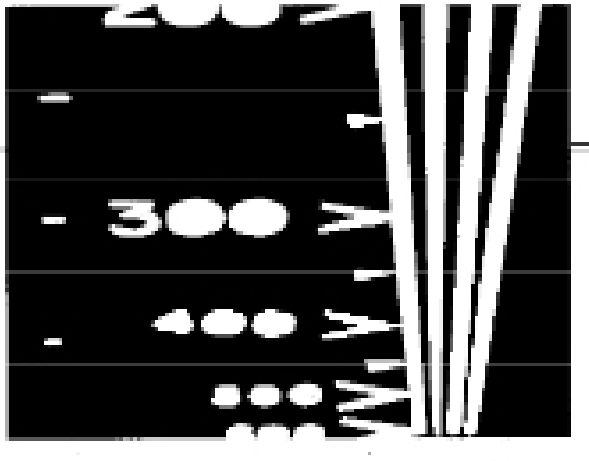


Predicting Where Faults Can Hide from Testing

JEFFREY VOAS, *NASA Langley Research Center*
LARRY MORELL, *Hampton University*
KEITH MILLER, *College of William and Mary*

◆ *Sensitivity analysis estimates the probability that a program location can hide a failure-causing fault. It does not require an oracle because correctness is not the issue.*

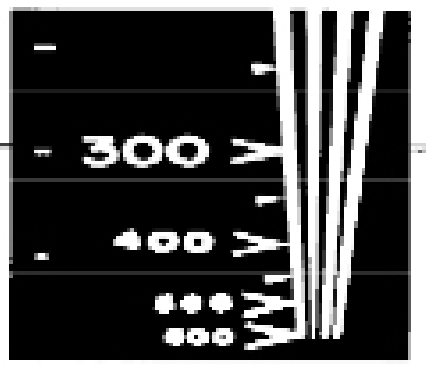
Testing seeks to reveal software faults by executing a program and comparing the output expected to the output produced. Exhaustive testing is the only testing scheme that can (in some sense) guarantee correctness. All other testing schemes are based on the assumption that successful execution on some inputs implies (but does not guarantee) successful execution on other inputs.

Because it is well known that some programming faults are very difficult to find with testing, our research focuses on program characteristics that make faults hard to find with random black-box testing. Given a piece of code, we try to predict if random black-box testing is likely to reveal any faults in that code. (By "fault," we mean those that have been compiled into the code.)

A program's testability is a prediction of its ability to hide faults when the program is black-box-tested with inputs selected randomly from a particular input distribution. You determine testability by the code's structure and semantics and by an assumed input distribution. Thus, two programs can compute the same function but may have different testabilities. A program has high testability when it readily reveals faults through random black-box testing; a program with low testability is unlikely to reveal faults through random black-box testing. A program with low testability is dangerous because considerable testing may make it appear that the program has no faults when in reality it has many.

A fault can lie anywhere in a program, so any method of determining testability must take into consideration all places in the code where a fault can occur. Although you can use our proposed techniques at different granularities, this arti-

0740-7459/91/0300/0041/$01.00 © 1991 IEEE



```
1.  read(a,b,c);
2.  if a <> 0 then begin
3.     d := b*b – 5*a*c;
4.     if d < 0 then
5.        x := 0
       else
6.        x := (–b + trunc(sqrt(d))) div (2*a)
    end
    else
7.     x := –c div b;
8.  if (a*x*x + b*x + c = 0) then
9.     writeln(x, ' is an integral solution')
    else
10.    writeln('There is no integral
                solution')
```

Figure 1. Example program.

cle concentrates on locations that roughly correspond to single commands in an imperative, procedural language.

We expect that any method for determining testability will require extensive analysis, a large amount of computing resources, or both. However, the potential benefits for measuring testability are significant. If you can effectively estimate testability, you can gain considerable insight into four issues important to testing:

- Where to get the most benefit from limited testing resources. A module with low testability requires more testing than a module with high testability. Testing resources can thus be distributed more effectively.
- When to use some verification technique other than testing. Extremely low testability suggests that an inordinate amount of testing may be required to gain confidence in the software's correctness. Alternative techniques like proofs of correctness or code review may be more appropriate for such modules.
- The degree to which testing must be performed to convince you that a location is probably correct. You can use testability to estimate how many tests are necessary to gain desired confidence in the software's correctness.
- Whether the software should be rewritten. You may use testability as a guide to whether critical software has been sufficiently verified. If a piece of critical software has low testability, you may reject it because too much testing will be required to sufficiently verify a sufficient level of reliability.

SENSITIVITY

We use the word "sensitivity" to mean a prediction of the probability that a fault will cause a failure in the software at a particular location under a specified input distribution. If a location has a sensitivity of 0.99 under a particular distribution, almost any input in the distribution that executes the location will cause a program failure. If a location has a sensitivity of 0.01, relatively few inputs from the distribution that execute would cause the program to fail, no matter what faults exist at that location.

Sensitivity is clearly related to testability, but the terms are not equivalent. Sensitivity focuses on a single location in a program and the effects a fault at that location can have on the program's I/O behavior. Testability encompasses the whole program and its sensitivities under a given input distribution. Sensitivity analysis is the process of determining the sensitivity of a location in a program. From the collection of sensitivities over all locations, we determine the program's testability.

One method of performing sensitivity analysis is successive analysis of program execution, infection, and propagation, which is dynamic in the sense that it requires execution of the code. You randomly select inputs from the input distribution and compare the code's computational behavior on these inputs against the behavior of similar code. Although this analysis is dynamic, it is not software testing, since you check no outputs against a specification or oracle.

FAULT/FAILURE MODEL

If the presence of faults in programs guaranteed program failure, every program would be highly testable. But this is not true. To understand why, you must consider the sequence of location executions that a program performs. Each set of variable values after the execution of a location in a computation is called a data state. After executing a fault, the resulting data state might be corrupted; if there is corruption in a data state, *infection* has occurred and the data state contains an error, which we call a "data-state error."

The program in Figure 1 displays an integral solution to the quadratic equation ax^2+bx+c for integral values of a, b, and c. (We have fixed a, b, and c so a and c fall between 0 and 10 and so b falls between 1 and 1,000.) The program has a fault at line 3: The constant 5 should be the constant 4. Each computation of the program falls into one of four categories:

- the fault is not executed,
- the fault is executed but does not infect any data state,
- the fault is executed and some data states are infected, but the output is nonetheless correct, and
- the fault is executed, infection occurs, and the infection causes an incorrect output.

Only computations in the final category would make the fault visible to a tester. Here are examples of each type of computation:

- Table 1 shows the computation for the input $(a,b,c)=(0,3,6)$. The value of $a=0$ causes the selection of a path that does not include location 3. Clearly, any such execution will not fail.

If the presence of faults in programs guaranteed program failure, every program would be highly testable. But this is not true. To understand why, you must consider the sequence of location executions that a program performs.

• Table 2 shows the computation for the input (3,2,0). The fault is reached, but the computation proceeds just as if there were no fault because c=0 prevents the fault from affecting the computation. No infection has occurred.

• For the input (1,–1,–12), the fault infects the succeeding data state, producing d=61 instead of d=49 (see Table 3). This data-state error then propagates to location 6 where it is canceled by the integer square-root calculation, because 7 is computed in either case.

• Executing the program with the input (10,0,10) executes the fault that then infects the succeeding data state so the data-state error propagates to the output (see Table 4).

The first computation type demonstrates that a program's execution can reveal only information about the part of the code that is executed. The second and third types provide a false sense of security to a tester because the fault is executed but no visible failure results. The fourth type shows three necessary and sufficient conditions for a fault to produce a failure:

• The fault must be executed.

• The succeeding data state must be infected.

• The data-state error must propagate to output.

These three phenomena comprise the fault/failure model. This model underlies our dynamic method to determine the sensitivity of a location in the code.

SENSITIVITY ANALYSIS

Sensitivity analysis requires that every location be analyzed for three properties: the probability of execution occurring, the probability of infection occurring, and the probability of propagation occurring. One type of analysis is required to handle each part of the fault/failure model.

Getting three estimates. You can make all three analyses at several different levels of abstraction — programs, modules, and statements are three such levels. The examples in this article show an analysis done on program locations where a location is a unit of code that changes a variable's value,

TABLE 1
THE VALUES (0,3,6) AS INPUT TO THE PROGRAM IN FIGURE 1

Location	a	b	c	d	x	Output
1	0	3	6	undefined	undefined	
7	0	3	6	undefined	–2	
8	0	3	6	undefined	–2	
9	0	3	6	undefined	–2	–2 is an integral solution

TABLE 2
THE VALUES (3,2,0) AS INPUT

Location	a	b	c	d	x	Output
1	3	2	0	undefined	undefined	
2	3	2	0	undefined	undefined	
3	3	2	0	4	undefined	
6	3	2	0	4	0	
8	3	2	0	4	0	
9	3	2	0	4	0	0 is an integral solution

TABLE 3
THE VALUES (1,–1,–12) AS INPUT

Location	a	b	c	d	x	Output
1	1	–1	–12	undefined	undefined	
2	1	–1	–12	undefined	undefined	
3	1	–1	–12	61	undefined	
6	1	–1	–12	61	4	
8	1	–1	–12	61	4	
9	1	–1	–12	61	4	4 is an integral solution

TABLE 4
THE VALUES (10,0,10) AS INPUT

Location	a	b	c	d	x	Output
1	10	0	10	undefined	undefined	
2	10	0	10	undefined	undefined	
3	10	0	10	–500	undefined	
4	10	0	10	–500	undefined	
5	10	0	10	–500	0	
10	10	0	10	–500	0	There is no integral solution