ADAPTIVE SEARCH TECHNIQUES APPLIED TO SOFTWARE TESTING

J.P. Benson
General Research Corporation
5383 Hollister Avenue
Santa Barbara, California 93111

Abstract

An experiment was performed in which executable assertions were used in conjuction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point.

Errors were interested in the program randomly using an error generation method based on published data defining common error types. Assertions were written for program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within the subranges. In this way a "grid" of test values was constructed over the program's input space.

The second used a search algorithm from optimization theory. This entailed using the assertions to define an error function and then maximizing its value. The program was then tested by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.

I. Introduction

Although Di jkstra's famous comment on testing, that it will never show the absence of bugs, only their presence, is undoubtedly true, testing is still the method most used for showing the correctness of software. If testing is to be used, ways must be found to make it more efficient and effective.

A paper by Alberts[1] presents data indicating that testing and validation efforts account for approximately 50% of the cost of developing a software system, where development includes the typical phases of conceptual design, requirements analysis, development, and operational use. This cost includes those associated with locating the errors, correcting the errors (which may include redesign), and checking that the corrections have removed the cause of the error. The testing process is a very labor-intensive activity, as is any aspect of software development. If methods could be found to automate the testing process, the cost of developing software could be reduced.

II. Problems With Testing

Two of the many problems involved in testing software are 1) how to develop test cases which identify errors and 2) how to check the results from these test cases. Before software testing can be automated and its cost reduced, these two problems must be solved.

Many methods have been proposed for identifying test cases which will show that a program performs correctly or indicate the errors which are present in the program. For examples of these methods see Howden[2] and Gannon[3]. Basically, the problem is one of complexity. For most programs, the number of different combinations of input values is practically infinite. Therefore, using exhaustive testing to show that a program works correctly is an impossible task.

Given the fact that programs cannot be tested by trying all test cases, what are the alternatives? Boundary value testing, path testing, and symbolic execution[4] have been some of the suggested solutions. The key problem is finding test cases which detect the errors present in the software. At present, there are no methods for deriving test cases with this property although many studies of the types of errors commonly found in software have been undertaken.[5-7]

The second problem has to do with checking whether a test has been successful. Even if there were a method for selecting test cases which was able to identify specific errors in a program, the process of evaluating whether or not the program ran successfully is a manual one. The output from the program must be compared with the expected results.

©1981 ACM 0-89791-038-9 /81/0003/0109 $00.75

For large programs composed of many functions this is a very time-consuming task.

III. A Proposed Solution

From the above discussion, it is evident that automating the testing of computer programs requires finding methods for developing effective test cases as well as methods for efficiently evaluating the results of using them. A method for solving these problems has been developed that combines the use of search algorithms from operations research with executable assertions from software verification research.

Finding the maximum or minimum value of a function of several variables, each subject to some set of constraints, is a common problem in operations research. Minimizing the cost of constructing a building given the choice of using brick, wood, and adobe materials in different proportions typifies problems of this sort. Many methods have been developed for solving such problems, for example, see Denn.[8] One of the simplest is to define the parameter of interest (e.g., cost) as a function of the possible alternative (e.g., brick, wood, adobe). The problem then is to find a minimum value of the function defined by the values of the alternatives (variables). Figure 1 illustrates this for two variables, brick and wood. The cost function defines a surface, with "hills" (maximum) and "valleys" (minimums).

The goal is to find a point on this surface which is a minimum (in the example of building cost). This point corresponds to a particular set of values of the alternatives or variables. Finding such a minimum value requires that this surface be searched. There are many methods for traversing the surface according to some search heuristic (for example, in the direction of the gradient) until a solution is found.

The problem of evaluating the results limits the application of these techniques to the testing of computer programs. That is, in operations research, we are usually trying to maximize or minimize the value of one variable, whereas in software testing we are usually trying to compute the value of many output variables with their expected values.

The solution of this problem has been found in "executable assertions," a technique developed for providing software correct and for checking it while it is running. Assertions are comments added to a program which specify how the program is to behave. They may specify a range of values for a variable, the relation the values of two ar more variables have to each other or compare the

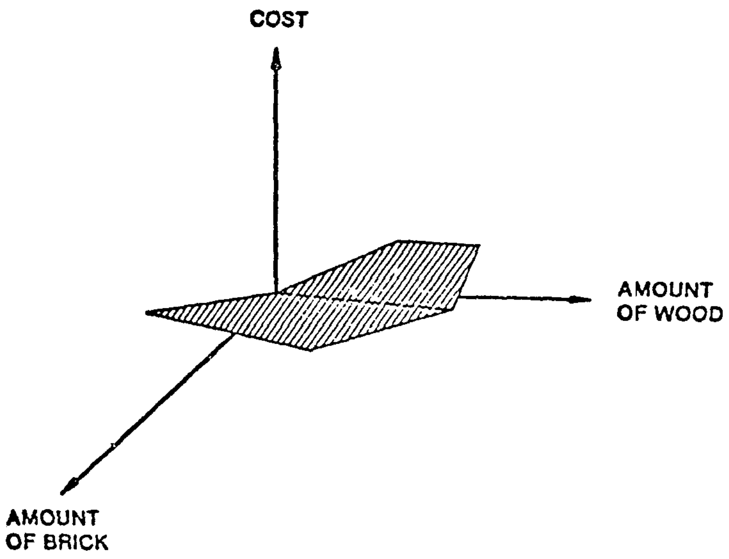

Figure 1. Cost as a Function of Building Material

the state of a present computation to that of a past computation. Figure 2 shows an example of two assertions that specifies the range of values that the variable Value can assume.

To make an assertion "executable," we merely translate it into machine language. Then while the program is running, the assertion can be evaluated. As in the case of a logical function, the assertion has a value of true or false. If the value of an assertion becomes false at any point in the execution of a program, then this can be reported as any other error message.

```
ASSERT (VALUE .GE. 0.0)
ASSERT (VALUE .LE. TWOPI)
```

Figure 2. Examples of Assertions

IV. Combining Assertions and Search Algorithms

Assertions give us a method for evaluating whether a program has run correctly without looking at all of its output. If the assertions are written correctly and they completely specify the algorithm, then the correctness of the program can be determined while the program is running. This is not to say that writing assertions to accomplish this is easy; a comprehensive and complete set of assertions for a program is difficult to develop. But if it can be done, then the problem of examining the output of a program to determine whether it executed a test case correctly has been solved.

Since using assertions means that we no longer have to examine the output of a program, the automated testing of computer programs becomes possible--provided we can automate the selection of test cases. If we can transform the output from the assertions into a function, we can utilize the search techniques from operations research to locate errors.

The basic idea is this: The function we define is the number of assertions that become false during the execution of a particular test case. The independent variables are the values of the input variables of the program. The search techniques will be used to find the values of the input variables for which the maximum number of assertions are violated. The function relating the number of assertions violated to the values of the input variables is called the "error function," and the surface that it describes is called the "error space."

If the search algorithm is to perform correctly, the error function must 1) not define a flat (uniform) surface and 2) not be discontinuous (have spikes) at any points. A previous experiment,[9] investigated the error function for a scheduling program. It was found that the error function for this program was neither uniform nor discontinuous. In a second experiment, described below, we have attempted to show that this is also true for another program "seeded" with several types of errors. We have also attempted to determine the efficiency of the search technique in locating these errors relative to other types of testing methods.

V. The Experiment

The experiment was to select a program, add assertions to it, and seed it with errors from a list of typical software errors. The location of the errors was determined randomly. Each of the errors was inserted in the program one at a time and the program was then tested by systematically choosing combinations of values for the input parameters. This testing was done automatically by a program which varied the input parameters over the required values. After this, the program was tested by the search routine, first by allowing the search algorithm to vary the same variables that were varied in the first tests, and then allowing it to vary all of the input variables.

V.1 The Program

The program selected takes an orbit described by six independent parameters (longitude of the ascending node, inclination of the orbit plane, angle of the perigee, eccentricity, time at perigee, and semi-major axis) and converts this description into a state vector representation of a point on the orbit (time, position, velocity, and acceleration). The point is determined by the values of two other parameters. The range of values of one of these parameters is dependent upon the other. In all, there are ten input parameters is dependent upon the other. In all, there are ten input parameters, seven of which are independent of the other.

V.2 The Search Routine

The search routine chosen for the experiment was one developed by Box[10] called complex search. This algorithm constructs a hypertriangle, or complex, of the values of the function from several tests and then rotates, skrinks, expands, and projects the complex in order to locate a value which is larger (in the case of finding the maximum) than the worst point currently in the complex. The worst point is then replaced by the new point and the process continued until no further progress can be made.

V.3 The Test Driver

Several programs were also written in order to support the testing and make it as automatic as possible: 1) A test driver, which handled the selection of the testing method to be used and read in an initial test case was written, 2) a set of subroutines which implemented the constraints among the input variables used in generating new values for the search routine, and 3) a set of routines to count the number of assertions violated in each test and print the results.

V.4 The Assertions

Assertions added to the program were of three types: 1) Those that described ranges of variable values, 2) Those that describe the relationship between values of variables, and 3) Those which kept track of the history of the computation. Two routines were also written which included assertions to check the values of the input variables and the correctness of the results. These routines were invoked at the beginning of the test program and at the end of the test program.

V.5 Selecting Errors

Certain categories of errors were selected from a list of common software errors.[5] Errors of these types were inserted into the test program by randomly selecting sites (statements in the program) where the particular type of error could occur. Table 1 shows the errors used in the experiment.

V.6 Testing Techniques

The program was then tested by inserting one error at a time. First, the program was tested by taking combinations of values from three input variables. The permissible input range of each of the variables was divided up into equal subranges so that a reasonable number of test cases could be performed. Test values for each variable were selected by choosing the end-points of each subrange. The program was then testes using the selected values for the three input variables. The program was then tested using the selected values for the three

TABLE 1
ERRORS USED IN THE EXPERIMENT

Error Number	Category	Description
1	A200	incorrect use of parenthesis
3	A300	sign convention error
8	A600	incorrect/inaccurate equation used /wrong sequence
13	A100	incorrect operand in equation
14	A800	missing computation
28	B400	missing logic or condition tests
31	B400	missing logic or condition tests
36	B200	logic activities out of sequence
37	B200	logic activities out of sequence
40	B300	wrong variable being checked
41	D200	data initialization done improperly
46	D100	data initialization not done
47	D100	data initialization not done
48	D400	variable referred to by the wrong number
52	D600	incorrect variable type
54	D600	incorrect variable type
55	D600	incorrect variable type
56	D400	variable referred to by the wrong name
57	D300	variable used as a flag or index not set properly
62	F100	wrong subroutine called
64	F100	wrong subroutine called
67	F700	software/software interface error
74	F200	call to subroutine not made or made in wrong place
77	F700	software/software interface error

input variables. First, the values of two of the three variables were fixed at a value selected from their range of test values. Then, a test was run for each of the test values of the third variable. The value of the third variable was then fixed, and the first variable was varied over its set of test values. After this, the values of the first and third variable were fixed and the second was varied. The testing continued until all combinations of the test values for the test values for the three variables had been used. In this way a "grid" over the input space was obtained. The values of the variables which caused assertions to be violated and the number of assertions violated were recorded.

A majority of the errors (15 out of 24) were not detected by the original assertions for a number of reasons. Two of the errors were not detected since they occurred only if another error had occurred previouly during program execution. For other errors, it was found to be very difficult to write assertions that would detect them. Finally, eight of the errors were not detected simply because the program did not contain enough assertions. In order to investigate the performance of the search algorithm, new assertions were added to the program and the grid tests were run again. Errors which were not detected in this second set of tests were removed from the list of errors used in the experiment.

Next, the errors were again inserted one at a time and the search routine was allowed to vary only the variables which were varied in the grid tests. The number of assertions violated and the input values which caused the violations were recorded.

Finally, the errors were again used one at a time; but this time the search routine was allowed to vary any of the seven independent variables in order to locate a maximum. Again, the assertions violated and the input values which caused the violations were recorded.

VI. Results

The results from the grid tests demonstrated the effectiveness of the assertions in detecting the errors. Table 2 shows the results of these tests. Of the original 24 errors, nine (thirty-eight percent) were detected by the original assertions, and eight (thirty-three percent) were detected by the assertions that were added. (The seven errors, twenty-nine percent, which could not be detected by assertions, were not tested).

The relative effectiveness of the search testing methods versus the grid testing method is summarized in Table 3. (In this table, and those following, the "error number" column refers to a unique number assigned to each error by the error generation method.) In one case, the grid technique caused an assertion violation violation which neither search technique caused. In another case, the search technique using all variables was not able to cause an assertion violation that was caused by the grid technique and the search varying three variables. On the other hand, the search technique using all variables was able to cause an assertion violation which neither the grid technique nor the search using three variables was able to cause. Finally, in one case the search technique using three variables caused an assertion violation that the grid technique did not cause while the search using all variables caused another assertion violation in addition to the one discovered by the search using three variables. In all other tests, each of the methods caused the same assertions to be violated.

The efficiency of the search technique was not measured directly, but an estimate of the behavior of the all-variable search technique in relation to the grid technique in relation to the grid technique can be given. Except for error 52, which required 683 tests, the grid technique required 317 tests. In the case for each error, the number of the test in which the first assertion violation was detected. In all, fifteen of the seventeen detectable errors were detected by the seventh test in the search.

VII. Discussion

The results from the experiment show that it is possible to detect errors automatically using assertions and search techniques. The major limitation of the technique as we see it is the difficulty in writing the assertions. The number of assertions which need to be written, the conditions they should describe and where they should be placed are all questions which are difficult to answer. In addition, the assertions are difficult to write and the task of writing them is not pleasant. On the other hand, the search testing technique aids in the refinement of the assertions.

Unfortunately, our results have also shown the limitations of assertions. There is sometimes no way to easily express exactly what is wanted by using the current semantics. In some cases, it seems that other techniques are more suited to detecting certain types of errors.

One may also argue with the technique of "error seeding," but we believe it to be a very effective way in which to control

TABLE 2

RESULTS FROM GRID TESTS

	Number	Percentage
Errors Detected by Original Assertions	9	38
Errors Detected by Added Assertions	8	33
Errors Not Detected by Assertions	7	29
Total	24	100

TABLE 3

EFFECTIVENESS OF SECOND TESTING TECHNIQUES

	Number of Assertion Violations Detected by Testing Technique		
Error Number	Grid	3-Variable Search	All-Variable Search
14	1	2	3
28	1	0	0
47	2	2	1
74	7	7	8

TABLE 4

DETECTION OF ASSERTION VIOLATIONS BY SEARCH METHOD

Error Number	Test Number of First Assertion Violation
1	5
3	2
13	7
14	5
28	*
31	4
37	5
41	3
47	57
48	3
52	3
54	3
56	5
57	7
64	2
67	5
74	2

*No assertion violations detected.

some of the problems in an experiment such as this. Using programs from actual development efforts containing unknown errors would introduce factors into the experiment which could not be controlled. Interpreting the results of such an experiment would therefore be more difficult.

Equating assertion violations with errors is also a point which may be argued. In this experiment, it was assumed that once an assertion violation was detected, the error would become self-evident. This is obviously not true in the case. This will be true only if assertions are placed in the correct spot and describe the nature of the error. Again, only further experimentation can determine how useful the technique is at locating errors.

The way in which the error function was constructed to allow the search routine to be used can also be questioned. Simply summing the number of assertions to determine the value of the function is a crude technique. The search technique is thereby driven to select input values which maximize the number of assertions violated. We have found some evidence to indicate that errors are not randomly distributed; that they occur in groups. Therefore, searching for maximums of the error function should locate most of the errors in a program. However, this is still a crude method. We are investigating a method which takes the content of the assertions into account in generating new input values. This technique is taken from artificial intelligence research and will be the basis for further experiments.

In addition to the new experiments described above, we also believe that the techniques need to be applied to cases where more than one error occurs in the software, and to types of programs other than arithmetic computations (e.g., compilers). The efficiency of the technique relative to other types of testing should also be investigated.

We believe that the experiment successfully demonstrated the value of the search testing method. We were able to locate errors in a program automatically and relieved ourselves of the necessity of inventing test cases. In addition, the technique identified errors in our conception of the operation of the program as embedded in the assertions.

References

1. D.S. Alberts, "The Economics of Software Quality Assurance" in AFIPS Conference Proceedings: 1976 National Computer Conference, Vol. 45, AFIPS Press, Montvale, N.J. pp. 433-442.
2. W.E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, July 1978.
3. C. Gannon, "Error Detection Using Path Testing and Static Analysis," Computer, Vol. 12, August 1979.
4. L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-4, September 1976.
5. T.A. Thayer et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, Redondo Beach, Calif., August 1976.
6. M.J. Fries, Software Error Data Acquisition, Boeing Aerospace Company, RADC-TR-77-130, Seattle, Washington, April 1977.
7. Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.
8. M.M. Denn, Optimization by Variational Methods, New York, McGraw-Hill, 1969.
9. J. Benson, A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February 1980.
10. M.J. Box, "A New Method of Constrained Optimization and Comparison with Other Methods," Computer Journal, Vol. 8, 1965.