Adaptive random data generation for computer software testing

by STEPHEN F. LUNDSTROM
Burroughs Corporation
Carlsbad, California

INTRODUCTION

This paper discusses computer-assisted generation of input data for program testing. The test data generation system must produce input data that is per specification, unbiased by prior analysis of the program to be tested. The test data generators evaluated use random generation techniques to produce the test data. To reduce the number of sets of test data needed to test a program, summary information about the performance of previously generated sets of input data is used to modify the probability distributions upon which the next set of test data is based. Four test data generators were evaluated and used to generate test data to exercise five testcase programs of various complexity. Observations of the results of the actual evaluation runs and of the types of structures involved led to the establishment of some guidelines for future testing operations. Program verification of any sort was not attempted. Manual checks of the normal program outputs did detect a number of software errors. No attempt was made to automatically isolate or even detect software faults. Nor was any comparison with other methods, manual or automatic, made.

The Software Testing System developed to use as a tool to evaluate the test data generators is similar to others previously reported.1-7 These systems require repetition of steps to guide and assist test case selection, to execute the instrumented program and to analyze testing coverage until the test goals have been achieved. The system used here to evaluate the test data generators not only provided the summary reports of testing progress similar to the above systems, but also provided progress summary information to the test data generators which was then used to modify the probability distributions controlling the generation of the test data.

Most previous work relating to the generation of test data for program testing in some way base the test data on analysis of the program to be tested. R. L. Sauder determined the format and description of the test data by analysis. Other systems9-12 use analysis of the program to guide choice of test data when the main goal of testing was usually to execute all paths through a program.

Program analysis is subject to any related programming errors in the program to be tested which may, in turn, cause the generation of test data which is inconsistent with the original specification. If any paths are missing in a program to be tested, they would not normally be detected through use of test data generators which are based on program analysis.

Program test data should be based, as much as possible, on the program specifications rather than on the program implementation. Sauder's work was unique in that it required the user to specify the data relationships thus allowing the "semantics" of the test data to be derived from the specification and whatever test plan might exist. The systems reported by W. H. Burkhart,10 and K. V. Hanford14 are specifically concerned with testing compilers and generate test data from the formal syntax of the language to be processed. These systems are unable to take any semantic information, such as relationships or limitations between syntactic variables, into account in the generation of test data. A new program product recently introduced by Burroughs Corporation19 does allow specification of both the desired syntax and semantics of the desired test data. Each of these three systems generate data based on a uniform random distribution generator without any concern as to the efficiency of such an approach.

The only reported work involving random test data generation where techniques were evolved to improve the efficiency of the generators was in the area of hardware test data generators used to test digital logic.16-17 The original concepts have been further extended by K. P. Parker.20-22 Parker's system allows the probability of a particular value of any test input to be based on the average or most probable value of previously successful sets of test data.

Although the hardware techniques were developed without regard to software problems, there are many areas in common and some of the hardware experience can be applicable.23 Although the concept appears valid, hardware testing is the process of searching for equipment malfunction (assuming a correct design), but software testing is the process of somehow verifying the design and implementation of the software itself.
SOFTWARE TESTING SYSTEM

The software testing system used for evaluation of the test data generators is organized into two phases as shown in Figure 1. The first phase, program analysis, inserts monitors in the program to be tested. The monitors allow observation of flow through the control structure of the program, and also observation of changes in the values of variables during execution. The second phase, program test and evaluation, is the phase which generates test data, runs the instrumented program and evaluates the results of the tests. In each of the repeated cycles, a new set of input data is produced, the program (with monitors) is executed, and the results are evaluated. Three of the four test data generators evaluated were adaptive and utilized monitor-generated summary information relating to the success of previous tests in their heuristic algorithms.

The actual format and values of the input data must be as specified for the particular program under test. Special programs supported this function for each of the five test programs used. In a production environment, a special input data specification language would be developed together with a processor for that language. Even though each program to be tested has special requirements for the form and range of its input data, the procedures used for selection of the data are generalized and apply regardless of the program to be tested.

Monitoring requirements

When developing a testing system of any sort, it is important to know what the test procedure has tested and to what extent. This information is important both in determining what is faulty if a problem is found and in determining when sufficient testing has been done. The system developed monitors both the control structure and range of variables as a means of measuring the ability of the generators to "thoroughly" test the software.

Branch monitoring

All possible branches within the program graph are monitored. The obvious goal is to attempt to exercise all possible branches. Even if all possible branches are exercised, all possible paths in the program being tested may not have been exercised. In addition, there may be paths which have not yet been implemented and which cannot be monitored. The simpler measure of branch executions was chosen as one measure useful in demonstrating the viability of the proposed adaptive random data generators.

Range of value monitoring

The user may define a desired range of values to be monitored for each variable of interest. Each range is divided into "N" equal size segments (another simple measure chosen as sufficient to demonstrate the viability of the generators, in this case N=10). Assignments to these variables are monitored with the goal of observing data values in each of the segments of the range for each variable. The specified range of interest is not required to be the same as the observed range of values for any variable.

Termination conditions

The need for well-defined termination conditions are particularly important in a testing system where there may be no guarantee that all testing requirements can ever be met. The following conditions were adopted, any of which can force immediate termination of any testing in progress:

1. "M" test data sets have been applied to the program being tested. (M=50 was used.)
2. All possible branches have been observed to have ex-
executed and values have been observed in each segment of the range of values for each monitored variable.

3. The last "P" test data sets were inefficient. In this case, if $P=10$ successive test data sets were not productive, termination would occur. A test data set was considered not productive if the sum of "new" branch executions observed and "new" range of value segments observed is "R" or less ($R=2$ during testing). "New" branches are those observed which had not been executed previously; "new" segments are those in which values were observed which previously had not been.

Software test sequence

The actual procedure of software testing can start after the program analysis phase has generated the instrumented source program. The system developed (see Figure 2) consists of a central control which, after any necessary initialization, sequences through automatic test data generation, execution of the instrumented source program and test evaluation. The conditions which cause termination were defined previously. These conditions are checked during the test evaluation phase. If termination is not yet required, the central control restarts the test generate, execute and evaluate sequence. The automatic test data generators are discussed in detail below.

Manual verification

The software testing system developed is not capable of automatic verification of the validity of the outputs of a program given the automatically generated inputs. The system produces summaries of the results of all testing (including which branches have not been executed and what range of value segments have been observed). This information together with the normal program outputs must be manually inspected as the final step of program testing.

AUTOMATIC TEST DATA GENERATION

The major problem faced in automatic test data generation is how to devise adequate tests. Program development proceeds through a sequence of four steps; specification, design, implementation, and verification. If the specification is assumed correct, then the design and implementation should be checked. The test data generation system described below generates input data based on the specification. It also allows use of manually generated or previously saved input data. By monitoring the execution of the program, the testing process covers both the design and implementation. Other systems, which generate test data based on analysis of the program being tested only have the capability of exercising the implementation. Analysis-based test data generators produce test data with implementation errors reflected in the test data. Therefore, many kinds of implementation errors are masked. Specification-based test data generation does not have this problem. Since input data is based directly on the specification, any errors due to the design or implementation procedure are vulnerable to detection. Errors such as missing control paths or incorrect handling of input data are likely to be observed using specification-based test data.

In a production-oriented software testing system, an input data definition language should be available to allow the representation of the portion of the program specification which defines expected input data in an unambiguous, machine-recognizable form. In the system implemented, that portion of each generator which would normally interpret an input data description was specialized for each of the programs tested. The specialization was based on the specification of the input data to be produced and not on analysis of the programs tested.

Recall that the software testing system monitors both the sequence of execution and range of value segments. The sum of the number of "new" branches and "new" range of value segments (where "new" indicates previously unobserved) is used as a means of comparison of the productivity of two test data sets and is used during evaluation of the termination condition.

The judicious selection of a small number of input test...
data sets from an extremely large domain is another major problem when developing an automatic test data generation system. The capabilities of four input test data generators were evaluated. The four generators are called UNIFORM, GAUSS, INVERSE, and NEXT-BEST. All of these generators, except UNIFORM, are designed to adapt to the most effective areas of the input data space.

**UNIFORM test data generator**

The UNIFORM test data generator is the only generator evaluated which is not adaptive. Values are chosen randomly over the given legal range of the particular test data element being chosen. A uniform probability distribution guides the choice so that each value in the range is equally likely to be chosen as any other value in the range. In addition to allowing direct use of the UNIFORM generator, each of the other generators use the UNIFORM generator under circumstances explained below.

**GAUSS test data generator**

The development of the GAUSS test data generator was motivated by the observation that many program organizations are not symmetric. The simple example below illustrates the asymmetry. The example procedure (Figure 3) receives an input parameter X which is an integer in the range of 0 through 100. If X is less than 50, a parameter Y is incremented. If X is less than 50 and is even, then a parameter Z is multiplied by X. If input data to test the example procedure is chosen with a uniform distribution, one half of the cases, on the average, will test only the first conditional. Only one quarter of the cases, on the average, will test both the code which increments Y and which multiplies Z by X. If those were major program blocks, it is clear that much of the work of a UNIFORM test data generator would be ineffective in testing the example program. The asymmetry also exists in computation of variable values. The GAUSS test data generator is designed to bias choice of test data by consideration of previously useful test data sets.

Test data generators are called productive if the data produced exercise portions of the program not previously tested. In terms of the productivity requirement, the program organization (or at least that part of interest) is dynamic. After each application of a set of inputs, that portion of the program which actually was exercised “disappears” and is no longer used in evaluation of the performance of the generator. The goal of the generators is to produce test data which exercises the “remaining (untested) portion” of the code.

Once the generator produces inputs which force execution of a branch of the program not previously executed, the generator becomes productive. The productivity might be maintained by choosing inputs to again enter the same branch, but perhaps test a different subbranch. One can usually assume that entry to a particular branch is controlled by some subset of the input variables. Thus, if the proper input variables are kept “the same” and others are modified, it is likely that various subbranches will be exercised. However, no analysis is performed which would determine the subset of inputs which control entry of some branch. If input values are chosen randomly under control of a Gaussian distribution, and if the standard deviation is small, and if the mean represents values needed to enter the branch, the generation of values the same as before becomes likely. The variation in values should be limited enough to continue generation of tests of a productive branch but, at the same time, be sufficient to exercise other subbranches. Eventually, all the subbranches have been exercised, the productivity of test inputs based on that distribution will decrease. As this occurs, other successful test sets will be used to compute the distribution, and inputs will be produced to test other portions of the program.

**Initial GAUSS distribution**

Unless the user provides guidance, the GAUSS test data generator has no means of knowing where to set the mean and standard deviation initially. In order to provide some initial history, the GAUSS generator produces the first “C” cases with the UNIFORM generator. For purposes of evaluation, “C” was arbitrarily chosen to be four. The UNIFORM cases are not generated if any manual or previously generated cases are input initially.

**GAUSS generation technique**

The GAUSS generator bases the test data produced on a Gaussian probability distribution. Each variable generated with GAUSS has a corresponding mean and standard deviation which, together with the defined range of the variable,
control the values produced. The mean and standard deviation for each variable are not fixed. These parameters are adaptive. Each time the GAUSS generator is activated, the mean and standard deviation of each variable is recomputed based on the latest “successful” tests.

It remains to be determined which tests are successful. First, a test set is assumed to have certain productivity, defined for purposes of evaluation to be the sum of the number of “new” branches executed and the number of “new” variable value ranges observed. Recall that “new” simply means “not previously observed.” The test data chosen to compute the means and standard deviations are those whose productivity exceeds the average productivity of the “M” most recent sets of test data.

The size of the collection of test data sets considered, “M”, also is adaptive. If a number of successive data sets are not productive, the range of “M” is doubled up to an arbitrary limit. On the other hand, if a number of successive data sets are productive, the range of “M” is halved, but not less than the original value (in this case, 5). The size of the collection of data sets used to define the variable value distributions varies because of many of the arguments summarized earlier. A period of high productivity can often be maintained by basing distributions on a small number of recently successful tests. A period of low productivity is often characterized by a search for another productive area. A search of this sort is often successful when not biased by only a few sets of previously successful data, but by a broad representation of all previously successful tests.

The standard deviations of the distributions used to produce the test values vary widely. When a particular value $V_a$ of a variable $V$ is useful to force control into a particularly productive set of tests, the standard deviation of $V$ may become very small as more and more tests with values of $V$ near $V_a$ are produced. If the value of a particular data element is not important in that way, the standard deviation is likely to be very large, often large enough that the Gaussian distribution over the legal range of values is essentially uniform.

**INVERSE test data generator**

The development of the INVERSE test data generator was motivated by impatience with the dynamics of the GAUSS generator. In particular, because of the procedures used to compute the distributions of the various input data values, a number of test sets would be produced before the deviation was large enough to allow an effective search for new productive areas. The problem appeared to be that data generated based on current distributions was often non-productive and no reliable information was available as to which part of the remaining portion of the input space would be productive.

**INVERSE and GAUSS commonality**

The resulting INVERSE data generator operates the same as the GAUSS generator during computation of variable value distributions, establishment of an initial history with cases produced by the UNIFORM generator, adjustment of the size of the collection of test cases used for distribution computations, and random generation of data based on a Gaussian distribution. Only after the normal GAUSS procedures become non-productive is the INVERSE generation technique activated.

**INVERSE generation technique**

The only difference between the GAUSS and INVERSE generators is how data values are produced after non-productive test data sets. If the previous test data set is not productive, the INVERSE search procedure is activated to attempt to find a new productive region in the input space. Successive activations of the search procedure alternate between standard generation of input data with the GAUSS generator and generation of input data with the INVERSE procedure. A true inverse distribution is not really appropriate. The inverse GAUSSIAN distribution would tend to generate probable values at the boundaries of the input data space. Rather, it is desirable to produce values not likely to be produced with the GAUSS generator, but still provide an unbiased search of the rest of the input space. The UNIFORM generator can accomplish this easily with the addition of a simple test of each of the variables generated. If a UNIFORM generated value is in the “GAUSSIAN-likely” range (MEAN–STANDARD DEVIATION through MEAN+STANDARD DEVIATION) where MEAN and STANDARD DEVIATION are as normally computed for the GAUSS generator, then another replacement value is produced (through use of the UNIFORM generator once again). When none of the generated values lie within the “GAUSSIAN-likely” range, the data values are output and applied to the program being tested. Analysis of the productivity and recomputation of the variable distributions then proceed as previously described with GAUSS. In many respects, the INVERSE test data generator can be thought of as a random data generator controlled by a UNIFORM distribution which has a hole in it, centered at the variable MEAN and with radius of the STANDARD DEVIATION. It allows a uniformly distributed random search of the area not recently productive.

**NEXT-BEST test data generator**

The NEXT-BEST test data generator was motivated by the observation that once a particularly productive input data set is chosen, other productive data sets can often be produced solely by changing only one of the variables in the input data set. Calculation of most variable values can be considered to be some function of the input variables. Variations of only one input variable will usually affect the computed values of some of the program values. Similarly the expressions which control branches between basic blocks in the program are also functions of the input variables. The appropriate modification of only one input variable
can cause the execution of different basic blocks. The effect of modifying only one input variable is essentially an orthogonal search of the input data space for other productive cases given a known productive case. Any arguments in favor of this technique for execution of basic blocks is equally valid for testing ranges of variables.

**Initial NEXT-BEST distribution**

The NEXT-BEST test data generator must have at least one set of input data values before modification of individual, randomly-chosen input values can be done. If no manual or previously specified data sets are input initially, one input data set is produced using the UNIFORM generator. If more than one manual or previously specified data set is input, the most productive will be used during the next phase of the NEXT-BEST procedure, input variable distribution initialization. The case chosen will be called the base case.

**Input variable distribution initialization**

The input variables which are randomly selected will not be equally productive during the use of the NEXT-BEST generator. In an attempt to further improve efficiency, the most productive variables are given a somewhat greater probability of being used to generate new data sets under the implementation of the NEXT-BEST generator. Before normal processing begins, each of the input variables is used to generate at least one input data set which is a modification of the base case chosen after the Initial Distribution phase. If the data set generated by modification of a particular input variable is productive, then the same variable will be used again. If non-productive, the next variable is used. The values generated for each of the chosen variables are produced through use of the UNIFORM generator. A record of the total productivity of each variable is kept both during this and the final phase. The productivity information is utilized during input variable selection.

**NEXT-BEST generation technique**

After Initial Distribution and Input Variable Distribution initializations, the major test data set generation phase begins. This phase consists of a number of subsequences of test data set generation steps. Each subsequence begins with the choice of a base case. The new base case chosen is that test data set previously generated which has the largest productivity of all those not yet used as a base case. Upon completion of a subsequence this base case is marked as used.

After a base case is chosen to start a subsequence, a new input variable is chosen. The choice is made randomly but with a preference toward previously productive variables. To make the choice, an assignment of each variable to a portion of the range of real numbers from 0 through 2 is made. Let $S_i$ be the size of the portion of the range assigned to the $i$th variable. Assume there are $n$ variables and that $P_i$ is that fraction of the total productivity to date contributed by the $i$th variable. Then:

$$S_i = \frac{1}{n} + P_i,$$

A random number between 0 and 2 is then chosen under control of a uniform distribution. That number will be within the portion of the range from 0 through 2 assigned to one of the variables and will be used to select that variable. The order of the variables across the range does not vary and is the same as the order of selection during variable distribution initialization phase.

Once a variable is chosen, a new value is selected for it to replace its original value in the current base case. The selection is under control of the UNIFORM generator. If the resulting test data set is productive, the same variable is used again for production of the next data set. When a data set is not productive, the next variable in the same order as chosen during the variable distribution initialization phase is selected. By sequencing to other variables in this way, variables with poor past productivity can develop a more complete history of productivity. The subsequence ends when the end of the sequence of variables is reached. At that point, the base case is marked as used and a new subsequence is initiated.

**EVALUATION AND RESULTS**

The software testing system together with the automatic test data generators were evaluated through use of some programs chosen as testcases. Limitations of manpower and computer time reduced the number of programs chosen to five. They were chosen to represent a wide variety of program application and complexity. Three of the programs are student submissions from an introductory programming course and represent table lookup, search techniques, interpolation, numerical approximation, divided difference, boundary value and game playing applications. In these three cases, the general problem was chosen as representative and one of the programs submitted was arbitrarily chosen for each. The other two programs chosen are production programs written (and selected) by a staff analyst other than the author. These two programs are numerical analysis programs concerned with the analysis of stresses in turbine blades. The five programs were called:

- Temperature Distribution in a Conducting Solid
- The Series Magnetic Circuit with an Air Gap
- Tictactoe (a tournament)
- Aerospace Analysis
- Turbine Blade Analysis

(See Lundstrom" for a more complete description of each of the five testcase programs.)
Test procedures

Each of the five testcase programs was “tested” by each of the four test data generators. Four of the testcases were also “tested” with at least one prespecified set of test data values. In order to gather reasonable statistics, each of the procedures above was performed ten times.

Measurement and observation of results

Control flow and range of value monitors were chosen since they reflect, at least to some degree, the thoroughness of a collection of test data sets. In the future, other measures may prove more useful. The monitors are observed and recorded after each test data set is applied to a testcase program. This information, when averaged over the ten times each generator was used to test each testcase program, gives a good feeling for the expected performance of each generator.

Summary of results

The software test system proved to be useful in detecting errors in the programs tested. More importantly, even though the input test data generators used cannot be guaranteed to find all errors, the generators produce unbiased test data which is more likely to thoroughly exercise a program, with respect to its specification, than its creator would.

The NEXT-BEST generator, as implemented, never demonstrated productivity better than the others. The results might have been different if more initial cases were produced, if more initial variable productivity history development was performed, and if variables known to be nonproductive were used less during the main body of the procedure.

The GAUSS generator proved to be most successful in situations where a very non-uniform mapping from input data space to the program occurs such that only one productive region exists. In similar cases with more than one productive region, the INVERSE generator was most productive, since distribution repositioning can be speeded through the UNIFORM “search.”

The UNIFORM generator represents the type of test data generator most used in the past and is most productive when the mapping described is uniform. If the mapping is not known, then the UNIFORM generator may be slightly more productive than the INVERSE generator, most likely due to an insufficient “UNIFORM” search for productive areas before the main INVERSE procedure begins. Although the INVERSE generator is never quite as productive as the UNIFORM generator when the mapping is uniform, it is sufficiently better when a non-uniform mapping exists (especially if a good manual case is input) that the INVERSE generator can be recommended as the generator to use in the many situations when the mapping is not known.

Recommendations

An improved generator, called COMPOSITE, has been considered, but not evaluated. It should offer better early productivity and less sensitivity to the mapping. COMPOSITE would make use of both the UNIFORM and INVERSE generators described. However, COMPOSITE would allow selection of the generator to be based on past performance. The mapping changes for each variable as more and more of the program is exercised. By allowing the data generator to reflect the current mapping, an increase in productivity should be observed. Additional suggested features of COMPOSITE are described in Lundstrom.

The values of many variables cannot be monitored realistically with the fixed-size, fixed-number range of variable value divisions utilized in the system implemented. It should be possible to specify the number of range segments together with the size of each segment (without the constraint that each segment be the same size). It is more important that the possible values be distributed uniformly across the segments of the range of values of a variable than to have equal-sized segments.

CONCLUSION

The goals of the work described here were to study random test data generation techniques as applied to software testing, to develop and evaluate random data generation algorithms, and to develop a complete software testing system to support the study. The software testing system was successfully implemented. The test data generators were evaluated by using them to test each of five programs chosen as testcases; programs which represent a wide variety of applications.

The four test data generators developed and evaluated were called UNIFORM, GAUSS, INVERSE and NEXT-BEST. While each of these was observed to be productive in some situations, only INVERSE was consistently productive. It was determined that knowledge of the type of mapping from the input space to the program could allow choice of the appropriate generator for most efficient operation. Since understanding of the mapping is often difficult or time-consuming, and since the system developed did not provide tools to support identification of the mapping, the INVERSE generator was determined to be the most broadly useful over the cases studied. An improved generator, called COMPOSITE, was proposed, but not evaluated.

Although program verification was not attempted, enough program faults were manually observed in the test output to demonstrate that input test data generation can be successfully based on the specification rather than on analysis of the program.
ACKNOWLEDGMENTS

The work described in this paper contributed to the fulfillment of the requirements of the Doctor of Philosophy degree while the author was enrolled at Texas A&M University. Dr. Dick B. Simmons provided motivation to investigate the importance of range of variable value monitoring as well as much needed encouragement throughout. Dr. Richard E. Fairley was the catalyst leading to the understanding of how the relationship between the input test data set domain and the program affects the choice of the test data generator. Computer facilities were provided by Texas A&M University and Burroughs Corporation.

BIBLIOGRAPHY