CASE STUDY OF SOFTWARE COMPLEXITY
AND ERROR DETECTION SIMULATION

N. F. Schneidewind
Naval Postgraduate School
Monterey, CA 93940

The history of developing and using a simulation model for the study of software error processes, complexity and structure is traced. Strong and weak points of simulation as they relate to model validity, accuracy and cost of implementation and use are discussed. The simulation model is compared to a similar analytic model. The history of an experiment in software complexity and error analysis is used to show the correspondence between empirical and model results. Empirical methods are contrasted with the use of models in terms of validity, accuracy, generality and cost. An assessment is made of the applicability of the techniques, based on these experiences.

Introduction

Some of the problems which have challenged computer software developers and testers are: (1) control of program complexity during development in order to avoid problems with debugging, testing and maintenance during later stages of the software life cycle, (2) identification of quantitative measures of program complexity which can be used in (1) and for allocating resources to testing, (3) estimation of difficulty of debugging as a function of program complexity and (4) estimation of degree of program checkout which can be accomplished with a given number of tests. A simulation model was developed which was addressed to assisting in the solution of the above problem areas. The history of a software error detection simulation model development and use, and related work, is described and evaluated for the purpose of informing software model developers and users about:

(1) Advantages of simulation for this application.
(2) Disadvantages of simulation for this application.
(3) Strengths of simulation relative to analytic models and empirical experiments.
(4) Lessons learned and things which will be done differently the next time.

Elements of Program Testing

The starting point for developing a functional test plan is to identify system outputs and the conditions under which these outputs should occur. Conditions refer to inputs, processing states and operating mode which are necessary to produce a given output. Operating mode refers to the environment in which processing takes place, e.g. time sharing. Inputs which have no meaning for a particular operating mode, e.g. batch inputs during a time sharing mode, are invalid inputs and are rejected. Subsequent discussions will pertain to an unchanging operating mode and it will be assumed that inputs correspond to the operating mode. Processing states are defined by the values of status indicators and by data base content and status. The number and types of inputs are the control variables which are available to the tester for constructing an efficient test plan. One view of an efficient test plan is one which uses only those inputs which are necessary for producing given outputs and does so with a minimum number of tests. Another view, and the one of interest for this paper, is a plan which maximizes program coverage for a given number of tests or, alternatively, minimizes number of tests for a given coverage. The latter concept is relevant for program quality assurance purposes. Coverage refers to the fraction of a program which has been executed. It is usually measured in terms of number of paths or branches which have been executed in relation to total number of paths or branches, respectively. In general, quality assurance is increased with increased coverage. If a program is viewed as a directed graph (Figure 1), coverage may be thought of as the portion of a program which is traversed by one or more inputs.

Model of Program Execution

In order to understand the test process, it is useful to have a model of program execution which includes inputs I, processing states P, and outputs O. If a program is initially in processing state P0, the input-output relationship can be viewed as one where P is caused to sequence thru P0 ... Pj ... Pn, and produce an output Oj, corresponding to Pj, where n is the final program state and j ≤ n. The sequencing of P is

---

1Supported by a grant from the Naval Air Development Center, Warminster, PA.
caused either by the sequence of inputs $I_1, \ldots, I_m$, or by an internal mechanism in the program which causes a transition to $P_j$.

An input is identified with a path, where a path is a connected and directed series of arcs (branches) which originate at start node $1$ and terminates at terminal node $t$. The series of instructions executed on a path is referred to as an execution sequence. Changes in $P$ occur as instructions are executed on a path. A path, and hence an input, is identified with a test. Thus coverage can be measured in terms of tests or inputs. An input, traversing an intended path $1 \rightarrow \ldots \rightarrow t$, may cause an incorrect value of $P$. This action results in an error. If errors are represented as occurring between a pair of nodes, $i$ and $i+1$, or on the arc connecting these nodes, the traversal of an input will be halted at this point, assuming the error is detected.

Test Methods

In some functional tests, inputs are chosen more or less at random in order to reduce bias in testing, or the nature of the inputs may actually be random. In other cases, input selection is deterministic, i.e., inputs are selected on the basis of criticality of the output associated with the input, sequence of inputs which is meaningful in the application, or for other reasons; in any event, the type and sequence of inputs is pre-determined. Independent of whether input selection is random or purposeful, a good strategy is to partition the program into paths which are as distinct as possible, that is, with paths that have a minimum number of branches in common.

One method of testing is by path. Each piece of test data, or input, defines a path. The ability to achieve high coverage is limited by the complexity of program structure. Complex structures contain many paths; it is difficult to cover a high percentage of paths in a complex structure without the number of inputs becoming prohibitive.

The most difficult facet of testing is lack of knowledge about the number, type and distribution of errors in a program. Whereas program structure is known, the location (distribution) of errors in the structure is unknown. Even error discovery does not help much. The detection of one error does not tell us much about the number of errors remaining or their locations in the program. This means that a test strategy can exploit a program's structure but not its error distribution.

Simulation Model Objectives

The objectives of the simulation model are to: (1) ascertain the relative difficulty of testing a set of programs prior to the test and (2) provide an evaluation of relative error detection prior to the test. This information is used to control the complexity of programs during their development in order to avoid excessive debugging during test. The information is also used for test planning by allocating test personnel and computer time to programs in accordance with anticipated relative difficulty of testing. The three types of outputs of the simulation model are: (1) expected percentage of residual errors versus number of inputs for a given program structure, (2) expected percent of coverage (arcs tested) versus number of inputs for a given program structure and (3) expected fraction of errors detected, for a given number of inputs, versus program structure complexity (number of paths). Examples of these outputs are shown in Figures 2, 3, and 4.

Description of Simulation Model

A simulation model of error detection must be general; it must have validity for a large number of cases. Experiments conducted with the model must be capable of replication and it must be possible to compute model statistics. For these reasons the selection of a path, which defines a path and test, is random. In the model, an input randomly branches (with equal branch probabilities) at each node until either an error is encountered between nodes $i$ and $i+1$ or node $t$ is reached. In the former case, the error is "removed" and the input retraces its previous path from $1$ to $i$; at $i+1$, a branch is randomly selected and the above process continues until $t$ is reached. In reality, the above process is no different than determining in advance, on a random basis, a path which the input will traverse.

Since the model must be general and since error distributions are not known in advance of testing, errors are randomly seeded in the program structure during the first phase of the simulation. A reasonable assumption concerning the error distribution is that the number of errors is proportional to number of statements in a program segment (arc). Accordingly, the distribution of error positions in an arc is uniform (equal probability) with respect to the beginning of the arc. This is equivalent to an exponential distribution of number of statements between errors, which is also equivalent to a Poisson distribution of number of errors per given number of statements.

Different programs will have different branching probabilities and a given program will have different branching probabilities depending upon the input data. In general, these probabilities will be unknown or, at least, very difficult to estimate. For this reason and because the model must be applicable to all programs, equal branching probabilities are used. The choice of branching probabilities has no significant effect on simulation results under the condition of random error distribution.

Evaluation of Simulation as a Tool for Software Error Analysis

Choice of Language

844
The type of simulation that is performed in software error analysis is not amenable to the use of a typical simulation language like GPSS, whose fundamental characteristic is the manipulation of queues. The analysis of wait times, queue sizes and the simulation of discrete events were not involved in the error simulation. The items that were important were program transportability and the ability to call subroutines, such as the Naval Postgraduate School random number generator, from our FORTRAN Library. In addition, program logic mainly involved branching decisions which simulated an input traversing a program and branching at decision nodes. This capability is readily accomplished with a general purpose language. For these reasons FORTRAN IV-G was used. No major difficulties were encountered in using this language.

Statistical and Computational Problems

The choice of probability distributions for error seeding and branching probabilities were the major problems in developing the model. This was the case because decisions had to be made about processes whose characteristics were unknown and where distributions could vary widely as a function of particular application and program characteristics. A related problem was the determination of number of replications which would be necessary for making statistical computations. All of these problems arose because it was not valid to use selected error distributions and selected program execution paths, since, as stated previously, program error distributions are unknown and distribution of path executions is input data dependent; therefore, simulations of particular cases would not have been meaningful. What was needed was a model which would have general applicability, and from which expected values and variances could be computed for error detection. The difficulty with this approach was the number of replications, say 100, that was necessary for statistical validity resulted in excessive CPU times, on the order of several minutes for large program structures. This was related to the fact that the program was written for batch processing use, with attendant long turnaround times. Under a batch system, the simulation could not be terminated or modified if processing were excessive or if the results were in error. The decision to implement the simulation for batch processing had far-reaching consequences. This decision had the following adverse effects in addition to the one which has been mentioned:

- Long debugging times.
- Lack of flexibility for making changes.
- Lack of capability for human interaction with the model for parameter changes or for changing the mode or course of the simulation as it was being executed.

Validation

Traditionally, validation of a simulation model has been "accomplished" by finding that simulation and analytic model outputs are equal. This result, of course, only demonstrates that the two methods have implemented the model logic in equivalent ways; there is no proof that either method properly reflects the real world conditions that are being modeled. An analytic model was developed for this project, but its use for validation was a secondary consideration. The analytic model was developed primarily to provide faster execution time and to be used in those instances where the detail provided by the simulation model (individual path traces) is unnecessary. Unlike the simulation model, the analytic model does not compute the variance of number of errors detected; this quantity is useful for many analyses. Thus, the two methods are complimentary, analytic for less detailed, quick results and simulation for detailed, time-consuming results. Secondary, the analytic model is used as a check against--not a proof of--the simulation model.

The validation tests which were conducted are described below.

a. Mean number of Errors Seeded.

Errors are seeded in the simulation model with exponentially distributed distances (number of source statement) between errors. This is equivalent to a Poisson distribution of number of errors seeded per arc, with the mean number seeded being proportional to arc length. The mean number of errors seeded in the directed graph is also Poisson distributed with mean $S/M$ and standard deviation $(S/M)^{1/2}$, where $S$ is number of source statements and $M$ is the mean number of statements between errors. Since the sample was $N = 100$ seedings, the normal approximation of the Poisson was used. Since the variance is known, the test statistic:

$$Z = \frac{(S/M) - \bar{X}}{(S/M)^{1/2}}$$

where $\bar{X}$ is the mean number of errors seeded over $N$ seedings with $M = 21$ statements between errors. A two-sided test was used with $\alpha = .05$. Eight Module 1 Naval Tactical Data System procedures were tested for error seeding.

$H_0$: $\mu = S/M$

$H_1$: $\mu \neq S/M$

Reject $H_0$ if $|Z| > 1.96$

The results of the hypothesis tests are shown in Table 1.

**Table 1**

| Procedure | S/M | $\bar{X}$ | $(S/M)^{1/2}$ | $|Z|$ |
|-----------|-----|-----------|----------------|-------|
| 8         | 10  | .476      | .550           | .690  | 1.072 |
| 14        | 9   | .429      | .430           | .655  | .015  |
| 25        | 8   | .381      | .400           | .617  | .308  |
| 34        | 15  | .714      | .780           | .845  | .761  |
| 39        | 17  | .870      | .890           | .900  | .033  |
| 67        | 12  | .571      | .680           | .756  | 1.442 |
| 48        | 13  | .619      | .700           | .787  | 1.029 |
| 53        | 11  | .524      | .630           | .724  | 1.464 |

845
Although $H_0$ would be accepted in each of the above tests, the simulation error seeding was consistently high.

Another test involved a graph with a single input and a single exit node. The arc joining these nodes had a length of 10 and $M$ was set to 1 so that the expected number of seeded errors was 10. The seeding subroutine was called 1000 times to seed errors in this arc, and the mean number of errors seeded was 9.995. This test was conducted by running a 1000 inputs through the graph and since each input traverses the single arc the number of errors found is the same as the number seeded ($Z = (9.9995 - 10)/10^{1/2} = .00005$).

b. Probability of Arc Traversal

Traversals on a given arc or path are independent and the probability of traversal is constant on successive trials. The number of traversals in an arc $ij$ is binomially distributed. The probability of arc traversal is $P_{ij}$ and the relative frequency of traversal obtained from simulation is $P_{ij}$, so that $E(P_{ij}) = P_{ij}$ and $V(P_{ij}) = P_{ij}(1-P_{ij})/N$, where $N$ is the number of independent trials (100 replications x 100 repetitions = 10,000 trials). Since a normal approximation can be used when $N$ is this large and the variance is known, the test statistic

$$z = (P_{ij} - P_{ij}) / (P_{ij}(1-P_{ij})N)^{1/2}$$

was employed for a two sided test with $\alpha = .05$. Eight procedures (different procedures than used in seeding tests) were randomly selected. A branch node of each of these eight procedures and its outgoing arcs were also randomly selected.

$H_0$: $\mu = P_{ij}$

$H_1$: $\mu \neq P_{ij}$

Reject $H_0$ if $|z| > 1.96$

The hypothesis $H_0$ was accepted in each case.

c. Numbers of Arc and Path Traversals

The branching mechanism was tested by including in the program a traversal counter which records the number of times each arc is traversed during a program run. The correct functioning of the counter was confirmed by obtaining detailed output for several different graphs and manually confirming that the count corresponded to the detailed output.

Several runs were made to test the actual traversal count against the expected number. One specific test is reported below:

The graph with a single input node connected to each of 4 terminal nodes was used in a run with 1 input, 4000 replications, and 1 seeding. The expected number of traversals on each arc is 1000, and the traversal counter showed 994, 1004, 1006, 996 as the observed frequencies. These values easily pass a chi square test at the 99% confidence level.

Comparison With Empirical Results

Both the simulation model results (Figure 4) and those obtained from the analytic model showed a significant relationship between program complexity and error detection and between complexity and test effort. Although gratifying, these results were insufficient because it was still not known whether model results would correspond to software error processes observed in practice. Both models served as excellent vehicles for studying the error/complexity problem, but now it was time to turn our attention to obtaining empirical data. By this time some proposals for software metrics had been advanced and there were tentative results which suggested a quantitative relationship between complexity and error properties, but substantive quantitative measurements of structural complexity were not available. Therefore an experiment was undertaken to obtain these data. The experiment involved the design, programming, debugging and testing of four programs which were coded in ALGOL-W for execution on an IBM 360/67 under OS/MVT and CP/MS operating systems. One of the major results of this experiment is shown in Table 2.

<table>
<thead>
<tr>
<th>Complexity Measure</th>
<th>No Errors</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Number</td>
<td>Mean No. of Value Procs.</td>
<td>1.70</td>
</tr>
<tr>
<td>No. of Source Statements</td>
<td>Mean No. of Value Procs.</td>
<td>9.36</td>
</tr>
<tr>
<td>No. of Paths</td>
<td>Mean No. of Value Procs.</td>
<td>10.1</td>
</tr>
</tbody>
</table>

The mean values of several structural complexity measures (e.g., number of paths) are tabulated for procedures with no errors against procedures with errors. Definitions of these complexity measures are given in reference 4. The higher the values of these measures, the more complicated are the corresponding program structures and, hence, the greater the difficulty encountered in writing and debugging these programs. Another important result was that the mean error finding and correcting times for procedures with errors were about twice the corresponding times for procedures without errors. Thus simulation, analytic and empirical approaches all showed significant relationships between complexity and error properties and between complexity and test effort. However, an interesting difference arose between the simulation/analytic model approach versus empirical methods. In the former, the number of errors
which was probabilistically seeded in a program structure was a known quantity. Thus it was possible to obtain either counts of detected errors or residual errors and to plot these quantities against complexity or number of inputs. This, of course, was not the case in the empirical study, where number of errors can never be known. Accordingly, whereas the models measured error detection capability, the purpose of the error experiment was to count errors made in design and programming and to relate these data to complexity and other factors.

Conclusions

Overall, the following conclusions can be drawn regarding the usefulness and applicability of the various techniques:

. The simulation and analytic models were valuable for analyzing and studying the problem, particularly in the beginning of the project, and provided valuable insight into the nature of the complexity-error properties relationships. However, the real demonstration of the validity of our hypothesis and accuracy of results could only be obtained by empirical methods.

. Although the use of FORTRAN as a language for simulation in this problem was satisfactory, hindsight indicates that the model should have been implemented by using this language under time sharing rather than batch processing.

. Although, as indicated above, empirical methods have the advantage in terms of validity and accuracy, they lack generality, are short lived in their applicability, and are relatively costly to undertake.

. The desirability of achieving generality in the models was a two-edged sword in that the lack of specificity (e.g., deterministically testing certain paths and error distributions) for certain applications is the most significant limitation.

. All methods were very useful for developing principles of program design and testing. However, the techniques were applied to small programs. The validation of these results against large-scale software projects is a challenge for future research.

References


Figure 4: Expected Fraction Errors Detected On First Input vs. Number of Paths ($P_p$)

<table>
<thead>
<tr>
<th>Module 1</th>
<th>Module 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

1 2 3 4 5 6 7 8 9 10 20 30 40 $P_p$