Chapter 16

Test Adequacy Assessment using Program Mutation

The purpose of this chapter is to introduce Program Mutation as a technique for the assessment of test adequacy and the enhancement of test sets. The chapter also focuses on the design and use of mutation operators for procedural and objected-oriented programming languages.

16.1. Introduction

Program mutation is a powerful technique for the assessment of the goodness of tests. It provides a set of strong criteria for test assessment and enhancement. If your tests are adequate with respect to some other adequacy criterion, such as the MC/DC coverage criterion, then chances are that these are not adequate with respect to most criteria offered by program mutation.

Program mutation is a technique to assess and enhance your tests. Hereafter we will refer to "program mutation" as "mutation." When testers use mutation to assess the adequacy of their tests, and possibly enhance them, we say that they are using *mutation testing*. Sometimes the act of assessing test adequacy using mutation is also referred to as *mutation analysis*.

Given that mutation requires access to all or portions of the source code, it is considered a white-box, or code-based, technique. Some refer to mutation testing as fault injection testing. However, it must be noted that fault-injection testing is a separate area in its own right and must be distinguished from mutation testing as a technique for test adequacy and enhancement.

Mutation has also been used as a black-box technique. In this case it is used to mutate specifications and, in the case of web applications, messages that flow between a client and a server. This chapter focuses on the mutation of computer programs written in high level languages such as Fortran, C, and Java.

While mutation of computer programs requires access to the source code of the applica-

tion under test, some variants can do without. Interface mutation requires access only to the interface of the application under test. Run-time fault injection, a technique similar to mutation, requires only the binary version of the application under test.

Mutation can be used to assess and enhance tests for program units, such as C functions and Java classes. It can also be used to assess and enhance tests for an integrated set of components. Thus, as explained in the remainder of this chapter, mutation is a powerful technique for use during unit, integration, and system testing.

A cautionary note: Mutation is a significantly different way of assessing test adequacy than what we have discussed in the previous chapters. Thus, while reading this chapter, you will likely have questions of the kind "Why this. . . " or "Why that. . . " With patience, you will find that most, if not all, of your questions are answered in this chapter.

16.2. Mutation and mutants

Mutation is the act of changing a program, albeit only slightly. If P denotes the original program under test and M a program obtained by slightly changing P, then M is known as a *mutant* of P and P the *parent* of M. Given that P is syntactically correct, and hence compiles, M must be syntactically correct. M might or might not exhibit the behavior of P from which it is derived.

The term "to mutate" refers to the act of mutation. To "mutate" a program means to change it. Of course, for the purpose of test assessment, we mutate by introducing only "slight" changes.

EXAMPLE 16.1. Consider the following simple program.

Program P16.1

```
1   begin
2     int x, y;
3     input (x, y);
4     if(x<y)
5       output(x+y);
6     else
7       output(x*y);
8   end
```

A large variety of changes can be made to P16.1 such that the resultant program is syntactically correct. Below we list two mutants of P16.1. Mutant M1 is obtained by replacing the $<$ operator by the $\leq$ operator. Mutant M2 is obtained by replacing the $*$ operator by the $/$ operator.

Mutant M1 of Program P16.1

```
1   begin
2     int x, y;
3     input (x, y);
4     if(x≤y)          ← Mutated statement
```

```
5     then
6       output(x+y);
7     else
8       output(x*y);
9   end
```

Mutant M2 of Program P16.1

```
1   begin
2     int x, y;
3     input (x, y);
4     if(x<y)
5     then
6       output(x+y);
7     else
8       output(x/y);        ← Mutated statement
9   end
```

Notice that the changes made in the original program are simple. For example, we have not added any chunk of code to the original program to generate a mutant. Also, only one change has been made to the parent to obtain its mutant. ■

16.2.1. First-order and higher-order mutants

Mutants generated by introducing only a single change to a program under test are also known as *first-order* mutants. Second-order mutants are created by making two simple changes, third order by making three simple changes, and so on. One can generate a second-order mutant by creating a first order mutant of another first-order mutant. Similarly, an n^{th} order mutant can be created by creating a first order mutant of an $(n-1)^{th}$ order mutant.

EXAMPLE 16.2. Once again let us consider program P16.1. We can obtain a second-order mutant of this program in a variety of ways. Here is a second-order mutant obtained by replacing variable y in the `if` statement by the expression $y+1$, and replacing operator $+$ in the expression $x+y$ by the operator $/$. ■

Mutant M3 of Program P16.1

```
1   begin
2     int x, y;
3     input (x, y);
4     if(x<y+1)          ← Mutated statement
5     then
6       output(x/y);      ← Mutated statement
7     else
8       output(x*y);
9   end
```

Mutants, other than first-order, are also known as *higher-order* mutants. First-order mutants are the ones generally used in practice. There are two reasons why first-order mutants are

preferred to higher order mutants. One reason is that there are many more higher order mutants of a program than there are first-order mutants. For example, 528,906 second order mutants are generated for program FIND that contains only 28 lines of Fortran code. Such a large number of mutants create a scalability problem during adequacy assessment. Another reason has to do with the coupling effect and is explained in Section 16.6.2.

Note that so far we have used the term "simple change" without explaining what we mean by "simple" and what is a "complex" change. An answer to this question appears in the following sections.

16.2.2. Syntax and semantics of mutants

In the examples presented so far, we mutated a program by making simple syntactic changes. Can we mutate using semantic changes ? Yes, we certainly can. However, note that syntax is the carrier of semantics in computer programs. Thus, given a program P written in a well defined programming language, a "semantic change" in P is made by making one or more syntactic changes. A few illustrative examples follow.

EXAMPLE 16.3. Let $f_{P16.1}(x, y)$ denote the function computed by Program P16.1. We can write $f(x, y)$ as follows.

$$f_{P16.1}(x,y) = \begin{cases} x + y & \text{if } x < y \\ x * y & \text{otherwise} \end{cases}$$

Let $f_{M1}(x, y)$ and $f_{M2}(x, y)$ denote the functions computed by M1 and M2, respectively. We can write $f_{M1}(x, y)$ and $f_{M2}(x, y)$ as follows

$$f_{M1}(x,y) = \begin{cases} x + y & \text{if } x \leq y \\ x * y & \text{otherwise} \end{cases}$$

$$f_{M2}(x,y) = \begin{cases} x + y & \text{if } x < y \\ x / y & \text{otherwise} \end{cases}$$

Notice that the three functions $f_{P16.1}(x, y)$, $f_{M1}(x, y)$, and $f_{M2}(x, y)$ are different. Thus we have changed the semantics of P16.1 by changing its syntax. ■

The previous example illustrates the meaning of "syntax is the carrier of semantics." Mutation might, at first thought, seem to be "just" a simple syntactic change made to a program. In effect, such a simple syntactic change could have a drastic effect, or it might have no effect at all, on program semantics. The next two examples illustrate why.

EXAMPLE 16.4. Nuclear reactors are increasingly relying on the use of software for control. Despite the intensive use of safety mechanisms, such as the use of 400,000 liters of cool heavy water moderator, the control software must continually monitor various reactor parameters and respond appropriately to conditions that could lead to a meltdown. For example, the Darlington Nuclear Generating Station located near Toronto, Canada, uses two independent computer

based shutdown systems. Though formal methods can be, and have been, used to convince the regulatory organizations that the software is reliable, a thorough testing of such systems is inevitable.

While the decision logic in a software-based emergency shutdown system would be quite complex, the highly simplified procedure below indicates that simple changes to a control program might lead to disastrous consequences. Assume that the $checkTemp$ procedure is invoked by the reactor monitoring system with three most recent sensory readings of the reactor temperature. The procedure returns a danger signal to the caller through variable `danger`.

Program P16.2

```
1   enum dangerLevel {none, moderate, high, veryHigh};
2   procedure checkTemp (currentTemp, maxTemp){
3     float currentTemp[3], maxTemp; int highCount=0;
4     enum dangerLevel danger;
5     danger=none;
6     if (currentTemp[0]>maxTemp)
7       highCount=1;
8     if (currentTemp[1]>maxTemp)
9       highCount=highCount+1;
10    if (currentTemp[2])>maxTemp)
11      highCount=highCount+1;
12    if (highCount==1) danger=moderate;
13    if (highCount==2) danger=high;
14    if (highCount==3) danger=veryHigh;
15    return(danger);
16  }
```

Procedure `checkTemp` compares each of the three temperature readings against the maximum allowed. A "none" signal is returned if none of the three readings is above the maximum allowed. Otherwise, a "moderate," "high," or "veryHigh" signal is returned depending on, respectively, whether one, two, or three readings are above the allowable maximum. Now consider the following mutant of P16.2 obtained by replacing the constant "veryHigh" with another constant "none" at line 14.

Mutant M1 of Program P16.2

```
1   enum dangerLevel {none, moderate, high, veryHigh};
2   procedure checkTemp (currentTemp, maxTemp){
3     float currentTemp[3], maxTemp; int highCount=0;
4     enum dangerLevel danger;
5     danger=none;
6     if (currentTemp[0]>maxTemp)
7       highCount=1;
8     if (currentTemp[1]>maxTemp)
9       highCount=highCount+1;
10    if (currentTemp[2])>maxTemp)
```