1975

Simulation of Error Detection in Computer Programs

Schneidewind, Norman F.; Green, Thomas F.

SIMULATION OF ERROR DETECTION IN COMPUTER PROGRAMS*

Norman F. Schneidewind and Thomas F. Green
Naval Postgraduate School
Monterey, California 93940

ABSTRACT

The relationship between computer program complexity and error detection capability is investigated by representing a program as a directed graph and simulating the detection and correction of errors. Variables of interest are test coverage, number of inputs, residual errors, execution time, correction time and node-arc-loop relationships. One application is in software design where the information provided by the model would be used to select program structures which are easy to test. A second application is in software testing where test strategies and allocation of test effort would be based on error detection and complexity considerations.

INTRODUCTION

It is generally accepted that computer programs with complex structure are harder to debug and test and more errors persist after release than for programs with more simple structure. By structure we refer to the topological arrangement of the parts of the programs. Here we develop a simulation model** to investigate the relationship of program structure complexity to error detection and test effort. Since structure can be controlled during the design phase and measured through all phases of a computer project, the study of the relationship between structure and error characteristics is valuable to the manager of a software project. Complex program structures with poor error characteristics should be avoided. In cases where complex program structures may be necessary to help meet program size or speed limitations, it is useful to have an indication of the additional testing which may be caused by complex structures. It is also useful to be able to compare the error characteristics of design alternatives that have different program structure (1).

Program structure complexity may be described by characteristics such as program size, incidence of branch instructions, incidence of loops, incidence of subroutine calls and variety of instructions. In addition to the complexity of the structure, the error characteristics of a program are a major determinant of the ability to detect errors. Error characteristics refer to the number and locations of errors in the program. Errors can be caused by incorrect design, incorrect implementation of design logic or clerical mistakes in programming. Finally, input characteristics--the sequence of inputs and the particular threads of the program which are traversed by the inputs--will have an effect on the number of errors detected.

Most computer programs have a large number of potential inputs; each may exercise the program in a different way. The sequence of instructions of the program that results from a particular input is called the path or thread associated with that input. Testing by submitting inputs to the program checks only the paths associated with those inputs. For programs with a very large number of inputs, testing can only be a relatively small sampling of all possible inputs (1).

TESTING AND ERROR DETECTION

In many moderate and large computer projects, a programmer writes and debugs a module and then gives it to a test group. The test group tests the module, integrates it with other modules and then continues testing. The module is tested by supplying an input to the module and then comparing the outcome to the known correct outcome. If there is a mismatch between observed and correct output, an error has been detected. When an error is detected the module is given to a programmer who locates and corrects the error (i.e. debugs the module) and then returns the module to the test group. Notice the distinction between testing, which is supplying inputs and observing outputs, and debugging, which is the highly individualized program analysis needed to locate and correct errors. In debugging, the programmer needs a detailed knowledge of the structure and operation of the module. The tester is frequently unaware of module structure and operation; his primary interest is the function of the module (1).

* This work was supported by a contract from the Naval Air Development Center, Warminster, PA.

** The suggestion to use a simulation model to study software error detection was given to the authors by Dr. Samuel Litwin, a consultant to the Naval Air Development Center.

SIMULATION OF ERROR DETECTION

It is very difficult to do experimentation with program structure in actual software projects because the cost of duplicate implementations of the same application is very high for all but small projects. For this reason analysis is performed on a model. Structure may be modeled as a set of nodes and arcs as shown in Figure 1. In the directed graph, nodes represent connection points where parts of the program may merge and/or branch and arcs represent a sequence of nonbranching instructions such as computation and input/output. Instructions are located in arcs and errors are located in some of the instructions. An input defines a path from the start node to the exit node. The arcs of a path are selected randomly at each node. Beginning at the start node an input causes execution of the instructions on its path, consuming test time, until an error is encountered. After the error is thus detected, it is corrected, consuming correction time; there is some risk that the correction will introduce a new error in some instruction. Then restarting at the initial node execution is begun again with the same input. This process is repeated until there are no errors on the path (1).

Measures of program complexity which are applicable to directed graph representations of program structure are: ratio of actual arcs to maximum arcs; ratio of nodes to arcs; and ratio of loop arcs to total arcs.

A simulation program for analyzing error detection and program structure has been programmed in FORTRAN IV and was developed using the Naval Postgraduate School IBM 360/67 operating under OS/MVT. The program is approximately 600 statements in length and consists of the following main program and subroutines:

Main Program

- Simulate Error Detection

Subroutines

- Construct Directed Graph
- Add an Arc Between Nodes
- Seed the Graph with Errors
- Insert New Errors
- Display the Graph by Using Plotter

The program requires 158K bytes of main memory and executes in approximately 1.5 minutes of CPU time. The program was written for a Master of Computer Science Thesis at the Naval Postgraduate School (2).

By representing a program as a directed graph, it is possible to simulate the execution of a program and the detection and correction of program errors. Prior to the execution of the simulation, the directed graph is established as follows:

- The input to the simulation is the definition of the directed graph in terms of nodes, arcs and loops.
- The graph is represented by an adjacency matrix.
- Arc lengths are established by randomly selecting the number of instructions between nodes.
- Programming errors are randomly seeded among the instructions in the arcs.
- The number of iterations per loop is randomly selected.

During the simulation execution, path traversal, instruction execution time, error correction time and new error insertion (as the result of correcting an error) are simulated. The probability distributions which are used to establish the graph and simulate events are listed below.

Property or Event	Probability Distribution
Instructions per arc	Exponential
Instruction execution time	Exponential
Original error occurrence	Exponential
Time to correct an error	Exponential
Iterations per loop	Uniform
Number of instructions affected by correction	Uniform
New error occurrence	Uniform
Arc selected for new error insertion	Uniform
Arc selected at branch point	Uniform

Since little is known about the type of probability distribution which is associated with the above program properties and execution events, the selection of type of distribution was, of necessity, based on assumptions. However, it is felt that the assumptions are reasonable. For example, the seeding of original errors is based on the number of instructions between errors being exponentially distributed or, equivalently, the presence of an error is independent of the presence of any other error. A second example is that instructions are placed in arcs according to an exponential distribution, or equivalently, the number of instructions between branch points is exponentially distributed. This implies that the number of instructions between two branch points is independent of the number of instructions between other branch points. Although the choice of distribution may have significant effect on the simulation results for a given structure, the objective is to evaluate results on a relative basis across the various structures so that choice of distribution is not critical. A sensitivity analysis could be made of the effect on error detection of the choice of distribution and distribution parameters.

When an input reaches an exit node, several types of data are printed:

- complexity data, e.g. ratio of nodes to arcs

- number of remaining errors in the program and their locations
- percent remaining errors
- total execution time
- total error correction time
- percent arcs traversed
- average arc traversal time
- number of instructions
- number of original errors.

The above procedure is repeated for as many inputs as desired. In addition, various program structures can be defined and simulated.

APPLICATION OF SIMULATION MODEL

The model can be used to influence the software design decisions on program structure by making it possible to compare the error detection characteristics of design alternatives. Since error detection characteristics are good indicators of the time and resources consumed by testing, it is valuable to relate design decisions to testability. The program structure can be converted to a directed graph; the model is then seeded with errors and subjected to random inputs.

The model can also be used to identify the measure or measures of complexity that best predict the ability to detect errors. To do this, it is necessary to estimate the error detection characteristics of a variety of different structures. The complexity of actual programs would be determined and the estimates of error detection characteristics would be compared.

It is also of interest to develop a relative ranking of the difficulty of detecting a given number of errors in various program structures. This information can be obtained from the model by seeding a specified number of errors and replicating the simulation a large number of times for various structures. The ranking obtained would be used during the design process as a guide for selecting structures which are easy to test and during testing as a guide for allocating test resources on the basis of anticipated difficulty of testing.

A difficult facet of program testing involves the selection of inputs. We wish to choose inputs so that a high percentage of the critical parts of the program will be exposed to testing. However, this objective must be weighed against the cost of machine time for debugging and the cost of programming personnel for error correction. It is infeasible to subject a program to all possible input combinations. Various software packages are available for recording and analyzing the following types of data: count and frequency distribution of types of instructions executed; indication of code which is not executed; and indication of code which is impossible to reach (3). Although this type of instrumentation is helpful for tracing program behavior, once a set of inputs is selected, it does not solve the problem of selecting the number and type of inputs in the first place. Thus, another use of the model is to examine the relationships between number of inputs and paths traversed, for a given program structure, and the number of remaining errors, fraction of program exposed to testing, execution time and correction time. In addition, it is of interest to determine the number of inputs required to achieve a specified number of remaining errors for various structures, when the same number of original errors is used for each structure.

ANALYSIS OF SELECTED SIMULATION RESULTS

Typical output from 1) submitting a sequence of inputs to a single program, and 2) submitting three inputs to a sequence of increasingly complex programs is described and discussed. The former case is illustrated by a program with 30 nodes, 40 arcs and 435 instructions. Errors were initially placed on 30 instructions. A sequence of 30 inputs (i.e. 30 paths through the program determined by random numbers) were submitted to the program. Recall that each input detects all errors on its path; each error is corrected but there is some probability (in this example it was small) that new errors are introduced by correcting detected errors. For each input the number of errors corrected, the number of new arcs tested, execution time (related to computer costs) and correction time (related to personnel costs) are shown in Figure 2.

Figure 2 illustrates the action of the model. Initially there are many errors detected with each input; then the number of errors detected drops off as errors are corrected. Note that many paths do not traverse any new arcs. Although the paths through the program are, in general, different from previous paths, these paths may involve only arcs that have been previously traversed. In the testing of actual software, after an initial burst of errors there are often long periods with no error detection followed by a new group of errors. The new group of errors results from testing previously untested parts of the program. The error detection model captures this phenomenon; for example, see Figure 2.

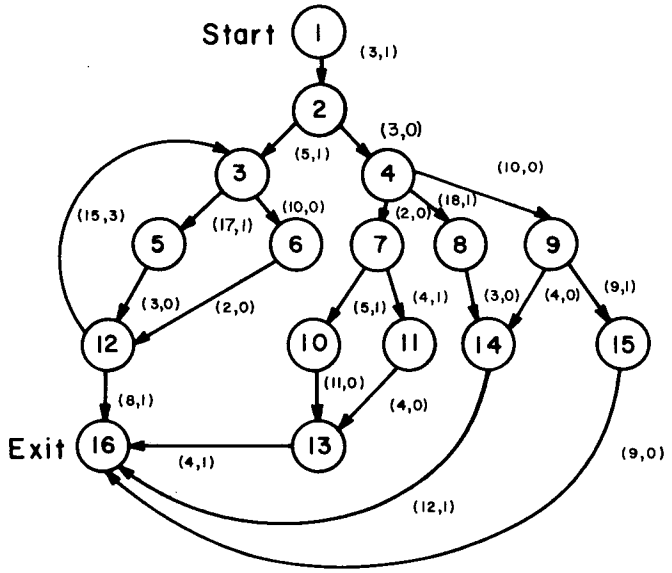
Figure 3 shows the results of running three successive inputs into 20 programs of increasing complexity. The initial program had 30 nodes and 40 arcs; the complexity was increased by adding an arc to each successive program. For each program the number of instructions and the number and location of errors were different. Because the number of errors and arcs varied, the results in Figure 3 are normalized. One would expect that the number of errors detected would decrease as the number of arcs increased; however, since the number of errors changed it was not possible to draw firm conclusions from only 20 runs (1).

BIBLIOGRAPHY

1. Bradley, Gordon H., Green, Thomas F., Howard, Gilbert T. and Schneidewind, Norman F., "Structure and Error Detection in Computer Software," Proceedings of the American Institute of Industrial Engineers National Meeting, May 21, 1975, Washington, D.C., 6 pages.

2. Green, Thomas F., "Software Error Detection Model," Master of Science in Computer Science Thesis, Naval Postgraduate School, Monterey, Monterey, California, June 1975.
3. Stucki, L. A., "Automatic Generation of Self-Metric Software," Record of 1973 IEEE Symposium on Computer Software Reliability, New York, New York, April 30-May 2, 1973, pages 94-100.

13 Errors originally; Number of instructions is 161.
(,) indicates number of instructions, followed by number of errors in arc. E Signifies an error found and NE Signifies a new error inserted in designated arc.



Input 1 Traversals: 1→2, E; 1→2→3,E; 1→2→3→6→12→3, E;
NE, 9→14; 1→2→3→6→12→3, E; NE,9→14; 1→2→3→6→12→3, E;
NE, 9→14; 1→2→3→6→12→3→6→12→16, E; 1→2→3→6→12→16.
Execution time = 6.59 hrs, Repair time = 395.67 mins.
6 Errors found, 3 Errors inserted, 10 Errors remain.
Input 2 Traversals: 1→2→3→6→12→3→5, E;
1→2→3→6→12→3→5→12→3→6→12→3→5→12→16.
Execution time = .061 hrs, Repair time = 3.65 mins.
1 Error found, 9 Errors remain.
Input 3 Traversals: 1→2→3→5→12→16.
Execution time ≈ 0 hrs, Repair time = 0 mins.
0 Errors found, 9 Errors remain.
Figure 1 -- Simulation Example (3 Inputs Used)

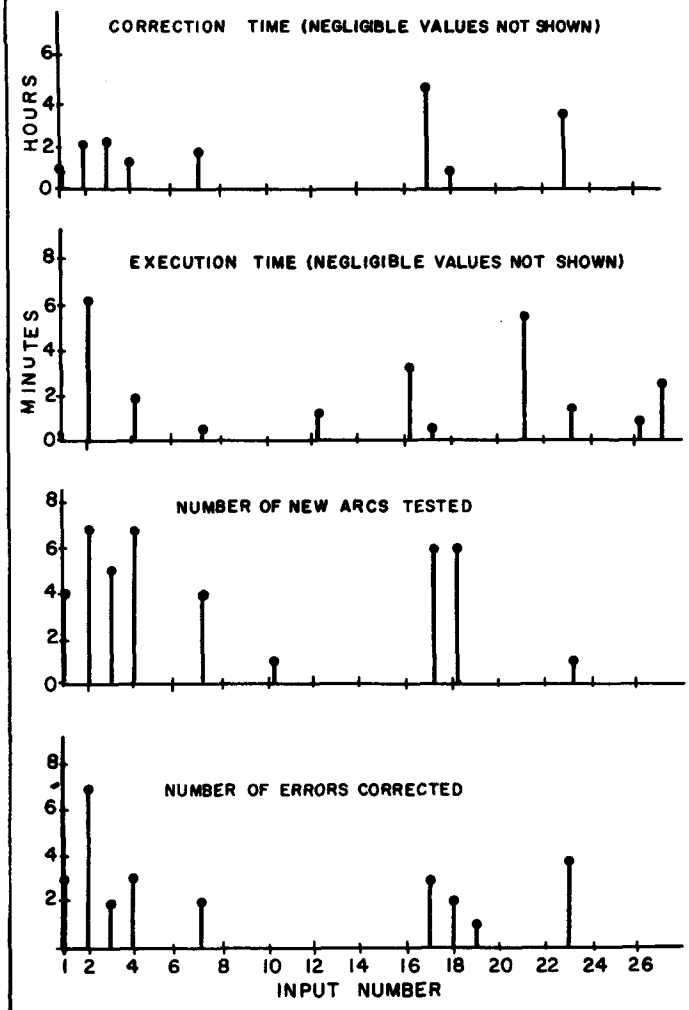


Figure 2. Simulation Results Obtained by Varying Inputs and Keeping Complexity Constant

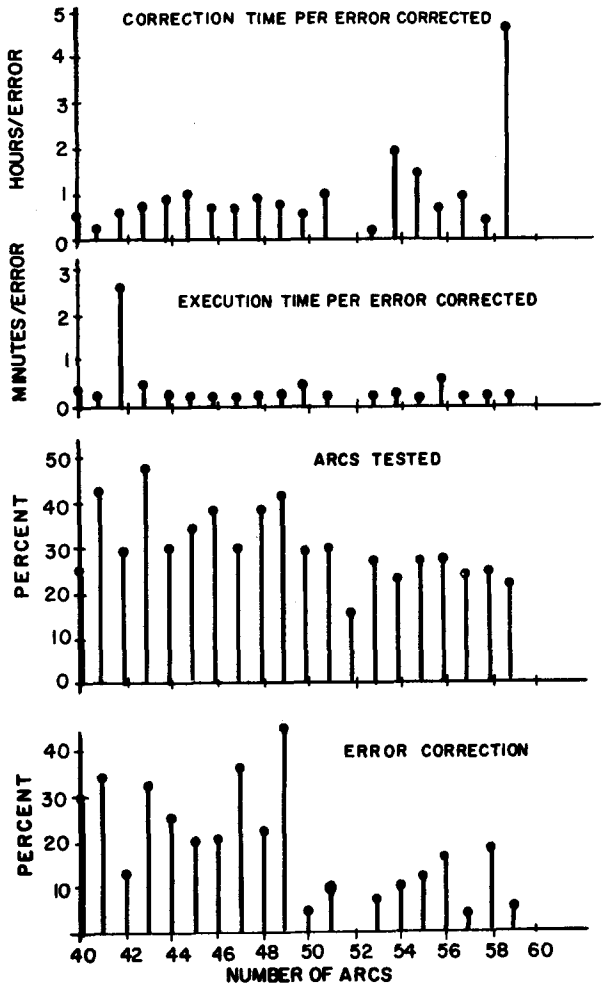


Figure 3. Simulation Results Obtained by Varying Complexity and Using Same Inputs

NORMAN F. SCHNEIDEWIND

Professor of Information Science at the Naval Postgraduate School since 1971. Teaches courses in information systems, computer science and operations research. Current research activities involve software reliability and the modelling and evaluation of computer systems. Prior to that time he held management and technical positions in computer systems at the System Development Corporation, Planning Research Corporation, Computer Usage Company and UNIVAC. He holds degrees in electrical engineering, operations research and management from the University of California (Berkeley) and the University of Southern California.

THOMAS F. GREEN

Lt. Thomas Green graduated from Kansas University in 1967 with a BA degree in Mathematics and will graduate from the Naval Postgraduate School in 1975 with an MS degree in Computer Science. Starting as an enlisted electronics technician in submarines, he has served in destroyers since being commissioned in 1967.