AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION

D W ANDREWS, J P BENSON

1980

UNCLASSIFIED

AFOSR-TR-81-0057

END
This paper describes an automated testing methodology and an experiment performed to determine its effectiveness. The method is to insert in the program to be tested a number of "executable assertions," statements about the program that trigger error signals whenever they are evaluated to be false (violated). A testcase is then developed for the program using actual values of the input variables. When the program is run, a plot is generated of the number of assertions violated versus the input variable values used. The resulting function is called the "error function." Heuristic search algorithms...
can then be used to maximize this function and thereby automatically locate input values which cause the most errors to occur. The experiment included developing assertions for the program to be tested, choosing and inserting representative errors into the program, and implementing search and data collection algorithms for testing. The results indicate that combining executable assertions with heuristic search algorithms is an effective method for automating the testing of computer programs.
AN AUTOMATED PROGRAM TESTING METHODOLOGY
AND ITS IMPLEMENTATION

Dorothy M. Andrews* and Jeoffrey P. Benson
General Research Corporation
P.O. Box 6770
Santa Barbara, California 93111

ABSTRACT

This paper describes an automated testing methodology and an experiment performed to determine its effectiveness. The method is to insert in the program to be tested a number of "executable assertions," statements about the program that trigger error signals whenever they are evaluated to be false (violated). A test case is then developed for the program using actual values of the input variables. When the program is run, a plot is generated of the number of assertions violated versus the input variable values used. The resulting function is called the "error function". Heuristic search algorithms can then be used to maximize this function and thereby automatically locate input values which cause the most errors to occur. The experiment included developing assertions for the program to be tested, choosing and inserting representative errors into the program, and implementing search and data collection algorithms for testing. The results indicate that combining executable assertions with heuristic search algorithms is an effective method for automating the testing of computer programs.

INTRODUCTION

In recent years, an active research area in computer science has been the development of methods for showing that computer programs operate correctly. One result of this research has been executable assertions. If assertions are used to specify the desired behavior of a program, then the program's correctness (relative to the assertions) can be checked automatically. This is done by using the number of assertions violated during a test as a correctness measure for the program. This value indicates whether the program is operating correctly on its input data; the number of assertions violated defines an "error function" over the input space of the program. This removes the need to examine a program's output in detail.

The error function can also be used to automatically generate test cases. It allows standard techniques for maximizing and minimizing functions in multi-dimensional spaces to be applied to the problem of program testing. Automated search techniques such as complex search and heuristic search can be used to find the maximum values of the error function. The input values for which assertions are violated are the input values for which the program fails to work correctly; therefore, it is desirable to find the regions with the maximum violations.

PROBLEMS ASSOCIATED WITH TESTING SOFTWARE

Testing has always been a problem of software development. The method used for testing a program is often a product of the idiosyncrasies of the tester. Typically the test criterion is to execute the program for a certain length of time or run a large program through the system. The critical nature of many current software systems makes it imperative to develop a generalized methodology for testing; one that can be applied to many types of programs, thus avoiding the subjective nature of present testing techniques. One way to minimize subjectivity is, of course, to have someone who has not been involved in writing the program do the testing. But this solution in itself brings new problems. The most obvious is that extra time must be allowed to train a new person until he is familiar enough with the program to intelligently make up test cases and procedures for testing. Although computer hardware testing and quality assurance is often a separate department in an organization, this is not commonly the case for software testing.

Test cases

There are two ways to make testing more reliable: increase the number of test cases or make the test cases more specific to the problem. Developing test cases requires much human ingenuity to discover the weak spots in a program and develop input data to test for them. This contributes to the skyrocketing cost of testing since each set of test cases must be tailored for each software system.

It is important to choose test cases which uncover errors early in the development cycle, not only because the cost of fixing errors increases dramatically with time, but also because of catastrophes that can result from latent errors.

There have been many papers on the subject of choosing test cases since it represents one of the most intriguing problems of testing. How does the tester know when enough
input data has been chosen for meaningful testing? In fault-tolerant applications, the
test data must include not only the expected
values of input data, but also the unexpected
values to look for intermittent hardware faults. Since software can have so many states, the
number of testcases required increases exponen-
tially.

Test Results

Software testing is unlike hardware
testing in that there is no "touchstone" that
can be used as a basis for determining if the
results from the testing are correct. Test
results must be checked manually. In some
applications, e.g., ballistic-missile-defense
software, checking test results from one run can
take several weeks. Unfortunately, with
software, it is not a matter of determining if a
switch is on or off; there is a lot of output to
read and analyze.

Last but not least is the psychological
aspect of testing that works against a produc-
tive testing phase. Once the software is com-
pleted, the programmer is anxious to get onto
some other project. The challenging and inter-
esting part is designing and implementing the
code, not testing it. No one really wants to
find errors in his own code, and, furthermore,
checking the output is so tedious that it makes
the testing process seem routine and boring.

EXECUTABLE ASSERTIONS AND ADAPTIVE TESTING

The major theme that connects most of the
problems associated with testing is that of
time. It takes time to construct good test-
cases, time to run them, and time to look at the
results. Therefore, one of the ways to address
the problem of testing is to automate as much of
the testing sequence as possible and to elimi-
nate as much subjectiveness and human interven-
tion as is practical. Fortunately, the basic
mechanism to do this, called an Adaptive Test-
er,4,5 has been developed over the past several
years in response to the need of the Ballistic
Missile Defense Advanced Technology Center to
develop tests for complex software. The Adaptive
Tester is a software system with the following
functional components:

- Machine aids for specification of the
testing environment
- Automatic preparation of initial test
cases
- Automatic performance evaluation
- Adaptive or learning algorithms for
selecting test cases

The research effort described in this
paper has utilized the components of the Adap-
tive Tester which generate testcases by auto-
matic perturbation of the input parameters,
evaluate past performances of the constructed
testcases, and, using this information in a
feedback system, generate subsequent testcases.

To adapt this powerful capability to this
particular application, executable assertions
were used as a means of providing data to the
performance evaluator. Executable assertions
allow the method to be prescribed in general
terms and used for any application, since the
only thing that varies from one application to
another are the assertions themselves.

Executable Assertions

Executable assertions have been found
very effective as a simple debugging technique
and have been utilized extensively in the
development of the Software Quality Laboratory
(a large verification system). The primary
motivation for adding them was to make debugging
easier and quicker since the exact statement
number of an assertion that is evaluated to
"false" during program execution is stated in a
message in the output. For example, if the
assertion INITIAL ( J.GE. 0 AND, J.JE. MAXJ ), is evaluated as "false", then it is clear
that J is negative or it has exceeded the
maximum value for J (MAXJ). Without assertions
to direct attention to the parts of the program
that are not operating as expected, it is often
impossible to find the source of the errors that
are causing the problems.

Not only are assertions useful for debug-
ing when new code is being added, but the
presence of assertions with a special statement
to invoke an error recovery routine usually
prevents premature termination and allows the
program to continue to perform its function.

Assertions also have proved their worth from the
aspect of maintenance and documentation of the
system. The Software Quality Laboratory is so
large that no one person can be familiar with it
all. Assertions which specify the acceptable
range of variables help immensely when new code
is being written to interface with existing
code.

The Adaptive Tester

The Adaptive Tester has been developed in
response to the need of the Ballistic Missile
Defense Advanced Technology Center to develop
tests for software that simulates actual battle
conditions. Devising tests in the conventional
fashion takes an inordinate amount of time
because of the number of parameters that can be
varied. Additionally, about a month is required
to examine the results of a single run. To get
around this situation, ways to perturb the input
variables and evaluate the results automatically
have been developed. Various search algorithms
from artificial intelligence have been implemen-
ted to construct new test cases from the results
of previous tests.
The search algorithm selected for this experiment employs complex search. The technique was developed by Box for solving for the maximum or minimum of a nonlinear function. It involves choosing a set of independent values for the function at random and determining the value of the function for each of these values. The function values define a set of points on a surface called a "complex". Figure 1 gives an example of a complex in three dimensions. The function may have many independent variables, but, for the search routine to function correctly, there must be one more point in the complex than the number of independent variables being perturbed. Assume that the goal is to maximize the value of the function. At each step of the algorithm the point in the complex with the minimum function value is replaced by a new point. The algorithm first attempts to locate the new point on a line connecting the rejected point and the centroid of the other points in the complex.

The distance between the rejected point and the centroid is calculated and the exact location of the new point is then determined. The algorithm has six choices in locating the new point. These are shown in Figure 2 for a complex that is a triangle. Reflection locates the new point at the same distance from the centroid as the rejected point by reflecting the triangle through the centroid. Expansion reflects the triangle through the centroid to locate the new point, but increases the distance between the new point and the centroid. Contraction reflects the triangle through the centroid and reduces the distance that the new point lies from the centroid. Centroid substitution uses the centroid as the new point of the triangle. If none of these operations results in a new point with a function value greater than the rejected point, then two other operations on the triangle can be performed. The triangle can be shrunk by reducing the lengths of one of its sides or rotating it about its centroid.

The coefficients of expansion, contraction, shrinkage and rotation are defined by the user of the algorithm. These operations are performed until a larger function value is found, a predefined limit on the number of operations is reached, or the function attains a value predefined as the maximum value that the algorithm is to locate.

METHODOLOGY FOR ADAPTIVE TESTING WITH ASSERTIONS

The testing methodology used in the experiment is very similar to that used in the Adaptive Testing project. The test configuration consists of several distinct software subsystems: a test driver, the Adaptive Tester, and an assertion evaluator. The program being tested is called the test object. The architecture of the software is shown in Figure 3.

Figure 1. A Complex in Three Dimensions

Figure 2. Complex Transformations
The test object is the program to be tested; it must contain executable assertions. Since they are useful throughout the entire software cycle, they should be included in the code as it is being written. This allows the correctness of the assertions to be validated as the code is tested. Since an existing program was used in the experiment, it was necessary to write assertions and then execute the program to be sure the assertions were correct.

Some assertions, such as range checks, are simple to write and do not require in-depth familiarity with the algorithm. For example, in a DO-loop with a variable as the upper bound, it is easy to write an assertion which specifies that the value of that variable is greater than zero. More difficult to write are assertions which check results of computations or that express a relationship between the variables. Since it is necessary to have a firm understanding of the program to write these assertions, it is generally best for the person who implements the code to write these assertions. The success of this testing technique depends on having a sufficient number of assertions expressing tight bounds on variables, thereby enabling them to detect errors.

Once the assertions have been written, the only other part of the testing process requiring human intervention is setting up the first testcase. The remaining part of the testing is automatic: the performance is evaluated and new testcases are generated until a given performance value is attained.

Figure 3. Software Architecture

To initiate the testing process, the tester must specify the following data:

- The input parameters to be altered by the search algorithm
- A set of initial values for each input parameter
- The range of values that each input parameter may assume, in other words, the constraints for each variable
- The maximum number of assertion violations expected

The function of the test driver is to read these values, initiate the testing process, and interface with the Adaptive Tester.

The function of the assertion evaluator is to maintain a test results file containing the following information:

- The value of each input parameter for every test
- The number of assertion violations and the number of different assertions violated
- The statement and module number where each violation occurred
- How many times each assertion was violated

The information in the test results file is used as input to the search algorithm so it can find values of the input variables which cause the maximum assertion violations. The search algorithm constructs a new test case using the operations described in the previous section and returns control to the test driver which executes the next test. At the conclusion of the tests, the test results are output in a final report.

EXPERIMENTS

Two experiments were performed to determine the usefulness of executable assertions in testing. The purpose of the first experiment was to determine if executable assertions could locate errors; and, if so, what the resulting error space looked like. The Adaptive Tester was not used in the first experiment; instead the values of the input parameters were methodically stepped-up in regular intervals across a "grid." The first experiment has been described elsewhere, but the results indicated that executable assertions were effective in detecting errors and that the error function did not include singularities.

The prominent research issues for the second experiment were as follows:

- Behavior of the error function - Does it confirm the results obtained in the first experiment?
- Applicable search techniques - Pending determination of the behavior of the error function, what search technique is the most effective in finding errors?
• Application to large input spaces - What happens when there is a large number of input parameters?

The second experiment was more comprehensive since it actually combined the Adaptive Tester with the use of executable assertions.

Since one purpose of this experiment was to provide corroborative evidence of the first experiment, a new test object was selected. The function of the new program was to input an orbit described by a set of eight parameters (orbital elements) and produce a state vector representation of a point on the orbit. Since this program had been in use for twelve years and had been the test object in another experiment, it was assumed to be error free. Yet, once the assertions were added to the program, they uncovered latent errors that were completely unsuspected! In most cases, these were errors that occurred only at boundary conditions.

In this second experiment, three modes of operation were implemented in the test driver:

Grid -

The values of the input parameters were varied in a uniform pattern, a grid, over the input space. The results from these grid tests were used as a baseline by which to evaluate the search technique.

Search -

Given one value for each of the tested inputs, the search algorithm constructed all subsequent test cases.

Grid and Search -

Instead of constructing the initial points on the complex from random test cases, a set of values for each of the inputs was derived for input to the search algorithm. These values were derived by sorting on the number of assertion violations obtained from the grid tests; the input values associated with the highest number of violations were passed to the search routine.

In each mode of operation, three variables were varied: MODE, VALUE, and the eccentricity of the orbit. To examine the effect of varying a large number of input parameters, additional tests were run in which all ten of the input parameters were perturbed. The Adaptive Tester was able to construct test cases and even found another assertion violation.

Error Seeding

Errors were generated for the test program using a procedure developed by Brooks. The method uses error types and frequencies from a previous study to randomly select a set of errors to be "seeded" in the program. Some types of errors were not chosen for the study, such as documentation, data definition, etc., because the experiment was specifically concerned with detecting run-time errors. The types of errors used were computational errors, logic errors, data handling errors, and interface errors. In generating errors for the experiment, statement types and other descriptive information about the test program were generated automatically using the Software Quality Laboratory. Each statement in the program was classified by type, and a table matching error categories to statement types was constructed. This resulted in a list of available error sites. Potential error sites were then randomly selected. Once the assertions were written and checked out, errors were introduced one at a time to determine how effective adaptive testing using assertions was in detecting errors.

For each error, a grid test was run. Then tests using the automated search technique were run to see if the results were the same. The search technique was used in two ways: first by perturbing three variables, MODE, VALUE, and one other variable; and then by perturbing all the variables of the orbit at one time. The search routine was allowed to run until it found a preset number of assertion violations (representing the performance value); then this number was automatically stepped up by one and the search algorithm tried to find another combination of input values which would cause a greater number of violations to occur. In this way, the performance value was maximized. The testing process was arbitrarily set to terminate when one hundred tests were run, but each test actually consisted of several subtests because the values of MODE and VALUE were varied within each test. The report that is produced at the conclusion of the runs is shown in Figure 4. In this test MODE, VALUE, and one other variable, ORBIT(6), the eccentricity, were varied.

Test Results

The results of the experiment demonstrated the effectiveness of the assertions in detecting errors. Of the original 24 errors, nine (38 percent) were detected by original assertions, and eight (33 percent) were detected by assertions that were added after the grid testing. There were seven errors that could not be detected by assertions. The reasons why these errors were not detected by assertions were:

• The seeded error was in a section of code that was only traversed after an error occurred.

• Assertions could not be written for some types of errors that were seeded. These included a misspelled variable name, a REAL variable declared as INTEGER, and the wrong number of arguments in a subroutine call.

• The FORTRAN compiler at this installation initializes all variables to zero, therefore a deleted DATA statement caused no problems.
<table>
<thead>
<tr>
<th>#RUN</th>
<th>INPUTI</th>
<th>#FALSE ASSERTION</th>
<th>#DIFFERENT ASSERTION</th>
<th>MODE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.7526</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2477545.659</td>
</tr>
<tr>
<td>9</td>
<td>0.6048</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>9649931.060</td>
</tr>
<tr>
<td>12</td>
<td>0.2700</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>13958923.49</td>
</tr>
<tr>
<td>13</td>
<td>0.900</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>24389119.03</td>
</tr>
<tr>
<td>24</td>
<td>0.2899</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8871067.739</td>
</tr>
<tr>
<td>25</td>
<td>0.3679</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1760571.330</td>
</tr>
<tr>
<td>30</td>
<td>0.2910</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>20758672.74</td>
</tr>
<tr>
<td>34</td>
<td>0.7346</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>22330022.80</td>
</tr>
<tr>
<td>35</td>
<td>0.1852</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>27015515.91</td>
</tr>
<tr>
<td>37</td>
<td>0.3555</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>19513234.41</td>
</tr>
<tr>
<td>44</td>
<td>0.6973</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4044234.171</td>
</tr>
<tr>
<td>43</td>
<td>0.6273</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>0.5851</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4533190.345</td>
</tr>
<tr>
<td>49</td>
<td>0.9000</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>0.7234</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4533190.345</td>
</tr>
<tr>
<td>53</td>
<td>0.9000</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3737662.000</td>
</tr>
<tr>
<td>55</td>
<td>0.7234</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7402021.345</td>
</tr>
<tr>
<td>63</td>
<td>0.7053</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>0.6261</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4533190.345</td>
</tr>
<tr>
<td>73</td>
<td>0.7474</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>0.6471</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4533190.345</td>
</tr>
<tr>
<td>83</td>
<td>0.8071</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>0.6769</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4533190.345</td>
</tr>
<tr>
<td>86</td>
<td>0.1774</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>23124989.06</td>
</tr>
<tr>
<td>87</td>
<td>0.9000</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1415222.264</td>
</tr>
<tr>
<td>89</td>
<td>0.7228</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5588024.749</td>
</tr>
<tr>
<td>90</td>
<td>0.9000</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1939816.571</td>
</tr>
<tr>
<td>91</td>
<td>0.1557</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6107643.223</td>
</tr>
<tr>
<td>92</td>
<td>0.5503</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>9951144.293</td>
</tr>
<tr>
<td>93</td>
<td>0.5530</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8029393.758</td>
</tr>
<tr>
<td>94</td>
<td>0.4955</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>11674092.79</td>
</tr>
<tr>
<td>95</td>
<td>0.8433</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>18860857.79</td>
</tr>
<tr>
<td>96</td>
<td>0.5648</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>11113683.73</td>
</tr>
<tr>
<td>97</td>
<td>0.7040</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>14998170.76</td>
</tr>
<tr>
<td>98</td>
<td>0.6108</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>17228371.99</td>
</tr>
<tr>
<td>99</td>
<td>0.2554</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3876077.474</td>
</tr>
<tr>
<td>100</td>
<td>0.5485</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>11161715.66</td>
</tr>
<tr>
<td>101</td>
<td>0.4019</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>7518896.567</td>
</tr>
<tr>
<td>102</td>
<td>0.1000</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>7331062.939</td>
</tr>
</tbody>
</table>

**INPUTI** = ORBIT(6)

**MODULE** | **STMT#** | **TYPE** | **FAILURES**
---|---|---|---
ORBP | 109 | ASSERT | 34
OUTCHK | 142 | ASSERT | 38

* HOW MANY RUNS EACH ASSERTION FAILED IN **102** RUNS

Figure 4. Summary of Search Testing for Error 3
ACkNOWLEDGEMENT

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under Contract F49620-79-C-0115. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

REFERENCES


Figure 5. Efficiency of Search Method

Some of the errors that assertions could not detect would have been detected by static-analysis tools which test the consistent use of variables.

For all but four of the errors, the search methods detected the same errors as the grid tests; but they were able to do so much more efficiently and used much less computer time. Figure 5 shows the efficiency of the search technique when all variables are varied; it plots the number of the test in which the first assertion violation was detected versus the error number. Fifteen of the seventeen errors were detected within the first seven tests devised by the search technique. In contrast, the grid technique was run for 317 tests and discovered all but one of the errors; but 68 tests had to be run to detect error number 11.

CONCLUSION

One of the themes emphasized at recent conferences is that new methods for system development and testing are necessary. The need to make software less labor intensive must result in new automated programming tools.

The results from this experiment indicate that this automated testing technique has the potential for finding errors (logic, computational, etc.) that are difficult to find in other ways. In addition, the search algorithm eliminates the subjectiveness in constructing testcases and increases the variety of input values. By automating the testing process, the cost of testing can be reduced dramatically.

Combining executable assertions with adaptive search has resulted in a tool which allows more automation of the software development process and a more accurate testing environment by which to provide software reliability.


*Dorothy M. Andrews is now at Xerox Corporation, Palo Alto, California.