Chapter 16

Fault-Based Testing

A model of potential program faults is a valuable source of information for evaluating and designing test suites. Some fault knowledge is commonly used in functional and structural testing, for example when identifying singleton and error values for parameter characteristics in category partition testing, or when populating catalogs with erroneous values, but a fault model can also be used more directly. Fault-based testing uses a fault model directly to hypothesize potential faults in a program under test, and to create or evaluate test suites based on its efficacy in detecting those hypothetical faults.

Required Background

- Chapter 9:
  The introduction to test case selection and adequacy sets the context for this chapter. It is not strictly required, but is helpful in understanding how the techniques described in this chapter should be applied.

- Chapter 12:
  Some basic knowledge of structural testing criteria is required to understand the comparison of fault-based with structural testing criteria.

16.1 Overview

Engineers study failures to understand how to prevent similar failures in the future. For example, failure of the Tacoma Narrows Bridge led to new understanding of oscillation in high wind, and the introduction of analyses to predict and prevent such destructive oscillation in subsequent bridge design. The causes of an airline crash are likewise extensively studied, and when traced to a structural failure they frequently result in a directive to apply diagnostic tests to all aircraft considered potentially vulnerable to similar failures.

Experience with common software faults sometimes leads to improvements in design methods and programming languages. For example, the main purpose of automatic memory management in Java is not to spare the programmer the trouble of releasing unused memory, but to prevent the programmer from making the kind of memory management errors (dangling pointers, redundant deallocations, and memory leaks) that frequently occur in C and C++ programs. Automatic array bounds checking cannot prevent a programmer from using an index expression outside array bounds, but can make it much less likely that the fault escapes detection in testing, as well as limiting the damage incurred if it does lead to operational failure (eliminating, in particular, the buffer overflow attack as a means of subverting privileged programs). Type checking reliably detects many other faults during program translation.

Of course, not all programmer errors fall into classes that can be prevented or statically detected using better programming languages. Some faults must be detected through testing, and there too we can use knowledge about common faults to be more effective.

The basic concept of fault-based testing is to select test cases that would distinguish the program under test from alternative programs that contain hypothetical faults. This is usually approached by modifying the program under test to actually produce the hypothetical faulty programs. Fault seeding can be used to evaluate the thoroughness of a test suite (that is, as an element of a test adequacy criterion), or for selecting test cases to augment a test suite, or to estimate the number of faults in a program.

16.2   Assumptions in Fault-Based Testing

The effectiveness of fault-based testing depends on the quality of the fault model, and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program.

Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure detection of all such faults. This is known as the *competent programmer hypothesis*, an assumption that the program under test is "close to" (in the sense of textual difference) a correct program.

Some program faults are indeed simple typographical errors, and others that involve deeper errors of logic may nonetheless be manifest in simple textual differences. Sometimes, though, an error of logic will result in much more complex differences in program text. This may not invalidate fault-based testing with a simpler fault model, provided test cases sufficient for detecting the   simpler faults are sufficient also for detecting the more complex fault. This is known as the *coupling effect*.

The coupling effect hypothesis may seem odd, but can be justified by appeal to a more plausible hypothesis about interaction of faults. A complex change is equivalent

Δ competent programmer hypothesis

Δ coupling effect hypothesis

Fault Based Testing: Terminology

Original program: The program unit (e.g., C function or Java class) to be tested.

Program location: A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and boolean expressions, and procedure calls.

Alternate expression: Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

Alternate program: A program obtained from the original program by substituting an alternate expression for the text at some program location.

Distinct behavior of an alternate program R for a test t: The behavior of an alternate program R is distinct from the behavior of the original program P for a test t, if R and P produce a different result for t, or if the output of R is not defined for t.

Distinguished set of alternate programs for a test suite T: A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in T.

---

to several smaller changes in program text. If the effect of one of these small changes is not masked by the effect of others, then a test case that differentiates a variant based on a single change may also serve to detect the more complex error.

Fault-based testing can guarantee fault detection only if the competent programmer hypothesis and the coupling effect hypothesis hold. But guarantees are more than we expect from other approaches to designing or evaluating test suites, including the structural and functional test adequacy criteria discussed in earlier chapters. Fault-based testing techniques can be useful even if we decline to take the leap of faith required to fully accept their underlying assumptions. What is essential is to recognize the dependence of these techniques, and any inferences about software quality based on fault-based testing, on the quality of the fault model. This also implies that developing better fault models, based on hard data about real faults rather than guesses, is a good investment of effort.

16.3   Mutation Analysis

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by "seeding" faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns for changing program text are called *mutation operators*, and each variant program is called a *mutant*.

Δ mutation
operator

Δ mutant

Mutation Analysis: Terminology

Original program under test: The program or procedure (function) to be tested.

Mutant: A program that differs from the original program for one syntactic element, e.g., a statement, a condition, a variable, a label, etc.

Distinguished mutant: A mutant that can be distinguished for the original program by executing at least one test case.

Equivalent mutant: A mutant that cannot be distinguished from the original program.

Mutation operator: A rule for producing a mutant program by syntactically modifying the original program.

Δ valid mutant
Δ useful mutant

Mutants should be plausible as faulty programs. Mutant programs that are rejected by a compiler, or which fail almost all tests, are not good models of the faults we seek to uncover with systematic testing.   We say a mutant is valid if it is syntactically correct.   We say a mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases.

A mutant obtained from the program of Figure 16.1 by substituting while for switch in the statement at line 13 would not be valid, since it would result in a compile-time error. A mutant obtained by substituting 1000 for 0 in the statement at line 4 would be valid, but not useful, since the mutant would be distinguished from the program under test by all inputs and thus would not give any useful information on the effectiveness of a test suite. Defining mutation operators that produce valid and useful mutations is a non-trivial task.

Since mutants must be valid, mutation operators are syntactic patterns defined relative to particular programming languages. Figure 16.2 shows some mutation operators for the C language. Constraints are associated with mutation operators to guide selection of test cases likely to distinguish mutants from the original program. For example, the mutation operator *svr* (scalar variable replacement) can be applied only to variables of compatible type (to be valid), and a test case that distinguishes the mutant from the original program must execute the modified statement in a state in which the original variable and its substitute have different values.

Many of the mutants of Figure 16.2 can be applied equally well to other procedural languages, but in general a mutation operator that produces valid and useful mutants for a given language may not apply to a different language or may produce invalid or useless mutants for another language. For example, a mutation operator that removes the "friend" keyword from the declaration of a C++ class would not be applicable to Java, which does not include friend classes.

```
1
2    /** Convert each line from standard input */
3    void  transduce() {
4        #define BUFLEN 1000
5        char buf[BUFLEN];   /* Accumulate line into this buffer   */
6        int   pos = 0;          /* Index for next character in buffer */
7
8        char inChar; /* Next character from input */
9
10       int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12       while ((inChar = getchar()) != EOF ) {
13          switch (inChar) {
14          case LF:
15            if (atCR) {   /* Optional DOS LF */
16               atCR = 0;
17            } else {          /* Encountered CR within line */
18               emit(buf, pos);
19               pos = 0;
20            }
21            break;
22          case CR:
23            emit(buf, pos);
24            pos = 0;
25            atCR = 1;
26            break;
27          default:
28            if (pos >= BUFLEN-2) fail("Buffer overflow");
29            buf[pos++] = inChar;
30          } /* switch */
31       }
32       if (pos > 0) {
33          emit(buf, pos);
34       }
35    }
```

Figure 16.1: Program transduce converts line endings among Unix, DOS, and Macintosh conventions. The main procedure, which selects the output line end convention, and the output procedure emit are not shown.

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by abs(e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

16.4   Fault-Based Adequacy Criteria

Given a program and a test suite T, mutation analysis consists of the following steps:

Select mutation operators: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants: Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants: Execute the original program and each generated mutant with the test cases in T. A mutant is *killed* when it can be distinguished from the original program.

Figure 16.3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in Figure 16.2. Test suite TS

$$TS = \{1U, 1D, 2U, 2D, 2M, End, Long\}$$

kills M_j, which can be distinguished from the original program by test cases $1D$, $2U$, $2D$, and $2M$. Mutants M_i, M_k, and M_l are not distinguished from the original program by any test in TS. We say that mutants not killed by a test suite are *live*.   live mutants
A mutant can remain *live* for two reasons:

- The mutant can be distinguished from the original program, but the test suite T does not contain a test case that distinguishes them, i.e., the test suite is not adequate with respect to the mutant.

- The mutant cannot be distinguished from the original program by any test case, i.e., the mutant is equivalent to the original program.

Given a set of mutants SM and a test suite T, the fraction of non-equivalent mutants killed by T measures the adequacy of T with respect to SM. Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not, or by counting some equivalent mutants among the remaining live mutants.

The adequacy of the test suite TS evaluated with respect to the four mutants of Figure 16.3 is 25%. However, we can easily observe that mutant M_i is equivalent to the original program, i.e., no input would distinguish it. Conversely, mutants M_k and M_l seems to be non-equivalent to the original program, i.e., there should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of TS, measured after eliminating the equivalent mutant M_i, is 33%.

Mutant M_l is killed by test case *Mixed*, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines. We would expect that *Mixed* would kill also M_k, but this does not actually happen: both M_k and the original program produce the same result for *Mixed*. This happens because both the mutant and the original program fail in the same way.[1] The use of a simple oracle for checking

---

[1] The program was in regular use by one of the authors and was believed to be correct. Discovery of the fault came as a surprise while using it as an example for this chapter.

ID	Operator	line	Original/ Mutant	1U	1D	2U	2D	2M	End	Long	Mixed
M_i	ror	28	`(pos >= BUFLEN-2)` `(pos == BUFLEN-2)`	-	-	-	-	-	-	-	-
M_j	ror	32	`(pos > 0)` `(pos >= 0`	-	x	x	x	x	-	-	-
M_k	sdl	16	`atCR = 0` *nothing*	-	-	-	-	-	-	-	-
M_l	ssr	16	`atCR = 0` `pos = 0`	-	-	-	-	-	-	-	x

Test case	Description	Test case	Description
1U	One line, Unix line-end	2M	Two lines, Mac line-end
1D	One line, DOS line-end	End	Last line not terminated with line-end sequence
2U	Two lines, Unix line-end	Long	Very long line (greater than buffer length)
2D	Two lines, DOS line-end	Mixed	Mix of DOS and Unix line ends in the same file

Figure 16.3: A sample set of mutants for program Transduce generated with mutation operators from Figure 16.2.

the correctness of the outputs (e.g., checking each output against an expected output) would reveal the fault. The test suite TS_2 obtained by adding test case *Mixed* to *TS* would by 100% adequate (relative to this set of mutants) after removing the fault.

16.5   Variations on Mutation Analysis

The mutation analysis process described above, which kills mutants based on the outputs produced by execution of test cases, is known as strong mutation. It can generate a number of mutants quadratic in the size of the program. Each mutant must be compiled and executed with each test case until it is killed. The time and space required for compiling all mutants and for executing all test cases for each mutant may be impractical.

The computational effort required for mutation analysis can be reduced by reducing the number of mutants generated and the number of test cases to be executed. Weak mutation analysis reduces the number of tests to be executed by killing mutants when they produce a different intermediate state, rather than waiting for a difference in the final result or observable program behavior.

weak mutation analysis

With weak mutation, a single program can be seeded with many faults. A "meta-mutant" program is divided into segments containing original and mutated source code, with a mechanism to select which segments to execute. Two copies of the meta-mutant are executed in tandem, one with only original program code selected, and the other with a set of live mutants selected. Execution is paused after each segment to compare the program state of the two versions. If the state is equivalent, execution resumes with the next segment of original and mutated code. If the state differs, the mutant is marked as dead, and execution of original and mutated code is restarted with a new selection

Mutation Analysis vs Structural Testing

For typical sets of syntactic mutants, a mutation-adequate test suite will also be adequate with respect to simple structural criteria such as statement or branch coverage. Mutation adequacy can simulate and subsume a structural coverage criterion if the set of mutants can be killed only by satisfying the corresponding test coverage obligations.

Statement coverage can be simulated by applying the mutation operator sdl (statement deletion) to each statement of a program. To kill a mutant whose only difference from the program under test is the absence of statement S requires executing the mutant and the program under test with a test case that executes S in the original program. Thus to kill all mutants generated by applying the operator sdl to statements of the program under test, we need a test suite that causes the execution of each statement in the original program.

Branch coverage can be simulated by applying the operator cpr (constant for predicate replacement) to all predicates of the program under test with constants *True* and *False*. To kill a mutant that differs from the program under test for a predicate P set to the constant value *False*, we need to execute the mutant and the program under test with a test case that causes the execution of the *True* branch of P. To kill a mutant that differs from the program under test for a predicate P set to the constant value *True*, we need to execute the mutant and the program under test with a test case that causes the execution of the *False* branch of P.

A test suite that satisfies a structural test adequacy criterion may or may not kill all the corresponding mutants. For example, a test suite that satisfies the statement coverage adequacy criterion might not kill an sdl mutant if the value computed at the statement does not effect the behavior of the program on some possible executions.

of live mutants.

Weak mutation testing does not decrease the number of program mutants that must be considered, but it decreases the number of test executions and compilations. This performance benefit has a cost in accuracy: weak mutation analysis may "kill" a mutant even if the changed intermediate state would not have an effect on the final output or observable behavior of the program.

statistical mutation analysis

Like structural test adequacy criteria, mutation analysis can be used either to judge the thoroughness of a test suite or to guide selection of additional test cases. If one is designing test cases to kill particular mutants, then it may be important to have a complete set of mutants generated by a set of mutation operators. If, on the other hand, the goal is a statistical estimate of the extent to which a test suite distinguishes programs with seeded faults from the original program, then only a much smaller statistical sample of mutants is required. Aside from its limitation to assessment rather than creation of test suites, the main limitation of statistical mutation analysis is that partial coverage is meaningful only to the extent that the generated mutants are a valid statistical model of occurrence frequencies of actual faults. To avoid reliance on this implausible assumption, the target coverage should be 100% of the sample; statistical sampling may keep the sample small enough to permit careful examination of equivalent mutants.

Fault seeding can be used statistically in another way: To estimate the number of faults remaining in a program. Usually we know only the number of faults that have been detected, and not the number that remains. However, again to the extent that the fault model is a valid statistical model of actual fault occurrence, we can estimate that the ratio of actual faults found to those still remaining should be similar to the ratio of seeded faults found to those still remaining.

Once again, the necessary assumptions are troubling, and one would be unwise to place too much confidence in an estimate of remaining faults. None the less, a prediction with known weaknesses is better than a seat-of-the-pants guess, and a set of estimates derived in different ways is probably the best that one can hope for.

While the focus of this chapter is on fault-based testing of software, related techniques can be applied to whole systems (hardware and software together) to evaluate fault tolerance. Some aspects of fault-based testing of hardware are discussed in the sidebar on page 326.

Open research issues

Fault-based testing has yet to be widely applied in software development, although it is an important research tool for evaluating other test selection techniques. Its limited impact on software practice so far can be blamed perhaps partly on computational expense, and partly on the lack of adequate support by industrial strength tools.

One promising direction in fault-based testing is development of fault models for particular classes of faults. These could result in more sharply focused fault-based techniques, and also partly address concerns about the extent to which the fault models conventionally used in mutation testing are representative of real faults. Two areas

Estimating population sizes

Counting fish   Lake Winnemunchie is inhabited by two kinds of fish, a native brown trout and an introduced species of chub. The Fish and Wildlife Service wish to estimate the populations to evaluate their efforts to eradicate the chub without harming the population of native trout.

The population of chub can be estimated statistically as follows. 1000 chub are netted, their dorsal fins are marked by attaching a tag, then they are released back into the lake. Over the next weeks, fishermen are asked to report the number of tagged and untagged chub caught. If 50 tagged chub and 300 untagged chub are caught, we can calculate

$$\frac{1000}{\text{untagged chub population}} = \frac{50}{300}$$

and thus there are about 6000 untagged chub remaining in the lake.

It may be tempting to also ask fishermen to report the number of trout caught, and to perform a similar calculation to estimate the ratio between chub and trout. However, this would only be valid if trout and chub are equally easy to catch, or if one can adjust the ratio using a known model of trout and chub vulnerability to fishing.

Counting residual faults   A similar procedure can be used to estimate the number of faults in a program: Seed a given number S of faults in the program. Test the program with some test suite and count the number of revealed faults. Measure the number of seeded faults detected, D_S, and also the number of natural faults D_N detected. Estimate the total number of faults remaining in the program, assuming the test suite is as effective at finding natural faults as it is at finding seeded faults, using the formula

$$\frac{S}{\text{total natural faults}} = \frac{D_S}{D_N}$$

If we estimate the number of faults remaining in a program by determining the proportion of seeded faults detected, we must be wary of the pitfall of estimating trout population by counting chub. The seeded faults are chub, the real faults are trout, and we must either have good reason for believing the seeded faults are no easier to detect than real remaining faults, or else make adequate allowances for uncertainty. The difference is that we cannot avoid the problem by repeating the process with trout — once a fault has been detected, our knowledge of its presence cannot be erased. We depend, therefore, on a very good fault model, so that the chub are as representative as possible of trout. Of course, if we use special bait for chub, or design test cases to detect particular seeded faults, then statistical estimation of the the total population of fish or errors cannot be justified.

Hardware fault-based testing

Fault based testing is widely used for semiconductor and hardware system validation and evaluation, both for evaluating the quality of test suites and for evaluating fault tolerance.

Semiconductor testing has conventionally been aimed at detecting random errors in fabrication, rather than design faults. Relatively simple fault models have been developed for testing semiconductor memory devices, the prototypical faults being "stuck-at-0" and "stuck-at-1" (a gate, cell, or pin that produces the same logical value regardless of inputs). A number of more complex fault models have been developed for particular kinds of semiconductor devices, e.g., failures of simultaneous access in dual-port memories. A test vector (analogous to a test suite for software) can be judged by the number of hypothetical faults it can detect, as a fraction of all possible faults under the model.

Fabrication of a semiconductor device, or assembly of a hardware system, is more analogous to copying disk images than to programming. The closest analogue of software is not the hardware device itself, but its design — in fact a high-level design of a semiconductor device is essentially a program in a language that is compiled into silicon. Test and analysis of logic device designs faces the same problems as test and analysis of software, including the challenge of devising fault models. Hardware design verification also faces the added problem that it is much more expensive to replace faulty devices that have been delivered to customers than to deliver software patches.

In evaluation of fault tolerance in hardware, the usual approach is to modify the state or behavior rather than the system under test. Due to a difference in terminology between hardware and software testing, the corruption of state or modification of behavior is called a "fault," and artificially introducing it is called "fault injection." Pin-level fault injection consists of forcing a stuck-at-0, a stuck-at-1, or an intermediate voltage level (a level that is neither a logical 0 nor a logical 1) on a pin of a semiconductor device. Heavy ion radiation is also used to inject random faults in a running system. A third approach, growing in importance as hardware complexity increases, uses software to modify the state of a running system, or to simulate faults in a running simulation of hardware logic design.

in which researchers have attempted to develop focused models, expressed as sets of mutation operators, are component interfaces and concurrency constructs.

Particularly important is development of fault models based on actual, observed faults in software. These are almost certainly dependent on application domain and perhaps to some extent vary also across software development organizations, but there is too little empirical evidence available now on the degree of variability.

Further Reading

Software testing using fault seeding was developed by Hamlet [Ham77] and independently by DeMillo, Lipton, and Sayward [DLS78]. Underlying theory for fault-based testing, and in particular on the conditions under which a test case can distinguish faulty and correct versions of a program, were developed by Morell [Mor90] and extended by Thompson, Richardson, and Clarke [TRC93]. Statistical mutation using a Bayesian approach to grow the sample until sufficient evidence has been collected has been described by Sahinoglu and Spafford [SS90]. Weak mutation was proposed by Howden [How82]. The sample mutation operators used in this chapter are adapted from the Mothra software testing environment [DKMO88].

Exercises

16.1. Consider the following C function, used to determine whether a misspelled word differs from a dictionary word by at most one character, which may be a deletion, an insertion, or a substitution (e.g., "text" is edit distance 1 from "test" by a substitution, and edit distance 1 from "tests" by deletion of "s").

```
1
2    /* edit1( s1, s2 ) returns TRUE iff s1 can be transformed to s2
3     * by inserting, deleting, or substituting a single character, or
4     * by a no-op (i.e., if they are already equal).
5     */
6    int  edit1( char *s1, char *s2) {
7       if (*s1 == 0) {
8          if (*s2 == 0) return TRUE;
9          /* Try inserting a character in s1 or deleting in s2 */
10         if ( *(s2 + 1) == 0) return TRUE;
11         return FALSE;
12      }
13      if (*s2 == 0) { /* Only match is by deleting last char from s1 */
14         if (*(s1 + 1) == 0) return TRUE;
15         return FALSE;
16      }
17      /* Now we know that neither string is empty */
18      if (*s1 == *s2) {
19         return edit1(s1 +1, s2 +1);
```

```
20      }
21
22      /* Mismatch; only dist 1 possibilities are identical strings after
23       * inserting, deleting, or substituting character
24       */
25
26      /* Substitution: We "look past" the mismatched character */
27      if (strcmp(s1+1, s2+1) == 0) return TRUE;
28      /* Deletion: look past character in s1 */
29      if (strcmp(s1+1, s2) == 0) return TRUE;
30      /* Insertion: look past character in s2 */
31      if (strcmp(s1, s2+1) == 0) return TRUE;
32      return FALSE;
33   }
```

Suppose we seed a fault in line 27, replacing s1 +1 by s1 + 0. Is there a test case that will kill this mutant using weak mutation, but not using strong mutation? Display such a test case if there is one, or explain why there is none.

16.2. We have described weak mutation as continuing execution up to the point that a mutant is killed, then re-starting execution of the original and mutated program from the beginning. Why doesn't execution just continue after killing a mutant? What would be necessary to make continued execution possible?

16.3. Motivate the need for the competent programmer and the coupling effect hypotheses. Would mutation analysis still make sense if these hypotheses did not hold? Why?

16.4. Generate some invalid, valid-but-not-useful, useful, equivalent and non-equivalent mutants for the program in Figure 16.1 using mutant operators from Figure 16.2.