Partition Testing, Stratified Sampling, and Cluster Analysis*

Andy Podgurski
Charles Yang
Computer Engineering and Science Department
Case Western Reserve University†

Wassim Masri
Picker International, NMR Division‡

Abstract

We present a new approach to reducing the manual labor required to estimate software reliability. It combines the ideas of *partition testing* methods with those of *stratified sampling* to reduce the sample size necessary to estimate reliability with a given degree of precision. Program executions are stratified by using automatic *cluster analysis* to group those with *similar features*. We describe the conditions under which stratification is effective for estimating software reliability, and we present preliminary experimental results suggesting that our approach may work well in practice.

1 Introduction

Partition testing or **subdomain testing** comprises a broad class of software testing methods that call for dividing a program's input domain into **subdomains** and then selecting a small number of tests (usually one) from each of them.[1] [2] Each subdomain is defined so that the inputs it contains are treated *similarly* by the program, in some sense. It is assumed that this similarity makes it likely that if the program fails on one input in a subdomain then it also fails on a significant portion of the others. To *economize* on tests, subdomain testing requires the selection of only a few representatives from each subdomain. Like many testing methods, subdomain testing is used for two distinct, but often confused, purposes: to reveal a program's defects and to assess its reliability.[3] Unfortunately, partition testing has not been proven efficacious for either purpose, despite considerable research. Duran and Ntafos [Dura84] and Hamlet and Taylor [Haml90] have concluded that partition testing is not particularly effective for detecting defects, although Weyuker and Jeng are more sanguine [Weyu91]. Partition testing *per se* cannot provide *objective* assessments of software reliability, because it does not incorporate a well-founded approach to scientific inference. These facts are disappointing insofar as partition testing has a strong intuitive appeal.

Statistical inference can provide objective estimates of software reliability. Several researchers have investigated using partition testing ideas to estimate reliability statistically (see Section 10). Some have proposed partitioning a program's input domain as a way of approximating a program's **operational input distribution**—the hypothetical probability distribution of its operational inputs [Brow75, Mill92]. Others have evaluated the utility of partitioning for deriving a confidence bound on a program's reliability [Dura80, Tsou91]. Still others did not state what they hoped to achieve! A significant obstacle to applying any of this work is the difficulty of actually *constructing* a parti-

*Professor Podgurski's research was supported by NSF Research Initiation Award CCR-9009375

†10900 Euclid Ave., Cleveland, Ohio 44106

‡5500 Avion Park Dr., Highland Heights, Ohio 44143

[1] "Input" refers here to the complete collection of external values required to execute a program, *not* to the elements of this collection.

[2] Strictly speaking, the term "subdomain testing" is more general than "partition testing", because the elements of a set partition are *disjoint* subsets whereas subdomains are not necessarily disjoint. Several testing methods use *overlapping* subdomains.

[3] By **reliability** we mean any objective, quantitative measure of how well a program satisfies its functional requirements. Reliability measures are usually more meaningful to software users than are characterizations of program *form*, e.g., defect counts.

SIGSOFT '93/12/93/CA, USA
© 1993 ACM 0-89791-625-5/93/0012...$1.50

tion of a program's inputs that is suitable for *probability sampling*. The lack of reported applications is telling evidence of this.

Partition testing bears a strong resemblance to a well-known statistical technique for estimating parameters of finite populations: **stratified sampling**. The principal purpose of stratified sampling is to increase the **efficiency** of estimation (the precision achievable with a given sample size), and considerable gains in efficiency are often realized with it Basic stratified sampling calls for partitioning a population into subpopulations, called **strata**. Strata are formed by grouping elements having similar values of one or more auxiliary variables, called **stratification variables**, whose values for each population element are already known or easily determined. It is intended that this makes each stratum relatively *homogeneous* with respect to the values of the study variable, whereas different strata are *heterogeneous*. A sample is drawn from each stratum and used to estimate a stratum parameter (e.g., the stratum mean). The population parameter of interest is estimated by a weighted average of the stratum estimates. The actual efficiency of a stratified sampling design is checked by computing a *variance estimate* for the stratified estimator [Coch77], using the stratified sample. Unlike (non-statistical) partition testing, stratified sampling *does* provide reasonably objective estimates, by virtue of statistical inference.

It is very desirable to reduce the sample size necessary to estimate software reliability with a given degree of precision, because a program's behavior during testing must often be evaluated *manually*, at least in part. Checking program behavior is one of the most time-consuming aspects of testing. Thus, it is natural to consider recasting partition testing as a form of stratified sampling.[4] This entails resolving the aforementioned problem of finding a practical way to construct an input partition from which to sample. In survey sampling, stratification criteria are typically simple; there is usually a single scalar stratification-variable. However, it seems clear that no such variable can adequately characterize program executions. The criteria used in partition testing to define subdomains are much more complex, because they involve program semantics. Undecidability results suggest it is infeasible to construct a partition of a program's entire domain of possible inputs automatically. Even partitioning a few hundred inputs would be extremely laborious if undertaken manually. The problem is complicated by the need to consider a program's operational usage when estimating reliability and by the need to control the number, size, and homogeneity of strata to achieve efficient stratified sampling.

[4] Stratified sampling has been proposed as a variance reduction technique in a related area, *simulation* [McGe92, Nels87].

We propose a new approach to statistical partition testing that is based explicitly on stratified sampling methodology. A program's reliability over a *concrete population of operational executions* is estimated by stratifying the population *automatically*, using a well-known multivariate data analysis technique called *cluster analysis* [Ande73, Kauf90]. Strata are formed by computing and analyzing numerical measures of similarity or dissimilarity between vectors of *feature* values characterizing program executions. Cluster analysis permits the use of stratification criteria that *generalize* ideas from partition testing research. We are considering a variety of features of program inputs and executions to identify those most likely to yield an effective stratification. For example, we have used data from *execution profiling* [Bent87] for clustering. We have obtained both mathematical and (preliminary) experimental evidence that stratified sampling based on cluster analysis can significantly lessen the cost of estimating software reliability. Moreover, it seems practical to apply this approach to real software when reliability is important. Although the approach is computationally intensive, this is offset by a reduction in the manual labor necessary to assess reliability. There is even reason to think that it may work *better* with large programs than with small ones, although this remains to be established experimentally.

In Section 2, we present some examples of partition testing methods. In Section 3, we discuss the use of finite population sampling techniques for estimating software reliability. Section 4 describes a technique for selecting a random sample of operational executions. Stratified random sampling is outlined in Section 5. In Section 6, the conditions under which stratification is effective for estimating a program's failure-frequency are derived. Section 7 briefly describes cluster analysis. We consider the kind of stratification criteria that might be effective for estimating software reliability in Section 8. The results of a preliminary experimental evaluation of our approach are presented in Section 9. Related work on probabilistic and statistical approaches to partition testing is surveyed in Section 10. Finally, conclusions and suggestions for further research are described in Section 11.

2 Examples of Partition Testing

Perhaps the most widely used form of partition testing is **functional testing**. This approach requires selecting test data to exercise each aspect of functionality identified in a program's requirements specification. The inputs that invoke a particular feature comprise a subdomain. For example, functional testing of a word proces-

sor calls for using each required feature of the program at least once. Such features might include: editing, formatting, search, and file manipulation commands, display options, type sizes and fonts, etc. Functional testing is a form of **black box** testing, because a program's internal form is not considered in the selection of test data.

Coverage testing, on the other hand, call for exercising certain elements of a program's internal structure; it is a form of **white box** testing. Typically, coverage testing methods require executing each program element of a certain type at least once. The inputs that execute a particular element form a subdomain. For example, **control flow coverage** involves executing elements of a program's control structure, such as statements, basic blocks, branches, or paths. **Data flow coverage** methods specify that certain patterns of data flow identified by data flow analysis should be induced. For example, the **all-uses** coverage criterion [Rapp85] requires that each definition-use chain in a program be exercised at least once if possible. The number of times a program element is revisited during a test is significant with some coverage testing methods. For example, **boundary-interior testing** [Howd75] entails selecting at least two tests for each loop: one that causes the loop to be entered but not iterated and another that causes the loop to be iterated at least once.

Mutation testing [DeMi79] involves creating many altered versions of a program, called **mutants**. Each mutant is created by making a small change to the original program, e.g., changing an operator symbol, variable name, or constant. These **mutations** are intended to correspond to typical programming errors; they can be viewed as "fixing" hypothesized defects. Test data is selected that distinguishes the mutants from the original program (unless they are equivalent to it). Such data is said to **kill** the mutants. If the original program contains a single defect corresponding to one of the mutants, this data will reveal it. It is hoped that test data that kills mutants will also reveal other defects as well, including complex ones. Mutation testing can be viewed as a partition testing method in which a subdomain consists of the inputs that kill a particular mutant [Haml90].

Richardson and Clarke proposed a method of partition testing called **partition analysis** that *combines* black box and white box testing [Rich81]. Their method forms subdomains using information from both a program's specification and its implementation. Essentially, partition analysis involves overlaying two partitions of a program's input domain. In one of these partitions, the elements of each subdomain are treated uniformly by the specification; in the other partition, the elements of each subdomain are treated uniformly by the implementation. The subdomains formed by partition analysis are treated uniformly by both the specification and the implementation.

3 Estimating Reliability by Sampling

A program's reliability can be quantified in several ways, e.g., by its frequency of failure, its mean time to failure, or the mean-squared deviation of its output from the desired output (if the output is numerical). In each case, reliability can be viewed as a parameter of a population of actual or potential executions associated with one or more users, operating environments, and time periods. Several authors have advocated estimating reliability statistically by *sampling* this population, e.g. [Brow75, Cho87, Curr86, Dura80, Musa87, Thay76, Weis86].[5] This intuitively-appealing approach is complicated by a number of subtle issues, however. One of these is the nature of operational usage.

If a program fails on any input, its reliability depends upon how often that input arises. Hence for a reliability estimate to be predictive, the sample must reflect future usage. It is often reasonable to assume that future usage will resemble past usage, and so estimate reliability based on the latter. However, it is not generally valid to assume that *short term* usage is typical or representative. Thus, it is advisable to estimate reliability based on an *extended* period of operational use. Cost will often preclude carefully evaluating all executions from this period, in which case it is necessary to sample them.[6] Many authors have characterized usage in terms of an *operational input distribution*, which was mentioned in the Introduction. Given an adequate approximation to an operational distribution, one may sample from it to estimate reliability. Methods have been proposed for constructing a representation of an operational distribution, e.g. [Musa93]. They require considerable effort however, which must be repeated if usage changes significantly.

We are investigating an approach to estimating software reliability in which a random sample of executions is "captured" directly from an actual population of operational runs, e.g., during beta testing. Our approach

[5] Butler and Finelli [Butl91] argue that empirical methods are inadequate for demonstrating **ultra-high reliability** (e.g., demonstrating that a program's probability of failure is less than 10^{-9}), because an adequate demonstration would require evaluating an inordinate number of program executions. However, Littlewood and Strigini question the *realism* of some ultra-high reliability requirements [Litt92].

[6] Complete system failures such as crashes are obvious, of course, and so do not require careful evaluation to detect.

is based on **finite-population sampling** methodology [Coch77, Sarn92, Sukh84]. This methodology has two attractive properties: its validity does not depend upon *modelling assumptions*, and it provides means of exploiting *auxiliary information* about a population to obtain efficient estimators. In **design-based** finite-population sampling, *explicit randomization* is used to select a **probability sample** of population elements.[7] The basic statistical properties of estimators, like *unbiasedness* and *consistency* [Sarn92], depend only on the *sampling design*, not on modelling assumptions. This is particularly important when the population is poorly understood—as is usually the case in software reliability assessment. Thus, design-based sampling is relatively free of questionable assumptions. This distinguishes it from **reliability growth modelling** [Goel85, Litt90, Musa87], the prevalent statistical approach to reliability prediction. The latter employs elaborate models intended to reflect the effect of debugging on software reliability, and is perhaps better suited for *project planning* than for validating software.

Statistical models and other auxiliary information may be used to *assist* design-based sampling without altering the basic statistical properties of estimators [Sarn92]. The sampling design provides protection against a poor model, and variance estimation is used to assess an estimator's actual precision. Stratified sampling is one example of design-based, model-assisted sampling, and there are many others. We are investigating the general problem of how auxiliary information about program executions may be exploited to estimate software reliability more efficiently. Existing sampling methodology was developed with *survey sampling* in mind. Much of this methodology may be applicable to studying software behavior, but the relevant auxiliary information is apparently quite different from that usually employed in survey sampling. For example, we use high-dimensional multivariate data for stratification, necessitating the use of cluster analysis. It is to be expected that some new sampling techniques will be required in the study of software behavior.

Note: *We shall hereafter refer to "design-based, model-assisted finite-population sampling" simply as "sampling".*

Because design-based reliability estimation does not depend upon models of the software debugging process in the way reliability growth modelling does, it is generally necessary to *re-estimate* a program's reliability when it is modified.[8] Given the frequency with which some software is modified, this may seem like an onerous requirement. However, we are investigating ways in which the cost of reassessment can be reduced when changes to software are *limited* in scope, as is often the case.[9] In some cases, resampling can be restricted to those executions that are affected by a change. In other cases, old and new software can be compared automatically over a large sample of inputs to obtain precise reliability estimates with little manual labor.

The reliability of a program may change even if the program is not modified, because of changes in its usage. Since a program's usage typically evolves over time, reliability may be viewed as a statistical *time series* [Chat84]. Special sampling techniques have been developed for studying population changes over time [Bind88]. However, abrupt and unexpected changes in usage may thwart attempts at reliability prediction. Although sampling methodology requires minimal assumptions, it is necessary to assume some regularity of usage to predict reliability. Fortunately, there is no evidence that software usage is inherently too irregular to be studied effectively using statistical methods. Prediction of all sorts entails assuming that the future will resemble the past in certain respects. It is prudent, though, to seek evidence periodically that the assumptions underlying a reliability prediction remain valid. The safest way to do this is to re-estimate reliability. A less expensive method is to instrument software to collect statistics about its own usage and to analyze these for signs of change.

When a program is intended to be used in many different environments, it is usually impractical to study its reliability in all of them. Instead, it is appropriate to *survey* reliability by conducting a second level of sampling: a probability sample of prospective users is provided with the program and the reliability it exhibits for each user is estimated. The resulting estimates are used in turn to study variations in reliability across environments. (For example, mean reliability might be estimated for different classes of users.) To gain the degree of cooperation necessary to conduct an acceptable survey, it may be necessary to offer inducements to prospective users. Traditional alpha and beta testing of software might be viewed as rudimentary software-reliability surveys.

4 Sampling Operational Executions

To employ sampling methodology to estimate a program's reliability, one must evaluate a random sample

[7] Using random sampling to estimate reliability statistically is quite different from simply selecting test cases randomly, which is sometimes called **random testing** [Dura84].

[8] However, we think it is unrealistic to assume that all steps in estimating reliability will be repeated until no failures occur.

[9] This work will be reported elsewhere.

of its operational executions. If the inputs to all of a program's operational executions can be saved, a sample of these inputs can be selected off-line and used to recreate the corresponding runs. Performance degradation and storage requirements will often preclude saving all inputs, however. Podgurski has proposed a simple scheme called **random input-logging** to circumvent this problem [Podg92]. Random input-logging involves capturing a program's input on randomly selected runs and writing it to a permanent file called the **log**. This can be done either by instrumenting the program to log its own inputs or by trapping its system calls. If random input-logging is employed during operational use of a program, the log will accumulate a random sample of operational inputs over time. The logging-probability can be adjusted to control the sample size and to minimize overhead. The program or an instrumented version of it can be reinvoked later using the logged inputs, in order to estimate its reliability or study other aspects of its behavior. Random input-logging is similar to techniques used for selecting a random sample from a file of elements [Knut81]. It can be used to implement a variety of sampling designs.

5 Stratified Random Sampling

A basic stratified-sampling design called **stratified random sampling** was employed in the experiments reported here [Coch77]. It calls for selecting a **simple random sample** (without replacement) from each stratum of a population independently. A simple random sample has a predetermined size n, and each sample of n distinct elements has the same probability $1/\binom{N}{n}$ of selection, where the sample is drawn from among N elements. Consider a population U of size N with a value y_i associated with its ith element for $i = 1, \ldots, N$. Let U have H strata and let stratum h have size N_h for $h = 1, \ldots, N$. Suppose that a simple random sample of n_h elements is drawn from stratum h. We estimate the mean $\mu = \sum_{i=1}^{N} y_i/N$ by the stratified estimator

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

where $\overline{y}_h = \sum_{i=1}^{n_h} y_{hi}/n_h$ is the sample mean for stratum h, and $W_h = N_h/N$ is the relative size of stratum h. The experimental results in Section 9 address the special case in which the study variable y is binary and y_i is 1 if and only if the ith execution of a program fails. Then the population mean is a *proportion*—the program's failure frequency. We will denote this proportion by φ and the estimator $\widehat{\mu}_{st}$ by $\widehat{\varphi}_{st}$ when it is used to estimate φ.

The estimator $\widehat{\mu}_{st}$ is **unbiased**; that is, $E(\widehat{\mu}_{st}) = \mu$.[10] The precision of an estimator $\widehat{\theta}$ of a parameter θ is usually measured by the estimator's **variance** $V(\widehat{\theta}) = E[(\widehat{\theta} - E(\widehat{\theta}))^2]$. The variance of an unbiased estimator is equal to the estimator's *mean squared error*. The variance of $\widehat{\mu}_{st}$ is

$$V(\widehat{\mu}_{st}) = \sum_{h=1}^{H} W_h^2 \frac{1 - f_h}{n_h}\sigma_h^2$$

where $f_h = n_h/N_h$ is the sampling fraction for stratum h,

$$\sigma_h^2 = \begin{cases} \sum_{i=1}^{N_h}(y_{hi} - \mu_h)^2/(N_h - 1) & \text{if } N_h > 1 \\ 0 & \text{if } N_h = 1 \end{cases}$$

is the variance in stratum h, and $\mu_h = \sum_{i=1}^{N_h} y_{hi}/N_h$ is the mean of stratum h. An unbiased estimator of $V(\widehat{\mu}_{st})$ is

$$\widehat{V}(\widehat{\mu}_{st}) = \sum_{h=1}^{H} W_h^2 \frac{1 - f_h}{n_h} s_h^2$$

where

$$s_h^2 = \frac{1}{n_h - 1}\sum_{i=1}^{n_h}(y_{hi} - \overline{y}_h)^2$$

is the sample variance in stratum h. This **variance estimator** is used to assess the efficacy of a stratification; a small variance-estimate is evidence that the stratification will improve the efficiency of estimation. Variance estimates are also used in computing confidence intervals [Coch77].

An important aspect of stratified sampling is how the total sample size n is **allocated** among the H strata. Various allocation methods exist [Sarn92]. A particularly simple one is **proportional allocation**, in which the sample size allocated to a stratum is approximately proportional to its size; that is, $n_h \approx n \cdot W_h$.[11] In the experiments reported in Section 9 we employed a modified version of proportional allocation. The clustering algorithms we used identified a number of small clusters/strata whose calculated sample size was zero. For these strata we used samples of size one instead, allowing the total sample size to increase above its initial size. We used the *final* sample size in computations, of course.

An assumption of basic stratified sampling is that the members of each population stratum are known. Determining stratum membership for all elements of a large population of program executions may be impractical. For example, this might entail instrumenting a

[10] We denote the *expected value* of a random variable X by $E(x)$.

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

production program to log its inputs on *all* executions, which might harm its performance. One way to circumvent this difficulty is to use **double sampling** [Coch77]. This involves the following steps: (1) selecting a large simple random sample S' of size n' (by random input-logging, for example); (2) using S' to estimate stratum weights precisely; and (3) subsampling from S' to obtain a stratified random sample S of total size n, which is used to estimate reliability. (Note that stratum weights can be estimated automatically, whereas estimating reliability may involve manual evaluation of program behavior.) Suppose that n'_h elements of S' are observed to be members of stratum h, for $h = 1, 2, \ldots, H$. The population proportion $W_h = N_h/N$ of elements in stratum h is estimated by the sample proportion $w_h = n'_h/n'$. The stratified subsample S is obtained by selecting n_h elements from the n'_h elements of S' belonging to stratum h, for $h = 1, 2, \ldots, H$. Thus, $n = \sum_{h=1}^{H} n_h$. For the double sampling design just described, an unbiased estimator of a population mean μ is

$$\widehat{\mu}_{ds} = \sum_{h=1}^{H} w_h \overline{y}_h$$

where $\overline{y}_h$ is the sample mean for stratum h. The variance of this estimator is

$$V(\widehat{\mu}_{ds}) = \frac{N - n'}{N n'} \sigma^2 + \sum_{h=1}^{H} \frac{W_h \sigma_h^2}{n'} \left(\frac{n'_h}{n_h} - 1 \right)$$

where $\sigma^2 = \sum_{i=1}^{N} (y_i - \mu)^2/(N-1)$ is the population variance.

6 When Stratification is Effective

In this section, 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. Specializing formula 3.7.26 of [Sarn92] 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 $\widehat{\varphi}_{st}$ under stratified random sampling with proportional allocation (see Section 5):

$$V(p) = V_{pr}(\widehat{\varphi}_{st}) + \frac{N - n}{n(N-1)} D$$

where N is the population size, n is the total sample size, and

$$D = \sum_{h=1}^{H} W_h (\varphi_h - \varphi)^2 - \frac{1}{N} \sum_{h=1}^{H} (1 - W_h) \sigma_h^2$$

Here H is the number of strata, $W_h = N_h/N$ is the relative size of stratum h, φ_h is the proportion of failures in stratum h, φ is the population failure proportion, and σ_h^2 is the variance of stratum h. We see that $V(p) - V_{pr}(\widehat{\varphi}_{st})$ is proportional to D and that $\widehat{\varphi}_{st}$ is more efficient than p provided that D 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

$$\begin{aligned} D &= \sum_{h \in F} W_h (\varphi_h - \varphi)^2 + \sum_{h \notin F} W_h (\varphi_h - \varphi)^2 - \\ &\quad \frac{1}{N} \sum_{h \in F} (1 - W_h) \sigma_h^2 - \frac{1}{N} \sum_{h \notin F} (1 - W_h) \sigma_h^2 \end{aligned}$$

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

$$D = \sum_{h \in F} W_h (\varphi_h - \varphi)^2 + \sum_{h \notin F} W_h \varphi^2 - \frac{1}{N} \sum_{h \in F} (1 - W_h) \sigma_h^2$$

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

$$D = \sum_{h \in F} W_h \varphi_h^2 - \frac{1}{N} \sum_{h \in F} (1 - W_h) \sigma_h^2 - \varphi^2$$

This may be rewritten symbolically as

$$D = SSF - SSW - \varphi^2$$

where SSF is a weighted sum of squared stratum failure proportions and SSW is an *oppositely* weighted sum of variances within strata. Whether $\widehat{\varphi}_{st}$ is more efficient than p and by how much depends on whether and by how much SSF exceeds $SSW + \varphi^2$. SSF grows with the number, relative sizes, and failure proportions of the failure strata. It grows quickly with stratum failure proportions, because these are squared. For a reasonably reliable program, φ^2 is extremely small for the same reason. SSW grows with number and variances of the failure strata but *decreases* as the relative sizes of the failure strata increase. Note that the variance σ_h^2 in stratum h is

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

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

$$\begin{aligned} SSW = \frac{1}{N} \sum_{h \in F} (1 - W_h) \sigma_h^2 &\leq \frac{1}{2N} \sum_{h \in F} (1 - W_h) \\ &< \frac{|F|}{2N} \end{aligned}$$

7 Cluster Analysis

Cluster analysis algorithms attempt to group objects so that objects within a group are more similar to each other than to objects outside the group [Ande73, Kauf90]. The **dissimilarity** between two objects is identified with the value of a distance metric applied to their feature vectors. Two commonly used distance metrics are Euclidean distance and Manhattan distance. If $\mathbf{x}$ and $\mathbf{y}$ are two feature vectors with n components, the **Euclidean distance** between $\mathbf{x}$ and $\mathbf{y}$ is

$$d_E(\mathbf{x},\mathbf{y}) = \left[\sum_{i=1}^{n}(x_i - y_i)^2\right]^{1/2}$$

The **Manhattan distance** between $\mathbf{x}$ and $\mathbf{y}$ is

$$d_M(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n}|x_i - y_i|$$

Distance metrics exist for binary, nominal, ordinal, interval, and ratio variables, as well as for mixed variables. There are two basic approaches to cluster analysis: **partitioning** methods construct a single partition of a set of objects, whereas **heirarchical** methods construct partitions of each possible size. **Agglomerative** heirarchical methods merge two clusters at each step to form a new cluster; the objects themselves are the initial clusters. **Divisive** heirarchical methods split a cluster at each step, starting with the set of all objects.

The initial choices made by a heirarchical clustering algorithm may prevent it from finding a good clustering into a given number of clusters. Hence, we judged partitioning methods more appropriate for stratification. In the experiments reported in Section 9, we stratified program executions using the partitioning program PAM (for Partitioning Around Medoids) developed by Leonard Kaufman and Peter Rousseeuw [Kauf90]. 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. First, it selects a set of initial representatives. 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; otherwise, it stops. Clusters are formed by associating each data object with the nearest medoid.

8 Stratification Criteria for Estimating Reliability

Stratification is most effective for estimating a population mean when the stratum means differ significantly but the variance within strata is small [Coch77]. The results of Section 6 indicate that in the special case of estimating the proportion of program's executions that fail, it desirable that strata containing failures have a high proportion of them. Apparently, if stratification is to be useful for estimating software reliability, stratification criteria must be identified that tend to group together program executions with similar outcomes. Of course, the circumstances causing a program to perform poorly are generally unknown during reliability estimation. The ideas of partition testing suggest how effective stratifications might be obtained, however: by grouping executions based on their easily-observed features, e.g., those that can be identified via instrumentation. The hypothesis of this approach is that executions with similar features are likely to have similar outcomes. Since a deterministic program's behavior is completely determined by its input, one might stratify based on features of inputs as well. Cluster analysis provides a general mechanism for grouping program executions based on complex feature-data.

What kinds of features of program executions should be used to stratify them? Consideration of partition testing methods suggests many possibilities. The ideas of functional testing suggest stratifying the input population of a program based on what aspects of the program's requirements specification are relevant to each input. Coverage testing methods suggest using information obtained by execution profiling, such as the execution frequencies of statements, branches, procedure calls, and data flow relationships. One might even attempt to stratify inputs based on what mutations they kill. Richardson and Clarke's work on the Partition Analysis Method [Rich81] suggests stratification criteria that *combine* information from a program's specification and its implementation. Since a specification and implementation can *both* neglect to address certain important conditions, it is wise to seek features of a program's inputs that are not reflected in either the program's specification or implementation yet are relevant to its behavior. In principle, the more information that is captured about program executions, the more effectively they can be stratified. However, the amount of information collected must be balanced against the cost of collecting and analyzing it. The performance of pattern recognition techniques like cluster analysis can actually deteriorate due to excessive feature data.

Stratified sampling will not always improve the efficiency of estimation. However, there is little to be lost

by using it when it is not expensive, because it rarely performs worse than simple random sampling. Hence, the use of stratified sampling for estimating software reliability might be amply justified were it effective in 50% of applications, say. (We reiterate that the actual efficiency of a stratified estimator may be checked by computing an estimate of its variance from the sample data.) It is plausible that certain stratification criteria might be generally effective for one type of software but not for others. As long as this were understood, such criteria might prove quite useful. The amount of variance reduction required to justify the use of stratified sampling depends on the cost of stratification and the precision required in estimation. For example, if it is crucial to obtain a narrow confidence interval for a parameter using a sample of limited size, then stratification may be a necessity.

9 Experimental Evaluation

9.1 Method

It is difficult to be certain *a priori* which stratification criteria will prove most useful for estimating software reliability. Clearly, the efficacy of any proposed criterion should be evaluated empirically. We chose to evaluate *cluster analysis of execution profiles* first, because we had the wherewithal to implement it quickly. We report here the results of preliminary experiments in which the execution counts of *conditional branches* were used for stratification. We employed branch profiling because it is relatively easy to do and because in many programs, failures are correlated with certain patterns of branching. In our experiments, the estimator $\widehat{\varphi}_{st}$, used with the stratified random sampling design and the modified form of proportional allocation described in Section 5, was compared to the sample proportion $p = \sum_{i=1}^{n} y_i/n$, used with simple random sampling. The relative precision of these estimators was assessed for execution populations of eight programs. The *same total sample size* was used to compute both estimators' variances for a particular execution population. The variance ratio $V(\widehat{\varphi}_{st})/V(p)$ (called the **design effect** [Sarn92]) was used to characterize the estimators' relative precision.[12] Note that we compared the *true* variances of the estimators, not estimated variances.

The eight subject-programs in this preliminary experiment were written by students (CE and CS seniors and graduate students) as class projects. Four were spelling checkers, assigned in a software engineering class. These were supposed to take a dictionary file and a text file as input and produce an alphabetized list of the misspelled words from the text file. Their average length was about 200 lines. The other four programs were lexical analyzers for ANSI C, assigned in a compiler design class. These were supposed to take a source program as input and produce a stream of token identifiers and corresponding lexemes. Their average length was about 1000 lines. All eight programs were written in C. None of the students knew when they wrote their programs that they would be used in our experiment. The source programs were instrumented by hand for branch-profiling, and each instrumented version was compared to the original over the input population to ensure that profiling did not affect reliability.

For the spelling checkers, 2000 news group postings were extracted and used with a single dictionary as an input population. For the lexical analyzers, 1000 C source files were obtained from Internet sources. The input files were chosen without regard for their contents. An execution was judged a failure if the output did not match that of an oracle program we developed.[13] The subject programs were chosen from among a larger group of projects because they failed on some but not all input files.[14] However, some of the selected programs failed far more often than we would expect a program undergoing formal reliability assessment to fail. To make our experiment more realistic, we modified the input population for the spelling checkers as follows. Characters that caused any spelling checker to fail excessively were identified and stripped from all but twenty of the input files, which were chosen arbitrarily. These twenty files were then distributed randomly in the input population. No changes were made to input files once any experimental results were obtained. The eight subject-programs were profiled over their entire input population. Executions that terminated abnormally (crashes) were removed from the execution population for the corresponding program. This slightly reduced the size of some programs' execution population.[15] The actual proportion φ of each program's executions that failed is shown in Table 1.

To obtain feature vectors for cluster analysis, each branch execution count was supplemented with a binary variable that was 1 if and only if the corresponding count was nonzero. This was done so that the pro-

[12] The variance of p under simple random sampling is

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

where P is the population proportion.

[13] The Unix utility program `diff` was used for comparing output files.

[14] Note that the estimators $\widehat{\varphi}_{st}$ and p each have variance *zero* when used with a program whose true failure frequency is zero.

[15] In a practical application of stratification, a separate stratum consisting of only these executions should be created. This will usually reduce estimator variance, because the added stratum will have failure-proportion 1 and variance 0.

Spelling Checker	φ	*Lexical Analyzer*	φ
Spell1	0.0613	**Lex1**	0.0130
Spell2	0.0045	**Lex2**	0.0411
Spell3	0.0040	**Lex3**	0.0610
Spell4	0.0200	**Lex4**	0.0470

Table 1: Proportion of each subject-program's executions that failed

gram DAISY, which we used for computing dissimilarities between executions (see Section 7), would weigh the distinction between execution and nonexecution of a branch as heavily as the difference between the largest and smallest number of executions of any branch. Thus, a vector of $2m$ binary and nonnegative-integer values was obtained from each vector of m branch execution counts. All the mixed-variable vectors for a subject program were input to DAISY to produce a dissimilarity matrix that was input to the clustering program PAM. DAISY used the following formula to compute the dissimilarity $d(i,j)$ between executions i and j:

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

The value $d_{ij}^{(f)}$ is the contribution of the fth variable to the dissimilarity between i and j. Let x_{if} and x_{jf} denote the values of the fth variable for object i and object j, respectively. For a binary variable, $d_{ij}^{(f)}$ is the exclusive-OR of x_{if} and x_{jf}; for an interval variable, we have

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

Variables for which the range in the denominator was zero were eliminated. Each spelling-checker's executions were clustered into 100, 200, and 300 strata and each lexical analyzer's executions were clustered into 50, 100, 150, and 200 strata. Each clustering produced strata with a range of sizes, including a number of singleton strata.

The total sample size used in our modified proportional allocation was initially equal to one-fifth the population size: 400 for the spelling checkers and 200 for the lexical analyzers. The total sample size after allocation was generally somewhat larger; this size was used in computing variances.

9.2 Results

The variance ratios $V(\widehat{\varphi}_{st})/V(p)$ obtained with the spelling checkers are shown in Table 2. The blank entries indicate attempted clusterings that took excessively long and had to be terminated before they could complete. The variance ratios obtained with the lexical analyzers are shown in Table 3.

9.3 Analysis

The results are promising. For six of the eight subject-programs (all but **Spell1** and **Lex1**), stratification resulted in substantial variance reductions. For three programs, **Spell2**, **Spell3**, and **Lex4**, there were very large reductions. The estimator $\widehat{\varphi}_{st}$ had zero variance for the **Spell3** population because the stratification was ideal: each stratum contained only failures or only successes. The variance reductions generally increase with the number of clusters used. This pattern was not observed with with **Spell1** and **Lex1**, however. The distribution of failures into clusters for **Spell1** seems to be fairly random. Failures are somewhat more localized in the **Lex1** clusterings. In the clusterings that yielded substantial variance reductions, there were generally one or more clusters with a moderate to high proportion of failures.

A few words about the efficiency of the clustering program PAM are in order. PAM was run on DEC 3000 Model 400 and 500 Alpha AXP™ workstations (manufactured by Digital Equipment Corporation), which have clock rates of 133MHz and 150MHz, respectively. These machines had 64MB of main memory. The time required to run PAM generally increased with the number of clusters requested but was also dependent on the data values. Some runs took only 25 minutes; others had to be aborted after three days. As its authors explain [Kauf90], PAM is not really designed for large data sets or for finding large numbers of clusters. We hoped PAM's limitations would be offset by the high-performance workstations we used, but it is now clear to us that other clustering methods should be used for our purposes.

	Number of Clusters		
Program	100	200	300
`Spell1`	0.97 (418)	0.97 (459)	0.99 (515)
`Spell2`	0.40 (420)	0.31 (466)	0.16 (522)
`Spell3`	0.00 (417)	0.00 (467)	
`Spell4`	0.62 (417)		

Table 2: Variance ratios $V(\widehat{\varphi}_{st})/V(p)$ for spelling-checker executions (actual sample sizes in parentheses)

	Number of Clusters			
Program	50	100	150	200
`Lex1`	1.03 (203)	0.89 (231)	0.92 (271)	1.05 (312)
`Lex2`	0.89 (203)	0.74 (219)	0.72 (240)	0.49 (273)
`Lex3`	0.90 (200)	0.79 (216)	0.75 (241)	0.63 (280)
`Lex4`	0.65 (205)	0.42 (225)	0.22 (259)	0.15 (296)

Table 3: Variance ratios $V(\widehat{\varphi}_{st})/V(p)$ for lexical-analyzer executions (actual sample sizes in parentheses)

10 Related Work

A number of authors have previously considered ways of combining the ideas of partition testing, on the one hand, with probabilistic or statistical methods, on the other. Their papers are divided into two categories: (1) those that investigate the probability of detecting defects using partition testing; and (2) those that, like this paper, apply partitioning to reliability estimation.

In the first category of papers [Dura84, Haml90, Weyu91], partition testing is compared to random testing (simple random sampling) with respect to the probability that at least one failure occurs during testing.[16] A form of partition testing is used in which inputs are drawn randomly from each subdomain of a partition. Different combinations of partition sizes, subdomain and overall failure probabilities, subdomain execution probabilities, and subdomain sample sizes are considered. Duran and Ntafos [Dura84] conclude that random testing is often more cost effective than partition testing. Hamlet and Taylor [Haml90] and Weyuker and Jeng [Weyu91] 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 the elements of these subdomains must have low execution probability. Note that similar conditions also tend to make a stratified estimator of the population failure-proportion efficient. Although none of the papers in this category 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.

In the second category of papers [Brow75, Dura80, Mill92, Nels78, Schi78, Thay76, Tsou91], various methods of using partitioning to estimate reliability are proposed. In a survey of reliability models, Schick and Wolverton suggested 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. [Schi78]

Schick and Wolverton do not pursue this idea. Other papers propose methods for estimating reliability that resemble conventional stratified sampling, although they do not mention this similarity.[17] None of these papers is explicitly concerned with variance reduction; none employs cluster analysis for forming partitions; and none applies a stratified reliability-estimator to real programs.

[16] Duran and Ntafos [Dura84] also consider the expected number of failures during testing.

[17] Stratified sampling was well-known at the time these papers were written. It was described by Neyman in 1934 [Neym34] and is discussed at length in most texts on finite population sampling.

Brown and Lipow [Brow75] present what is essentially an unbiased stratified estimator of a program's reliability. However, they do not consider this estimator's variance. Rather, they use the estimator strictly as a way of accounting for the nature of a program's operational usage. They advocate partitioning a program's input domain and associating operational probabilities with subdomains. These probabilities are used to weight subdomain estimates. Thayer, Lipow, and Nelson [Thay76] subsequently derive the variance of this estimator and present an unbiased variance estimator, although they do not suggest that the stratified reliability-estimator is more efficient than others. They do not evaluate the stratified estimator experimentally. Nelson [Nels78] presents *ad hoc* rules for estimating reliability based on partition testing.

Duran and Wiorkowski [Dura80] derive upper confidence bounds on a program's failure probability for the special case where no failures occur during testing. Bounds are derived for random testing and (randomized) partition testing, respectively, and these are found to be approximately equal when subdomain sample sizes are proportional to subdomain execution probabilities. Tsoukalas, Duran, and Ntafos [Tsou91] 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 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.

Miller *et al* [Mill92] present what is essentially a stratified estimator of a program's failure probability, for the special case that no failures occur during testing. However, the authors use partitioning only to account for operational usage, as in [Brow75]. They do not consider their estimator's variance. The estimator makes use of *prior assumptions* about the probability of failure and, unlike the stratified estimators we employ, is actually *biased* if these assumptions are violated.

In summary, the work described by Thayer *et al* [Thay76] is the most similar to ours, in that they present a design-unbiased stratified estimator of reliability, possibly for the purpose of variance reduction. Their work differs from ours in two principal respects: (1) they do not employ cluster analysis for forming partitions and (2) they do not experimentally evaluate their stratified estimator. To our knowledge, the application of cluster analysis we describe is original. It is significant because the practical application of partition testing has been hampered by the difficulty of actually partitioning a program's inputs. Cluster analysis allows this task to be automated. Moreover, it permits a much wider variety of partitioning criteria than those described in the partition testing literature. Any binary, categorical, numeric, or mixed data characterizing program executions can be used for clustering. (For example, we cluster using a mixture of binary data, characterizing program coverage, and numeric data, characterizing execution frequency.) To get different clusterings, one may vary the number of clusters, cluster diameter, dissimilarity metrics, and clustering algorithms. This is important for stratification since different clusterings yield different variance reductions. Cluster analysis also provides information about the quality of a clustering that can be used in sample allocation (determining stratum sample sizes) to minimize estimator variance.

11 Conclusion

We have introduced an approach to reducing the manual labor required to estimate software reliability. It works by reducing the sample size necessary to estimate reliability with a given degree of precision. This approach uses the ideas of *partition testing* methods to create designs for *stratified sampling.* It thereby unifies ideas from software testing and statistical reliability assessment. To form strata, automatic *cluster analysis* methods are used to group program executions that have similar features. We have described the conditions under stratification is effective for estimating reliability and reported the results of a preliminary experimental evaluation of our approach. These results suggest that stratified sampling based on clustering *execution profiles* can lead to significant reductions in estimator variance.

Much more experimentation is necessary to confirm our initial results and to explore the utility of alternative stratifications, estimators, profiling and clustering methods, etc. It is especially important to apply our approach with a variety of programs, including production ones. This is necessary to judge its generality and to identify classes of programs to which particular stratification criteria are best suited. Large programs are a stumbling block for much proposed software engineering methodology. However, we suspect that stratification may tend to work *better* with large programs than with small ones, because the former usually have a greater variety of distinct behaviors. Naturally, this requires experimental confirmation.

The notion of using multivariate dissimilarity metrics to distinguish program executions may have applications beyond stratified sampling, because it provides a means of identifying *unusual* executions. For example, a dissimilarity metric that applies to program inputs could be used to identify very unusual inputs *prior* to execution, permitting appropriate intervention.

References

[Ande73] Anderberg, M. R. *Cluster Analysis for Applications*, Academic Press, New York, 1973.

[Bent87] Bentley, J. Profilers. *Communications of the ACM*, Vol. 30, No. 7 (July 1987), pp. 587–592.

[Bind88] Binder, D. A. and Hidiroglou, M. A. Sampling in time. In *Handbook of Statistics 6: Sampling*, P. R. Krishnaiah and C. R. Rao editors, North Holland, Amsterdam, 1988.

[Brow75] Brown, J. R. and Lipow, M. Testing for software reliability. *Proceedings of the International Conference on Reliable Software* (Los Angeles, April 1975), pp. 518–527.

[Butl91] Butler, R. W. and Finelli, G. B. The infeasibility of experimental quantification of life-critical software reliability. *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems* (New Orleans, December 1991), ACM Press, New York, 1991, pp. 66-76.

[Chat84] Chatfield, C. *The Analysis of Time Series: An Introduction*, Chapman and Hall, London, 1984.

[Cho87] Cho, C. *Quality Programming*, Wiley, New York, 1987.

[Coch77] Cochran, W. G. *Sampling Techniques*, Wiley, New York, 1977.

[Curr86] Currit, P. A., Dyer, M., and Mills, H. D. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, (January 1986), pp. 3–11.

[DeMi79] DeMillo, R. A., Sayward, F. G., and Lipton, R. J. Program mutation: a new approach to program testing. *Infotech International State of the Art Report: Program Testing*, Infotech International, 1979.

[Dura80] Duran, J. W. and Wiorkowski, J. J. Quantifying software validity by sampling. *IEEE Transactions on Reliability*, Vol. R-29, No. 2 (June 1980), pp. 141–144.

[Dura84] Duran, J. W. and Ntafos, S. C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4 (July 1984), pp. 438–444.

[Goel85] Goel, A. L. Software reliability models: assumptions, limitations, applicability. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12 (December 1985), pp. 1411–1423.

[Haml90] Hamlet, D. and Taylor, R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, Vol. 16, No. 12 (December 1990), pp. 206–215.

[Howd75] Howden, W. E. Methodology for the generation of program test data. *IEEE Transactions on Computers*, Vol. c-24, No. 5 (May 1975), pp. 554–559.

[Kauf90] Kaufman, L. and Rousseeuw, P. J. *Finding Groups in Data*, Wiley, New York, 1990.

[Knut81] Knuth, D. E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison Wesley, Reading, MA, 1982.

[Litt90] Littlewood, B. Modelling growth in software reliability. In *Software Reliability Handbook*, P. Rook editor, Elsevier, New York, 1990, pp. 137–154.

[Litt92] Littlewood, B. and Strigini, L. The risks of software. *Scientific American* (November 1992), pp. 62–75.

[McGe92] McGeoch, C. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Computing Surveys*, Vol. 24, No. 2 (June 1992), pp. 195–212.

[Mill92] Miller, K. W., Morell, L. J., Noonan, R. E., Park, S. K., Nichol, D. M., Murrill, B. W., and Voas, J. M. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, (January 1992), pp. 33–42.

[Musa87] Musa, J. D., Iannino, A., and Okumoto, K. *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

[Musa93] Musa, J. D. Operational profiles in software-reliability engineering. *IEEE Software* (March 1993), pp. 14–32.

[Nels78] Nelson, E. N. Estimating software reliability from test data. *Microelectronics and Reliability*, Vol. 17 (1978), pp. 67–74.

[Nels87] Nelson, B. L. A perspective on variance reduction in dynamic simulation experiments. *Communications on Statistical Simulation*, Vol. 16, No. 2 (1987), pp. 385–426.

[Neym34] Neyman, J. On two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. *Journal of the Royal Statistical Society*, Vol. 97, pp. 558–606.

[Podg92] Podgurski, A. The role of statistical reliability assessment. *Proceedings of the 25th Symposium on the Interface: Computing Science and Statistics* (College Station, Texas, March 1992).

[Rapp85] Rapps, S. and Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4 (April 1985), pp. 367–375.

[Rich81] Richardson, D. J. and Clarke, L. A. A partition analysis method to increase program reliability. *Proceedings of the Fifth International Conference on Software Engineering* (1981), IEEE Computer Society, Los Alamitos, CA, pp. 244–253.

[Sarn92] Sarndal, C.-E., Swensson, B., and Wretman, J. *Model Assisted Survey Sampling*, Springer-Verlag, New York, 1992.

[Schi78] Schick, G. J. and Wolverton, R. W. An analysis of competing software reliability models. *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2 (March 1978), pp. 104–120.

[Sukh84] Sukhatme, P. V., Sukhatme, B. V., Sukhatme, S., and Asok, C. *Sampling Theory of Surveys With Applications*, third edition, Iowa State University Press, 1984.

[Thay76] Thayer, T. A., Lipow, M., and Nelson, E. C. *Software Reliability*, TRW technical report TRW-SS-76-03, March 1976.

[Tsou91] Tsoukalas, M. Z., Duran, J. W., and Ntafos, S. C. On some reliability estimation problems in random and partition testing. *Proceedings of the International Symposium on Software Reliability Engineering* (Austin, Texas, May 1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 194–201.

[Weis86] S. N. Weiss and E. J. Weyuker, "A generalized domain-based definition of software reliability," *Proceedings of the Workshop on Software Testing* (Banff, Alberta, July 1986) IEEE Computer Society Press, Los Alamitos, CA, pp. 98–107.

[Weyu91] Weyuker, E. J. and Jeng, B. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, Vol. 17, No. 7 (July 1991), pp. 703–711.