Estimation of Software Reliability by Stratified Sampling

ANDY PODGURSKI, WASSIM MASRI, YOLANDA MCCLEESE, and FRANCIS G. WOLFF
Case Western Reserve University
and
CHARLES YANG
Massachusetts Institute of Technology

A new approach to software reliability estimation is presented that combines operational testing with stratified sampling in order to reduce the number of program executions that must be checked manually for conformance to requirements. Automatic cluster analysis is applied to execution profiles in order to stratify captured operational executions. Experimental results are reported that suggest this approach can significantly reduce the cost of estimating reliability.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.5 [**Operating Systems**]: Reliability

General Terms: Reliability

Additional Key Words and Phrases: Beta testing, cluster analysis, operational testing, software reliability, software testing, statistical testing, stratified sampling

1. INTRODUCTION

In software testing there is an important dichotomy between *synthetic* and *operational* techniques. Synthetic testing involves selecting test data systematically, based on an analysis of a program or its specification. Operational testing or *beta testing* involves having the intended users of software employ it in the field as they see fit. Synthetic testing and operational testing are complementary: synthetic testing is not a good predictor of operational reliability, but it may help to make software reliable enough for

Authors' addresses: A. Podgurski, W. Masri, Y. McCleese, and F. G. Wolff, Electrical Engineering and Computer Science Department, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, OH 44106; email: andy@alpha.cs.ces.cwru.edu; C. Yang, Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139-4307.

beta testing. The preponderance of software testing research has addressed synthetic testing techniques, of which many have been proposed. However, conventional beta testing suffers from a number of problems that merit investigation.

One of the problems with conventional beta testing is that it does not provide objective estimates of software reliability. Such estimates are necessary for making informed decisions about a software product's fitness for general release. Another problem with conventional beta testing is its reliance on ordinary users to detect and report failures in the software being tested. Although such users provide valuable feedback, they are unlikely to have detailed knowledge of the software's requirements specification, and testing is not their primary occupation. Consequently, they may fail to observe or accurately report some software failures. This problem can be addressed by employing *capture/replay* tools to capture operational executions so that they can be replayed off-line and reviewed carefully by trained personnel. However, the manual effort needed to review many captured executions may be infeasible.

We describe a new technique for operational testing that does not rely solely on ordinary beta users and which is intended to provide accurate and economical estimates of the reliability software *has exhibited* in the field. (It is assumed that past reliability is often a good predictor of future reliability.) The technique involves collecting execution profiles of captured beta-test executions and applying automatic *cluster analysis* to the profiles in order to partition the executions based on the dissimilarity of their profiles. A *stratified random sample* of executions is then selected, reviewed for conformance to requirements, and used to estimate the proportion of failures in the entire population of captured executions. These steps ensure that executions with unusual profiles are considered when reliability is estimated. We report an experiment in which executions of several programs were clustered based on branch traversal counts. Stratified random sampling produced significantly more accurate estimates of failure frequency than did simple random sampling, without requiring larger samples.

We now outline the remainder of the article. A brief introduction to stratified sampling is presented in Section 2. The use of cluster analysis for stratifying program executions is motivated in Section 3. Our procedure for estimating reliability is described in Section 4. Experimental results are presented in Section 5. The principal assumptions and the costs of our approach are discussed in Sections 6 and 7, respectively. Related work is surveyed in Section 8. Conclusions are presented in Section 9. Finally, possible future research is discussed in Section 10.

2. STRATIFIED SAMPLING

Stratified sampling is a classical survey sampling technique, which is used to estimate population parameters efficiently when there is substantial variability between subpopulations [Cochran 1977; Neyman 1934]. It in-

volves partitioning a population into disjoint *strata*, by grouping elements having similar values of one or more *stratification variables*. The values of the stratification variables are known for the entire population and are assumed to be correlated with the study variable. A stratified sample is constructed by selecting a probability sample from each stratum independently. The sample from a stratum is used to estimate a parameter of the stratum, such as the stratum mean. The population parameter of interest (e.g., the population mean) is estimated by a weighted average of the stratum estimates. *Stratified random sampling* [Cochran 1977] is a commonly used form of stratified sampling. It involves selecting a simple random sample from each stratum (without replacement). Consider a population having N elements and H strata. Let y be a study variable; let y_i be its value for the ith element of the population, $i = 1, \ldots, N$, and let N_h be the size of stratum h, for $h = 1, \ldots, H$. Suppose that a stratified random sample is selected from the population, with n_h elements drawn from stratum h. To estimate the population mean $\mu = \sum_{i=1}^{N} y_i/N$, the estimator

$$\hat{\mu}_{st} = \frac{1}{N}\sum_{h=1}^{H} N_h \bar{y}_h = \sum_{h=1}^{H} W_h \bar{y}_h$$

is often used, where $\bar{y}_h = \sum_{i=1}^{n_h} y_{hi}/n_h$ is the sample mean for stratum h and where $W_h = N_h/N$ is the relative size of stratum h. When the study variable is binary with $y_i = 1$ if and only if the ith population element has a given property, then μ is the *proportion* of elements having the property. In this case, we denote the population proportion by P and the aforementioned estimator by p_{st}.

Basic stratified estimators are unbiased, regardless of the stratification criteria used.[1] The variance of a stratified estimator will be small if the study variable's variance *within* strata is much less than its variance *between* different strata.[2] In practice, stratified sampling is often much more efficient than simple random sampling and is rarely less efficient [Cochran 1977]. An important aspect of stratified sampling is how the total sample size n is allocated among the H strata. Various allocation methods exist [Sarndal et al. 1992]. A particularly simple one is *proportional allocation*, in which the sample size allocated to a stratum is approximately proportional to the stratum's size, i.e., $n_h \approx nW_h$.[3]

[1]An estimator is *unbiased* if its expected value is equal to the population parameter to be estimated.

[2]An estimator's variance for a particular population can be estimated using the same sample data used to estimate a population parameter.

[3]The allocated stratum sample size is not always exactly proportional to the stratum size, because of rounding.

3. CLUSTER ANALYSIS OF EXECUTION PROFILES

In order for stratified sampling to be useful for estimating software reliability, program executions must be stratified so that some failures are *concentrated* in one or more strata or are *isolated* by themselves. (See Appendix A.) This must be done without knowledge of which executions actually fail. Although survey populations are often stratified based on a single variable, detailed execution profiles are a more suitable basis for stratifying program executions during reliability estimation, because the occurrence of a software failure may involve only a small part of a complex execution. Since it may be necessary to stratify thousands of executions, each of whose profile may contain thousands of elements, an automated procedure for analyzing profiles and forming strata is essential. Cluster analysis provides such a procedure.

Cluster analysis is a well-known multivariate data analysis technique which partitions a population into *clusters*, each of whose elements are more similar to one another than to objects of other clusters [Anderberg 1973; Kaufman and Rousseeuw 1990]. Typically each object is characterized by a vector of *feature* or *attribute* values, which may be binary, nominal, ordinal, interval, or ratio variables. The *dissimilarity* between two objects is measured by applying a metric, such as Euclidean or Manhattan distance, to their feature vectors. There are two basic approaches toward cluster analysis: *partitioning* methods construct a single partition of a set of objects, given a desired partition size or cluster diameter, whereas *hierarchical* methods construct hierarchies of partitions by merging or splitting clusters. Cluster analysis provides control over the number, size, and homogeneity of strata, as required for efficient stratified sampling. Further information about clustering algorithms is presented in Appendix B.

In most applications of cluster analysis, the goal is to identify meaningful groups within a population. For estimating software reliability by stratified sampling, it would be ideal if cluster analysis could separate successful executions from failures. However, this is not likely, because failures are often caused by small defects in a large program. Two executions may differ only in regard to reaching a particular defect, with the result that one execution fails, while the other does not. Conversely, two otherwise dissimilar executions may fail because they each encounter a certain defect. Hence, failures may not cluster together even if they have the same cause. Nevertheless, if a failed execution has an *unusual* profile, say because it reaches seldom-executed code, the failure may be isolated in a small cluster. This is often sufficient to make stratified sampling effective.

4. ESTIMATION PROCEDURE

Our approach to estimating software reliability involves the following procedure:

(1) The software is instrumented so that its inputs and other information necessary to replay executions are captured (logged to permanent storage).

(2) The software is subjected to extended operational use in one or more environments.

(3) The captured data are retrieved.

(4) A "laboratory" version of the software is instrumented to produce execution profiles and is then reexecuted using the captured data.

(5) Cluster analysis is applied to the execution profiles to stratify the captured executions.

(6) A stratifed random sample of captured executions is selected and rigorously checked by testing personnel for conformance to requirements.[4]

(7) The software's reliability is estimated from the sample, using a stratified estimator.

5. EXPERIMENTAL RESULTS

Podgurski, Yang, and Masri describe experiments in which the efficiency of stratified random sampling was compared to that of simple random sampling, with respect to estimating the failure frequency of eight subject programs [Podgurski et al. 1993]. Stratified random sampling was substantially more efficient than simple random sampling for six of the eight programs, and the two sampling designs were about equally efficient for the other two programs. The subject programs were rather small, however. In this section we describe additional experiments in which larger subject programs were used.

5.1 Subject Programs

Six subject programs were used in the experiments reported here. All were written in the C programming language. Five were recursive-descent parsers for ANSI C [Kernighan and Ritchie 1988], hand-written by students in a graduate-level course on compiler design. The other subject program was a version of a project-scheduling system that was used internally by a large company. The parsers ranged in size from 1624 to 6578 source lines; their average length was 3446 source lines. An execution population was generated for each parser by executing it on 1000 C source files obtained from system directories. Executions that terminated abnormally were classified as failures.[5] Every other execution was classified as a success or failure by comparing its output to that of both the GCC compiler

[4]Note that because operational inputs are captured directly there is no need to construct an *operational profile* [Musa 1993] (a model of operational usage) in order to generate them.
[5]Executions that took excessively long were timed-out automatically.

and a parser we constructed with the YACC parser-generator. The project-scheduling system comprised approximately 17,000 lines of source code, much of it involving the user interface. It was not possible for us to monitor the operational use of this system, so a program was developed to randomly generate valid inputs for it. Six defects were identified from a list of known problems. Each defect involved one element of the scheduling system's complex input. For $i = 1,2, \ldots 6$, the class C_i of inputs that revealed the ith defect was determined. For $k = 1,2, \ldots, 5$ and for $k = 10$, an input population of size 1000 was created by randomly generating k inputs from each class C_i, $i = 1,2, \ldots, 6$, and $1000 - 6k$ inputs that did not cause failures. No input induced more than one type of failure. Execution populations were obtained by executing the project-scheduling system on each input population. Thus, six execution populations were created having 6, 12, 18, 24, 30, and 60 failures, respectively. Note that the mechanism used to generate failure-causing inputs differed from that used to generate other inputs with regard to only one element of the input.

5.2 Method

For each population of executions, the parameter to be estimated was the proportion P of executions that failed.[5] Hence, the study variable y was binary with $y_i = 1$ if and only if the ith execution failed. We compared the efficiency of the stratified estimator p_{st} (see Section 2), used with stratified random sampling, to the efficiency of the sample proportion $p = \sum_{i=1}^{n} y_i/n$, used with simple random sampling. As is customary in sampling theory, the relative efficiency of the estimators was characterized by a ratio of their variances. Since all executions were checked, not just a sample, it was possible to use standard formulas to compute true estimator variances instead of variance estimates. The variance of p is

$$V(p) = \frac{N - n}{N - 1} \frac{P(1 - P)}{n}$$

where N is the population size, and n is sample size. The variance of p_{st} is

$$V(p_{st}) = \sum_{h=1}^{H} W_h^2 \frac{1 - f_h}{n_h} \frac{N_h}{N_h - 1} P_h(1 - P_h)$$

where H is the number of strata; N_h is the size of stratum h; $W_h = N_h/N$ is the relative size of stratum h; n_h is the sample size for stratum h; $f_h = n_h/N_h$ is the sampling fraction for stratum h; and P_h is the failure proportion for stratum h. For each population of executions, the relative efficiency $V(p_{st})/V(p)$ of p compared to p_{st} was computed, using the *same*

[6]In each case, P was greater than zero. Note that if $P = 0$, both estimators have zero variance.

total sample size for both estimators. A modified form of proportional allocation was used with stratified random sampling to ensure that the sample size for each stratum was at least one.[7] First, one sample element was allocated to each stratum; then remaining sample elements were allocated to strata proportionally. The actual size of the complete sample often differed slightly from the desired size, due to rounding.

Strata were formed using the two-stage cluster analysis algorithm described in Appendix B, which forms a set of initial clusters and then partitions them independently. Twenty clusters were created in the initial stage; the final number of clusters was varied systematically. The actual number of clusters often differed somewhat from the desired number, due to rounding and to the way first-stage clusters were subdivided. Program executions were clustered based on dynamic control flow. To profile this, the source code of each subject program was instrumented, using a tool we developed, to associate a counter with each conditional branch and to increment it whenever the branch was traversed during execution. (The average number of branches for the parsers was 622; the project-scheduling system had 2364 branches.) Checksums were used to help ensure that profiles were not corrupted by erroneous program behavior. In computing dissimilarities, each branch-traversal count was augmented with a binary variable that took on the value 1 if the count was nonzero. This was done so that the distinction between traversing a branch at least once and not traversing it at all would be weighted as heavily as the difference between the maximum and minimum number of traversals of the branch. The dissimilarity between the ith and jth executions was measured with the formula

$$d(i,j) = \frac{\sum_{f=1}^{2B} d_{ij}^{(f)}}{2B}$$

where B is the number of branches, and $d_{ij}^{(f)}$ is the contribution of the fth variable to the dissimilarity between executions i and j [Kaufman and Rousseeuw 1990]. Let x_{if} and x_{jf} denote the values of the fth variable for object i and object j, respectively. If the fth variable is binary then $d_{ij}^{(f)}$ is the exclusive-OR of x_{if} and x_{jf}; if this variable is a count then

$$d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{\max_h x_{hf} - \min_h x_{hf}}$$

provided the range in the denominator is nonzero; otherwise, $d_{ij}^{(f)}$ is zero.

Note that the longest clustering took about 7.5 hours on a DECstation *TM* model 5000/200 computer with 32MB of RAM. This computation

[7]With ordinary proportional allocation, the computed sample size for a very small stratum may round to zero.

Table I. Experimental Results for ANSI C Parsers (#*Strata* is the actual number of strata, and *Sample Size* is the actual sample size)

Program	P	#Strata	Sample Size	$V(p_{st})$	$V(p_{st})/V(p)$
Parse1	0.406	100	328	0.00018	0.363
		151	333	0.000166	0.343
		200	324	0.000182	0.361
		249	314	0.000172	0.325
Parse2	0.305	102	313	0.000123	0.264
		145	311	0.000133	0.283
		176	313	0.000136	0.293
		210	307	0.000162	0.339
Parse3	0.041	102	331	0.000045	0.563
		151	330	0.000042	0.531
		200	328	0.000044	0.547
		250	295	0.000051	0.538
Parse4	0.001	99	332	0.000003	1.408
		154	337	0	0
		202	324	0	0
		253	294	0	0
Parse5	0.43	101	330	0.000094	0.189
		150	334	0.000091	0.186
		200	331	0.000103	0.207
		249	305	0.000111	0.198

involved extensive paging, which could have been avoided with additional memory.

5.3 Results

The execution population for each parser was clustered into approximately 100, 150, 200, and 250 strata. The execution populations for the project-scheduling system were clustered into approximately 100, 150, and 200 strata. Visual inspection of the clusters indicated that a significant number of failures were isolated in very small clusters of one to several elements. (Figure 1 shows some of the 201 clusters identified in the 60-failure execution population of the project-scheduling system.) In all cases, the actual sample size was approximately one-third of the population size. The results obtained with the parsers are summarized in Table I. The results for the project-scheduling system are summarized in Table II.

5.4 Analysis

In 37 of the 38 cases summarized in Tables I and II, the stratified estimator p_{st} was substantially more accurate than the sample proportion p. The average relative-efficiency $V(p_{st})/V(p)$ over all cases was 0.38. Table II suggests that the gain from stratification increases with the number of

```
4 576 4 904 54
3 606 146 920
16 405 122 133 177 510 563 575 608 52 651 682 689 772 808 995 28
7 143 *190 218 641 549 848 407
1 767
3 723 853 531
9 84 241 274 414 491 645 702 340 318
1 *222
1 *854
1 14
8 111 114 157 580 664 528 821 754
4 104 267 658 960
8 26 191 468 568 738 228 978 401
2 107 353
1 *351
2 545 622
1 *266
4 234 *358 524 685
12 113 225 232 269 474 493 592 712 778 815 *994 673
9 105 847 556 600 757 929 959 967 53
2 *402 *22
1 *728
1 *309
3 *633 *117 *759
2 343 *242
1 *643
1 979
2 *603 *733
3 20 880 932
4 8 278 362 243
4 290 892 931 326
12 57 64 717 515 555 560 588 619 708 812 862 25
```

Fig. 1. Some of 201 clusters identified in the 60-failure execution population of the project-scheduling system. The first number of each cluster is the cluster size; other numbers are execution labels. The label of each failed execution is preceded by an asterisk.

strata employed. Such a tendency is not evident in Table I, however.[8] The

[8]Note that with the method of allocation we used there is a tendency for the computed total sample size to be lower for a large number of strata than for fewer strata. This makes it more difficult to see the true effect of the number of strata on $V(p_{st})/V(p)$, because in our experience the performance of p_{st} relative to p tends to improve as the total sample size increases (the same sample size being used with both estimators).

Table II. Experimental Results for Project-Scheduling System (#*Strata* is the actual number of strata, and *Sample Size* is the actual sample size)

Failures/Defect	P	#Strata	Sample Size	$V(p_{st})$	$V(p_{st})/V(p)$
1	0.006	100	329	0.000004	0.345
		151	328	0.000003	0.204
		201	318	0.000003	0.26
2	0.012	98	333	0.000017	0.699
		151	331	0.000011	0.454
		202	328	0.000013	0.53
3	0.018	99	325	0.000023	0.634
		152	331	0.000015	0.424
		200	318	0.000014	0.369
4	0.024	100	327	0.000023	0.47
		150	329	0.00002	0.424
		201	321	0.000013	0.262
5	0.03	99	330	0.000034	0.576
		151	333	0.000022	0.37
		200	322	0.000022	0.351
10	0.06	98	329	0.000047	0.408
		151	331	0.000042	0.368
		201	320	0.000039	0.323

average, over all programs and execution populations, of the relative efficiency achieved with the maximum number of strata was 0.32. The case of `Parse4` is noteworthy. For 99 strata the relative efficiency was 1.4, by far the worst performance by stratified sampling that we have observed with any program. For 154, 202, and 253 strata, however, the variance of the stratified estimator dropped to zero! Note that there was only one failure in `Parse4`'s execution population. When 99 strata were formed, it was clustered with 23 other executions. In the other three clusterings, it was *isolated* in a cluster by itself.

To illustrate the effect of stratified sampling, Figure 2 shows histograms of empirical sampling distributions of p and p_{st}. Each distribution was obtained with 1000 samples of size 328 from the six-failure execution population of the project-scheduling system; p_{st} was used with 151 strata. It is evident that the distribution of p_{st} is much more tightly concentrated around the true failure-proportion 0.006 than is the distribution of p. The gaps in the histograms are due to the fact that only a small number of distinct estimates are possible with just six failures in the population.

5.5 Summary of Experimental Results

The results of our experiments suggest the following conclusions:

__Clustering of executions based on dissimilarity of branch traversal profiles *can* isolate certain failures in programs of at least several thousand lines.

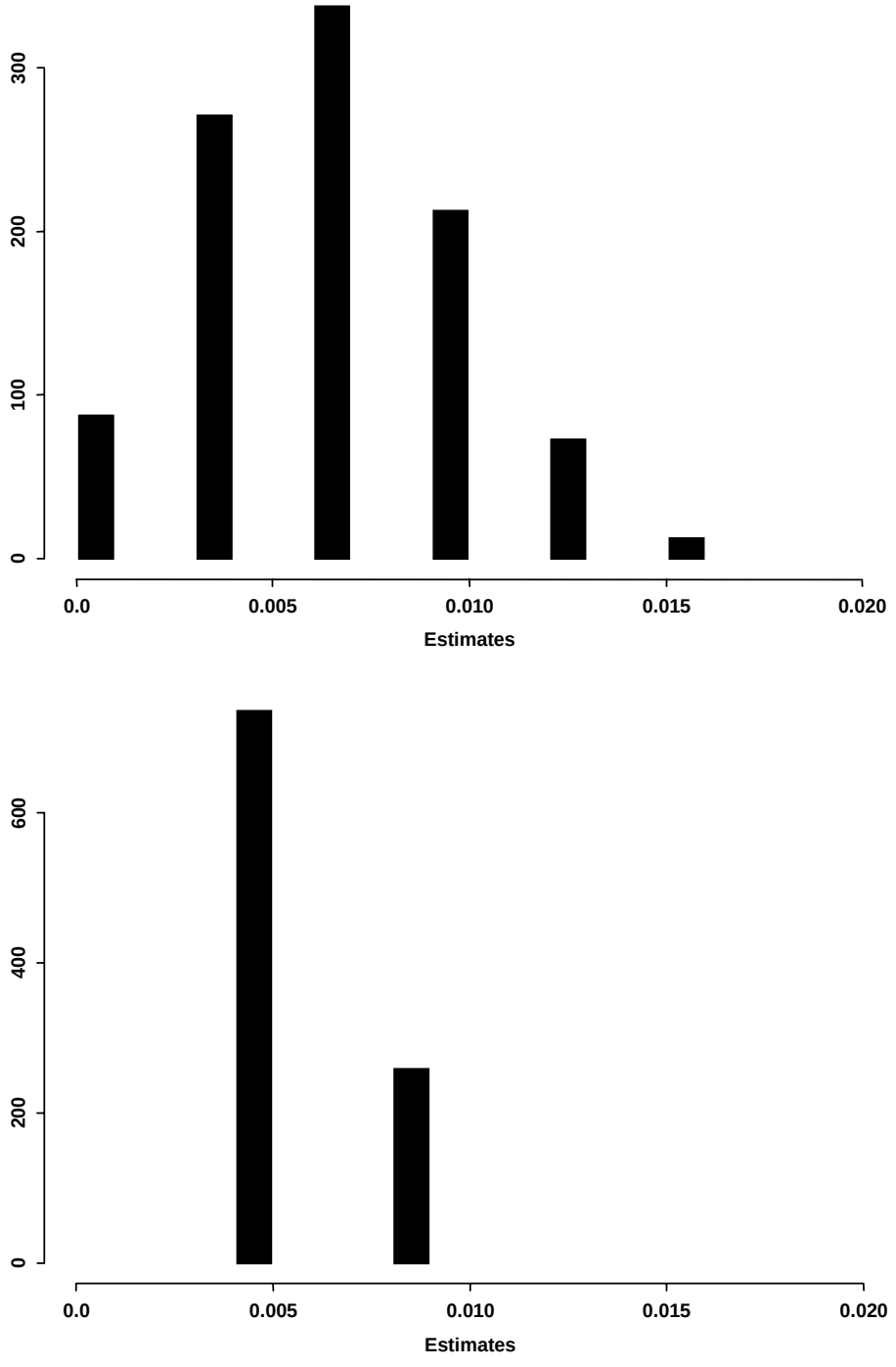

Fig. 2. Histograms of empirical distributions of p (top) and p_{st} (bottom). Each distribution was obtained with 1000 samples of size 328 from the six-failure execution population of the project-scheduling system; p_{st} was used with 151 strata.

—The computational cost of such clustering is acceptable for programs of at least this size.

—For estimating failure frequency, stratified random sampling of the clusters *can be* significantly more accurate than simple random sampling.

Substantial further experimentation is required before general conclusions can be drawn about the efficacy of our approach and the criteria for its applicability.

6. DISCUSSION OF ASSUMPTIONS

The principal assumptions underlying our approach to estimating software reliability are as follows:

(1) The future reliability of software can be predicted based upon its reliability during operational testing.

(2) It is possible to identify ways of profiling program executions such that the occurrence of program failures will often be reflected by unusual profile features.

(3) The cost saved by reducing the number of executions that must be checked for conformance to specifications is greater than the cost of the additional data collection and analysis entailed by our approach.

For assumption (1) to hold, there must be regularity in how the software is used, which persists after operational testing. Such long-term regularity is often evident in summary statistics of survey populations, if not in individuals. Statistical regularity is also likely to be evident in the usage patterns of a population of software users. Nevertheless, usage of a software system can change over time due to changes in the user population, new applications of the system, or other factors, and such changes call for reassessing its reliability. Feature usage can be monitored automatically to detect changes. To guard against unobserved changes in usage, it is prudent to reestimate reliability periodically. Note that it is possible for usage to change substantially without this affecting reliability, e.g., because most features were implemented correctly.

Assumption (2) is related to the assumptions underlying synthetic testing research. All synthetic testing techniques are based on an assumption that certain features of program executions (or of program inputs) are relevant to the occurrence of failures. The results of Section 5 lend some support to assumption (2), but further experiments are needed before broad conclusions can be drawn.

We believe assumption (3) is likely to hold when conformance to specifications must be checked manually, because of the painstaking attention to detail that is required.

7. COSTS OF THE APPROACH

The cost of checking executions for conformance to requirements is usually a large part of the overall cost of software testing. We have provided evidence that stratified sampling can reduce the number of executions that must be checked to estimate software reliability accurately. However, our approach entails costs that other testing techniques do not, such as the costs of

(1) arranging and conducting beta testing,

(2) capturing, re-creating, and profiling operational executions,

(3) analyzing execution profiles, and

(4) sampling and computing estimates.

Activities (2)–(4) can be largely automated, so the personnel time they require is negligible compared to that required to check executions. Manual evaluation of executions may take days or weeks. Although cluster analysis of profiles is computer intensive, we believe it will generally require much less time, even for large programs. Capture/replay of executions entails run-time overhead and either implementation costs or, if commercial tools are used, licensing fees. Auxiliary hardware may be necessary in some cases, e.g., for monitoring a real-time system.

The largest costs entailed by our approach to reliability estimation are likely to be those associated with operational testing in general. Considerable effort may be needed to arrange for a beta version of software to be used in the field. A developer may have to offer incentives to obtain the cooperation of users. Most importantly, operational testing delays the release of a product significantly. To expedite the delivery of software, many developers choose to omit operational testing. This is risky, however, because the short-term benefit may be negated by the long-term consequences of delivering an unreliable product. To make well-informed decisions about whether software is ready for general release, reliability measurements are necessary. The only valid basis for making such measurements is operational testing, because reliability depends upon usage.

Once software has been released and is in regular use, it may be relatively easy to capture operational inputs. Consequently, the cost of *reestimating* its reliability by stratified sampling may be significantly lower than the cost of estimating it the first time.

8. RELATED WORK

Our use of stratified sampling for estimating software reliability is related to the idea of *partition testing* or *subdomain testing* of software. Partition testing is a framework for selecting a test set of manageable size that exercises a program thoroughly. It involves dividing a program's input domain into a number of *subdomains* and then selecting a few test cases (often one) from each of them. The subdomains are chosen so that the

inputs within each one are treated similarly by the program. (The subdomains may or may not actually form a true partition of the input domain—a collection of disjoint, nonempty sets whose union is the entire domain.) Many testing strategies can be viewed as forms of partition testing, including functional testing [Howden 1980], control flow coverage [Adrion et al. 1982], data flow coverage [Rapps and Weyuker 1985], mutation testing [DeMillo et al. 1978], and partition analysis [Richardson and Clarke 1981].

The software testing and reliability research most closely related to ours concerns probabilistic or statistical approaches toward partition testing. Some of this work focuses on defect detection, the rest on reliability estimation.

In Duran and Ntafos [1984], Hamlet and Taylor [1990], and Weyuker and Jeng [1991], partition testing is compared to random testing (simple random sampling from a program's entire input domain) with respect to the probability that at least one failure occurs during testing.[9] A form of partition testing is evaluated in which inputs are selected randomly from each subdomain of a partition. Different combinations of partition size, subdomain failure probabilities, overall failure probability, subdomain execution probabilities, and subdomain sample sizes are considered. Duran and Ntafos conclude that random testing is often more cost effective than partition testing [Duran and Ntafos 1984]. Hamlet and Taylor [1990] and Weyuker and Jeng [1991] conclude that partition testing is significantly more effective than random testing only when one or more subdomains have a relatively high failure probability. Hamlet and Taylor also stipulate that these subdomains must have low execution probability. Note that similar conditions also tend to make a stratified estimator of a program's failure frequency efficient. Although none of these papers consider the variance of reliability estimators, Weyuker and Jeng do question whether the probability that at least one failure occurs during testing is an adequate measure of partition testing's effectiveness.

Frankl and Weyuker explore the defect-revealing ability of partition-testing criteria under random selection from subdomains. In Frankl and Weyuker [1993a], they define the *properly covers* relation between criteria and prove that if criterion C_1 properly covers criterion C_2 and if one test is selected randomly and uniformly from each subdomain then the probability that C_1 will reveal at least one defect is no less than the probability that C_2 will do so. Frankl and Weyuker [1993b] prove that if C_1 properly covers C_2 then the expected number of failures discovered with C_1 is no less than that discovered with C_2. They also relate a number of partition-testing criteria in terms of the properly covers relation.

Thévenod-Fosse and Waeselynck [1993] examine the use of probabilistic test generation to satisfy behavioral and structural criteria for fault detection. They emphasize automatically generating enough tests so that

[9]Duran and Ntafos [1984] also consider the expected number of failures during testing.

each element of a testing criterion is addressed by several test cases. Although they call their approach "statistical testing," it does not involve estimating reliability. They describe it as follows:

> The *second direction* [which the authors pursue] involves larger test sets that should be tedious to determine manually; hence the need for an automatic generation of test sets. And this is the *motivation of statistical testing* designed *according to a criterion*, which aims to combine the information provided by imperfect (but not irrelevant) criteria with a practical way of producing large test sets, that is, a random generation.

Thévenod-Fosse and Waeselynck apparently view the evaluation of tests as relatively *inexpensive*. They describe applications of their approach in which behavioral and structural testing criteria were successfully employed to reveal known defects.

Techniques for using partition testing to estimate software reliability are proposed in Brown and Lipow [1975], Duran and Wiorkowski [1980], Miller et al. [1992], Nelson [1978], Schick and Wolverton [1978], Thayer et al. [1976], and Tsoukalas et al. [1993]. In a survey of reliability models, Schick and Wolverton suggest the possibility of using stratified sampling to estimate reliability:

> There are clearly numerous methods possible for sampling. For example, one might want to use a stratified sampling approach. Even cost can enter here ....One might attach a cost to the length of the running time and use stratified sampling with cost [Schick and Wolverton 1978].

Schick and Wolverton do not pursue this idea. Other papers describe techniques for estimating reliability that resemble conventional stratified sampling. None of the papers is explicitly concerned with estimator efficiency, employs cluster analysis for forming partitions, or applies a stratified reliability estimator to real programs.

To account for operational usage, Brown and Lipow present a reliability estimator

$$\hat{R} = 1 - \sum_i \frac{f_i}{n_i}\pi_i$$

where i indexes a partition of the input domain; f_i is the number of failed tests from subdomain i; n_i is the total number of tests from subdomain i; and π_i is the operational probability of an input from subdomain i [Brown and Lipow 1975]. This is essentially a stratified estimator, although Brown and Lipow do not fully specify a corresponding sampling design. The variance of $\hat{R}$ is not considered in Brown and Lipow [1975]. Thayer et al. [1976] consider $\hat{R}$ in more detail. They assume it will be used with simple random sampling within subdomains and claim the estimator is unbiased—provided that all subdomains are sampled. This claim is incorrect, however:

$\hat{R}$ is generally *biased* even when all subdomains are sampled.[10] This is because all elements of a subdomain are treated as *equally likely* to arise in operational usage, whether or not they actually are.[11] For example, suppose that $\hat{R}$ is employed with a single subdomain, equal to the entire input domain. Then $\pi_1 = 1$, and $\hat{R}$ actually estimates the proportion of *all possible inputs* that do not cause failure. This proportion is different from the operational frequency of success unless all possible inputs are equally likely to arise in operational use, a condition that seldom holds. It is conceivable that subdomains could be delimited so as to make the bias of $\hat{R}$ small, but $\hat{R}$ is not certain to be unbiased unless each subdomain has size one. Thayer et al. present a variance formula, variance estimator, and sample-allocation formula for $\hat{R}$.[12] They do not experimentally evaluate the efficiency of $\hat{R}$.

Duran and Wiorkowski [1980] derive upper confidence bounds on a program's failure probability for the special case where no failures occur during testing. They derive bounds for random testing and (randomized) partition testing; these are approximately equal when subdomain sample sizes are proportional to subdomain execution probabilities. Tsoukalas et al. [1993] derive confidence bounds on the mean *failure cost* of a run, for random testing and (randomized) partition testing In the case of partition testing, they assume that an input partition is given and that the cost of a failure is uniform within a subdomain. Based on simulations with randomly generated partitions, they conclude that their methods generally yield tighter confidence bounds for partition testing than for random testing. Tsoukalas et al. also present a stratified estimator, based on $\hat{R}$, for the mean failure cost of a run:

$$\hat{C}_p = \sum_{i=1}^{k} c_i \frac{f_i}{n_i} \pi_i$$

Here i indexes a partition of the input domain; c_i is the failure cost for domain i; and f_i, n_i, and π_i are defined as for $\hat{R}$. The variance of $\hat{C}_p$ is not investigated in Tsoukalas et al. [1993].

Miller et al. [1992] present a stratified estimator of a program's failure probability, for the special case that no failures occur during testing. The

[10]These remarks correct ones in Podgurski et al. [1993], which repeated the claim from Thayer et al. [1976] that $\hat{R}$ is unbiased (assuming all subdomains are sampled).

[11]Note that this is not the case with the stratified sampling designs we employ. These designs are intended for sampling from a concrete population of operational executions, not for sampling from a program's entire input domain. Typically, many possible inputs will not occur during a period of operational use, and others may occur multiple times.

[12]Their allocation method closely resembles Neyman's method of optimal allocation in stratified sampling [Neyman 1934].

authors use partitioning of a program's input domain to account for operational usage, as in Brown and Lipow [1975]; they do not consider their estimator's variance. As with $\hat{R}$, the elements of a subdomain are treated as equally likely to occur in operational use. The estimator incorporates *prior assumptions* about the probability of failure.

9. CONCLUSIONS

We have presented a new approach to estimating software reliability, which is based on operational testing and which uses stratified sampling to reduce the number of program executions that must be checked manually for conformance to requirements. Captured executions are stratified automatically, by cluster analysis of detailed execution profiles. Experimental results were reported that suggest this approach is computationally feasible, can isolate failures, and can significantly reduce the cost of estimating software reliability. Automatic clustering of captured executions provides a concrete way to apply the intuition underlying proposed methods of partition testing, for which it is often difficult to generate test data synthetically. Stratified sampling of the resulting clusters provides a means of integrating synthetic testing concepts with those of software reliability estimation.

10. FUTURE WORK

To ascertain the utility of our approach, it is necessary to replicate the experiments described in Section 5 with other programs. Applying the approach to different types of programs will help to clarify the conditions for its applicability. The experiments described here may also be extended in several directions. It is natural to consider profiling other aspects of program execution besides control flow, including function execution [Howden 1980], data flow [Rapps and Weyuker 1985], boundary values [Adrion et al. 1982], message passing and dynamic binding in object-oriented programs [Harrold et al. 1992], event sequencing in concurrent programs [Taylor et al. 1992], and mutation coverage [DeMillo et al. 1978]. Likewise, other dissimilarity metrics, clustering algorithms, and stratified sampling techniques merit investigation.

Stratified sampling is only one example of how *auxiliary information* about a population can be used to improve the efficiency of sampling. A large body of other sampling designs and estimators has been developed to exploit such information [Sarndal et al. 1992; Thompson 1992]. Some of these, such as regression estimators, may be applicable to software reliability estimation. Similarly, other multivariate data analysis techniques besides cluster analysis, such as multidimensional scaling and logistic regression, might be useful for analyzing and reducing program profile data. In this article, we have focused on improving the efficiency of software reliability estimation. However, automated analysis of execution profiles might also be useful for revealing software defects. In this context, it could

be used to filter out executions with unusual profiles on the assumption that they are most likely to reveal failures.

APPENDIX

A. WHEN STRATIFICATION IS EFFECTIVE

In this appendix, we mathematically characterize conditions under which stratified random sampling is more efficient than simple random sampling for estimating the proportion of a program's executions that fail. Modifying formula 3.7.26 of Sarndal et al. [1992], which applies to estimated totals, yields the following relationship between the variance of the sample proportion $p = \sum_{i=1}^{n} y_i/n$ under simple random sampling and the variance of the estimator p_{st} under stratified random sampling with proportional allocation (see Section 2):

$$V(p) = V_{pr}(p_{st}) + \frac{N - n}{n(N - 1)} SSD$$

where N is the population size; n is the total sample size for each estimator; and

$$SSD = \sum_{h=1}^{H} W_h(P_h - P)^2 - \frac{1}{N} \sum_{h=1}^{H} (1 - W_h)\sigma_h^2$$

Here H is the number of strata; $W_h = N_h/N$ is the relative size of stratum h; P_h is the proportion of failures in stratum h; P is the population failure proportion; and σ_h^2 is the variance of stratum h. The "sums-of-squares difference" SSD is a difference of two components: the first characterizes variation between strata, and the second characterizes variation within strata. We see that $V(p) - V_{pr}(p_{st})$ is proportional to SSD and that p_{st} is more efficient than p provided that SSD is positive and $1 \leq n < N$. By letting F be the set of labels of *failure strata* (strata containing failures) and by partitioning sums, we obtain

$$SSD = \sum_{h \in F} W_h(P_h - P)^2 + \sum_{h \notin F} W_h(P_h - P)^2$$

$$- \frac{1}{N} \sum_{h \in F} (1 - W_h)\sigma_h^2 - \frac{1}{N} \sum_{h \notin F} (1 - W_h)\sigma_h^2.$$

Since $P_h = \sigma_h^2 = 0$ for any $h \notin F$, we have

$$SSD = \sum_{h \in F} W_h(P_h - P)^2 + \sum_{h \notin F} W_h P^2 - \frac{1}{N} \sum_{h \in F} (1 - W_h)\sigma_h^2.$$

By expanding the squared factor in the leftmost sum and then simplifying, we find

$$SSD = \sum_{h \in F} W_h P_h^2 - \frac{1}{N} \sum_{h \in F} (1 - W_h)\sigma_h^2 - P^2.$$

This may be rewritten symbolically as

$$SSD = SSF - SSW - P^2$$

where SSF is a weighted sum of squared stratum failure proportions, and SSW is an *oppositely* weighted sum of variances within strata. Whether p_{st} is more efficient than p and by how much depends on whether and by how much SSF exceeds $SSW + P^2$. Note that the variance σ_h^2 in stratum h is

$$\sigma_h^2 = \frac{N_h}{N_h - 1} P_h(1 - P_h)$$

if $N_h \geq 2$. (We have $\sigma_h^2 = 0$ for $N_h = 1$.) Differentiation shows that the maximum value of σ_h^2 is attained when $P_h = 1/2$. Hence, $\sigma_h^2 \leq 1/2$, which implies that

$$SSW = \frac{1}{N} \sum_{h \in F} (1 - W_h)\sigma_h^2 \leq \frac{1}{2N} \sum_{h \in F} (1 - W_h)$$

$$< \frac{|F|}{2N}.$$

The conditions under which stratified sampling is beneficial for estimating P are difficult to characterize intuitively, because of dependencies between factors like P_h and σ_h^2. Results we have obtained from simulations and from experiments with actual programs (see Section 5) suggest the following rough guidelines:

> Stratified sampling is beneficial for estimating a program's failure frequency when a moderate-to-high proportion of all failures are distributed among strata whose individual failure proportions are themselves moderate to high. Hence, when few failures are expected, it is desirable to use many small strata.

B. CLUSTERING ALGORITHMS

The initial choices made by a hierarchical clustering algorithm may prevent it from finding a good clustering into a given number of clusters. Hence, we first judged partitioning methods to be more appropriate for stratified sampling, where the number of strata is chosen to maximize estimator efficiency. In the experiments reported in Podgurski et al. [1993],

program executions were stratified using the partitioning program PAM (for Partitioning Around Medoids) developed by Kaufman and Rousseeuw [1990]. The inputs to PAM were dissimilarities computed with the program DAISY developed by the same authors. PAM takes the desired number k of clusters as input. It searches for k representative objects called *medoids* using an iterative relocation algorithm called the *k-medoid algorithm*. PAM first selects a set of initial representatives and then tries to improve this set. For each selected object a and unselected object b, it determines the effect of swapping a and b on the average dissimilarity $\overline{d}$ between objects and their closest representative. If any reduction in $\overline{d}$ is possible, PAM makes the swap causing the greatest reduction and then tries to improve the new set of representatives. Otherwise, it stops. Clusters are formed by associating each data object with the nearest medoid.

The running time of iterative-relocation algorithms like the k-medoid algorithm grows quickly with the population size and the desired number of clusters [Kaufman and Rousseeuw 1990]. This is problematic for stratification of program executions, which may entail clustering thousands of executions into hundreds of clusters. In some cases, we were unable to cluster a population of executions with PAM in over 30 hours of computation time. In the experiments reported here, a two-stage clustering algorithm was used that combines aspects of hierarchical clustering and partitioning. In the first stage, the k-medoid algorithm is used to cluster a population into a relatively small number of clusters, say 20. In the second stage, the k-medoid algorithm is applied to each of the first-stage clusters. The number of subclusters created from each first-stage cluster is chosen to be proportional to the internal dissimilarity of that cluster. Our experiments indicate that two-stage partitioning is several times faster than ordinary partitioning.

Our experiments suggest that much of the benefit of clustering execution profiles derives from isolating unusual executions in singleton clusters, rather than from grouping failures together. If this is generally true, then it may be preferable to use relatively inexpensive heirarchical clustering algorithms instead of partitioning algorithms. We are currently investigating this issue.

REFERENCES

ADRION, W. R., BRANSTAD, M. A., AND CHERNIAVSKY, J. C. 1982. Verification, validation, and testing of computer software. *ACM Comput. Surv. 14*, 2 (June), 159–192.

ANDERBERG, M. R. 1973. *Cluster Analysis for Applications*. Academic Press, Inc., New York, NY.

BROWN, J. R. AND LIPOW, M. 1975. Testing for software reliability. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, CA, Apr.). 518–527.

COCHRAN, W. G. 1977. *Sampling Techniques*. John Wiley & Sons, Inc., New York, NY.

DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1978. Hints on test data selection: Help for the practising programmer. *Computer 11*, 4, 34–41.

DURAN, J. W. AND WIORKOWSKI, J. J. 1980. Quantifying software validity by sampling. *IEEE Trans. Reliab. R-29*, 2 (June), 141–144.

DURAN, J. W. AND NTAFOS, S. C. 1984. An evaluation of random testing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July), 438–444.

FRANKL, P. G. AND WEYUKER, E. J. 1993a. A formal analysis of the fault detecting ability of testing methods. *IEEE Trans. Softw. Eng. 19*, 3, 202–213.

FRANKL, P. G. AND WEYUKER, E. J. 1993b. Provable improvements on branch testing. *IEEE Trans. Softw. Eng. 19*, 10 (Oct.), 962–975.

HAMLET, D. AND TAYLOR, R. 1990. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng. 16*, 12 (Dec. 1990), 1402–1411.

HARROLD, M. J., MCGREGOR, J. D., AND FITZPATRICK, K. J. 1992. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering* (ICSE '92, Melbourne, Australia, May 11–15), T. Montgomery, Ed. ACM Press, New York, NY, 68–80.

HOWDEN, W. E. 1980. Functional program testing. *IEEE Trans. Softw. Eng. SE-6*, 2 (Mar.), 162–169.

KAUFMAN, L. AND ROUSSEEUW, P. J. 1990. *Finding Groups in Data*. John Wiley & Sons, Inc., New York, NY.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ.

MILLER, K. W., MORELL, L. J., NOONAN, R. E., PARK, S. K., NICOL, D. M., MURRILL, B. W., AND VOAS, J. M. 1992. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. Softw. Eng. 18*, 1 (Jan. 1992), 33–43.

MUSA, J. D. 1993. Operational profiles in software-reliability engineering. *IEEE Softw.* (Mar.), 14–32.

NELSON, E. N. 1978. Estimating software reliability from test data. *Mircroelectron. Reliab. 17*, 67–74.

NEYMAN, J. 1934. On two different aspects of the representative method: The method of stratified sampling and the method of purposive selection. *J. Royal Stat. Soc. B. 97*, 558–606.

PODGURSKI, A. AND YANG, C. 1993. Partition testing, stratified sampling, and cluster analysis. *SIGSOFT Softw. Eng. Notes 18*, 5 (Dec. 1993), 169–181.

RAPPS, S. AND WEYUKER, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 367–375.

RICHARDSON, D. J. AND CLARKE, L. A. 1981. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference on Software Engineering* (Los Alamitos, CA). 244–253.

SARNDAL, C.-E., SWENSSON, B., AND WRETMAN, J. 1992. *Model Assisted Survey Sampling*. Springer-Verlag, New York, NY.

SCHICK, G. J. AND WOLVERTON, R. W. 1978. An analysis of competing software reliability models. *IEEE Trans. Softw. Eng. SE-4*, 2 (Mar.), 104–120.

TAYLOR, R. N., LEVINE, D. L., AND KELLY, C. D. 1992. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng. 18*, 3 (Mar. 1992), 206–215.

THAYER, T. A., LIPOW, M., AND NELSON, E. C. 1976. Software reliability. Tech. Rep. TRW-SS-76-03. TRW.

THÉVENOD-FOSSE, P. AND WAESELYNCK, H. 1993. STATEMATE applied to statistical software testing. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis* (ISSTA, Cambridge, MA, June 28–30, 1993), T. Ostrand and E. Weyuker, Eds. ACM Press, New York, NY, 99–109.

THOMPSON, S. K. 1992. *Sampling*. John Wiley & Sons, Inc., New York, NY.

TSOUKALAS, M. Z., DURAN, J. W., AND NTAFOS, S. 1993. On some reliability estimation problems in random and partition testing. *IEEE Trans. Softw. Eng. 19*, 7 (July), 687–697.

WEYUKER, E. J. AND JENG, B. 1991. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng. 17*, 7 (July 1991), 703–711.

Received: March 1994; revised: December 1994, October 1996, and July 1998; accepted: September 1998