Abstract—This paper presents a new, statistical approach to evaluating software products that transform complex inputs into complex outputs. This approach, called multistage stratified input/output (MSIO) sampling, combines automatic clustering of multidimensional I/O data with multistage sampling and manual examination of data elements, in order to accurately and economically estimate summary measures of output data quality. We report results of two case studies in which MSIO sampling was successfully applied to evaluating complex graphical outputs.

Keywords: software testing, software reliability, complex data, stratified sampling, multistage sampling, MSIO sampling, estimation, bioinformatics.

I. INTRODUCTION

Many software products transform a complex input into an equally complex output, whose elements are in rough correspondence to the elements of the input. Examples include bioinformatics applications, health information systems, program analysis tools, document processors, drawing programs, computer games, and interactive simulators. To test such products in-house with realistic inputs, developers must either determine expected outputs in advance or manually check the actual outputs produced. Because realistic inputs and corresponding outputs are complex, both alternatives are labor-intensive and error prone. Consequently, developers often construct mainly simple test inputs, which do not resemble operational inputs, or they check complex outputs only superficially. The same issues also hamper the use of captured operational inputs [1, 2] in regression testing and software reliability estimation [3-5], which is unfortunate because such inputs reflect how end users actually employ a software product. Beta testing does address these issues, but it depends upon ordinary users to recognize and accurately report software failures to developers, which they often fail to do [6, 7].

Under assumptions that it is desirable to test software with realistic inputs and that the results should be checked carefully by developers or other application experts (e.g., bioinformatics experts or health informaticians), not just end-users, there is a compelling need for principled, workable techniques for evaluating transformations of complex inputs into complex outputs. Since complete correctness appears to be an unrealistic goal for large products, quantitative summary measures of operational output quality are needed to aid developers and their customers in deciding such questions as when a new product version is good enough for general deployment, when repairs to a deployed product are necessary, and which problems should be repaired first.

Consider, for example, an application that displays complex, annotated graphs such as biological networks [8]. Accurate estimates of, for example, the mean proportions of graph elements that have errors of specified types provide developers and users with objective evidence about its operational output-quality. Such estimates might, for instance, justify concluding that structural errors in outputs are likely to be small and rare, while stylistic errors (e.g., with colors or layout) are likely to be larger and more common. Well-chosen and accurate measures of output quality are obviously preferable to the subjective and hearsay evidence that software deployment and repair decisions are often based upon.

A software product’s overall proportion of failing executions is a familiar kind of software reliability measure, which can be estimated statistically after checking the outputs from a random sample of operational executions [4, 9, 10]. Measures of average output quality like those in the preceding example, which characterize the prevalence of erroneous elements in composite outputs, are not standard. Nevertheless, they can be also estimated statistically, after checking a random sample of output elements (e.g., graph vertices and edges) selected from a random sample of operational outputs. Random sampling and statistical estimation enable developers and users to assess output quality in a principled way yet also control assessment costs. The latter is done by adjusting sample size to balance estimator precision against developer effort.

Developers presumably won’t expend the effort needed to estimate a new product’s output quality until they think it is reasonably close to satisfying quality requirements. However, if incorrect output elements have low probability then basic sampling designs such as simple random sampling are likely to produce imprecise estimates of output quality, that is, estimates with excessive variance relative to...
the magnitude of the true value [11]. Consequently, estimated output quality may satisfy a specified requirement (e.g., a 95% upper confidence bound of 0.001 on the mean proportion of erroneous elements per output) though actual output quality does not satisfy the requirement, or the converse may hold.

To address this problem, we present a new technique, called multistage stratified I/O (MSIO) sampling, that enables developers or users to accurately estimate measures of the average quality of operational output, such as the mean proportion of an output’s elements that are erroneous, even when such elements are infrequent. MSIO sampling is based on two established ideas, multidimensional stratification of complex data and multistage sampling. We integrate these ideas to enable accurate estimation of data quality measures with a reasonable amount of human effort. To our knowledge, this has not been done previously.

MSIO sampling is most useful when occasional small errors in complex outputs can be or must be tolerated, at least temporarily. This situation is surprisingly common. For example, with entertainment applications such as computer games, small, infrequent errors are usually harmless. With office, finance, and business software, defects that can corrupt user documents or files [12, 13] are potentially serious, yet some remain unfixed for months [13, 14], e.g., because other issues take precedence. As a final example, errors in the output of important medical and scientific applications are not uncommon [15-17], and they are prioritized for attention [18], according to their likely impact, while the applications remain in use.

To evaluate MSIO sampling, we conducted two substantial case studies in which it was applied to applications that transform complex data: a program dependence graph generator and a bioinformatics application for visualizing metabolic pathways. The results indicate that MSIO sampling does in fact enable accurate estimation of output data quality at acceptable cost.

**Is it necessary?** One may wonder if it is really necessary to estimate output quality at all. Isn’t it sufficient to repair software defects revealed during operational use of a product and ensure that the defects were actually corrected (e.g. by rerunning existing tests)? The answer is no, unless the output quality is known to have been adequate before the defects were fixed. Defects revealed during a period operational use may comprise only a subset of the defects remaining in a product. Even if the former defects are repaired successfully, the product’s subsequent output quality may be inadequate if the latter defects are triggered. This is possible because the inputs to a product during different periods of operational use are subject to random variation. On the other hand, suppose that a valid statistical confidence bound on a product’s average output quality is computed based on operational testing results. Also suppose that the bound satisfies the criteria established for the product’s general release, even though one or more defects were revealed during operational testing. Then if the developer chooses to debug those defects, successful repair will ensure that the product’s output quality remains at least high enough for release, assuming the operational input distribution has not changed significantly. However, if the success of debugging is still in doubt after regression testing (e.g., because extensive changes were required), then further operational testing and output quality estimation is called for.

**Roadmap:** The remainder of the paper is organized as follows. Section II presents necessary background on sampling techniques and on multidimensional stratification. Section III introduces MSIO sampling. Section IV presents the two case studies. Section V discusses issues raised by MSIO sampling. Section VI surveys related work. Section VII concludes the paper.

II. BACKGROUND

**A. Stratified and Multistage Sampling**

Stratified sampling [19] is a classical sampling and estimation strategy used in surveys and other observational studies to accurately but economically estimate population parameters, such as the mean or total of a study variable $y$. In order to reduce estimation error without increasing overall sample size (which would increase costs), the population is partitioned into groups called strata, typically based on one or two auxiliary variables (covariates) whose values are believed to vary roughly with the value of $y$. In stratified random sampling, a simple random sample of elements is selected from each stratum and used to estimate the parameter of interest for that stratum. The stratum estimates are then combined, using a stratified estimator, to form an estimate of the population parameter. The standard stratified estimator for the population mean $\mu$ of $y$ is the weighted average

$$\hat{\mu}_{ST} = \sum_{h=1}^{L} \frac{N_h}{N} \bar{y}_h$$

where $N$ is the population size, $N_h$ is the size of stratum $h$, and $\bar{y}_h$ is mean of the sample from stratum $h$. If the $y$ values have lower variance within strata than between them, the stratified estimator will have lower variance, for a given sample size, than the mean of a simple random sample from the whole population. In this case the stratified estimator is said to be more efficient than the sample mean. Stratified estimators have other advantages too: they are unbiased and consistent, they seldom have higher variance than simple random sampling, and their variance can also be estimated from the stratified sample [19].

As an example of stratified random sampling, consider the problem of accurately estimating the mean household income in a large city. Stratified sampling could be applied by partitioning the city geographically into neighborhoods with similar housing costs (e.g., as determined from recent real estate and rent data) and then drawing a simple random sample of households from each neighborhood. If survey personnel visited each of these households and were able to determine its income, the mean household income in each neighborhood could be estimated by the sample mean for

---

3 An estimator $\hat{\theta}$ is unbiased for a population parameter $\theta$ if $E(\hat{\theta}) = \theta$; $\hat{\theta}$ is consistent if it converges in probability to $\theta$ as the sample size increases.
that neighborhood. The mean household income in the city as a whole could be estimated using the stratified estimator \( \hat{\mu}_{str} \). If the households in each neighborhood had roughly similar income, this estimate would tend to be much more accurate than one obtained with a simple random sample of households drawn from the entire city, assuming the same overall sample size was used with both estimators.

**Multistage sampling** designs are often used in surveys, for example, to reduce overall survey costs [19]. A multistage sampling design involves two or more nested levels of sampling units. Each stage of sampling after the first one involves sub-sampling from units drawn in the previous stage. The basic sampling designs used at each stage can be the same or different. For example, in two-stages of stratified random sampling, a stratified random sample of primary sampling units or PSUs (e.g., neighborhoods) is selected in the first stage; in the second stage, a stratified random sample of secondary sampling units or SSUs (e.g., households) is selected from each primary sampling unit. Such a design requires one criterion (e.g., location) for stratifying the population of primary sampling units and a second criterion (e.g., last year’s tax receipts) for stratifying the secondary sampling units within each primary unit. One important application of multistage designs is estimating the sizes (and other parameters) of rare and elusive animal and plant populations [11].

### B. Multidimensional Stratification

To establish that software is fit for general use, it is often appropriate to establish, with high confidence, that it rarely fails during beta testing. This calls for a very precise estimate of its failure rate. To address the facts that beta users may be likely to overlook or neglect to report certain failures, developers may be able to capture, replay, and review executions from the field [1, 2]. Podgurski et al [4] explored the application of stratified random sampling to estimating software failure rates from captured operational executions. Their goal was to obtain, for a given number of executions reviewed by developers, estimates that were as accurate as possible. Adapting the ideas of coverage testing [20], they stratified executions based on branch traversal profiles indicating how often each conditional branch in a program was traversed during an execution. Since branch profiles may contain thousands of elements, they employed automatic cluster analysis [21] to cluster similar executions together based on a measure of profile dissimilarity. They applied stratified sampling to the clusters, whose number was varied between 10\% and 25\% of the number of executions. This idea, which we shall call multidimensional stratification, reduced estimation variance substantially as compared to simple random sampling [4].

Subsequent work by a number of authors (e.g. [22-37]) has explored variants of multidimensional stratification intended for use in different applications, such as revealing software defects and grouping related failure reports from users. These techniques involve different profile types, sampling schemes, etc. Empirical evaluations of the techniques have indicated that they generally perform well.

### III. MSIO Sampling

Multidimensional stratification can reduce the cost of estimating a software reliability measure (e.g., failure rate) by decreasing the total number of executions that must be reviewed to achieve a given degree of precision, as measured by estimator variance.\(^4\) However, it does not itself address how the cost of reviewing executions varies with the size and complexity of their corresponding inputs and outputs. Manually checking a single large output may be more costly than checking a number of small outputs.

With software products for which it is possible to characterize the quality of an output with a quantitative measure that is defined in terms of corresponding elements of the input and output, the idea of estimating mean output quality is attractive, because it obviates the need to exhaustively check large outputs. Instead, one can employ, for example, a two-stage sampling design in which outputs are the primary sampling units and output elements are the secondary sampling units. For each sampled output, only a sample of its elements is checked. Given an appropriate sampling design and estimator, accurate estimates of mean output quality can be obtained at acceptable cost.

Our proposed technique, multistage stratified input/output sampling, combines multivariate stratification with multistage sampling to address the aforementioned issues with the cost of checking complex inputs and outputs. We consider two forms of MSIO sampling in more detail, both of which are two-stage sampling designs that employ stratified random sampling in the first stage.

One form of MSIO sampling that we consider employs stratified random sampling (ST sampling) with multivariate stratification to sample PSUs (outputs) in the first stage, and it employs simple random sampling without replacement (SIR sampling) to sample SSUs in the second stage. For brevity, we refer to it as ST/SIR sampling. The other form of MSIO sampling that we consider employs stratified random sampling in both the first and second stages. We refer to it as ST/ST sampling. These two forms of MSIO sampling are illustrated in Figure 1. In the first stage of each technique, outputs are stratified by clustering them directly or by clustering corresponding inputs or execution profiles.

Our two case studies (see Section IV) provide concrete examples of MSIO sampling. In one of them, we employed ST/ST sampling to evaluate a program dependence graph generator. The estimated quality measure was the mean proportion of erroneous vertices per output graph. In the first stage, dependence graphs were stratified by clustering the graphs based on their properties. In the second stage, the vertices of sampled graphs were also stratified, by clustering the vertices based on their attribute values. An SIR sample of vertices was selected from each second-stage cluster/stratum and checked for correctness.

\(^4\) It is assumed that the cost of developers manually reviewing executions dominates the cost of automatically collecting and analyzing profiles.
Figure 1. The two forms of MSIO sampling: ST/SIR sampling (A) and ST/ST sampling (B). In the first stage of ST/SIR sampling or ST/ST sampling, complex outputs are stratified by clustering them directly or by clustering corresponding inputs or profiles. In the second stage of ST/ST sampling, elements of a complex output are stratified, based on their own properties or those of corresponding input elements. In the illustration above, primary sampling units are denoted by small ellipses and secondary sampling units are denoted by triangles. Units with the same color have similar attribute values.

In the second case study, ST/SIR sampling was used to evaluate a sophisticated tool for visualizing metabolic pathways. The estimated quality measure was the mean proportion of constituent chemical reactions that were displayed incorrectly. In the first stage, the input pathways were stratified by clustering them based on their chemical reaction structure. In the second stage, an SIR sample of reactions was selected from each pathway sampled in the first stage, and each reaction was checked for correctness.

Mapping between input and output elements: Many programs that transform complex inputs into complex outputs do so in a way that preserves aspects of an explicit or implicit graphical structure. The transformation may be an isomorphism, a homomorphism, or a more loosely-defined mapping between input and output elements. One example of the latter is a sequence of editing and formatting operations applied to a written document. MSIO can be employed to evaluate such transformations if there is a reasonably straightforward way to map each basic element of the output to the corresponding element(s) of the input (if any exists) and vice-versa. The simplest way to provide such a mapping is for the program to record the associations between input and output elements. MSIO is a generalization of the program dependence graph (PDG) [41], which is a labeled, directed graph that models control and data dependences between elements of a program. A procedure dependence graph (pDG) is a dependence graph for a single procedure or function. In an SDG, pDGs are linked together by interprocedural data and control dependences between callers and callees.

TF-IDF weight is often used in information retrieval [42] to evaluate how important a given term \( t \) is to a particular document \( d \) from a corpus \( C \). It is the product of \( t \)'s frequency in \( d \) and the log inverse of its frequency in \( C \). We used TF-IDF weight to measure the importance of a vertex or edge in a pDG.

Figure 2. Example of applying JavaPDG to generate an SDG, consisting of two pDGs, from a Java class with two methods. The left side shows source code statements compiled into JVM byte code instructions. The right side shows pDG vertices (corresponding to JVM instructions) linked together by control dependence edges (blue) and data dependence edges (green). Intraprocedural dependences are shown as solid lines while interprocedural dependences are shown as dashed lines.

IV. CASE STUDIES

We evaluated the accuracy of MSIO sampling in two case studies involving applications with complex inputs and outputs.

A. Case Study 1: JavaPDG

We applied ST/ST sampling to JavaPDG [39] (see Figure 2), an automated code analysis tool developed by our laboratory, to estimate the quality of its outputs. JavaPDG inputs a Java project and performs whole-project, static interprocedural dependence analysis of both source code and byte code together, producing a system dependence graph. It currently comprises 15,991 lines of Java source code, excluding external library code.

Figure 2. Example of applying JavaPDG to generate an SDG, consisting of two pDGs, from a Java class with two methods. The left side shows source code statements compiled into JVM byte code instructions. The right side shows pDG vertices (corresponding to JVM instructions) linked together by control dependence edges (blue) and data dependence edges (green). Intraprocedural dependences are shown as solid lines while interprocedural dependences are shown as dashed lines.

1) Background Concepts

A system dependence graph (SDG) [40] is a generalization of the program dependence graph (PDG) [41], which is a labeled, directed graph that models control and data dependences between elements of a program. A procedure dependence graph (pDG) is a dependence graph for a single procedure or function. In an SDG, pDGs are linked together by interprocedural data and control dependences between callers and callees.
Cosine similarity is a measure of similarity between two \( n \)-dimensional vectors. It is equal to the cosine of the angle between them [43]. We applied cosine similarity to vectors of TF-IDF weights and hence the results were nonnegative.

**Partitioning Around Medoids (PAM)** [44] is a well-known clustering algorithm, which first computes \( k \) representative objects, called medoids. A medoid is defined as that object of a cluster whose average dissimilarity to all the objects in the cluster is minimal. Starting with an initial set of medoids, PAM constructs clusters by assigning each object of the data set to the nearest medoid. It iteratively tries to improve the clustering by swapping medoids with other objects. The objective is to minimize the sum of dissimilarities between each object and its nearest medoid.

2) **Subject Program, Inputs, and Outputs**

The output of the JavaPDG is a SDG, formed from a collection of pDGs generated from Java methods of the target project. The vertices in a pDG represent JVM instructions generated when the function is compiled. The target project used in this study was the component org.springframework.core-3.0.2.RELEASE of the open source Java application framework Spring Framework [45]. This contains 228 Java source files, 274 classes, and 1820 functions (excluding abstract and native types). The corresponding outputs of JavaPDG included 1,820 pDGs with 38,784 vertices and 117,461 edges in total. There is a simple correspondence between source code statements and the groups of pDG vertices that represent them.

In this study, we employed a buggy version of JavaPDG, which contains six known defects discovered during previous development and testing. (Two other defects were excluded because they caused very many output errors.) JavaPDG was instrumented so that its inputs (Java methods) and corresponding outputs (pDGs) were captured. For evaluation purposes, it was desirable to know the “ground truth” about which output pDG vertices were erroneous. Since it was impractical for us to check each output pDG manually for errors, we instead manually instrumented the subject version of JavaPDG to detect the trigger conditions for its six known defects and to report how many erroneous vertices were generated. A vertex was considered erroneous if its attributes, its set of incident edges, or those edges’ types were incorrect. It was easily verified that no vertices were omitted by JavaPDG, hence the frequency of omissions was not estimated. In general, doing so is straightforward.

3) **ST/ST Sampling Methodology for JavaPDG**

ST/ST sampling, which involves two stages of stratified sampling, was applied to JavaPDG outputs. In the first stage, pDGs were stratified by clustering the graphs; and in the second stage, the vertices of sampled pDGs were also stratified, by clustering the vertices of each graph. To evaluate the performance of ST/ST sampling, it was compared to SIR sampling with respect to estimating the mean proportion of erroneous vertices per pDG, which we denote by \( \pi \). To assess the effects of sampling variation, 500 full samples were drawn and 500 corresponding estimates were computed for each technique. For each replication, the same total sample size was used for SIR sampling as for ST/ST sampling. This size varied between replications because the clustering algorithm PAM is nondeterministic and because the order of the data was randomly permuted. In each case, pDGs were sampled from the same set of 1820 graphs.

The figures of merit used to compare ST/ST sampling to SIR sampling were the mean squared error (MSE) and the coefficient of variation (CV) of the estimates. The mean squared error of an estimator \( \hat{\pi} \) for the mean proportion \( \pi \) is defined as:

\[
MSE(\hat{\pi}) = E[(\hat{\pi} - \pi)^2]
\]

Note that MSE reflects both an estimator’s variance and its bias (if any) [19]. MSE was estimated by averaging, over 500 replications, the squared differences between the estimated value and the actual value of \( \pi \). (The actual value of \( \pi \) was 0.01923.) The coefficient of variation is the ratio of the standard deviation of the estimates to the mean:

\[
CV = \sigma / \pi
\]

It therefore characterizes the average size of the estimation errors relative to the magnitude of the estimated quantity \( \pi \). We estimated \( CV \) using \( \sqrt{MSE} \) in place of \( \sigma \). (We display \( CV \) values below as percentages.) Of course, in normal use ST/ST sampling would be done just once, to compute a single estimate of \( \pi \); a variance estimate may be computed from the sample and used to estimate \( CV \) or to derive a confidence interval for \( \pi \) [19].

**First-stage stratified sampling**: To stratify pDGs we used some pDG features to compute a measure of similarity between pDGs. (We also employed these features in our previous work on supervised learning of programming rules and bug patterns [46].) To derive the features, we considered different types of vertices and edges that could be misrepresented or (in the case of edges) missing in the generated pDG. The output graphs contained thirteen types of vertices and four types of dependences, including intraprocedural and interprocedural data dependences and control dependences. There are 676 possible combinations of one edge type with two vertex types; however, only 47 of them are plausible in the generated pDGs. Their presence or absence and their frequencies were recorded in a feature vector. TF-IDF weight was used to measure the importance of each pDG edge. Hence, the weight of an edge in a pDG \( G \) increased in proportion to the number of times it appeared in \( G \), but this value was offset by the edge’s frequency in the whole collection of pDGs. A matrix of dissimilarities between pDGs was computed by subtracting their cosine similarities from 1.

We used the \( R \) statistical computing environment [47] and the PAM (Partitioning Around Medoids) function from the package \( fpc \) [48] for performing cluster analysis. To ensure that our results were not sensitive to the order of data, we randomly permuted its order before starting each replication of clustering. The pDGs were partitioned into 364 clusters; this number is equal to 20% of the total number of pDGs (1820). After each clustering, a stratified random sample of pDGs was selected. The nominal “target” size for each complete sample was 455, which was 25% of the
population size. *Proportional allocation* [19] was used to allocate the total sample size to strata, so that the initial sample size for a cluster was proportional to its size. The initial sample size allocated for some small clusters was less than one. In this case we adjusted the sample size to equal one, which increased the overall sample size above the nominal target size. With each replication of pDG clustering, it happened that exactly 541 pDGs overall were selected as primary sampling units. Hence, the actual sampling rate for first-stage sampling was 30%.

**Second-stage stratified sampling:** To stratify pDG vertices, each vertex was characterized by the values of the features shown in Table 1. Each vertex had one of thirteen types, which were further categorized into six classes. Other features characterized a vertex’s incident edges. Again we applied the TF-IDF weighting scheme together with cosine similarity to determine the similarity between vertices.

### Table 1. Features for pDG vertices

<table>
<thead>
<tr>
<th>Vertex</th>
<th>13 types of PDG vertices categorized into 6 classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacent Edges</td>
<td># Edges</td>
</tr>
<tr>
<td></td>
<td># Interprocedural control dependence edges</td>
</tr>
<tr>
<td></td>
<td># Intraprocedural control dependence edges</td>
</tr>
<tr>
<td></td>
<td># Intraprocedural data dependence edges</td>
</tr>
<tr>
<td></td>
<td>In-degree</td>
</tr>
<tr>
<td></td>
<td>Out-degree</td>
</tr>
<tr>
<td></td>
<td>In-degree / Out-degree</td>
</tr>
</tbody>
</table>

In second-stage stratification, PAM was used to cluster each sampled pDG’s $|V|$ vertices into $[0.2|V|]$ clusters. Then $\min(|V|, 20)$ vertices in total were sampled from the clusters, using the modified proportional allocation scheme described previously. The number of vertices in each sampled pDG ranged from 4 to 685; the average was 40.

**Replications:** To produce a total of 500 ST/ST samples of pDG vertices, we ran fifty replications of clustering and ST sampling on the primary sampling units (pDGs), to produce fifty “top level” stratified samples. For each of these samples, we ran ten replications of clustering and ST sampling on the secondary sampling units (vertices). We compared the results obtained with ST/ST sampling against those obtained with 500 samples of vertices selected by simple random (SIR) sampling without replacement from among all pDG vertices (pooled from all pDGs). All sampled vertices were automatically checked for errors by employing the aforementioned JavaPDG instrumentation, and the results were cross-checked manually.

**Estimators:** The standard estimator used in (one stage of) stratified random sampling to estimate a population mean $\mu = \sum_{i=1}^{N} y_i / N$ can be written as [19]:

$$\hat{\mu}_{st} = \frac{1}{N} \sum_{k=1}^{H} N_k \frac{1}{n_h} \sum_{i=1}^{n_h} y_{hi} = \frac{1}{N} \sum_{k=1}^{H} \sum_{i=1}^{n_h} N_k \frac{y_{hi}}{n_h}$$

where $H$ is the number of strata, $N_k$ is the size of stratum $h$, and $n_h$ is the sample size for stratum $h$, and $y_{hi}$ is the value of the study variable for the $i$th sampled element from stratum $h$. We used this estimator with a binary study variable $\theta_{hi}$ (in place of $y_{hi}$) to compute, for each pDG $G$ in the first-stage stratified sample, an estimate $\hat{\pi}_{st}$ of the proportion of erroneous vertices in $G$. The value of $\theta_{hi}$ was 1 if and only if the $i$th vertex sampled from stratum $h$ of $G$ was erroneous. The estimator was used again to estimate the overall mean proportion $\pi$ of erroneous vertices per pDG. For each pDG $G$ in the first-stage sample, the estimate $\hat{\pi}_{st}$ was used as the value of the study variable for $G$.

### 4) Results

The results are summarized in Table 2 for increasing numbers of replications. Table 2 shows that for the full 500 iterations, the mean squared error for ST/ST sampling was substantially lower than that of simple random sampling. The ratio of the MSE for ST/ST sampling to the MSE for SIR sampling was 0.31. Table 2 also shows that, for 500 replications, the coefficient of variation for ST/ST sampling was about half of that for SIR sampling. Thus the estimates obtained with ST/ST sampling were significantly more accurate than those obtained with SIR sampling.

### Table 2. MSE and CV values for ST/ST sampling and SIR sampling.

(500 replications)

<table>
<thead>
<tr>
<th>Replications</th>
<th>MSE</th>
<th>CV × 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST/ST sampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.0000000633</td>
<td>27.75%</td>
</tr>
<tr>
<td>30</td>
<td>0.0000000528</td>
<td>28.18%</td>
</tr>
<tr>
<td>50</td>
<td>0.0000000501</td>
<td>26.77%</td>
</tr>
<tr>
<td>100</td>
<td>0.0000000490</td>
<td>26.15%</td>
</tr>
<tr>
<td>500</td>
<td>0.0000000496</td>
<td>25.85%</td>
</tr>
<tr>
<td>SIR sampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.0000001007</td>
<td>37.58%</td>
</tr>
<tr>
<td>30</td>
<td>0.0000002108</td>
<td>47.03%</td>
</tr>
<tr>
<td>50</td>
<td>0.0000001861</td>
<td>49.44%</td>
</tr>
<tr>
<td>100</td>
<td>0.0000001845</td>
<td>49.67%</td>
</tr>
<tr>
<td>500</td>
<td>0.0000001596</td>
<td>49.36%</td>
</tr>
</tbody>
</table>

Figure 3 shows histograms of the empirical sampling distributions of the ST/ST and SIR estimators for the JavaPDG study. The vertical red line in each histogram represents the true mean proportion $\pi$ of erroneous vertices per pDG. The tick marks below the histogram bars represent the individual estimates. The histograms show that the estimates obtained with ST/ST sampling are concentrated much more closely around $\pi$ than are the estimates obtained with SIR sampling.

![Figure 3](image-url)  
(A) ST/ST sampling  
(B) SIR sampling

**Computation times:** The machine used for the experiments was a 2.66 GHz Intel Core 2 with 4GB RAM. In the first-stage of ST sampling, it took about 4 hours to
compute the dissimilarity matrix for pDGs (which was calculated only once) and, for each replication, it took about 10 minutes in total to permute the data, cluster pDGs, and select a stratified sample of pDGs. In each replication of the second-stage of ST sampling, it took at most 10 minutes in total to compute the compute dissimilarities between pDG vertices, cluster them, select a stratified sample of vertices from each sampled pDG, and check the sampled vertices for errors. In total, the time for 500 replications of ST/ST sampling on 1,820 pDGs with 38,784 vertices was within 3,500 minutes. In actual use, there would be no need to repeat ST/ST sampling, so the total computation time would be about 4 hours, 20 minutes.

B. Case Study 2: The PathCase System

In the second case study, we applied ST/SIR sampling to the visualization functionality of the PathCase software system. PathCase is a sophisticated tool for querying and visualizing metabolic pathways, which are sets of chemical reactions catalyzed by enzymes [8]. In PathCase, a metabolic pathway is represented as a graph. As shown in Figure 4, the circular vertices represent molecular entities that participate in reactions as substrates or products, and the rectangles represent enzymes, which are connected by arrows to the molecular entities involved in reactions. For our case study, we used a development version of PathCase known to have bugs due a recent component update. The visualization subsystem alone comprised 10,366 SLOC.

![Pathway graph](http://nashuatest2.case.edu/PathwaysKegg/Web/)

**Figure 4. Pathway graph**

PathCase downloads pathway data from the KEGG PATHWAY Database site [49], parses the information into its own “PathCase format”, and inserts it into its database. In this way, PathCase’s database is kept up-to-date with the KEGG pathway data. In our study, we manually checked the consistency between sampled PathCase visualizations and the corresponding KEGG PATHWAY data. For example, the definition obtained from KEGG for the reaction “dihydrozeatin:NADP oxidoreductase” is:

\[
\text{Dihydrozeatin} + \text{NADP}^+ \iff \text{Zeatin} + \text{NADPH} + \text{H}^+ 
\]

which includes five different molecular entities. However, in the corresponding PathCase display, there were only two molecular entities. As shown in Figure 5, three molecular entities named “H””, “NADP”” and “NADPH” were missing. This error implied the existence of one or more defects in PathCase that caused the loss of information. In checking PathCase visualizations, we found that cofactor molecules such as “NADPH” and “NADP” were often missing in the pathway graphs. For the purposes of this study, we considered a displayed reaction to be erroneous if and only if it was missing a cofactor molecule.7

In this case study, ST/SIR sampling was applied to the 146 pathways displayed by the PathCase development version (using the KEGG data released on July 23, 2010), in order to estimate the mean proportion of erroneous reactions per pathway visualization, which we denote by \( \rho \). In the first stage of sampling, a stratified sample of pathways was selected; in the second stage, a simple random sample of reactions was selected from each sampled pathway for examination. Unlike in the case study involving JavaPDG, we could not automatically determine ground truth about which pathways and reactions were displayed incorrectly; they had to be checked manually. There were a number of large pathways (e.g., one had 148 reactions) and checking any of them exhaustively would be very time consuming. Hence, we did not determine the actual value of \( \rho \), and we could not replicate ST/SIR sampling and SIR sampling 500 times, to be very rigorous, and compute the MSEs of their estimates as in Case Study 1. Instead, we applied each sampling design 10 times, each time computing both an estimate of \( \rho \) and a variance estimate. (Note that in normal use ST/SIR sampling would be done just once, to compute a single estimate of \( \rho \).

1) Clustering and Sample Allocation

To partition the population of pathways into \( L \) non-overlapping strata, we again used the cluster analysis algorithm PAM. We chose to measure dissimilarity between pathways in the general manner proposed by Tohsato [50], in which each reaction of one pathway is mapped to the most similar reaction of another pathway. This approach assumes that enzymes with similar “EC numbers” will catalyze similar reactions, so that reaction similarity can be measured by the similarity between EC numbers. Based on Tohsato’s ideas, Chen and Hofest [51] defined a series of edit operations to delete, insert, and replace reactions. The dissimilarity between two pathways is scored by calculating the minimum number of edit operations to change one pathway into the other. To calculate these edit distances, we applied Munkre’s algorithm [52] to the KEGG data.

In this case study, we wished to use the optimal allocation method [19] for stratified sampling, which seeks to improve estimation accuracy by allocating more sample elements to strata with higher variance. With this method, the number of units in the sample from stratum \( h \) is ideally given by

\[
n_h = \frac{n N_h \sigma_h}{\sum_{k=1}^{\infty} N_k y_k}
\]

7 Initially, we considered other display errors as well, which caused the mean proportion of erroneous reactions to be over 0.5. To make the study scenario more typical, we chose to ignore those errors in this paper. Nevertheless, MSIO sampling performed similarly in both cases.
where \( n \) is the total sample size, \( N_h \) is the total number of units in stratum \( h \), \( \sigma_h \) is the (unknown) true standard deviation of the study variable \( y \) in stratum \( h \), and \( \bar{y}_k \) is the sample mean of \( y \) for stratum \( k \).

The benefit of optimal allocation is reduced when rough estimates of \( \sigma_h \) are employed. To overcome this problem, we adopted Manly’s technique for two phase adaptive stratified sampling [11]. (This was done within the ST stage of ST/SIR sampling.) In the first phase, we selected a number of pathways equal to 75\% of the total sample size \( N \), by sampling from each stratum a number of pathways proportional to its relative size \( N_h / N \). In the second phase, we allocated the remaining 25\% of the sample to strata so as to maximize the reduction in the coefficient of variation \( CV \).

This required computing some auxiliary estimates. Recall the standard stratified estimator of the population mean of a study variable \( y \):

\[
\hat{\mu}_st = \sum_{k=1}^{L} \frac{N_k}{N} \bar{y}_k
\]

where \( N_k \) is the size of stratum \( k \), \( N \) is the population size, and \( \bar{y}_k \) is the sample mean of stratum \( k \). Let \( s_k^2 \) be the sample variance of \( y \) for stratum \( k \). An estimate of the variance of \( \hat{\mu}_st \) is [53]:

\[
\hat{\text{var}}(\hat{\mu}_st) = \sum_{k=1}^{L} \left( \frac{N_k}{N} \right)^2 \left( \frac{N_k - n_k}{N_k} \right) s_k^2 / n_k
\]

The estimate of \( CV \) that we used in the second phase of Manly’s technique was

\[
\hat{CV} = \frac{\sqrt{\hat{\text{var}}(\hat{\mu}_st)}}{\hat{\mu}_st}
\]

2) ST/SIR Sampling Methodology for PathCase

In order to stratify pathways into \( L \) strata for the first stage of ST/SIR sampling, a matrix of dissimilarities between pathways was generated by computing reaction edit distances, and it was used in clustering the pathways with PAM. Then a stratified sample of pathways was selected according to Manly’s two-phase sample allocation technique. In the second stage of ST/SIR sampling, and SIR sample of reactions was selected from within each sampled pathway. The correctness of all sampled reactions was checked manually.

Let \( p_{ki} \) be the sample proportion of erroneous reactions from the \( i \)th pathway selected from the \( k \)th stratum. Then the estimator for the mean proportion \( \bar{y}_k \) of erroneous reactions in the \( k \)th stratum is

\[
\bar{y}_k = \frac{\sum_{i=1}^{n_k} M_{ki} p_{ki}}{\sum_{i=1}^{n_k} M_{ki}}
\]

where \( M_{ki} \) is the total number of reactions in the \( i \)th pathway sampled from the \( k \)th stratum and \( n_k \) is the number of pathways sampled from the \( k \)th stratum. For the whole population, the mean proportion \( \rho \) of erroneous reactions per pathway was estimated using the stratified estimator:

\[
\hat{\rho} = \frac{\sum_{k=1}^{L} N_k \bar{y}_k}{\sum_{k=1}^{L} N_k}
\]

An estimate of the variance of \( \bar{y}_k \) is [54]:

\[
\hat{\text{var}}(\bar{y}_k) = \frac{(N_k - n_k)}{N_k} \left( \frac{1}{n_k} \right) r_k^2 + \frac{1}{n_k N_k} \sum_{i=1}^{n_k} M_{ki}^2 \frac{(M_{ki} - m_{ki})}{m_{ki}} \left( \frac{p_{ki} q_{ki}}{m_{ki} - 1} \right)
\]

where \( r_k^2 \) is the sample variance of the \( M_{ki} p_{ki} \) terms, \( m_{ki} \) is the number of reactions sampled from the \( i \)th pathway selected from the \( k \)th stratum, \( \bar{M}_k \) is mean size of pathways in the \( k \)th stratum, and \( q_{ki} = 1 - p_{ki} \). The first term of the variance estimator for \( \bar{y}_k \) involves the variance between sampled pathways, while the second term involves the variance within sampled pathways.

Using the variance estimator above and the identity \( \text{var}(aX) = a^2 \text{var}(X) \), we derived the following variance estimator for \( \hat{\rho} \):

\[
\hat{\text{var}}(\hat{\rho}) = \frac{\sum_{k=1}^{L} N_k^2 \hat{\text{var}}(\bar{y}_k)}{(\sum_{k=1}^{L} N_k)^2}
\]

3) Results

In the first stage of ST/SIR sampling, the 146 pathways were clustered into 15 strata and Manly’s technique was used to select a stratified random sample of pathways. The initial (nominal) size for the overall sample was 30, which was 20.5\% of the population size. The sample size for each stratum was made proportional to the stratum’s size and then rounded to the nearest integer. This increased the overall sample size to 36. In the second phase of Manly’s technique, another 12 pathways were allocated to stratum samples. The size of the primary sample was adjusted to 48, which was 33\% of the population size. In the second stage of ST/SIR sampling, simple random sampling was used to select 20\% of the reactions within each sampled pathway for examination.

To evaluate the effectiveness of ST/SIR sampling, we compared it both with and without Manly’s technique to SIR/SIR sampling, with respect to estimator variance and coefficient of variation. Each sampling method was applied 10 times in total. The average results are shown in Table 3.

<table>
<thead>
<tr>
<th>Sampling Designs</th>
<th>( \hat{\rho} )</th>
<th>( \hat{\text{var}}(\hat{\rho}) )</th>
<th>( \frac{\hat{\text{var}}(\hat{\rho}) \times 100%}{\hat{\rho}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST/SIR with</td>
<td>0.16081</td>
<td>0.00019</td>
<td>8.57%</td>
</tr>
<tr>
<td>Manly’s technique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic ST/SIR</td>
<td>0.14932</td>
<td>0.00028</td>
<td>11.21%</td>
</tr>
<tr>
<td>SIR/SIR</td>
<td>0.17117</td>
<td>0.00066</td>
<td>15.01%</td>
</tr>
</tbody>
</table>

Table 3 shows that both variants of ST/SIR sampling had lower estimated variance than SIR/SIR sampling. Between the two forms of ST/SIR sampling, the one incorporating Manly’s technique had lower estimated variance. The reductions in the estimated coefficient of variation observed with ST/SIR sampling were smaller than those observed with
ST/ST sampling in Case Study 1, possibly because of the use of SIR sampling instead of ST sampling in stage two.

C. Threats to Validity

The generalizability of our results is limited by the fact that our case studies involved only two applications and input data sets. Follow-up studies of MSIO sampling are clearly needed. The variance estimates obtained in Study 2 are probably less reliable than the MSE estimates from Study 1, because only in Study 1 did we know the actual output errors before sampling, which permitted us to replicate sampling with each technique 500 times, and because Study 2 required us to manually examine outputs to find errors.

V. DISCUSSION

The usefulness of a particular output quality measure depends on how the output will be used, and it may be appropriate to characterize the quality of complex outputs with multiple measures. With PathCase, for example, if vertex omissions were suspected it would be appropriate to estimate their frequency. This would entail sampling input elements and then seeking corresponding elements in the output. It might also be desirable to estimate the frequencies of erroneous vertices and erroneous edges separately, rather than considering a vertex with an erroneous set of incident edges as erroneous itself. Also, with scientific applications like PathCase the quality of input data may be problematic [17], so it may be desirable to use the ideas of MSIO sampling to evaluate input quality as well as to evaluate output quality.

When implementing MSIO sampling with a particular software product, it is necessary to write a small program or script to extract feature values from outputs, inputs, or execution profiles. It is also desirable to provide whoever will check output elements for errors with a simple tool to help them quickly find and examine sampled elements and record whether they are erroneous or not. These utilities are reused whenever the product’s output quality is reassessed.

VI. RELATED WORK

Dickinson et al [23, 24], Leon et al [25, 55-57], and Masri et al presented variants of one-stage multidimensional stratification for use in discovering software defects [26]. Podgurski et al presented a technique that applies cluster analysis, supervised learning, and multivariate visualization techniques to execution profiles in order to classify reported software failures according to their cause(s) [27]. Francis et al presented two tree-based techniques for refining an initial classification of failures [58]. Augustine and Podgurski presented a technique that employs profile-based clustering of executions to corroborate noisy user failure reports [22].

Mao and Lu presented a technique that employs Markov models of program behavior, clustering of profiles, and priority-ranked n-per-cluster sampling to extract a representative sample of failed executions from a set of executions [31]. Liu and Han propose a dissimilarity metric called R-Proximity for comparing traces of failed executions, which regards two failing traces as similar if they suggest roughly the same fault location [29]. Sampath et al use formal concept analysis for clustering web-application user sessions, and they present a set of heuristics for test case selection [30]. Jones et al present a technique for parallel localization of multiple fault that automatically partitions a set of failing test cases into clusters that trigger different faults, using behavior models and execution data [33]. Ramanathan et al describe a framework called Phalanx that prioritizes regression tests according to a declustered linearization of nodes in a graph of test case dissimilarities [59]. Jagannath et al present a technique called “oracle-based test clustering” to reduce time spent inspecting automatically generated tests that fail. It clusters failing tests based on abstracted error messages [59]. Yan et al propose a dynamic cluster sampling strategy for test cases, called execution-spectra-based sampling (ESBS), which computes a suspiciousness value for each test case and uses it to select from a cluster the test that is most likely to fail [28]. The results of the test are checked, and the suspiciousness values of other tests are updated based on the results. None of the aforementioned work involves sub-sampling of complex outputs.

VII. CONCLUSION

The costs of determining expected outputs corresponding to complex inputs or of checking complex outputs for errors are important issues in software testing that have received little attention from researchers. MSIO sampling provides a feasible and principled approach to controlling these costs while obtaining valuable, objective information about output quality. The results of our case studies suggest that output quality estimates obtained with ST/ST sampling and ST/SIR sampling are more accurate, for a fixed sample size, than ones obtained with simple random sampling. The case studies also indicate that MSIO sampling is practical to apply to substantial applications with complex inputs and outputs. In future work we plan to conduct more empirical evaluation of MSIO sampling and to explore the use of additional sampling designs and estimators.

REFERENCES
