The Use of Simulation in the Evaluation of Software

Schneidewind, Norman F.

http://hdl.handle.net/10945/45153

Downloaded from NPS Archive: Calhoun
The Use of Simulation in the Evaluation of Software

N. F. Schneidewind
Naval Postgraduate School

Introduction

Two particularly nagging ailments faced by the computer industry today are the high cost and late delivery of software. The symptoms usually surface during software debugging, testing, and integration; but the ailments themselves can most often be traced back to the program design phase and the structural characteristics of the program. The significance of program structural characteristics has been recognized for some time, as witnessed by the emergence of structured programming, a methodology that sets out to (1) reduce programming errors; (2) design an understandable, readable, and therefore maintainable program; (3) increase our ability to detect errors; and (4) prove, if only informally, that the program is correct. But there is another tool available that has usually been overlooked in the software development process: simulation.

Simulation is not new to hardware and operating system design performance modeling; it is, however, relatively new to the evaluation and measurement of software—even though examples abound of simulation and analytical models that have been developed for modeling software error detection. This paper attempts to show how simulation can be used both to evaluate alternatives during design and to simulate the detection of errors during testing.

To improve program quality we must not only avoid errors during program design; we must also detect them during testing. Hence, one of the characteristics of a good design is a program structure that allows easy error detection.

A convenient way of describing program structure and simulating the detection of errors is to represent the program in a directed graph. As shown in Figure 1, nodes represent branch points (point A) or points where instruction sequences merge (points B and C). Arcs (BC) are the segments of a program between branch and merge points. The length of an arc is equal to the number of source statements in the segment. A path is a series of connected arcs that begins at start node S and terminates at terminal node T, such as the path SDBCEFT in Figure 1. Repetition of statement execution is implemented with cycles in the directed graph (cycle ABA).

By using a directed graph to represent the structure of a program and simulation to study program error detection, the following information can be obtained:

1. Error detection (number or fraction of errors detected) as a function of a program's structural characteristics, for a given number of tests. The test consists of beginning simulated program execution at the start node, detecting and correcting any errors, restarting at the start node, and repeating this process until a terminal node is reached.

2. Error detection as a function of number of tests for given structural characteristics.

Structural characteristics correspond to program characteristics. For example, numbers of nodes, arcs, paths and source statements correspond to branching and merging, arithmetic and data transfer operations, execution sequences, and size.

Program complexity and error characteristics

The structural characteristics of the directed graph indicates program complexity—i.e., the difficulty of error avoidance and detection during program design and testing. Program structures with many nodes and paths are difficult to write without committing programming errors. Complex structures are also difficult to test because the number of paths to test is large, the probability of traversing certain paths is small, and the probability of undetected errors is high.

Error characteristics are the numbers and locations of program errors and the probability of their being detected in the directed graph model. The structure of a real program is known but its error characteristics are not
known before testing. Therefore, the simulation model must be able to randomly seed errors in the directed graph. This feature allows the quantity and location of errors to be varied by repeating simulated error detection over a large number of error placements.

Error simulation model

The user of the model must specify the directed graph to be used in the error simulation. For example, the following inputs and outputs must be identified:

Inputs.
1. Number of nodes.
2. Nodal connections (arcs).
3. Arc length (number of source statements).
4. Either mean number of statements between errors (if error seeding is used) or numbers and locations (arcs) of errors (if user-specified error placement is used).
5. Number of tests.
6. Number of error seedings (repetitions) for each test.
7. Number of inputs (replications) for each seeding, where an input corresponds to a particular path traversal from the start node to a terminal node.

Outputs.
1. Mean number of errors detected per test computed over all seedings and inputs.
2. Standard deviation of errors detected per test computed over all seedings and inputs.
3. Number of times each arc is traversed.

Optional outputs.
1. Matrix of original errors as seeded.
2. Matrix of errors detected and path traversed, displayed after each input.

Error seeding.
The error seeding is accomplished as follows:
1. Draw a uniformly distributed random number between 0 and 1 from a random number generator.
2. Use this number to compute the value of a random variable with a mean of 1.0 from an error-seeding probability distribution function, which in this case is exponential.
3. Multiply the value of the random variable, with mean of 1.0, by the mean number of source statements between errors. The product is the number of statements to count from the last error, or, in the case of the first error placement, from the start node. The mean of the distribution can be obtained from historical program error data or it can be a hypothetical value specified by the model user. Error seeding continues in this manner until the terminal node is reached. For a large number of seedings, the number of statements between errors will have a distribution and mean that approximate the error-seeding probability function.

Since the number of statements between errors is exponentially distributed, they correspond to a Poisson distribution of the number of errors in a given interval of source statements. As a consequence, the intervals between errors are independent. Furthermore, the errors that occur in disjoint intervals are independent and are proportional in number to interval length.

The exponential distribution was chosen because of its memoryless property: the occurrence of an error in a program is independent of errors previously occurring in the program. Stated another way, the commission of an error by a programmer is viewed as an independent event unrelated to previous possible errors. This isn’t always so; for example, one can learn from past programming mistakes, or a new error can occur as a result of correcting a previous error. However, random error seeding is used in the usual situation where there is no knowledge of the error characteristics in a set of programs and the model user must account for variability in error placement by repeating the simulated error detection over a large number of seedings.

In contrast, the model user may wish to study error detection for known or assumed error locations, or for situations in which errors in different locations are related. For these cases, the error placement is specified in the simulation input and the errors are planted in the specified arcs rather than being randomly seeded.

Path selection. For many programs, the large number of possible paths precludes testing every path. Even if every path could be tested, it is not feasible to test all paths with all combinations of input data. Therefore, a representative sample of inputs is selected. In debugging, the inputs are usually well defined because the programmer traces specific paths. In contrast, functional testing involves exposing a program to a variety of inputs. Since the number of input combinations may be enormous, the tester may use random samples of inputs to subject the program to a representative sample of inputs and path traversals. This consideration leads us to use random path selection in the model.

Path traversals are simulated by randomly selecting an outgoing arc at each node. Outgoing arcs have uniform probability of selection. Since information about program branch probabilities is not usually available, there is no basis for selecting any other distribution. In some cases it may be possible to estimate branch probabilities for programs based on branch data dependencies. In these instances it is necessary to provide an option in the model for specifying branch probabilities.

Error detection. When an arc is traversed (DB in Figure 1, for example), all errors in the arc are found and the error count is updated. Then another arc (BC, for example) is traversed. Eventually all errors in a path (SDBCEFT, for example) are found and the next test is performed. Each succeeding test starts with the errors present at the beginning of the previous test minus the errors found during that test. In general, a different path (input) will be traversed on each test. For example, in Figure 1 path SDBABCEFT may be used on the second test. Although the procedure is implemented differently in the simulation, it is equivalent to the test sequence of testing, finding an error, correcting the error, retesting with the same input, and retracing the path to the point where the error was found. This process is continued for a specified number of tests.

To account for variations in path traversal that are caused by different inputs, one may replicate path selection for the various tests by changing the random number seed of the random number generator that governs arc selection. Also, to account for variations in error detection caused by different error placements, error seedings may be repeated for each set of replications.

Analytical model

In addition to the simulation model, an analytical model of error detection has been developed. In this model checks the validity of the simulation model. The analytical model may require less CPU time and storage.
space for some directed graphs. It computes the expected number of detected errors for each test. Designating the original expected number of errors in arc \(ij\) as \(\mu_{ij}\), and \(R_{ij}\) as the probability of traversing arc \(ij\), the expected number of detected errors for the first test is \(\mu_{ij}P_{ij}\). The expected number of errors remaining in arc \(ij\) after the first test is \(\mu_{ij}(1 - P_{ij})\). The expected number of errors detected on the second test is \(\mu_{ij}(1 - P_{ij})R_{ij}\). The expected number of errors detected on the \(k\)th test is

\[
e(k) = \mu_{ij}(1 - P_{ij})^{k-1}P_{ij}.
\]

When the expected number of detected errors is added over all arcs, we have

\[
E(k) = \sum_{ij} \mu_{ij}(1 - P_{ij})^{k-1}P_{ij}
\]

for the total expected number of detected errors on the \(k\)th test. The initial \(\mu_{ij}\) can be interpreted as the mean number of errors per arc originally present, in which case the \(\mu_{ij} = \overline{\mu}_{ij}\) are equal for all arcs, or as specified number of errors in each arc. In the latter case, \(\mu_{ij}\) will be different. When comparing simulation and analytical results, \(\overline{\mu}_{ij}\) is used in (1) if the simulation is repeated for a number of seedings and \(\mu_{ij}\) is used in (1) if only one seeding is used.

The calculation of the arc traversal probabilities \(P_{ij}\) is easy when there are no cycles (repeated arcs). The probability of traversing arc \(ij\) is computed by multiplying the probability of reaching node \(i\) by the branch probability for arc \(ij\), \(1/m\), where \(m\) is the number of arcs emanating from node \(i\). The probability of reaching node \(i\) is the sum of the traversal probabilities of arcs that enter node \(i\). For example, the probability of traversing arc 8-16 in Figure 2 is equal to the probability of reaching node 8 (0.25) multiplied by the branch probability of arc 8-16 (0.50) or 0.125. When cycles are present, as shown in Figure 2, for arcs 3-5, and 5-3, the calculation of \(P_{ij}\) is more complicated. For example, the probability of reaching arc 3-6 by first cycling in arcs 3-5 and 5-3 is as follows:

\[
(1/4)(1/3)(1/2), \text{one cycle}
\]
\[
(1/4)(1/3)(1/2)(1/3)(1/2), \text{two cycles}
\]
\[
(1/4)(1/3)(1/2)(1/3)(1/2)(1/3)(1/2), \text{three cycles}
\]

Thus, the probability of reaching arc 3-6 by cycling is

\[
(1/4) \sum_{n=1}^{\infty} (1/3 \times 1/2)^n = 0.05
\]

The probability of traversing arc 3-6 is then the sum of the probability of traversing it directly and the probability of reaching it by cycling \((0.25 + 0.05 = 0.30)\). Arcs 5-9 and 5-10 can be traversed directly by way of arc 3-5 or by cycling in arcs 3-5 and 5-3. By a similar analysis, their traversal probability is 0.10. Since inputs to arcs 6-11 and 6-12 divide equally, their traversal probability is one-half the traversal probability of arc 3-6 or 0.15. All other arcs in Figure 2 are not affected by the cycle 3-5, 5-3.

When an actual program is available for analysis, the number of source statements in an arc \(s_{ij}\) and the mean number of source statements between errors \(M\) (obtained from historical module error records) can be used to obtain the \(\mu_{ij}\) in (2) by the computation \(\mu_{ij} = s_{ij}/M\). Then (2) becomes

\[
E(k) = \sum_{ij} s_{ij}(1 - P_{ij})^{k-1}P_{ij}\frac{1}{M}
\]

If we want to calculate (3) as the fraction of original errors detected, then we divide (3) by \(U\), the expected number of original errors in a program. We obtain

April 1977
$U = S/M$, where $S$ is the total number of statements in the program. If the program is a procedure of a larger module, $S$ is the number of statements in the procedure and $M$ is the mean number of errors between statements of the module. Thus, when calculated on a fractional basis, (3) becomes

$$E(k)/U = \left(\sum_{ij} s_{ij}(1 - p_{ij})^{k-1} p_{ij}\right)/S \quad (4)$$

It is important to use (4) when comparing detected errors from programs of different size, since more errors would be expected in larger programs.14

A computer program has been written for the analytical model for computing traversal probabilities and the expected number of detected errors.15

### Comparison of simulation and analytical models

A potential advantage of the simulation over the analytical model is its computation of the variance. The variance computation is particularly important in that a model user wants variability information in addition to expected values, since the locations of errors are not known in advance of testing. In addition, input path information may be unknown or incompletely specified. Thus it is important to account for the variability in error detection contributed by different error placements and paths. At one time, only the simulation model could calculate the variance. However, the variance has been obtained analytically,15 partially negating an advantage of the simulation model. The simulation model still has the advantage of permitting the study of specific path traversals and error-detection histories. The advantage of the analytical model is computational speed, particularly for small graphs where the hand-held calculator can be used. The simulation of large numbers of repetitions and replications requires several minutes of CPU time on an IBM 360/67, even for small graphs. (The analytical model CPU time and storage requirements have not been completely evaluated.) In any case, both models are required so that one model can be used to check the accuracy of the other.

### Validation tests

Comparisons were made between simulation and analytical results for selected structures. Three of the comparisons were conducted as follows:

1. The structure shown in Figure 2 was used without arc 5-3. Nine errors were randomly seeded. The number of errors in each arc is shown in Table 1.

2. The structure shown in Figure 2 was used with arc 5-3. Nine errors were randomly seeded. The number of errors in each arc is shown in Table 2.

3. The structure shown in Figure 2 was used with arc 5-3. Eighteen errors were randomly seeded. The number of errors in each arc is shown in Table 3.

In each case one error seeding and 999 replications of inputs for 10 tests were used in the simulation program. The tables show arc identification, arc traversal probability, errors seeded, test number, and detected errors for the simulation and analytical models. The simulation results were obtained by computing the mean number of detected errors over 999 replications for each of 10 tests.
Table 4. NTDS error detection (Module 2, Procedure 48): 100 Replications, 100 Repetitions

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>$p_{ij}$</th>
<th>$p_{ij}$</th>
<th>$s_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1.0000</td>
<td>1.0000</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.4934</td>
<td>0.5000</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.5036</td>
<td>0.5000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.2444</td>
<td>0.2500</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.2490</td>
<td>0.2500</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0.1229</td>
<td>0.1250</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>0.1215</td>
<td>0.1250</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0.1869</td>
<td>0.1875</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0.1850</td>
<td>0.1875</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0.1869</td>
<td>0.1875</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>0.3719</td>
<td>0.3750</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>0.3719</td>
<td>0.3750</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>0.3719</td>
<td>0.3750</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0.3719</td>
<td>0.3750</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>0.3719</td>
<td>0.3750</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>0.0884</td>
<td>0.0938</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>0.0979</td>
<td>0.0938</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>0.3719</td>
<td>0.3750</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>0.3719</td>
<td>0.3750</td>
<td>0</td>
</tr>
</tbody>
</table>

Analytical Simulation

\[
E(1) = 0.1550, \quad E(1)/U = 0.4651
\]

The analytical results are expected values obtained by using (1) and (2). The $\mu_{ij}$ used in the analytical calculations of (1) and (2) were the errors seeded by the simulation. Figure 3 shows error detection as a function of number of tests for the simulation results of Tables 2 and 3.

Comparisons of simulation and analytical results are shown for two Naval Tactical Data Systems procedures in Table 5. The directed graphs for these data are Figures 4 and 5, respectively. The NTDS programs are used primarily for shipboard tracking of air targets. The system consists of large modules (2000 statements in some cases), each of which performs a function, such as tracking or display. Each module is divided into many procedures varying in length from approximately 10 to 100 source statements.

In Tables 4 and 5, the primed values refer to simulation results; unprimed values refer to analytical results. The errors detected were obtained for one test; $k = 1$ was used in (3) and (4) to obtain expected number of errors detected and expected fraction of errors detected, respectively. Other experiments have shown similar agreement between simulation and analytical results. The agreement does not prove the validity of the model as an accurate representation of error detection in computer programs; rather, the results only indicate that we have correctly implemented our concept of the model. Planned future work will compare model results with error detection and structure data provided by the NTDS programs.

Applications

The simulation and analytical models of error detection described here will help the program designer select program structures that have good error detection characteristics—i.e., high degree of error detection compared with the test effort expended. By comparing the relative error detection of proposed designs before testing, one can obtain an indication of the probable difficulty of error detection during the test phase. Although error detection is only one of many considerations in program design, the high cost of the test and integration phase of software development makes it an increasingly important one.

The test manager can use these models to obtain a relative ranking of the difficulty of testing new programs. He can use the models of the programs to be tested to obtain their relative error detection capabilities over some specified number of tests. Also, the rate at which error detection decreases asymptotically with the number of tests, as illustrated in Figure 3, can provide the test manager with one criterion for terminating tests.

![Graph showing detected errors for tree structure with cycle (simulation).]

April 1977
The model does have some limitations in its application to large programs. For example, the CPU time and storage space grow rapidly as the size of the program to be represented increases. Also, it is necessary to use a program that automatically converts source language programs or flowcharts to the directed graph input format required by the models, because manual conversion is tedious. With the increased use of modular programming, program size should not be a limitation to using these models. A problem in analyzing NTDS modules was the large module size. Only by analyzing procedures within modules could we feasibly use the models.

**NTDS program analysis.** We analyzed 44 NTDS procedures of the type shown in Figures 4 and 5 (31 procedures of Module 1 and 13 procedures of Module 2) to determine whether certain structural characteristics, (such as numbers of paths) correlate well with error detection. Remembering that correlation does not indicate a cause and effect relationship but only the degree of linear association between variables, we were interested in determining the degree of linearity for the NTDS programs. The following variables were used in the correlation analysis:

\[ N_n: \] Number of nodes, where dummy nodes (nodes inserted to implement parallel arcs) and transient nodes (nodes associated with called procedures) were eliminated.

**Figure 4.** Directed graph of NTDS procedure (Module 2, Procedure 48).

**Figure 5.** Directed graph of NTDS procedure (Module 2, Procedure 122).
Table 6. Correlation coefficients (44 NTDS procedures, 100 replications, 100 repetitions)

<table>
<thead>
<tr>
<th>E(1)/U</th>
<th>% Detection, 1 Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nn (nodes)</td>
<td>-.46</td>
</tr>
<tr>
<td>Na (arcs)</td>
<td>-.53</td>
</tr>
<tr>
<td>Np (paths)</td>
<td>-.55</td>
</tr>
<tr>
<td>S (source statements)</td>
<td>-.47</td>
</tr>
</tbody>
</table>

Table 7. Correlation coefficients after log_{10} transformation (44 NTDS procedures, 100 replications, 100 repetitions)

<table>
<thead>
<tr>
<th>E(1)/U</th>
<th>% Detection, 1 Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nn (nodes)</td>
<td>-.44</td>
</tr>
<tr>
<td>Na (arcs)</td>
<td>-.50</td>
</tr>
<tr>
<td>Np (paths)</td>
<td>-.65</td>
</tr>
<tr>
<td>S (source statements)</td>
<td>-.53</td>
</tr>
</tbody>
</table>

Na: Number of arcs, where transient arcs (arcs associated with called procedures) and arcs associated with dummy nodes were eliminated.

Np: Number of paths.

S: Number of source statements.

E(1)/U: Fraction or percentage errors detected for one test.

We modified Nn and Na to eliminate nodes and arcs that were not part of the procedures. The correlation coefficients are shown in Table 6. Although correlation exists, no coefficient is high and no coefficient is significantly higher than the others. In order to test the strength of a nonlinear relationship of the form \( y = ax^b \), the variables were subjected to a \( \log_{10} \) transformation before calculating the correlation coefficients. This provides the degree of linearity between \( x \) and \( y \) after the logarithmic transformation has been applied. The results are shown in Table 7. There is no significant difference between the data of Tables 6 and 7. These results indicate that, for NTDS procedures, no single structural characteristic dominates error detection, as represented by the model. Work is proceeding on relating multiple structural characteristics to error detection. Programs produced by structured programming will be similarly analyzed.

N.F. Schneidewind is professor of information science in the Departments of Computer Science and Administrative Sciences at the Naval Postgraduate School where he teaches courses in operating systems, real time systems, information systems, and operations research. His research activities include software engineering and computer performance modelling. He previously held technical and management positions with Systems Development Corporation, Planning Research Corporation, Computer Usage Company, and Univac. Dr. Schneidewind has degrees in electrical engineering, operations research, and management from the University of California, Berkeley, and USC. A holder of the Certificate in Data Processing, he is a member of ACM, ORSA, IEEE, and the IEEE Computer Society (Technical Committee on Simulation Newsletter Editor), as well asEta Kappa Nu, Alpha Pi Mu and Beta Gamma Sigma.

Acknowledgments

This paper is based on research sponsored by the Naval Air Development Center, Warminster, Pennsylvania. The technical contributions of the following individuals are gratefully acknowledged: Professors Gordon H. Bradley and Gilbert T. Howard, Naval Postgraduate School; Dr. Samuel Litwin and Mr. Richard J. Pariseau, Naval Air Development Center; Lieutenant Thomas F. Green, United States Navy; and Lieutenant Commander Michael Kirchgaessner, Federal German Navy.

References

15. S. Litwin and R. J. Pariseau, “Variance of the Number of Errors Detected During a Set of Random Passes Through an Error Laden Graph,” Naval Air Development Center Technical Memorandum, 52TM 76-STS-001, 6 April 1976.

April 1977