Chapter 2
Code-Based Test-Case Selection Method

J. C. Huang
Department of Computer Science
University of Houston

The main topic

This chapter presents a family of test-case selection methods that can be used to do debug testing. These methods make use of the information extracted from the source code to select test cases.

The central problem

How to construct a test set T (= $\{t_1, t_2, \ldots, t_n\}$) having a maximal probability of revealing a fault?

The probability to be optimized is

$$p(\neg OK(t_1) \vee \neg OK(t_2) \ldots \vee \neg OK(t_n))$$

$$= p((\exists t)_T(\neg OK(t)))$$

$$= p(\neg(\forall t)_T(OK(t)))$$

$$= 1 - p((\forall t)_T(OK(t)))$$

The central problem (continued)

We can construct the test set incrementally by letting $T = \{t_1\}$ first.

If there is any information available to find an input that has a high probability of revealing a fault, make it t_1. Otherwise arbitrarily choose one from the input domain. (More will be said about the choice of t_1 in Ch. 4.)

The central problem (continued)

Next we choose t_2 from the input domain such that the probability

$p(\neg OK(t_1) \vee \neg OK(t_2))$

$= p(\neg (OK(t_1) \wedge OK(t_2)))$

$= p(\neg (OK(t_2) \wedge OK(t_1)))$

$= 1 - p(OK(t_2) \wedge OK(t_1))$

$= 1 - p(OK(t_2)|OK(t_1))p(OK(t_1))$

is as high as possible.

The central problem (continued)

In choosing the third element of T, the probability to be maximized is

$p(\neg OK(t_1) \vee \neg OK(t_2) \vee \neg OK(t_3))$

$= p(\neg (OK(t_1) \wedge OK(t_2) \wedge OK(t_3)))$

$= p(\neg (OK(t_3) \wedge OK(t_2) \wedge OK(t_1)))$

$= 1 - p(OK(t_3) \wedge OK(t_2) \wedge OK(t_1))$

$= 1 - p(OK(t_3) | OK(t_2) \wedge OK(t_1)) p(OK(t_2) \wedge OK(t_1))$.

The central problem (continued)

In general, to add a new element to the test set $T = \{t_1, t_2, \ldots, t_i\}$, the (i+1)th test case t_{i+1} is to be selected to maximize the probability

$$p(\neg OK(t_1) \vee \ldots \vee \neg OK(t_i) \vee \neg OK(t_{i+1}))$$

$$= p(\neg (OK(t_1) \wedge \ldots \wedge OK(t_i) \wedge OK(t_{i+1})))$$

$$= p(\neg (OK(t_{i+1}) \wedge OK(t_i) \wedge \ldots \wedge OK(t_1)))$$

$$= 1 - p(OK(t_{i+1}) \wedge OK(t_i) \wedge \ldots \wedge OK(t_1))$$

$$= 1 - p(OK(t_{i+1}) | OK(t_i) \wedge \ldots \wedge OK(t_1)) p(OK(t_i) \wedge \ldots \wedge OK(t_1)).$$

The central problem (continued)

This probability can be maximized by minimizing the conditional probability:

$$p(OK(t_{i+1})|OK(t_i)\wedge\ldots\wedge OK(t_1)),$$

i.e., by selecting t_{i+1} in such a way that $\delta(t_1, t_{i+1})$, $\delta(t_2, t_{i+1})$, … $\delta(t_i, t_{i+1})$, are all minimal.

The central problem (continued)

In practice, there are several different ways to do this, each of which led to the development of a different test-case selection method discussed in this and the following chapters.

Essence of a test-case selection method

First, a type of programming construct, such as a statement or branch predicate, is identified as the essential component of a program, each of which in the program must be exercised during the test in order to reveal potential faults.

Second, a test-case selection criterion is established to guide the construction of a test set.

Third, an analysis method is devised to identify such constructs in a program, and to select test cases from the input domain so that the resulting test set is of a reasonable size, and its elements are computationally loosely coupled.

Path testing

In path testing, the component to be exercised is execution path. The test-case selection criterion is to select test cases to test the program to the extent that every feasible execution path in the program is traversed at least once during the test.

Path testing (continued)

Path testing is interesting in that

- The set of all subdomains formed by the inputs that traverse the same execution path constitutes a partition of the input domain.
- Any two inputs in the different subdomains are computationally loosely coupled.
- Any two inputs in the same subdomain are computationally tightly coupled.

Path testing (continued)

Path testing, therefore, is an ideal method for test-case selection.

Unfortunately, it's applicability is rather limited because most real programs contain loop constructs that are likely to expand into a prohibitively large number of feasible execution paths.

Path testing (continued)

Despite of its impracticality, the path testing is discussed here first because all other test methods discussed in the following can be viewed as an approximation of the path testing. Originally, these methods were developed independently based on the propositions that a certain subset of execution paths in a program is more important in some sense, and therefore should be exercised during the test.

Path testing (continued)

In the conceptual framework used in this book, however, each of these methods can be viewed as a different way to sample execution paths to be tested. It is the resulting reduction in the number of paths to be exercised that make these methods more practical to use but less effective in fault discovery.

Statement testing

For most programs, not every statement will occur in a given execution path. Therefore, if a program contains a statement in error and that statement is not executed during the test, we will not be able to detect any abnormality at all in the test result.

Thus, a possible test criterion is to have each and every statement in the program executed at least once during the test.

The futility

It must be emphasized here, however, that the use of such a set of test cases gives us no assurance that the presence of an error will be definitely reflected in the test result.

For instance, if a statement in the program, say, `x=x+y` is somehow erroneously written as `x=x-y`, and if the test case used is such that it sets y = 0 prior to the execution of this statement, the test result certainly will not indicate the presence of this error.

Example

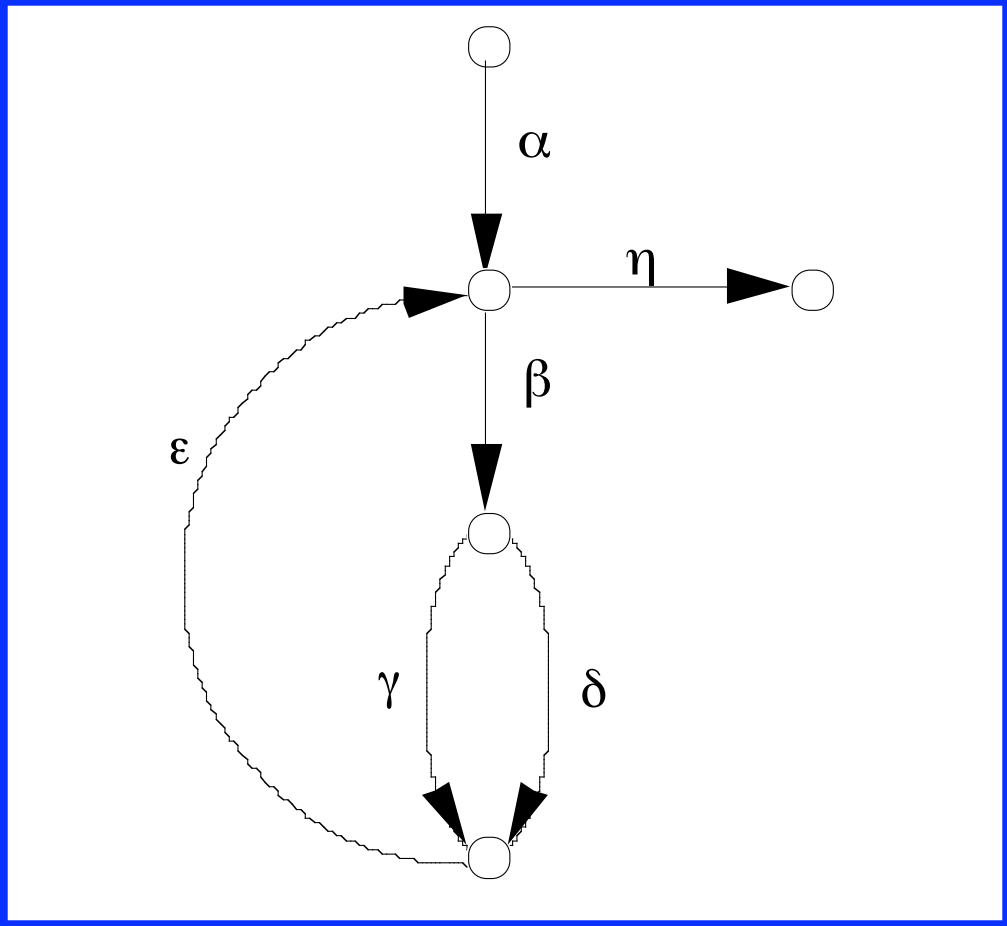
The following program is designed to find the abscissa within the interval (a, b) at which a function f(x) assumes the maximum value. The basic strategy used is that, given a continuous function that has a maximum in the interval (a, b), we can find the desired point on the x-axis by first dividing the interval into three equal parts. Then compare the values of the function at the dividing points a+w/3 and b-w/3, where w is the width of the interval being considered. If the value of the function at a+w/3 is less than that at b-w/3, then the leftmost third of the interval is eliminated for further consideration; otherwise the rightmost third is eliminated.

An example program

```
Main()
{
  float a, b, e, w, p, q, u, v;

  scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
  w = b - a;
  while (w > e) {
        p = a + w / 3;
        u = f(p);
        q = b - w / 3;
        v = f(q);
        if (u < v)
                a = p;
        else
                b = q;
        w = b - a;
  }
  max = (a + b) / 2;
  printf("5.2f\n", max);
}
```

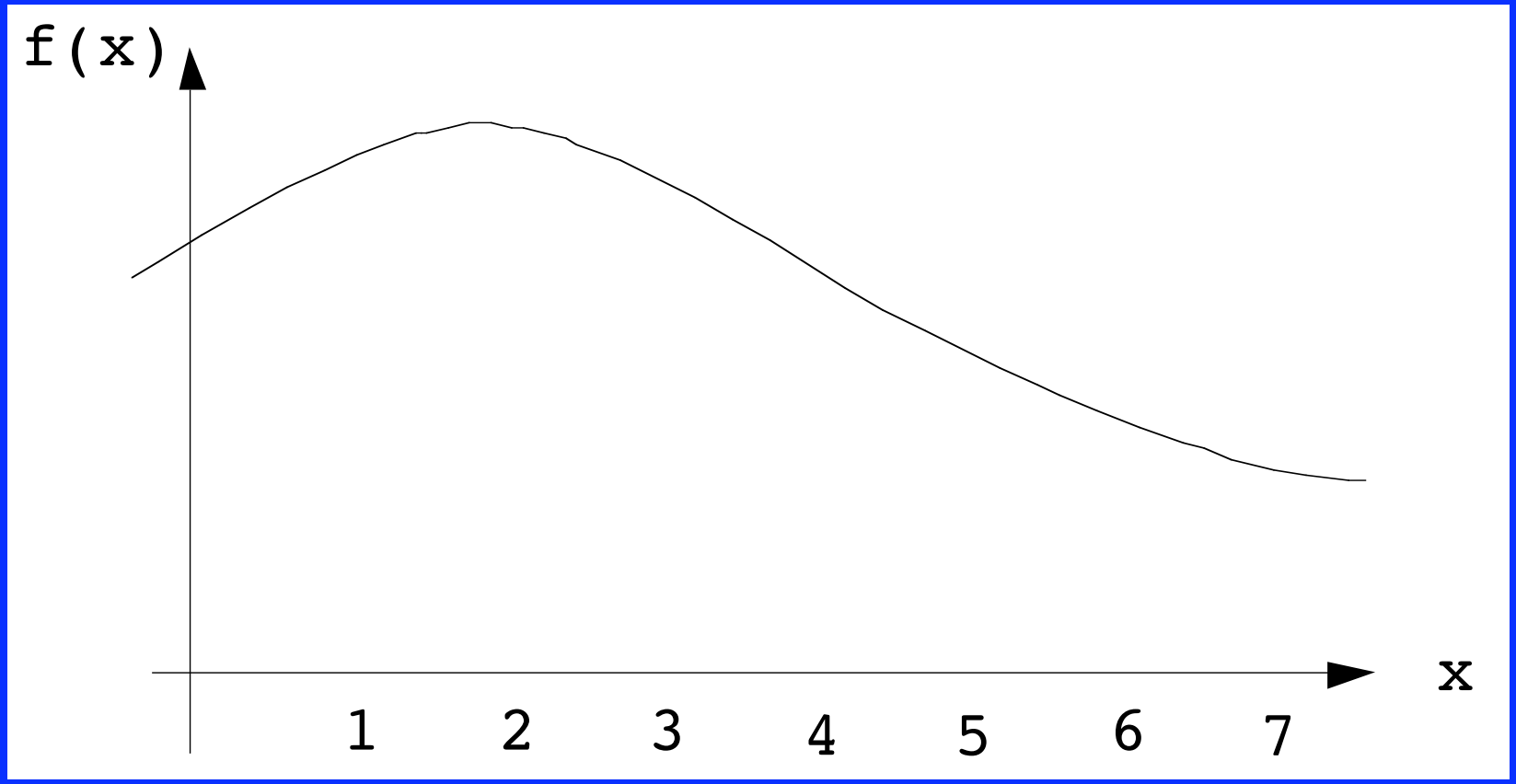
The program graph



where

```
α:   scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
     w = b - a;
β:   /\ w > e;
     p = a + w / 3;
     u = f(p);
     q = b - w / 3;
     v = f(q);
γ:   /\ !(u < v);
     b = q;
δ:   /\ u < v;
     a = p;
ε:   w = b - a;
η:   /\ !(w > e);
     max = (a + b) / 2;
     printf("5.2f\n", max);
```

The function



Example (continued)

Now suppose we wish to test this program for three different test cases, and assume that the function f(x) can be plotted as shown in Figure 2.2. Let us first arbitrarily choose `e` to be equal to 0.1, and choose the interval (a, b) to be (3, 4), (5, 6), and (7, 8). Now suppose that the values of `max` for all three cases are found to be correct in the test. What can we say about the design of this test?

Example (continued)

Observe that in all three intervals chosen the value of `u` will be always greater than `v` as we can see from the function plot. Consequently, the statement `a=p` in the program will never be executed during the test. Thus if this statement is for some reason erroneously written as, say, `a=q` or `b=p`, we will never be able to discover the error in a test using the three test cases mentioned above. This is so simply because this particular statement is not "exercised" during the test.

Example (continued)

The functional plot given in Figure 2.2 shows that u will always be less than v within the interval (0, 1). Thus if the set of test cases used includes the interval (0, 1) we will be able to discover the error described above.

Example (continued)

The point to be made here is that our chances of discovering errors through program testing can be significantly improved if we select the test cases in such a way that each and every statement will be executed at least once.

An observation

The use of such a set of test cases gives us no assurance that the presence of an error will be definitely reflected in the test result. For instance, if a statement in the program, say, `x=x+y` is somehow erroneously written as `x=x-y`, and if the test case used is such that it sets y = 0 prior to the execution of this statement, the test result certainly will not indicate the presence of this error.

A common programming error

There is a class of common programming errors that cannot be discovered in this way.

For instance, consider the type of error illustrated in Figure 2.3, where the flow of control is transferred to a wrong place as indicated by the dotted line.

A common programming error (continued)

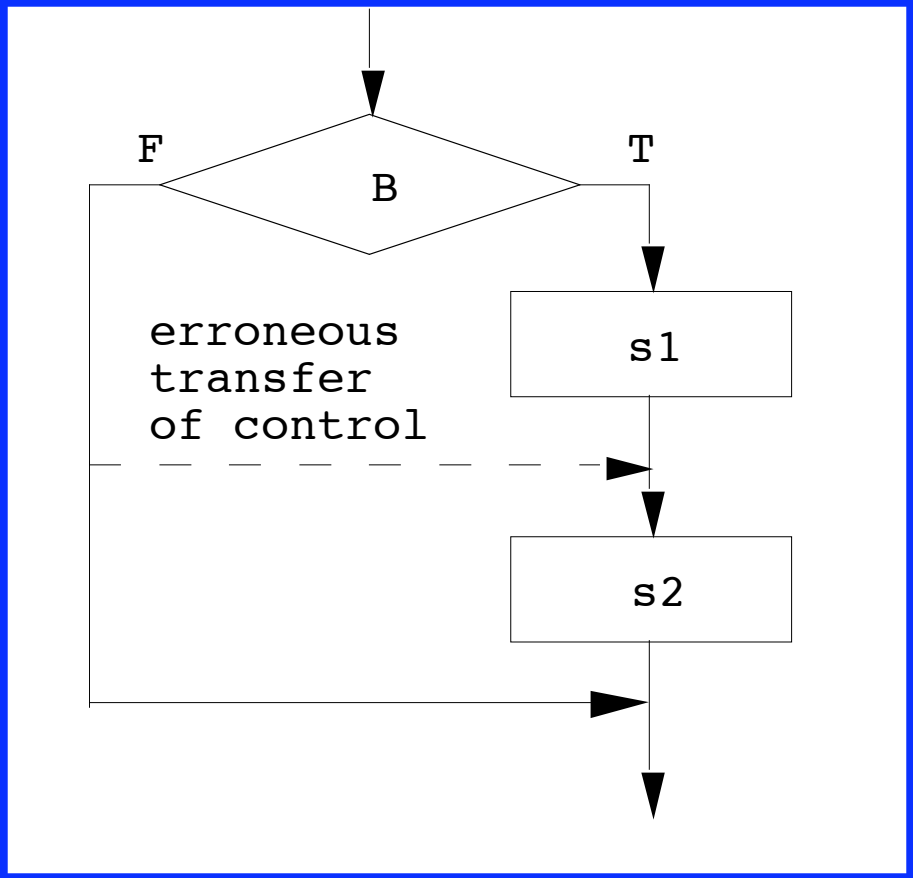


Fig. 2.3

A common programming error (continued)

This occurs when a C programmer mistakenly writes

```
if (B)
   s1;
   s2;
```

instead of

```
if (B) {
     S1;
     S2
}
```

A common programming error (continued)

In this case the program produces correct results as long as the input data cause B to be true when this program segment is entered.

The requirement of having each and every statement executed at least once is trivially satisfied in this case by choosing input data so that B is true. Obviously, the error will not be detected in this case.

Limitation of a statement test

The problem is that a program may contain paths from the entry to the exit (in its control flow) which need not be traversed in order to have each and every statement executed at least once. Since the present test requirement can be satisfied without having such paths traversed during the test, it is only natural that we will not be able to discover errors that occur on those paths.

Impracticality of path testing

An obvious solution to this problem would be to require that each and every control path in the program be traversed at least once during the test. However, this test requirement can be easily proved to be impractical because in practice almost every program contains loops, and a program with a loop contains at least as many different control paths as the number of times the loop can be iterated, which is prohibitively large in many cases.

Branch test

A more realistic solution is to require that each and every edge or branch (these two terms are used interchangeably throughout this article) in the program graph be traversed at least once during the test.

Branch test (continued)

In accordance with this new test requirement, we will have to use a new test case that makes `B` false, in addition to the one that satisfies `B`, in order to have every branch in Figure 2.3 traversed at least once. Hence our chances of discovering the error will be greatly improved, because the program will most likely produce an erroneous result for the test case that makes `B` false.

Branch test *covers* statement test

Observe that this new requirement of having each and every branch traversed at least once is more stringent than the previously stated requirement of having each and every statement executed at least once. In fact, satisfaction of the new requirement implies satisfaction of the previous one. Satisfaction of the previously stated requirement, however, does not necessarily entail satisfaction of the new one.

How to determine coverage?

A simple and practical way to determine to what extent the test coverage has been achieved is to instrument the program to be tested by using a set of software counters as explained in a later chapter. After having the program tested for a number of test cases, we can determine the coverage achieved by examining the resulting counter values. If the test requirement is to have each branch traversed at least once, and if the program is instrumented in such a way that there is one counter on each decision-to-decision path, then the requirement is satisfied when the values of all counters are non-zero.

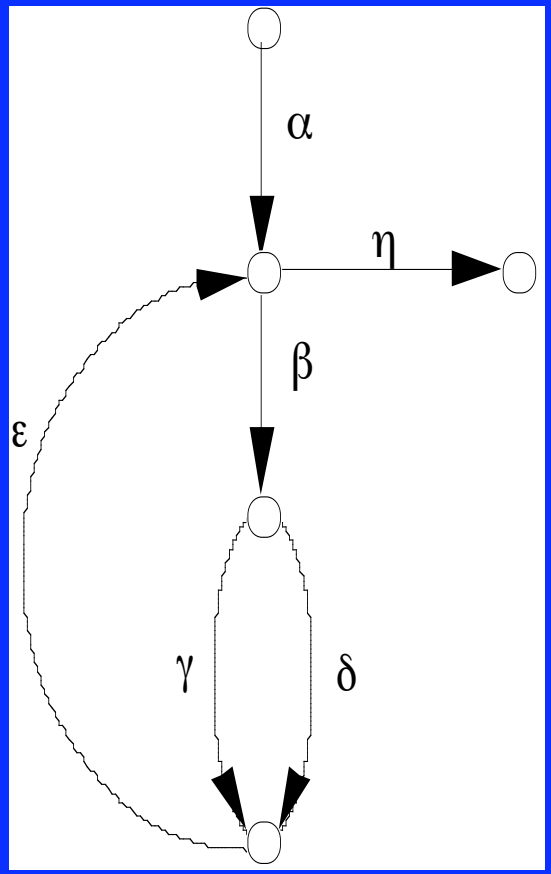
What test cases to use?

Alternatively, we can begin the test procedure by finding a (minimal) set of test cases that will test the program thoroughly. We then use this set of test cases, perhaps in conjunction with any other desirable test cases, to test the program. In this way the desired degree of thoroughness will be automatically achieved. Furthermore, by using a minimal set (i.e., a set with a minimal number of elements) of test cases, we can keep the required resources for program testing to a minimum.

How to generate test cases?

1. Find S, a minimal set of paths from the entries to the exits in the program graph such that every branch is on some path in S;
2. Find a path predicate for each path in S; and
3. Find a set of assignments to the input variables, each of which satisfies a path predicate obtained in step 2. This set is the desired set of test cases.

Example



Consider the program shown in Fig. 2.1.

Example (continued)

Consider the set S = {αβδεη, αβγεη} of two paths, traversal of which will cause every branch to be traversed at least once.

Trace subprogram αβδεη:

```
scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
w = b - a;
/\ w > e;
p = a + w / 3;
u = f(p);
q = b - w / 3;
v = f(q);
/\ u < v;
a = p;
w = b - a;
/\ !(w > e);
max = (a + b) / 2;
printf("5.2f\n", max);
```

Αβδεη becomes

```
scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
/\ b - a > e;
/\ f(a + (b - a) / 3) < f(b - (b - a) / 3);
/\ !(2 * (b - a) / 3 > e);
w = b - a;
p = a + w / 3;
u = f(p);
q = b - w / 3;
v = f(q);
a = p;
w = b - a;
max = (a + b) / 2;
printf("5.2f\n", max);
```

The path condition of path αβδεη is

```
b - a > e
&& f(a + (b - a) / 3) < f(b - (b - a) / 3)
&& !(2 * (b - a) / 3 > e)
```

which can be simplified to

```
b - a > e
&& f((b + 2a) / 3) < f((a + 2b) / 3)
&& !(2 * (b - a) / 3 > e)
```

Trace subprogram αβγεη

```
scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
w = b - a;
/\ w > e;
p = a + w / 3;
u = f(p);
q = b - w / 3;
v = f(q);
/\ !(u < v);
b = q;
w = b - a;
/\ !(w > e);
max = (a + b) / 2;
printf("5.2f\n", max);
```

αβγεη becomes

```
scanf("%5.2f %5.2f %1.4f", &a, &b, &e);
/\ b - a > e;
/\ !(f(a + (b - a) / 3) < f(b - (b - a) / 3));
/\ !(2 * (b - a) / 3 > e);
w = b - a;
p = a + w / 3;
u = f(p);
q = b - w / 3;
v = f(q);
b = q;
w = b - a;
max = (a + b) / 2;
printf("5.2f\n", max);
```

The path condition of αβγεη is

```
b - a > e
&& !(f(a + (b - a) / 3) < f(b - (b - a) / 3))
&& !(2 * (b - a) / 3 > e)
```

which can be simplified to

```
b - a > e
&& !(f((b + 2a) / 3) < f((a + 2b) / 3))
&& !(2 * (b - a) / 3 > e)
```

How to evaluate $\neg((f(x_1) < f(x_2))$?

It is observed that a < b because they stand for the lower and upper boundaries of an interval on the x-axis. Hence it is always true that (b+2a)/3 < (a+2b)/3. Now, from the function plot in Fig. 2.2 we see that $\neg((f(x_1) < f(x_2))$ will be true if $x_1 < x_2$ and x_1 is greater than or equal to 2. In other words, the second atomic expression will be true if (b + 2a)/3 = 2, or, equivalently, b + 2a = 6.

Modified path conditions

Thus, instead of the path predicates shown above, we may consider

```
b-a>e && b+2a>=6 && !(2*(b-a)/3>e)
```

as the path predicate of path αβδεη, and

```
b-a>e && !((b+2a)>=6) && !(2*(b-a)/3> e)
```

as that of path αβγεη for the purpose of finding test cases.

Test the program with
$a \leftarrow 0, \quad b \leftarrow 0.5, \quad e \leftarrow 1/3.$

If the program contains an error, say, `p=a+w/3` is somehow written as `p=a-w/3`, then the algorithm will not converge.

There is an infinite loop in the program.

Test the program with
$a \leftarrow 0, \quad b \leftarrow 0.5, \quad e \leftarrow 1/3.$

If the logical expression (associated with the second decision box in Figure 2.1) is erroneously written as `v<u` instead of `u<v`, then variable `max` will contain the abscissa at which f(x) assumes the minimum value (instead of the maximum). In other words, we will obtain as the test result max = Δ, i.e., max is within the distance Δ = e/2 = 1/6 from a = 0, which is clearly not a correct answer, as one can see from the function plot.

Test the program with
$a \leftarrow 0, \quad b \leftarrow 0.5, \quad e \leftarrow 1/3.$

If the program is correct, it should produce

$$\max = 0.5 - \Delta$$

as the result, Δ, where Δ is the error less than or equal to $e/2 = 1/6$.

Test the program with
$a \leftarrow 0, \quad b \leftarrow 0.5, \quad e \leftarrow 1/3.$

If the program is in error, say, the assignment statement `p=a+w/3` is somehow written as `p=a-w/3`, then the algorithm will not converge. Consequently, we have an infinite loop in the program and execution will not terminate.

Test the program with
$a \leftarrow 0, \quad b \leftarrow 0.5, \quad e \leftarrow 1/3.$

If the logical expression (associated with the second decision box in Figure 2.1) is erroneously written as `v<u` instead of `u<v`, then variable `max` will contain the abscissa at which f(x) assumes the minimum value (instead of the maximum). In other words, we will obtain as the test result max = Δ, i.e., max is within the distance Δ = e/2 = 1/6 from a = 0, which is clearly not a correct answer

Complicating factors

1) *Number of paths involved in a large program.*

In general, the number of paths that exist in a real world program is so large that test case selection cannot be done without automated tools.

Complicating factors (continued)

2) *Non-traversable paths in a program:*

Not every path in the control-flow diagram is a feasible execution path.

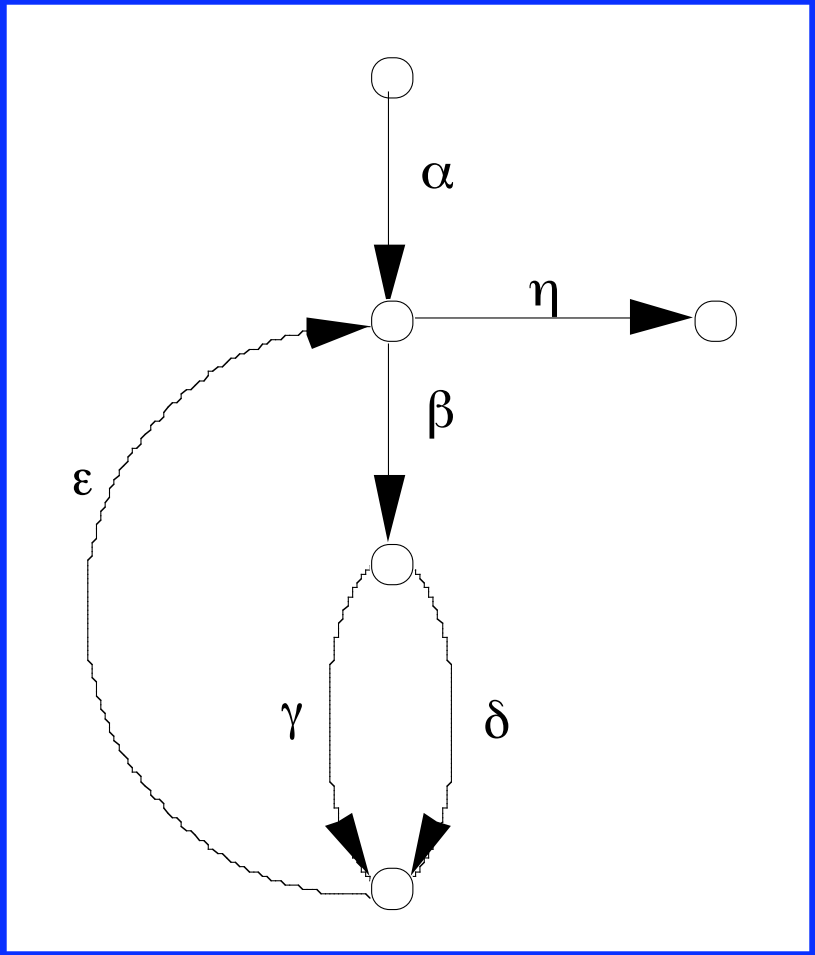
The fact that an unfeasible path cannot be identified on the basis of the graph structure of the flowchart but rather by the fact that it has an unsatisfiable path predicate, greatly complicates the problem.

Complicating factors: Example

```
main()
{
  int x, y, z;

  scanf("%d %d", &x, &y);
  z = 1;
  while (y != 0) {
            if (y % 2 == 1)
                  z = z * x;
            y = y / 2;
            x = x * x;
  }
  printf("%d\n", z);
}
```

Complicating factors: Example



Complicating factors: Example

```
α: scanf("%d %d", &x, &y);
   z = 1;
β: /\ y != 0;
γ: /\ !(y % 2 == 1);
δ: /\ y % 2 == 1;
   z = z * x;
ε: y = y / 2;
   x = x * x;
η: /\ !(y != 0);
   printf("%d\n", z);
```

Complicating factors: Example

This program computes x^y by a binary decomposition of y for integer $y \geq 0$. By inspection we see that every branch is on some path in the set given below:

$\{\alpha\beta\delta\varepsilon\eta, \alpha\beta\gamma\varepsilon\eta\}$

and thus is a candidate minimal covering set for test-case construction.

Complicating factors: Example

Thus the path predicates for these two paths are:

```
y != 0 && y % 2 == 1 && y / 2 == 0
```

and

```
y != 0 && y % 2 != 1 && y / 2 == 0,
```

the second of which is not satisfiable, and thus represent an infeasible execution path.

Complicating factors (continued)

3) *Loop structure in a program:*

It is desirable to iterate a loop as few times as possible for economical reasons.

Unfortunately, some loops have to be iterated for a constant number of times (e.g., FOR loops)

Complicating factors (continued)

4) *Subscripted variables:*

There will be more than one way to satisfy a predicate if an array element is involved. For example, consider the predicate: a[i+1] == a[j]. This predicate can be satisfied by letting i + 1 == j or by assigning the same value to a[i+1] and a[j]. Thus a degree of indeterminacy is added to the process of finding an assignment that satisfies a predicate.

Complicating factors (continued)

5) *Block structure and call of procedure or subroutine:* if the program is written in a language (such as C) that permits the use of block structures, or if it contains a function (procedure) call, then we need to be able to tell whether a given variable is local or global. Since the same identifier can be used to denote two distinct variables in the same program, we must keep track of the scopes in which the variables are defined. We also need to know whether an identifier stands for a "call by name" or a "call by value" parameter in order to construct a path predicate correctly.

Complicating factors (continued)

6) *Path predicates involving floating-point variables:* the truth values of such predicates may become unpredictable.

Significance of Path Testing

It is interesting to see what branch test means in terms of the tasks to be performed by a program. Mathematically speaking, a computer program may be considered as the definition of a function. This function usually is expressed as a union of a set of partial functions, each defined on a subset of the intended input domain.

Subprograms vs. partial functions

Each partial function is associated with an execution path in such a way that the sequence of non-control statements on the path is actually a subprogram that computes the values of that partial function.

Path predicate

The condition that a set of input data has to satisfy in order for a path to be traversed in execution is generally referred to as the *path predicate (condition)* of that path. The path predicate essentially defines the membership of a subdomain in which the corresponding partial function is defined.

Implication of branch testing

If every branch in the program is traversed at least once, it implies that most, but not necessarily all, of the possible execution paths will be traversed at least once. Therefore, to test a program by having every branch traversed at least once is to test the correctness of most partial functions for at least one point in the subdomain in which each is defined.

On path testing

If there is an error in the constituent statements of a certain path, it is most likely that we will discover the error because the corresponding partial function will be checked for at least one point in its domain.

On branch testing

We must remember, however, that some possible execution paths may not be covered in a branch test.

Furthermore, for some input data, some programs may produce results that are fortuitously correct, as we have illustrated before. This is why the requirement of having every branch traversed at least once is still not sufficient to ensure that the presence of an error will be definitely indicated in the test result.

Impracticality of path testing

Obviously, we can make a test more thorough by requiring that every possible execution path in the program be exercised at least once. But it is infeasible in practice because most programs contain loop constructs, each of which yields a prohibitively large number of executable paths.

A viewpoint

Having every branch traversed at least once can be seen as a practical way to obtain a well distributed sample of execution paths.

Other path-oriented methods

In addition to branch testing, there are at least three other methods for selecting paths to be tested, viz.,

- Boundary-interior testing
- McCabe method

Boundary-interior testing

The first, called the *boundary-interior testing*, is designed to circumvent the problem presented by a loop construct. A *boundary* test of a loop construct causes it to be entered but not iterated. An *interior* test causes a loop construct to be entered and iterated at least once.

Boundary-interior testing (continued)

To be more precise, if an execution path is expressed in a regular expression then the paths to be exercised in the boundary-interior test is described by replacing every occurrence of expression of the form α^* with $(\lambda+\alpha)$, where λ is the null path.

The examples listed in Table 2.1 should clarify this definition.

Table 2.1 Examples

paths in the program	paths to be traversed in the test
ab*c	ac + abc
a(b+c)*d	ad + abd + acd
ab*cd*e	ace + abce + acde + abcde

Boundary-interior testing (continued)

In practice, the paths prescribed by this method may be infeasible because certain types of loop construct, such as a "for" loop in C++ and other programming languages, has to iterate a fixed number of times every time it is executed.

Leave such a loop construct intact because it will not be expanded into many paths.

Boundary-interior testing (continued)

Semantically speaking, the significance of having a loop iterated zero and one time can be explained as follows. A loop construct is usually employed in the source code of a program to implement something that has to be recursively defined.

Boundary-interior testing (continued)

For example, a set D of data whose membership can be defined recursively in the form

(1) $d_0 \in D$, (initialization clause)

(2) If $d \in D$ and P(d) then f(d) is also an element of D, (inductive clause)

(3) Those and only those obtainable by a finite number of applications of (1) and (2) are the elements of D, (extremal clause)

Boundary-interior testing (continued)

where P is some predicate and f is some function. In this typical recursive definition scheme, the initialization clause is used to prescribe what is known or given, and the inductive clause is used to specify how a new element can be generated from the given ones.

Obviously, set D is correctly defined if the initialization and inductive clauses are correctly stated.

Boundary-interior testing (continued)

When D is used in a program, it will be implemented as a loop construct of the from

```
d := d0;
while P(d) do begin S; d := f(d) end;
```

where S is the program segment designed to make use of the elements of the set.

Boundary-interior testing (continued)

Obviously, a test execution without entering the loop will exercise the initialization clause, and a test execution that iterates the loop only once will exercise the inductive clause. Therefore, we may say that boundary-interior testing is an abbreviated form of path testing.

McCabe's testing method

The second method is proposed by Charles McCabe based on his complexity measure. It requires that at least a maximal set of linearly independent paths in the program be traversed during the test.

McCabe's testing method (continued)

A graph is said to be *strongly connected* if there is a path from any node in the graph to any other node.

It can be shown that, in a strongly connected graph G = <E, N>, where E is the set of edges and N is the set of nodes in G, there can be as many as v(G) elements in a set of linearly independent paths,

McCabe's testing method (continued)

where

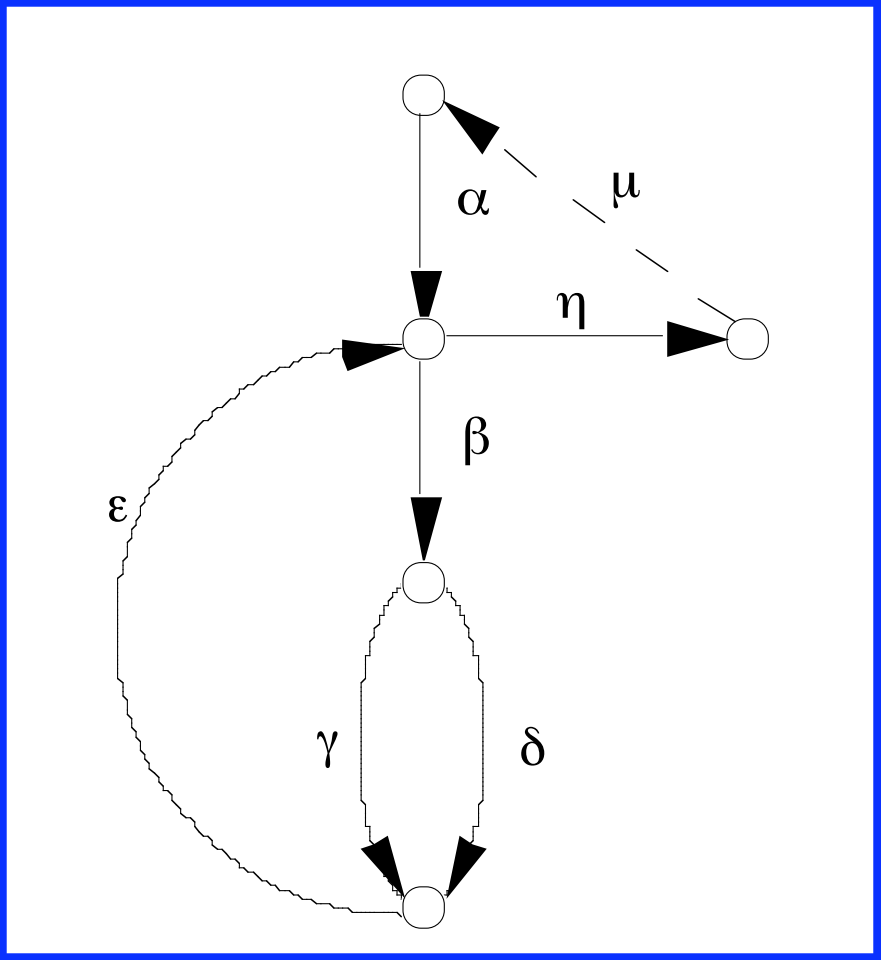
$v(G) = |E| - |N| + 1.$

The number v(G) is also known as McCabe's *cyclomatic* number, a measure of program complexity.

McCabe's testing method (continued)

Here we speak of a program (control) graph with one entry and one exit. It has the property that every node can be reached from the entry, and every node can reach the exit. In general, it is not strongly connected, but can be made so by adding an edge from the exit to the entry.

Example



For example, we can make the program graph in Fig. 2.4 strongly connected by adding the edge μ (in dashed line) as depicted on the left.

Example (continued)

Since there are 7 edges and 5 nodes in this graph, v(G) = 7 - 5 + 1 = 3 in this example. Note that, for an ordinary program graph without that added edge, the formula for computing v(G) should be:

$$v(G) = |E| - |N| + 2.$$

Example (continued)

For any path in G, we can associate it with a 1 by |N| vector, where the element on the i-th column is an integer equal to the number of times the i-th edge is used in forming the path. Thus, if we arrange the edges in the above graph in the order αβδεγμη, then the vector representation of path αβγεη is <1 1 0 1 1 0 1>, and that of βγεβγε is <0 2 0 2 2 0 0>. We shall write <αβγεη> = <1 1 0 1 1 0 1>, and <βγεβγε> = <0 2 0 2 2 0 0>.

Example (continued)

A path is said to be a *linear combination* of others if its vector representation is equal to that formed by a linear combination of their vector representations.

Thus, path βγεη is a linear combination of βγ and εη because <βγεη> = <βγ> + <εη> = <0 1 0 0 1 0 0> + <0 0 0 1 0 0 1> = <0 1 0 1 1 0 1>, and path αη is a linear combination of αβδεη and βδε because <αη> = <αβδε η> - <βδε> = <1 1 1 1 0 0 1> - <0 1 1 1 0 0 0> = <1 0 0 0 0 0 1>.

Example (continued)

A set of paths is said to be *linearly independent* if no path in the set is a linear combination of any other paths in the set.

Thus {αβδεη, αβγεη, αη} is linearly independent, but {αβδεη, αβδεβδεη, αη} is not (because <αβδεη > + <αβδεη> - <αβδεβδεη> = <αη>).

Example (continued)

A *basis set* of paths is a maximal set of linearly independent paths. In graph G given above, since v(G) = 3, {αβδεη, αβγεη, αη} constitutes a basis set of paths. Note that, although v(G) is fixed by the graph structure, the membership of a basis set is not unique. For example, {αβδεη, αβδεβγεη, αη} is also a basis set in G.

Properties of v(G)

- $v(G) \geq 1$.
- v(G) is the maximum number of linearly independent paths in G, and it is the size of the basis set.
- Inserting or deleting a node with out-degree of 1 does not affect v(G).
- G has only one path if $v(G) = 1$.
- Inserting a new edge in G increases v(G) by 1.
- v(G) depends only on the decision structure of the program represented by G.

Data-flow testing

The third approach to path sampling is based on the data flow in the program.

When a program is executed along a path, the value of each variable involved is defined first and then used later. By requiring such "define-use" relations be exercised during the test, the probability of error detection can be increased.

Data-flow oriented methods

Data flow testing is also a form of structure testing. The "component" that will be exercised during the test is a segment of control path that starts from the point where a variable is defined, and ends at the point where that definition is used. We need to introduce a few terms before we proceed to describe data-flow oriented methods for test case selection.

Properties of a data-flow path

A path is said to be *definition clear* with respect to a variable, say, x, if it begins at a point where x is defined, and contains no statement that causes x to be undefined or redefined.

A path is *loop-free* if every node on the path occurs only once.

Properties of a data-flow path (continued)

A *simple path* is a path in which at most one node occurs twice.

A *du path* of a variable, say, x, is a simple path that is definition clear with respect to x.

All-du-path testing

It requires that every du path form every definition of every variable in the program to every use of that definition be traversed at least once during the test.

Table 2.2

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ, αβγεη
	δ		δ, η	δεβδ, δεη

Table 2.3

	αη	αβδεη	αβδεβδεη	αβγεβδεη
αβδε		√	√	
αβγε				√
εβδε			√	√
εβγε				
αη	√			
εη		√	√	√
δεβδ			√	
δεη		√	√	√

Table 2.4

	αη	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδε		√	√		
αβγε				√	
εβδε			√	√	
εβγε					√
αη	√				
εη		√	√	√	
δεβδ			√		
δεη		√	√	√	

All-use testing

It requires that at least one definition-clear path from every definition of every variable to every use of that definition be traversed during the test

Table 2.5

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ, αβγεη
	δ		δ, η	δεβδ, δεη

Table 2.6

	αη	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδε		√	√		
αβγε					
εβδε			√	√	
εβγε					
αη	√				
εη		√	√	√	
δεβδ			√		
δεη		√	√	√	

All p-use/some c-use testing

It requires that at least one definition-clear path from every definition of every variable to every p-use (i.e., the definition is used in a predicate) of that definition be traversed during the test. If there is no p-use of that definition, replace "every p-use" in the above sentence with "at least one c-use (i.e., the definition is used in a computation)."

Table 2.7

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ
	δ		δ, η	δεβδ, δεη

Table 2.8

	αη	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδ		√	√		√
αβγ				√	
εβδ			√	√	√
εβγ					√
αη	√				
εη		√	√	√	√
δεη		√	√	√	√

All c-use/some p-use testing

It requires that at least one definition-clear path from every definition of every variable to every c-use of that definition be traversed during the test. If there is no c-use of that definition, replace "every c-use" in the above sentence with "at least one p-use."

Table 2.9

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ
	δ		δ, η	δεβδ, δεη

Table 2.10

	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδε	√	√		√
αβγε			√	
εβδε		√	√	√
εβγε				√
δεβδ		√		
δεη	√	√	√	√

All definitions testing

It requires that, for every definition of every variable in the program, at least one du-path emanating from that definition be traversed at least once during the test.

Table 2.11

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ
	δ		δ, η	δεβδ, δεη

Table 2.12

	αβδεη	αβδεβδεη	αβγεβδεη
αβδ	√	√	
εβδ		√	√
δεβδ		√	

All p-use testing

It derives from the all p-use/some c-use by dropping the "some c-use" requirement.

Table 2.13

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ
	δ		δ, η	δεβδ, δεη

Table 2.14

	αη	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδ		√	√		√
αβγ				√	
εβδ			√	√	√
εβγ					√
αη	√				
εη		√	√	√	√
δεη		√	√	√	√

All c-use testing

It derives from the all c-use/some p-use by dropping the "some p-use" requirement.

Table 2.15

variable	defined in	p-used in	c-used in	du-paths
x	α	δ, ε	δ, ε	αβδ, αβδε, αβγε
	ε		δ, ε	εβδ, εβγε
y	α	β, γ, δ, η	ε	αβ, αβγ, αβδ, αη, αβδε, αβγε
	ε	β, γ, δ, η	ε	εβ, εβγ, εβδ, εη, εβδε, εβγε
z	α		δ, η	αβδ
	δ		δ, η	δεβδ, δεη

Table 2.16

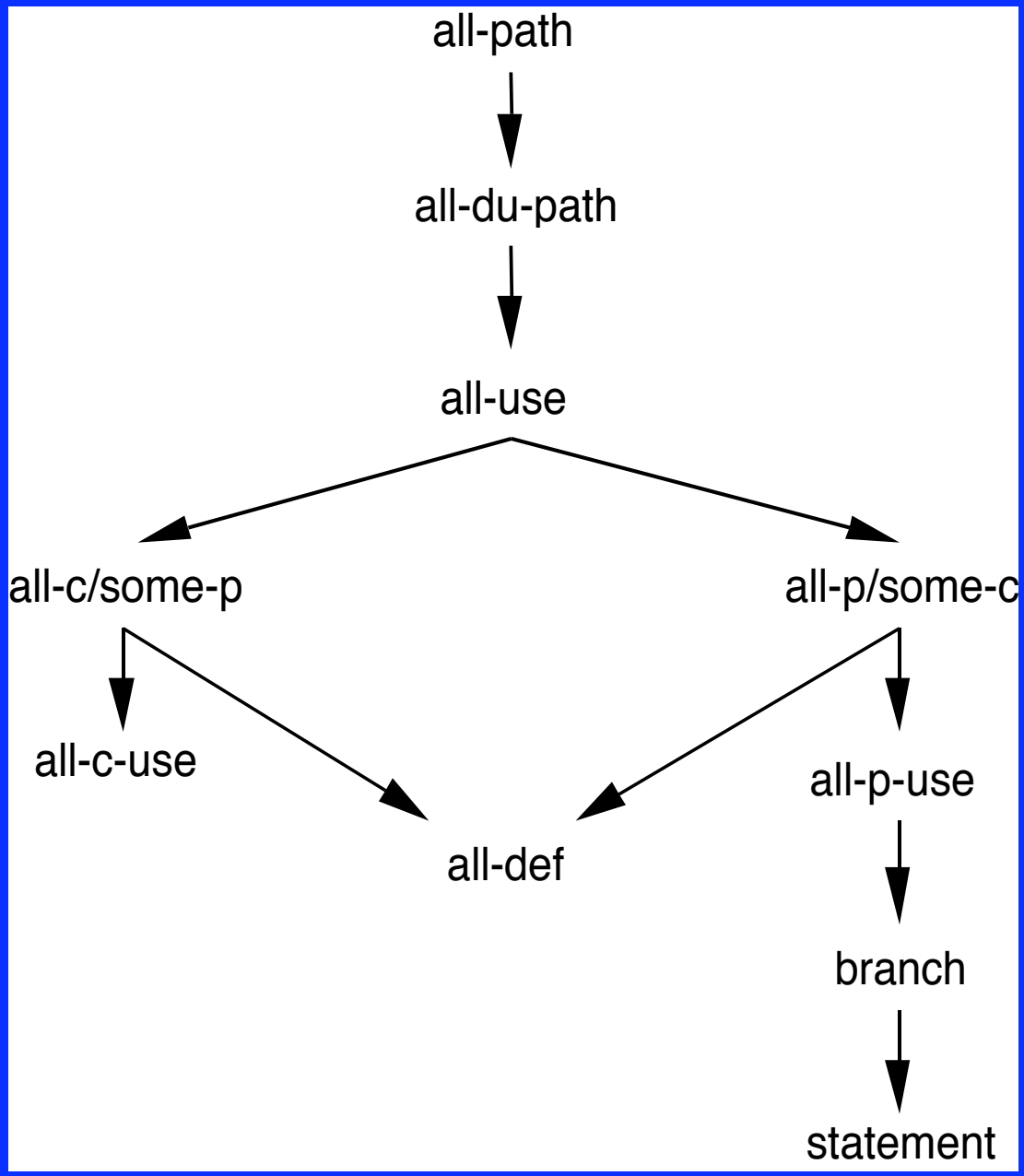
	αβδεη	αβδεβδεη	αβγεβδεη	αβδεβγεβδεη
αβδε	√	√		√
αβγε			√	
εβδε		√	√	
εβγε				√
δεβδ		√		
δεη	√	√	√	

Coverage relation

A test-case selection criterion C_1 is said to cover another criterion C_2 if satisfaction of C_1 implies satisfaction of C_2.

For example, branch test covers statement test, but not the other way around.

Coverage relation
(continued)



Error classification

An error categorization scheme is useful if it enables us to characterize a test method in terms of error type for which it is particularly effective.

A conceptual framework

In abstract, the intended function of a program can be viewed as a function f of the nature $f: X \rightarrow Y$. The definition of f usually is expressed as a set of subfunctions $f_1, f_2, ..., f_m$, where $f_i: X_i \rightarrow Y$ (i.e., f_i is f restricted to X_i for all $1 \leq i \leq m$), $X = X_1 \cup X_2 \cup ... \cup X_m$, and $f_i \neq f_j$ if $i \neq j$. We shall use f(x) to denote the value of f evaluated at $x \in X$, and suppose that each X_i can be described in the standard subset notation $X_i = \{x \mid x \in X \wedge C_i(x)\}$.

A conceptual framework (continued)

Note that, in the above, we require the specification of f to be *compact*, i.e., $f_i \neq f_j$ if $i \neq j$. This requirement makes it easier to construct the definition of a type of programming error in the following. In practice, the specification of a program may not be compact, i.e., f_i may be identical to f_j for some i and j. Such a specification, however, can be made compact by merging X_i and X_j.

A conceptual framework (continued)

Let (P, S) denote a program, where P is the condition under which the program will be executed, and S is the sequence of statements to be executed. Furthermore, let D be the set of all possible inputs to the program. Then the (valid) input domain of this program should be $X = \{ x \mid x \in D \wedge P(x)\}$, and the program should be composed of n paths, i.e.,

$$(P, S) = (P_1, S_1) + (P_2, S_2) + \ldots + (P_n, S_n),$$

such that for every $1 \leq i \leq n$, S_i is the sequence of statements designed to compute f_j for some $1 \leq j \leq m$ (note that n is not necessarily equal to m).

An error classification scheme

We shall use S(x) to denote the computation performed by an execution of S with x as input.

Two basic types of error may be committed in constructing the program (P, S):

(1) *Computational error:* the program has a computational error if

$$(\exists i)(\exists j)((P_i \supset C_j \wedge S_i(x) \neq f_j(x)).$$

(2) *Domain error:* the program has a domain error if

$$\neg(\forall i)(\exists j)(P_i \supset C_j).$$

It is different!

Note that the above definition is not identical to that given by Goodenough and Gerhart, Howden, or White and Cohen. More will be said about the differences later.

Domain strategy testing

The domain strategy testing described below is designed to detect domain errors, and is based on a geometrical analysis of the domain boundary, taking advantage of the fact that points on or near the border are most sensitive to domain errors.

Domain strategy testing (continued)

Test cases are to be selected for each border segment which, if processed correctly, determine that both the relational operator and the position of the border are correct.

Oracle

- An important assumption made in this work is that the user or an *oracle* is available who can decide unequivocally if the output is correct for the specific input processed.
- The oracle decides only if the output values are correct, and not whether they are computed correctly. If they are incorrect, the oracle does not provide any information about the error and does not give the correct output values.

Assumptions

The test method is applicable only if the program has the following properties:

(a) It contains only simple linear predicates of the form $a_1v_1 + a_2v_2 + ... + a_kv_k$ ROP C, where a_i's and C are constants, and ROP is a relational operator.

(b) The path predicate of every path in the program is composed of a conjunction of such simple linear predicates.

(c) Coincidental (fortuitous) correctness of the program will not occur for any test case.

Assumptions (continued)

(d) A missing path error is not associated with the path being tested.

(e) Each border is produced by a simple predicate.

(f) The path corresponding to each adjacent domain computes a different subfunction.

(g) Functions defined in two adjacent subdomains yield different values for the same test point near the border.

(h) Any border defined by the program is linear, and if it is incorrect, the correct border is also linear.

(i) The input space is continuous rather than discrete.

The method

Each border is a line segment in a k-dimensional space, which can be open or closed, depending on the relational operator in the predicate.

The method (continued)

A *closed* border segment of a domain is actually part of that domain and is formed by a predicate with ≥, =, or ≤ operator.

The method (continued)

An *open* border segment of a domain forms part of the domain boundary, but does not constitute part of that domain, and is formed by a $<$, $>$, or $\neq$ operator.

The method (continued)

The test points (cases) selected will be of two types defined by their relative position with respect to the given border. An *on* test point lies on the given border while an *off* test point is a small distance ε from, and lies on the open side of, the given border.

The method (continued)

When testing a closed border of a domain, the *on* test points are in the domain being tested, and each *off* test point is in some adjacent domain.

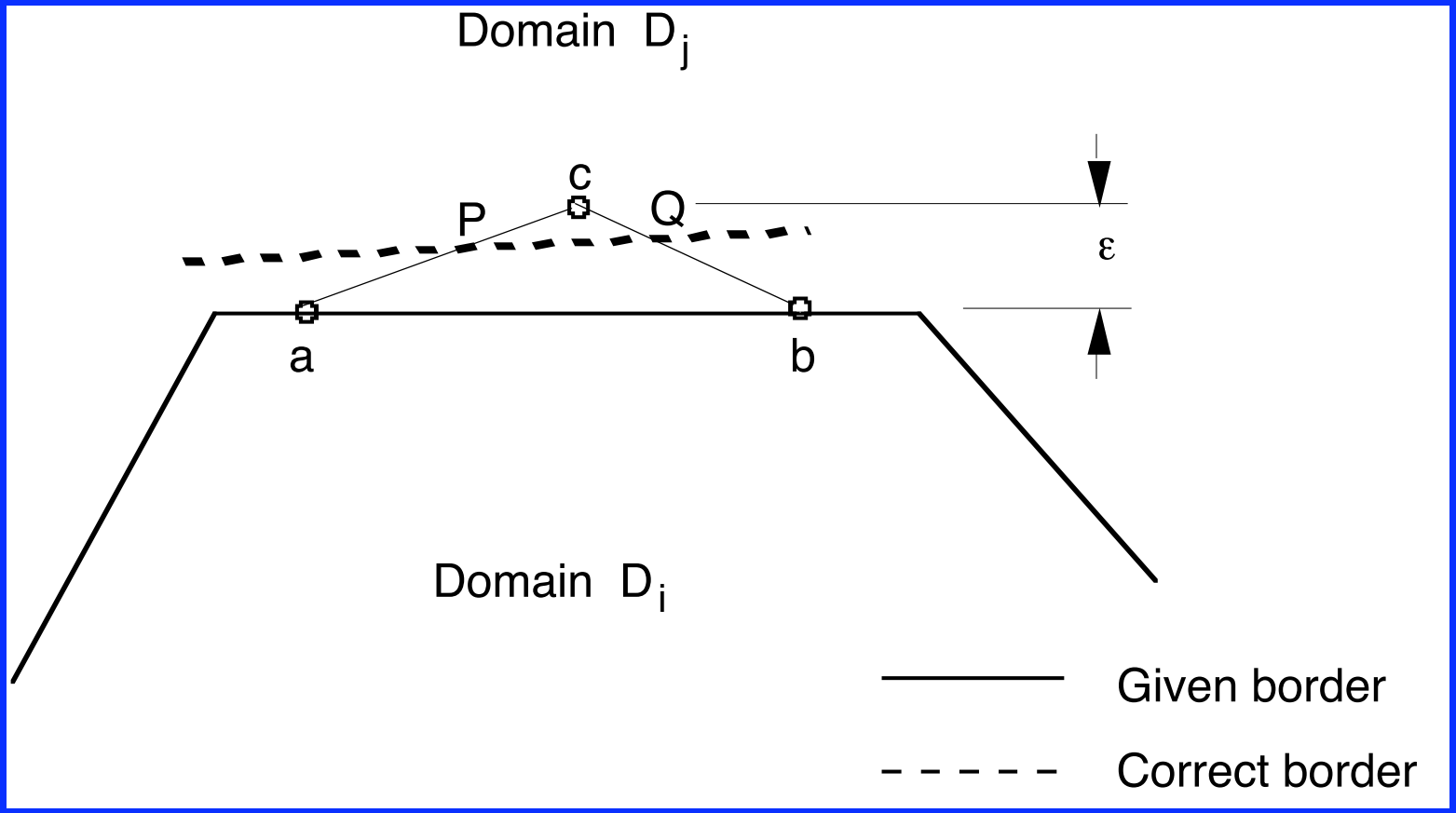
The method (continued)

When testing an open border, each *on* test point is in some adjacent domain while the *off* test points are in the domain being tested.

The method (continued)

Three test points will be selected for each border segment in an on-off-on sequence as depicted in Fig. 2.5.

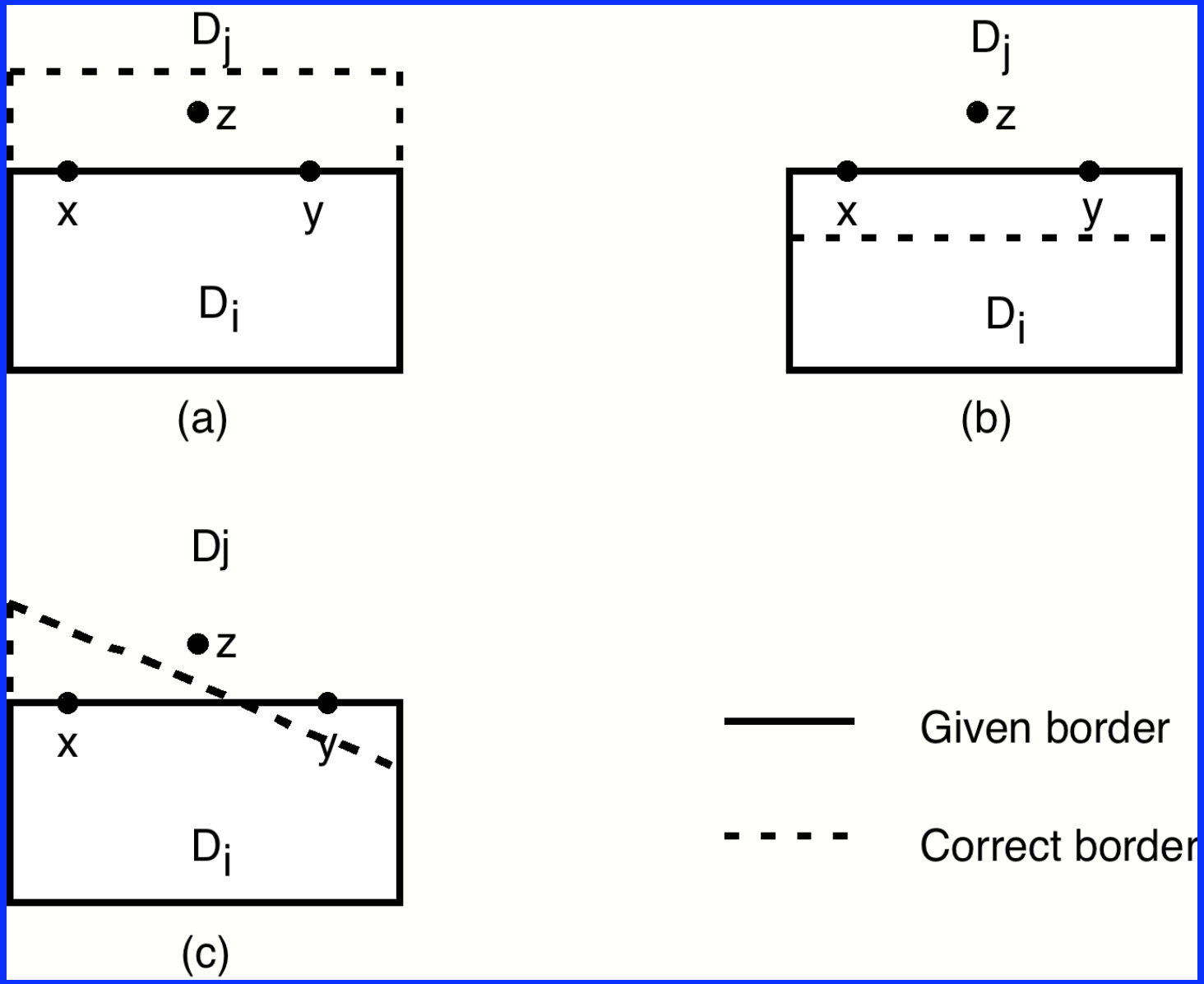
The method (continued)



The method (continued)

The test will be successful if the test points `a` and `b` are computed by the subfunction defined for domain D_i, and the test point `c` is computed by that defined for the neighboring domain D_j. This will be the case if the correct border is a line that intersects the line segments `ac` and `bc` at any point except `c`. In order to verify that the given border is identical to the correct one, we need to select the test point `c` in such a way that its distance from the given border is ε, an arbitrarily small number.

Three different types of error



Comments

Two important assumptions were made at the outset:

(1) all path predicates are numerical and linear, and

(2) functions defined in two adjacent subdomains yield different values for the same test point near the border.

The class of real-world programs satisfying assumption (1) is probably small. Assumption (2) is contrary to requirements in most practical applications: the common requirement is to have the functions defined in the adjacent subdomains to produce approximately the same, if not identical, values near the border.

CW+C error classification method

This method is a modified version of the Howden's method to be discussed later.

It is more operational in that the classification is based on the kind of changes needed to correct the error.

CW+C error classification method (contin'd)

A program can be viewed as

- establishing an exhaustive partition of the input space into mutually exclusive domains, each of which corresponds to an execution path, and
- specifying, for each domain, a set of assignment statements which compute the domain computation.

CW+C error classification method (contin'd)

Thus we have a canonical representation of a program, which is a (possibly infinite) set of pairs

$$\{(D_1, f_1), (D_2, f_2), ..., (D_i, f_i), ... \}$$

where D_i is the i-th domain and f_i is the corresponding domain computation function.

CW+C error classification method (contin'd)

Given an incorrect program P, let us consider the changes in its canonical representation as a result of modifications performed on P.

Domain-boundary modification

Definition: A *domain-boundary modification* occurs if the modification results in a change in the D_i component of some (D_i, f_i) pair in the canonical representation.

Domain-computation modification

Definition: A *domain-computation modification* occurs if the modification results in a change in the f_i component of some (D_i, f_i) pair in the canonical representation.

Missing-path modification

Definition: A *missing-path modification* occurs if the modification results in the creation of a new (D_i, f_i) pair such that D_i is a subset of D_j occurring in some (D_j, f_j) pair in the canonical representation of P.

Modification type is not uniquely defined

Notice that a particular modification (say a change of some assignment statement) can be a modification of more than one type. In particular, a missing-path modification is also a domain-boundary modification.

Error classification result is not unique

- Error occurs in a program can be classified on the basis of the modifications needed to obtain a correct program and consequent changes in the canonical representation.
- In general, there will be many correct programs and multiple ways to get a particular correct program. Hence, the error classification is not absolute, but relative to the particular correct program that would result from the series of modifications.

C+W error classification scheme

Definition: An incorrect program P can be viewed as having a *domain error (computational error) (missing-path error)* if a correct program P* can be created by a sequence of modifications, at least one of which is a *domain-boundary modification* (*domain-computation modification*) (*missing-path modification*).

Howden's error classification method

We will be concerned with programs which are either correct or can be considered deviations from a hypothetical correct program P*.

The "differences" between P and P* define errors in P. Each class of programs P will consist of correct programs P* together with incorrect programs P which differ from P* by some type of error.

Howden's classification method (contin'd)

A path through a program corresponds to some possible flow of control. A path may be *infeasible* in the sense that there is no input data which will cause the path to be executed.

In general, a program containing loops will have an infinite number of paths. The errors in a program can be categorized in terms of their effects on the paths through the program.

Howden's classification method (contin'd)

Associated with each path through a program is the subset of the input domain which causes the path to be followed and a sequence of computations which is carried out by the path.

Path domain and path computation

Definition: Suppose P_i is a path through a program P. Then the *path domain* $D_i = D(P_i)$ for P_i is the subset of the input domain which causes P_i to be executed. The *path computation* $C_i = C(P_i)$ for P_i is the function which is computed by the sequence of computations in P_i.

Equivalence of two paths

In general C_i may not be defined over all of D or, since P may contain errors, even over all of D_i.

In comparing two computations C_i and C_j, we say that C_i and C_j are equivalent ($C_i = C_j$) if C_i and C_j are defined for the same subset D' of D and $C_i(x) = C_j(x)$ for all $x \in D'$.

The effect of errors

The effect of program errors on the paths through a program can be described in terms of their effects on the path domains and path computations of the paths.

The definition of correctness

- If there is an isomorphism (one-to-one correspondence) between the paths P_i of P and the paths P_i^* of the correct version P* of P such that $D(P_i) = D(P_i^*)$ and $C(P_i) = C(P_i^*)$ for all paths, then P = P* and P is correct.
- If P is not correct, no isomorphism having these properties can be constructed. Either the domains or the computations, or both, of P and P* will be different.

(Path) computation error

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and the paths P_i* of P* such that for all pairs of paths (P_i, P_i*), $D(P_i) = D(P_i^*)$ but that for some pairs (P_k, P_k*), $C(P_k) \neq C(P_k^*)$. Then P contains a *path computation* or *computation error*.

(Path) domain error

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and the paths P_i* of P* such that for all pairs of paths (P_i, P_i*), $C(P_i) = C(P_i^*)$ but that for some pairs (P_k, P_k*), $D(P_k) \neq D(P_k^*)$. Then P contains a *path domain* or *domain error*.

Subcase error

Definition: Suppose P is an incorrect program for computing a function F and P* is a correct program. Suppose there is an isomorphism between the paths P_i of P and a subset of the paths P_i^* of P* such that $C(P_i^*) = C(P_i)$ and $D(P_i) \supset D(P_i^*)$ for all paths P_i in P. Then P contains a *subcase error*.

The assumptions

When a program contains a computation error we assume that the paths in P and P* have been indexed so that $D(P_i) = D(P_i^*)$ for all paths.

When it contains a domain or a subcase error we assume they have been indexed so that $C(P_i) = C(P_i^*)$ for all paths P_i in P.

Howden's theorem

Theorem: Suppose that P is an incorrect program and that the only difference between P and P* is in some statement which does not affect the flow of control in P. Then P has a computation error.

Another Howden's theorem

Theorem: Suppose that P is an incorrect program and that the only difference between P and a correct program P* is in some statement which affects the flow of control in P. Then P may have a computation, domain, or subcase error.

G+G error classification method

Missing Control Flow Paths. This type of error arises from failure to examine a particular condition; it results in the execution of inappropriate actions.

For example, failure to test for a zero divisor before executing a division may be a missing-path error. When a program contains this type of error, it may be possible to execute all control flow paths through the program without detecting the error. This is why exercising all program paths does not constitute a reliable test.

G+G error classification method (contin'd)

Inappropriate Path Selection. This type of error occurs when a condition is expressed incorrectly, and therefore, an action is sometimes performed (or omitted) under inappropriate conditions.

When a program contains this type of error, it is quite possible to exercise all statements and all branch conditions without detecting the error. This error can occur not merely through failure to evaluate the right combination of conditions, but also through failure to see that the method of evaluation is not adequate (e.g., determining whether three numbers are equal by writing (X+Y+Z)/3 = X).

G+G error classification method (contin'd)

Inappropriate or Missing Action. Examples are calculating a value using a method that does not necessarily give the correct result (e.g., d*d instead of d+d), or failing to assign a value to a variable, or calling a function or procedure with the wrong argument list. Some of these errors are revealed when the action is executed under any circumstances. Requiring all statements in a program to be executed will catch such errors. But sometimes the action is incorrect only under certain combinations of conditions; in this case, merely exercising the action (or the part of the program where a missing action should appear) will not necessarily reveal the error. For example, this is the case if d*d is written instead of d+d.

G+G error classification method (contin'd)

This classification of errors is useful because our goal is to detect errors by constructing appropriate tests. But insight into test reliability is given by the proposed classification. For example, consider the test-data selection criterion, "choose data to exercise all statements and branch conditions in an implementation." In evaluating the reliability of this criterion, we would ask, "Will all construction, specification, design, and requirements errors always be detected by exercising programs with data satisfying this criterion?" Clearly, if a design error, for example, is manifested as a missing path in an implementation, then this criterion for test data selection will not be reliable.

Program mutation

A mutant of a program P is defined as a program P' derived from P by making one of a set of carefully defined syntactic changes in P. Typical changes include replacing one arithmetic operator by another, one statement by another, and so forth.

Example program

```
main()   /* compute sine function */
{
   int i;
   float e, sum, term, x;

      scanf("%f %f" x, e);
      printf("x= %10.6f  e= %10.6f\n" x, e);
      term = x;
      for (i = 3; i <= 100 && term > e; i = i + 2)
      {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
      }
      printf("sin(x)= %8.6f\n" sum);
}
```

An example mutant

```
/* a mutant obtained by changing variable x to a constant 0 */
main()   /* compute sine function */
{
   int i;
   float e, sum, term, x;
      scanf("%f %f", x, e);
      printf("x= %10.6f  e= %10.6f\n", x, e);
      term = 0;
      for (i = 3; i <= 100 && term > e; i = i + 2)
      {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
      }
      printf("sin(x)= %8.6f\n", sum);
}
```

Another example mutant

```
/* a mutant obtained by changing a relational operator */
/* in i <= 100 to i >= 100                              */
main()   /* compute sine function */
{
   int i;
   float e, sum, term, x;
      scanf("%f %f", x, e);
      printf("x= %10.6f  e= %10.6f\n", x, e);
      term = x;
      for (i = 3; i >= 100 && term > e; i = i + 2)
      {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 0) sum = sum + term;
        else sum = sum - term;
      }
      printf("sin(x)= %8.6f\n", sum);
}
```

Yet another example mutant

```
/* a mutant obtained by changing constant 0 to 1 */
main()   /* compute sine function */
{
   int i;
   float e, sum, term, x;

      scanf("%f %f", x, e);
      printf("x= %10.6f  e= %10.6f\n", x, e);
      term = x;
      for (i = 3; i <= 100 && term > e; i = i + 2)
      {
        term = term * x * x / (i * (i - 1));
        if (i % 2 == 1) sum = sum + term;
        else sum = sum - term;
      }
      printf("sin(x)= %8.6f\n", sum);
}
```

Basic idea

Let P' be a mutant of some program P. A test case t is said to *differentiate* P' from P if an execution of P and P' with t produced different results.

Basic idea (contin'd)

If t failed to differentiate P' from P, either P' is functionally equivalent to P, or t is ineffective in revealing the changes (errors) introduced into P'. Thus a test method can be formulated as follows.

Mutation testing

Given a program P, which is written to implement function f,

Step 1: Generate Φ, a set of mutants of P, by using a set of mutation operations.

Step 2: Identify and delete all mutants in Φ which are equivalent to P.

Step 3: Find T, a set of test cases that as a whole differentiate P from every mutant in Φ, to test-execute P and elements of Φ.

The "competent programmer" assumption

These three steps constitute a *Φ mutant test*. The test is *successful* if P(t) = f(t) for all t∈T. A successful Φ mutant test implies that the program is free of any errors introduced into P in the process of constructing Φ. If we can assume that P was written by a competent programmer who had a good understanding of the task to be performed and was not capable of making any mistakes other than those introduced in constructing Φ, we can conclude that P is correct.

Construction of mutants

How can Φ be constructed in practice? Budd et al. suggested that a set of syntactic operations can be used to construct the desired mutants systematically. The definition of such operations obviously would be language dependent. For Fortran programs, the mutation operations may include the following.

The mutation operations

(1) *Constant Replacement:* Replacing a constant, say, C, with C+1 or C-1, e.g., statement A = 0 becomes A = 1 or A = -1.

(2) *Scalar Replacement:* Replacing one scalar variable with another, e.g., statement A = B - 1 becomes A = D - 1.

(3) *Scalar for Constant Replacement:* Replacing a constant with a scalar variable, e.g., statement A = 1 becomes A = B.

The mutation operations (contin'd)

(4) *Constant for Scalar Replacement:* Replacing a scalar variable with a constant, e.g., statement A = B becomes A = 5.

(5) *Source Constant Replacement:* Replacing a constant in the program with another constant found in the same program, e.g., statement A = 1 becomes A = 11, where the constant 11 appears in some other statement.

The mutation operations (contin'd)

(6) *Array Reference for Constant Replacement:* Replacing a constant with an array element, e.g., statement A = 2 becomes A = B(2).

(7) *Array Reference for Scalar Replacement:* Replacing a scalar variable with an array element, e.g., statement A = B + 1 becomes A = X(1) + 1.

The mutation operations (contin'd)

(8) *Comparable Array Name Replacement:* Replacing a subscripted variable with the corresponding element in another array of the same size and dimension, e.g., statement A = B(2, 4) becomes A = D(2, 4).

(9) *Constant for Array Reference Replacement:* Replacing an array element with a constant, e.g., statement A = X(1) becomes A = 5.

(10) *Scalar for Array Reference Replacement:* Replacing a subscripted variable with a nonsubscripted variable, e.g., statement A = B(1) - 1 becomes A = X - 1.

The mutation operations (contin'd)

(11) *Array Reference for Array Reference Replacement:* Replacing a subscripted variable by another, e.g., statement A = B(1) + 1 becomes A = D(4) + 1.

(12) *Unary Operator Insertion:* Insertion of one of the unary operators such as - (negation) in front of any data reference, e.g., statement A = X becomes A = -X.

(13) *Arithmetic Operator Replacement:* Replacing an arithmetic operator (i.e., +, -, *, /, **) with another, e.g., statement A = B + C becomes A = B - C.

The mutation operations (contin'd)

(14) *Relational Operator Replacement:* Replacing a relational operator (i.e., =, <>, <=, <, >=, >) with another, e.g., expression X = Y becomes X <> Y.

(15) *Logical Connector Replacement:* Replacing a logical connector (i.e., .AND., .OR., .XOR.) with another, e.g., expression A .AND. B becomes A .OR. B.

(16) *Unary Operator Removal:* Deleting any unary operator, e.g., statement A = -B/C becomes A = B/C.

The mutation operations (contin'd)

(17) *Statement Analysis:* Replacing a statement with a trap statement that causes the program execution to be aborted immediately, e.g., statement GOTO 10 becomes CALL TRAP.

(18) *Statement Deletion:* Deleting a statement from the program.

(19) *Return Statement:* Replacing a statement in a subprogram by a RETURN statement.

The mutation operations (contin'd)

(20) *Goto Statement Replacement:* Replacing the statement label of a GOTO statement by another, e.g., statement GOTO 20 becomes GOTO 30.

(21) *DO Statement End Replacement:* Replacing the end label of a DO statement with some other label, e.g., statement DO 5 I=2,10 becomes DO 40 I=2,10.

(22) *Data Statement Alteration:* Changing the values of variables assigned by a DATA statement (in FORTRAN), e.g., statement DATA Y /22/ becomes DATA Y /31/.

Application of the mutation operations

A mutant in Φ is created by applying one mutation operation to one statement in the program. The set Φ consists of all possible mutants constructed by applying every mutation operation to every applicable statement in the program.

Identification & removal of equivalent mutants

In the second step of the mutant test method, after all possible mutants are generated, one needs to identify and to remove mutants that are functionally equivalent to the program. In general, determining the equivalency of two programs is a problem unsolvable in the sense that there does not exist a single effective algorithm for this purpose. Although a mutant differs from the original program only by one statement, determination of equivalency may become problematic in practice. This difficulty remains to be a major obstacle in making program mutation as a practical method for program testing.

Significance of a discriminating test case

Observe that a mutant of program P is created by altering a statement in P. A test case would not differentiate the mutant from P unless this particular statement is involved in the test-execution. Thus, to find a test case to differentiate a mutant is to find an input to P that causes the statement in question to be "exercised" during the test. Of course, causing the statement to be exercised is only a necessary condition. For some input, a non-equivalent mutant may produce an output fortuitously identical to that of P. A sufficient condition, therefore, is that the mutant and P do not produce the same output for that input.

Example

Consider a C program that includes the following statement:

```
while (fahrenheit <= upper) {
   celsius    =    (5.0    / 9.0) *
(fahrenheit - 32.0);
   fahrenheit = fahrenheit + 10.0;
}
```

Example (contin'd)

A mutant obtained by replacing constant 5.0 with 4.0, for instance, thus includes the following statement:

```
while (fahrenheit <= upper) {
   celsius    =    (4.0    / 9.0) *
(fahrenheit - 32.0);
   fahrenheit = fahrenheit + 10.0;
}
```

Example (contin'd)

Obviously, any test case satisfying `fahrenheit > upper` will not be able to differentiate this mutant because the mutated statement in the loop body will not be executed at all. To differentiate this mutant, the test case must cause that statement to be exercised. In addition, the test case must cause the mutant to produce a different output.

Example (contin'd)

A test case that set `fahrenheit = upper = 32.0` just before the loop will not do it (because the factor `fahrenheit - 32.0` will become zero, and variable `celsius` will be set to zero regardless of the constant used there). Such a test case may satisfy the need of a statement-coverage test because it causes the statement in question to be executed, but not the need of this mutation test because it will cause the mutant to produce an output fortuitously identical to that of the original program.

An example test result

Test #	P	M1	M2	M3	M4	M5	M6	P*
1	3	3	3	3	3	5	3	3
2	7	4	7	7	7	7	7	7
3	8	8	9	8	8	8	8	8
4	11	11	13	14	11	11	11	11
5	22	22	22	22	7	22	22	22
6	5	5	5	5	5	5	9	5

This is a complete mutant test, and P is correct.

Mutant test vs. statement test

A Φ mutant test, therefore, is at least as thorough as a statement-coverage test (i.e., a test in which every statement in the program is exercised at least once). This is so because there is no program statement that can be made absolutely error-free, even if it is written by a competent programmer. That means Φ should contain at least one mutant from every statement in the program, if the Φ mutant test is to be effective. That in turn means that the set of test cases used should have every statement in the program exercised at least once, so that all mutants can be differentiated from the program.

Mutant test vs. statement test (contin'd)

A Φ mutant test may be more thorough than a statement-coverage test because if a test case failed to differentiate a non-equivalent mutant in Φ, additional test cases must be employed. These additional test cases make it possible to detect errors of the type induced by the mutation operation used.

The cost

This added thoroughness is achieved with an enormous cost, however.

A cost analysis

Suppose that a given program P has m mutants, and n test cases are used to differentiate all mutants.

In any other test method, the use of n test cases requires only n test executions,

In mutation test, additional test executions must be performed for the mutants.

A cost analysis (contin'd)

If every mutant is tested with every test case, then m×n test executions will be required for the mutants alone.

If every test result produced by a mutant that happens to be different from that produced by P is also found to be incorrect, then the number of test executions can be reduced. This is so because there is no point in testing a mutant again if it has already produced a different (and incorrect) result.

A cost analysis (contin'd)

In that case, the number of *mutant tests* (i.e., test-executions of mutant) needed depends on the number of mutants each test case is able to differentiate, and the order in which the test cases are used.

A cost analysis (contin'd)

In the best case, the first test case differentiates all but n-1 mutants with m test executions. The second test case differentiates one mutant with n-1 test executions. The third test case differentiates one mutant with n-2 executions, and so on and so forth.

A cost analysis (contin'd)

In general, the i-th test case differentiates one mutant with n-i+1 test executions (for all $1 < i = n$). Thus the total number of test executions required will be

$$m + (n - 1) + (n - 2) + \ldots + 1 = m + ((n - 1) + 1)/2 * (n - 1) = m + n(n - 1)/2$$

A cost analysis (contin'd)

In the worst case, each of the first n-1 test cases differentiates only one mutant, and the last test case differentiates the remaining m-(n-1) mutants. The total number of test executions required will be

$$m + (m - 1) + (m - 2) + \ldots + (m - (n - 1))$$
$$= mn - (1 + 2 + \ldots + (n - 1))$$
$$= mn - n(n - 1)/2$$

A cost analysis (contin'd)

These two figures represent two extreme cases. In average, the number of test executions required will be

$(m + n(n - 1)/2 + mn - n(n - 1)/2)/2 = (m + mn)/2 = m(n + 1)/2$

A cost analysis (contin'd)

Prog.	Size (no. of statements)	No. of mutants	No. of test cases needed	No. of mutant tests needed (minimum)	(maximum)	(average)
1	30	900	4	906	3,594	2,250
2	31	773	7	794	5,390	3,092
3	16	383	7	404	2,660	1,532
4	62	5,033	34	5,529	170,626	88,078
5	28	3,348	13	3,426	43,446	23,436
6	57	8,026	17	8,162	136,306	72,234
7	43	1,028	40	1,808	40,340	21,074
8	55	6,317	5	6,327	31,575	18,951
9	34	945	9	981	8,469	4,725
10	19	567	12	633	6,738	3,686

Comment on the cost

Note that in other test methods, the number of test-executions is equal to the number of test cases needed to complete the test. In the mutant test, additional executions of mutants have to be carried out with the same test cases. The last three columns in the above table indicate the minimum, maximum, and average number of test-executions required.

Comment on the cost

Take program 8 in the above table as example. Only 5 test cases (and hence test-executions) are required to complete a statement-coverage test. For a mutant test, somewhere between 6,327 and 31,575 additional test-executions are required. Assuming that each test-execution can be completed in 10 seconds (including the time needed to analyze the test result), these additional test-executions will consume somewhere between 17.5 and 87.7 hours of time. It remains to be shown that a mutant test is cost effective in comparison with other test methods.