Chapter 1
Preliminaries

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

The central problem

Given a computer program, how can we determine whether or not it will do exactly what it is intended to do?

Ideal solution

An ideal solution to this problem would be to develop certain techniques that can be used to construct the formal proof (or disproof) of the correctness of a program systematically.

The reality

There have been considerable efforts to develop such techniques, and many different techniques for proving program correctness have been reported as the results. However, none of them has been developed to the point where it can be used in practice at a reasonable cost.

A practical solution: testing

At present, a practical and more intuitive solution would be to test-execute the program with a number of test cases (input data) to see if it will do what it is intended to do.

Definition

By *program testing* here we mean a process in which we attempt to determine if a program will do what it is intended to do by actually executing it with a number of inputs.

The goals of program testing

We may want to test-execute a program in order to

- discover errors (debug testing), or
- assess the reliability (operational testing).

Exhaustive testing

Perhaps the most intuitive (and seemingly plausible) alternative is to consider the program as a black box and test it for all possible input cases to see if it will produce the correct outputs. This test strategy is commonly referred to as *exhaustive testing*.

A successful exhaustive test constitutes a direct proof of the correctness.

Exhaustive testing (continued)

Unfortunately, it is impractical to do exhaustive testing on any real program because the number of test cases involved is prohibitively large.

Exhaustive testing (continued)

For example, consider a program that takes three (short) integers as input, such as Program 1.0.1 in the text. The number of possible inputs is

$$2^{16} \times 2^{16} \times 2^{16} = 2^{48} \approx 256 \times 10^{12}$$

If we can test-execute the program at the rate of one microsecond per test-execution in average, it would take about 8 years, 24/7, to complete!

Exhaustive test (continued)

This example clearly indicates that exhaustive testing is impractical. In general, we will never have enough time and resources to do that.

In practice, we have to settle for testing the program with a manageably small subset of the input domain.

The central problem

Test-case selection is the central problem in program testing.

How do we select test cases? It depends, among other things, the reason why we wish to test the program.

Debug vs. operational testing

Debug testing is performed by a software developer to improve reliability of a product by detecting and removing latent faults in the program.

Operational testing is performed by an expert user to assess the reliability of a program when an accurate assessment is needed to decide whether to pay the vendor, or to use the program for production run.

The problem of test-case selection

For debug testing, we need to find a set of test cases that, in the aggregate, has a high probability of revealing at least one latent fault in the program.

For operational testing, we need to find a set of test cases having the highest probabilities of being used in production runs. (More about this in Chapter 4)

Importance of the problem

The problem of test-case selection is central to the study of debug testing. The essence is to maximize the fault-discovery probability with a minimal set of test cases.

The way the test cases are selected affects the cost and effectiveness of a test to a great extent.

Some terminologies

The *input domain* of a program is the set of all possible inputs to the program, each of which is sufficient to cause the program to be executed completely.

For example, the input domain of Program 1.1 is

$$D = \{\langle x, y, z\rangle \mid x, y, \text{and } z \text{ are 16-bit integers}\}$$

Program 1.1

```
main ()
{
int i, j, k, match;
cin >> i >> j >> k;
cout << i << j << k;
if (i <= 0 || j <= 0 || k <= 0
|| i+j <= k || j+k <= i || k+i <= j)
match = 4;
else if !(i == j || j == k || k == i)
match = 3;
else if (i != j || j != k || k != i)
match = 2;
else match = 1;
cout << match << endl;
}
```

Some terminologies (continued)

An *input* (data) is an element of the input domain.

A *test case* is an input used to perform a test execution.

The set of all test cases used in testing is called a *test set*.

OK(P, d)

Let D be the input domain of a given program P, let d be an element of D, and let OK(P, d) be a predicate that becomes TRUE if an execution of program P with input d terminates and produces a correct result, and FALSE otherwise. Predicate OK(P, d) can be shortened to OK(d) if the omission of P would not lead to any confusion.

Oracle

An *oracle* is some contraption that can be used to determine if the test result is correct.

Example:

The target-practice equipment used in testing the software that controls a computerized gun sight is a good example of an oracle. When the gun is fired at the target, a "hit" indicates the test is successful, and a "miss" indicates otherwise.

Successful test

A test using test set T is said to be *successful* if the program terminates and produces a correct result for *every* test cases in T. Formally, a successful test, denoted by predicate SUCCESSFUL(T), is defined as

$$\text{SUCCESSFUL}(T) \equiv (\forall t)_T(\text{OK}(t))$$

The test fails if *at least one* input causes the program to produce incorrect result, i.e.,

$$\neg\,\text{SUCCESSFUL}(T) \equiv (\exists t)_T(\neg\,\text{OK}(t))$$

Maximizing the probability

Suppose we wish to construct $T = \{t_1, t_2, \ldots, t_n\}$, a test set of n elements. The probability that at least one fault will be revealed is

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

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

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

This probability is what we wish to optimize in the process of test-case selection.

Maximizing the probability (continued)

To see how we may optimize, suppose we somehow choose test case t_1 to test the program first, and then select another test case t_2 to test the program further. What relationship must hold between t_1 and t_2 so that the joint fault discovery probability is arguably enhanced?

Maximizing the probability (continued)

Note that

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

Obviously we can increase this probability by choosing some t_2 with small $p(OK(t_2)|OK(t_1))$.

Maximizing the probability (continued)

We define

$$\delta(t_1, t_2) = p(OK(t_2)|OK(t_1)) - p(OK(t_2))$$

as the *computational coupling coefficient* between the two inputs t_1 and t_2.

Maximizing the probability (continued)

It can be shown that

$$p(\neg OK(t_1) \vee \neg OK(t_2))$$
$$= 1 - \delta(t_1, t_2)p(OK(t_1)) - p(OK(t_2))p(OK(t_1))$$

The tester cannot change $p(OK(t_1))$ or $p(OK(t_2))$ because these are intrinsic to the program, but can choose t_2 to change the value of $\delta(t_1, t_2)$: the smaller this coefficient is, the greater the value of the fault-discovery probability $p(\neg OK(t_1) \vee \neg OK(t_2))$.

Maximizing the probability (continued)

$\delta(t_1, t_2) = 0$ if $p(OK(t_2)|OK(t_1)) = p(OK(t_2))$, i.e., if $OK(t_1)$ and $OK(t_2)$ are two totally independent events.

The events $OK(t_1)$ and $OK(t_2)$ probably would never be totally independent because they may reflect the competence of the same (team of) programmer(s).

Maximizing the probability (continued)

Nevertheless, we can make $\delta(t_1, t_2)$ as small as possible by choosing t_2 in such a way that the sequences of programming components invoked by t_1 and t_2 be as different as possible.

In words, the choice of t_2 should be such that it is computationally as loosely coupled to t_1 as possible.

Maximizing the probability (continued)

It is obvious that two test cases are computationally loosely coupled if they cause the program to traverse different paths, or to execute different sequences of components.

This fact provides us a simple general rules for selecting test cases.

Definition of a component

How do we define the “component” just mentioned?

It could be a program statement, an execution path, or a subsystem, depending on the size of the program being tested, or the level of testing being performed.

It is chosen to determine the granularity of the programming constructs being scrutinized.

An important observation

- Not every program component is involved in a test-execution.
- If there is a fault in a component, and that component is not involved in a test-execution, then that fault will not be reflected in the test result.
- Thus a test set will have a higher probability of revealing a fault if it will cause every component in the program to be exercised at least once.

First principle of test-case selection

In choosing a new element for the test set being constructed, the preference should be given to those candidates that are computationally as loosely coupled as possible to the existing elements in the set.

Second principle of test-case selection

A test set should include enough elements to cause every component in the program to be exercised at least once during the test.

How to choose initial elements?

The users and designers of the program often have test cases to offer. Accept them as the initial elements. Otherwise, choose as an input

- for which the program is likely to fail,
- for which some complex computation is required,
- that is most likely to be used in production run,
- for which the corresponding output is known, or
- any element in the input domain.

A unified conceptual framework

Most existing methods for debug testing can be seen as an instantiation of this general method:

1. Select an initial element, or set of initial elements, as just explained.
2. Use the first principle of test-case selection to add elements to the set repeatedly.
3. Use the second principle to determine when to stop the process.

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 $i \neq j$.

A conceptual framework (continued)

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

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

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

Where S(x) denotes the output produced by executing S with input x.

Other published classifications

Previously published methods include that of Goodenough and Gerhart [GOGE77], Howden [HOWD76], and White and Cohen [WHCO80]. All three include one more type of fault called *subcase* or *missing-path fault*, which occurs when the programmer failed to create a subdomain required by the specification, i.e., when $\neg(\forall i)(\exists j)(C_i \subset P_j)$. Since such a fault also manifests as a computational fault, it will be omitted in this work for simplicity.

Tasks involved in testing

1. test-case selection,
2. test execution,
3. test-result analysis, and, if it is debug testing,
4. fault removal and regression testing.

Test-case selection

1. Identify all input variables.
2. Identify all components to be exercised.
3. Find a set of inputs that will cause all components to be exercised during the test.

The cost is proportional to the number of test cases needed, and complexity of the analysis required to find them.

Test execution

Test execute the program to produce test results.

The cost is proportional to the execution time and the number of test cases used.

Test-result analysis

Record and analyze the test results.

The cost is proportional to the number of test cases used, and the amount of effort needed to find the correct output for each and every test case.