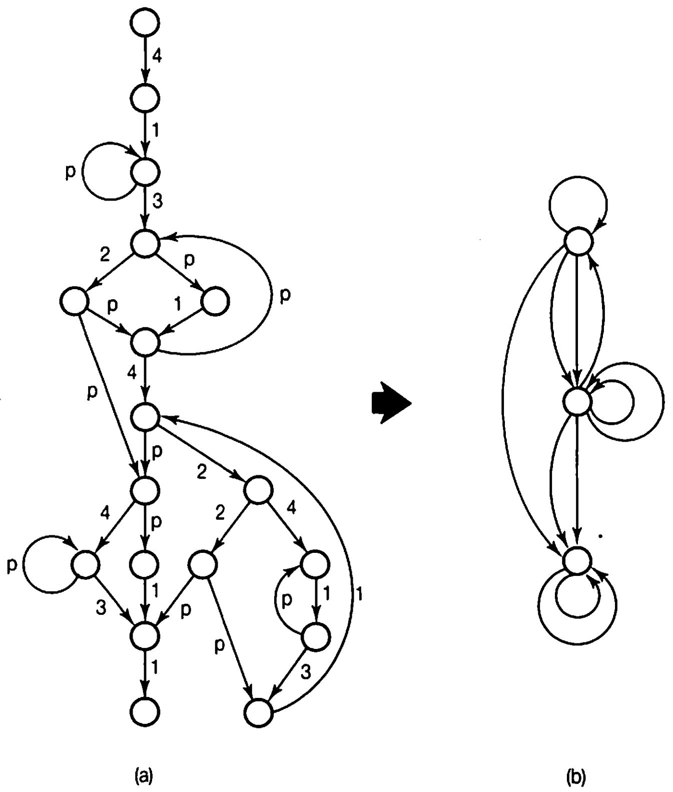


Figure 7.14 *An example of the application of Algorithm 1: (a) a control flow graph; (b) the inheritor-reduced graph.*

7.3.4.2 A tool for new coverage measure

The tool for the measurement of C_{pr}, SCORE, was developed and used for comparisons between C_{pr} and C_1. SCORE is applicable to Pascal programs and is composed of the following four phases:

- P1: a Pascal program is transformed to the control flow graph.
- P2: the control flow graph is transformed to the inheritor-reduced graph by Algorithm 1.
- P3: an instrument code is embedded into any place in the source program corresponding to any arc in the inheritor-reduced graph.
- P4: the coverage rate C_{pr} and unexecuted essential branches are printed out after the code-embedded program is executed.

7.3.5 Reduction of branches to be monitored

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

(1) PL0 parser: the parser for the language PL0, whose source code is given on pp. 314–19 of Wirth's book [13].
(2) SCORE: the tool itself for C_{pr}, whose four phases (P1, P2, P3 and P4) are used separately.
(3) PARSE: a structure editor developed by our group [14].

The PL0 parser was selected to ensure objectivity and the others were selected as examples of rather large programs.

The overall result is shown in Table 7.2. Item 2 is the total number of branches in a tested program; all these branches are monitored when

Table 7.2 *Reduction of branches to be monitored when applying Algorithm 1 to a Pascal program*

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	11027
(2) The number of branches (the denominator of C_l)	136	582	533	230	553	2914	4948
(3) The number of eliminated branches:							
(a) using the R2 rule	40	87	73	44	52	754	1050
(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 (the denominator of C_{pr})	81	343	298	157	323	1874	3076
The ratio of (4):(2)	0.60	0.59	0.56	0.68	0.58	0.64	0.62

C_1 is applied as the coverage measure for branch testing. Item 3 is the number of branches eliminated as non-essential branches by reduction rules R2, R3 and R4 in Algorithm 1. Table 7.2 omits the number of arcs eliminated by reduction rule R1 because they have no relation to the comparison between C_{pr} and C_1. 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% by using C_{pr} instead of C_1.

As a result, this reduction implies the following advantages when C_{pr} is used for preparing additional test data so that otherwise unexecuted branches will be executed:

- Effectiveness: the possibility of selecting redundant test data is less because attention is only paid to essential branches.
- Efficiency: it is easier to select additional test data because the number of branches under consideration is fewer.

7.3.6 *Application to test data selection*

7.3.6.1 Algorithms for test data selection

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

- Consideration should be limited to arcs in an inheritor-reduced graph, that is, to essential branches.
- 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 is 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) On the basis of this inheritor-reduced graph, select test data to include as many of the following (in decreasing order of priority) arcs as possible among those arcs not yet included in the executed paths of selected test data:

- a self-loop;
- a backward arc [11];
- 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. When this algorithm is used, the amount of redundant test data decreases 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.

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

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

$$\begin{aligned}
&N = \{n_1, \ldots, n_k\}\\
&(n_i, n_{i+1}) \in A \quad \text{for} \quad i = 1, \ldots, k-1\\
&(n_i, n_j) \notin A \quad \text{for} \quad j \neq i+1\\
&\mathrm{IN}(n_1) = 0\\
&\mathrm{OUT}(n_i) = \mathrm{IN}(n_{i+1}) \geq 2 \quad \text{for} \quad i = 1, \ldots, k-1\\
&\mathrm{OUT}(n_k) = 0
\end{aligned}$$ ■

Let h_i be the number of (n_i, n_{i+1}) represented by $a_{i1}, \ldots, a_{ih_i}$ as shown in Figure 7.15.

□ **Theorem 4:**
Under condition 2, the minimum possible number of paths required to cover all arcs is

$$\max_i (h_i)$$ ■

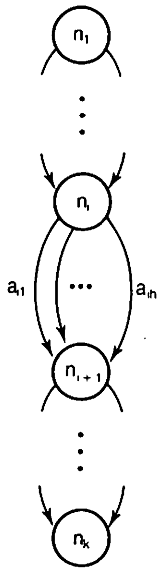


Figure 7.15 *A chain of nodes.*

□ **Theorem 5:**
Under condition 2, the maximum possible number of paths required to cover all arcs is

$$\sum_{i=1}^{k-1} (h_i - 1)$$ ■

The proofs of these theorems are obvious and are therefore omitted.

7.3.6.2 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% 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:

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

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

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

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

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

(5) Repeat steps (3) and (4) 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 only primitive arcs are treated.

7.4 Conclusions

A functional testing tool (AGENT) and a structural testing tool (SCORE) have been introduced.

First, the AGENT method has been proposed for systematic test-case generation for functional testing. The AGENT program automatically generates test cases from a function diagram. Generated test cases satisfy the following criteria for a function diagram which expresses a functional specification:

(1) Each state includes true and false cases of input and output conditions.

(2) All transitions of a state transition are retrieved at least once, and bypassing and getting through all loops in a state transition are included.

Experiences with the AGENT method and its program have shown it to be effective for standardizing test-case generation and simplifying test-case management.

Second, 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 non-essential 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% of all branches.

As a result, the new measure had the following advantages in comparison with 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.

Acknowledgements

The author wishes to express his gratitude to Dr. Jun Kawasaki for providing the opportunity to conduct this study. He is also indebted to Kenroku Nogi, Zengo Furukawa and Katsuyuki Yasuda, who developed AGENT, for their invaluable technical assistance.

References and further reading

[1] Howden W.E. (1978). A survey of dynamic analysis methods. In *Tutorial: Software Testing & Validation Techniques*, IEEE Catalog No. EHO 138-8, pp. 184–206. New York: IEEE
[2] Chusho T. (1983). HITS: a symbolic testing and debugging system for multilingual microcomputer software. In *Proc. AFIPS NCC '83*, pp. 73–80
[3] Goodenough J.B. and Gerhalt S.L. (1975). Toward a theory of test data selection. *IEEE Trans. Software Engineering*, **1**, 156–73
[4] Furukawa Z., *et al.* (1982). AGENT: automatic generation of test-cases with cause–effect graphs. In *Proc. 6th ICSE Poster Session*
[5] Furukawa Z., *et al.* (1985). AGENT: an advanced test-case generation system for functional testing. In *Proc. NCC '85*, pp. 525–35

[6] Howden W.E. (1976). Reliability of the path analysis testing strategy. *IEEE Trans. Software Engineering*, **2**, 208–14

[7] Miller E.F. (1977). Program testing: art meets theory. *IEEE Computer*, **10**, 42–51

[8] Huang J.C. (1977). Error detection through program testing. In *Current Trends in Programming Methodology*, Vol. 2, pp. 16–43. Englewood Cliffs NJ: Prentice-Hall

[9] Myers G.J. (1979). *The Art of Software Testing*. New York: Wiley

[10] Lanzarone (1982). G.A. Automatic functional test case generation for real-time control systems. In *Proc. 6th ICSE Poster Session*

[11] Hecht M.S. (1978). *Flow Analysis of Computer Programs*. New York: North-Holland

[12] Chusho T. (1984). Coverage measure for path testing based on the concept of essential branches, *J. Information Processing*, **6**, 199–205

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

[14] Chusho T., *et al.* (1983). A language-adaptive programming environment based on a program analyzer and a structure editor. In *Proc. IFIP '83*, pp. 621–6