Finding Failures by Cluster Analysis of Execution Profiles

William Dickinson, David Leon, and Andy Podgurski
Electrical Engineering & Computer Science Department
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106 USA
+1 330 656 4228, +1 216 368 4231, +1 216 368 6884
bilkat@adelphia.net, dzl@po.cwru.edu, andy@eecs.cwru.edu

Abstract

We experimentally evaluate the effectiveness of using cluster analysis of execution profiles to find failures among the executions induced by a set of potential test cases. We compare several filtering procedures for selecting executions to evaluate for conformance to requirements. Each filtering procedure involves a choice of a sampling strategy and a clustering metric. The results suggest that filtering procedures based on clustering are more effective than simple random sampling for identifying failures in populations of operational executions, with adaptive sampling from clusters being the most effective sampling strategy. The results also suggest that clustering metrics that give extra weight to unusual profile features are most effective. Scatter plots of execution populations, produced by multidimensional scaling, are used to provide intuition for these results.

Keywords: Observation-based testing, software testing, operational testing, beta testing, cluster analysis, multidimensional scaling

1. Introduction

Traditional software testing techniques call for constructing or generating a set of test inputs, executing the software under test on those inputs, and then checking the results for conformance with requirements. *Observation-based testing* [12], on the other hand, involves the following steps: taking an existing set of program inputs (possibly quite large); executing an instrumented version of the software under test on those inputs to produce *execution profiles* characterizing the executions; analyzing the resulting profiles, with automated help; and finally selecting and evaluating a subset of the original executions for conformance to requirements. It is assumed that substantial manual effort is required to evaluate executions for conformance to requirements or, equivalently, to determine expected output. Observation-based testing seeks to reduce this effort by filtering out a subset of the original set of executions that is more likely to contain failures than is a randomly chosen subset. (A *failure* is defined here as an erroneous execution — not an erroneous aspect of an execution). The filtering process itself must be relatively inexpensive, which implies that the analysis of execution profiles must be largely automated.

Observation-based testing was conceived as an operational testing (beta testing, field testing) technique. In many cases, it seems reasonable to assume that software being beta tested fails infrequently and that executions which do fail have unusual properties that may be reflected in execution profiles. Hence, if beta test executions are captured and later filtered to identify those with unusual profiles, and if the selected executions are evaluated by trained testing personnel, this may reveal failures that beta users overlooked or did not report [20]. More generally, observation-based testing is potentially applicable whenever it is desirable to filter a large set of potential test cases in order to identify a promising subset to evaluate for conformance to requirements. Other examples of such applications include filtering "legacy" test suites and filtering automatically generated tests. Observation-based testing techniques can also be used to *augment* a set of test cases [12].

An observation-based testing method is defined by specifying: (a) the form of execution profiling to be used; (b) a *filtering procedure* for selecting the set of executions to be evaluated for conformance to requirements; and (c) an optional *augmentation procedure* for augmenting the executions selected by the filtering procedure. The form of profiling that is used should reflect runtime events associated with software failures, such as the execution of erroneous program statements or the execution of other statements under the wrong conditions. Software testing research and practice suggest many possible alternatives, including statement/basic-block profiling, branch profiling, path profiling, function-call profiling, and various forms of data flow profiling. (Harrold, *et al* evaluate several

0-7695-1050-7/01 $10.00

alternatives in [10].) Indeed, for every proposed test-data selection technique, there seems to be a corresponding form of profiling.

This paper focuses on the choice of filtering procedure to be used in observation-based testing. It addresses the hypothesis that in filtering executions, it is often desirable to select those exhibiting unusual behaviors, as reflected by their profiles. In particular, filtering procedures based on automatic *cluster analysis* [11] are evaluated. This type of procedure, which we call *cluster filtering*, entails clustering executions based on their profiles and then selecting one or more executions from each cluster or from certain clusters such as small ones. Thus, a cluster filtering procedure comprises a choice of a clustering technique and of a strategy for sampling from clusters. Cluster filtering was suggested by the work of Podgurski, *et al* [15,16], who used cluster analysis of execution profiles together with stratified random sampling to estimate software reliability efficiently. Their results suggested that software failures were often isolated in small clusters, but they did not actually evaluate the effectiveness of cluster analysis for finding failures.

We describe experiments in which cluster filtering is applied to execution populations of several defective programs. Two cluster sampling strategies, which we call *one-per-cluster sampling* and *adaptive sampling*, respectively, are compared to simple random sampling in terms of their effectiveness for locating the failures present in these populations. One-per-cluster sampling involves selecting one execution at random from each cluster. Adaptive sampling involves initially selecting one execution at random from each cluster and then including the remaining executions from any cluster if the first one selected from it is a failure. Although one basic clustering algorithm (agglomerative hierarchical clustering) is used throughout, the dissimilarity metric used to form clusters is varied systematically, as is the number of clusters formed.

Our results suggest the following: (1) cluster filtering is more effective than simple random sampling for finding failures in populations of operational executions; (2) adaptive sampling is more effective than one-per-cluster sampling; and (3) dissimilarity metrics which give extra weight to unusual profile features are most effective. Scatter plots of execution populations, produced by multidimensional scaling of their profiles, are used to provide intuition for these results. We also describe several interesting research issues raised by our experiments.

2. Cluster analysis

Cluster analysis is a multivariate analysis method for finding groups or *clusters* in a population of objects. Each object is characterized by a vector of *attribute* values. The goal of cluster analysis is to partition a population into clusters in such a way that objects with similar attribute values are placed in the same cluster, while objects with dissimilar attribute values are placed in different clusters. The similarity or dissimilarity of objects is measured using a *dissimilarity metric*, such as n-dimensional Euclidean distance.

There are two main approaches to cluster analysis: partitioning and hierarchical clustering. The *partitioning* approach initially divides the population into k clusters, for a chosen k. The quality of the partitioning is improved iteratively by reassigning objects to different clusters. Although partitioning produces high-quality clusterings, it is computationally expensive.

The *hierarchical* approach to cluster analysis is a stepwise process for generating clusters. There are two types of hierarchical clustering methods: *agglomerative* and *divisive*. Agglomerative methods initially assign each object to its own cluster. At each step a pair of clusters that is minimally dissimilar is chosen and merged to form a new cluster that replaces the chosen pair. The process continues until k clusters remain, for a chosen k. Divisive methods, on the other hand, initially assign all the objects to one cluster. At each step, a cluster is divided into two clusters. Again, the process is continued until a desired number of clusters is produced. Hierarchical methods are faster than partitioning methods, but they may not produce as good a clustering, because once a merge or division operation is carried out it cannot be undone. Hierarchical clustering has the added benefit of being able to generate partitions with varying numbers of clusters in a single run.

3. Cluster filtering

The rationale for cluster filtering is this: if failures in a population of executions are infrequent and have unusual profiles, then it should be possible to cluster the population so as to isolate a significant proportion of the failures in small clusters. This would enable the failures to be found by sampling those clusters. We reasoned that for cluster filtering to succeed, the number of clusters formed must be large – a significant fraction of the population size – because failures are infrequent. Note that it is *not* necessary for failures to cluster together. If a failure is isolated in a singleton cluster, then the sampling strategies we consider will find it. Two failures may be placed in different clusters because they have different causes or because they differ with respect to the non-defective code they traverse. On the other hand, a successful execution may be placed in the same cluster with a failure because the two executions are similar with respect to the non-defective code they traverse.

To fully specify a cluster filtering procedure, it is necessary to select the following: the particular clustering technique to be used; the number of clusters to be formed; and a strategy for sampling from the clusters. In the remainder of this section we consider each of these aspects of cluster filtering.

3.1. Clustering technique

Agglomerative hierarchical clustering is used in all of the experiments described in Section 4. This type of clustering was chosen because it is faster than partitioning and because it performed reasonably well in preliminary experiments. With all but one of the subject programs for our experiments, the inputs to cluster analysis were *function caller/callee profiles*. In this type of profile, there is an entry for each pair of program functions *f* and *g*, giving the number of times that *f* called *g* during the corresponding execution. With one larger subject program (GCC), *simple function call profiles* were used instead, because they are smaller. In this type of profile, there is an entry for each program function, giving the number of times the function was executed.

Although agglomerative hierarchical clustering was used in all of our experiments, a number of different dissimilarity metrics were studied. One of these was *n*-dimensional Euclidean distance. Several of the other metrics work by applying transformations to profiles before applying the Euclidean distance formula:

- *Binary metric*: replaces nonzero profile counts with 1 in order to emphasize differences in the coverage of program elements rather than differences in the frequency of coverage.
- *Proportional metric*: normalizes each attribute. The range of values for each attribute is computed, and then each value is mapped to its relative position within the range.
- *SD metric*: also normalizes each attribute. The standard deviation of each attribute is computed, and then each attribute value is divided by the standard deviation.
- *Histogram metric*: for each attribute, a histogram indicating the relative frequency of different attribute values is constructed. Each attribute value is mapped into 1 minus the corresponding relative frequency. This is done to emphasize attribute values that occur infrequently.
- *Linear regression metric*: suggested by initial observations. It was noted that many attributes were correlated with the size of the program input. This metric attempts to remove that effect and expose underlying differences. Since the size of the program input is not known to the clustering program, the attribute that is best correlated with all the other attributes is used instead. A linear regression of this attribute with each of the other attributes is computed. For each profile, the value predicted by the regression is subtracted from each attribute value.

The remaining metrics are hybrids, which are intended to combine the best features of some of the metrics described above:

- *Count-binary metric*: suggested by Podgurski *et al* [15]. It gives equal weight to differences in program coverage and differences in the frequency of coverage.
- *Proportional-binary metric*: combines the binary and proportional metrics. It is similar to the count-binary metric, but uses the proportion value (see above) instead of the count value.

3.2. Number of clusters

Any number of clusters, up to the number of runs, can be formed using cluster analysis. This number affects the degree to which runs are separated. If too many clusters are formed, then even executions that are highly similar will be placed in separate clusters. If too few clusters are formed, then failures may be placed in large clusters containing mostly successful executions, making them difficult to find. Also, with some of the sampling strategies described below the sample size depends on the number of clusters. In the experiments described in Section 4, a number of different cluster counts were used for each subject program. Rather than use fixed counts, fixed fractions of the number of program executions in our populations were used. We considered 1%, 2.5%, 5%, 10%, 15%, 20%, 25%, and 30% of the runs as candidate cluster counts.

3.3. Sampling strategy

Following the cluster analysis step, a subset of the population of executions is chosen for evaluation against requirements. This subset can be chosen in a number of ways. We focused on three sampling strategies: simple random sampling from the population as a whole, one-per-cluster sampling from all clusters, and adaptive sampling from all clusters.

The most basic sampling strategy is *simple random sampling* from the entire population. This involves selecting *m* executions from the population randomly and without replacement. Each sample of size *m* has equal probability of being selected. Note that this procedure does not require cluster analysis. We use it as the baseline against which other procedures are compared.

The *one-per-cluster* sampling strategy selects one member at random from each cluster. The number of executions to be evaluated is equal to the number of clusters. With this strategy, more executions are selected from small clusters than from large ones, because small clusters are typically much more numerous. We expect a significant proportion of failures to be isolated in small clusters, so this strategy should be effective at finding them.

The *adaptive sampling* strategy first selects one execution at random from each cluster. All selected executions are evaluated for conformance to requirements.

If a selected execution is found to be a failure, then all other members of its cluster are also selected. The number of executions to be selected and evaluated is at least the number of clusters produced by cluster analysis. The additional number of executions to be evaluated depends on the distribution of erroneous runs over the clusters. Like one-per-cluster sampling, this strategy favors executions in small clusters. It is likely to be more effective than one-per-cluster sampling when multiple failures are clustered together.

Two other sampling strategies were also considered. One of these strategies is a variation of the one-per-cluster strategy, called *n-per-cluster* sampling. It calls for selecting n executions, where $n > 1$ is a small number, from each cluster containing at least n executions. All executions in clusters of size less than n are selected. The *small cluster* sampling strategy calls for selecting entire clusters randomly and without replacement, starting with the smallest clusters and progressing to larger ones. This process continues until the total number of executions selected exceeds a previously chosen threshold.

4. Experimental results

We ran a series of experiments to evaluate the effectiveness of cluster filtering in finding failures in execution populations. We examined the following issues: (1) Is our hypothesis about the distribution of failures among clusters correct — will a significant number of failures often be isolated in small clusters? (2) How likely is each of the various sampling strategies to find at least one failure, if any are present? (3) How *efficient* are the sampling strategies, in terms of the number of failures they find? (4) Which of the various dissimilarity metrics and which cluster counts are most effective for finding failures?

In the remainder of this section we describe our experimental protocol, the programs that were the subjects of our experiments, the results of our experiments, and our interpretation of those results.

4.1. Experimental protocol

Our experiments had a few basic ingredients: a program with known failures, a large set of inputs, and a way to automatically determine whether an execution of the program was successful or not. An instrumented or monitored version of the program was run on the inputs to produce execution profiles, and cluster filtering was applied to the resulting profiles. Since we knew which executions failed, we could determine how well cluster filtering worked.

Three series of experiments were run, involving three sets of subject programs (the last set contained just one program). In the first two series, all but one of the original subject programs used were assumed to be correct. Defective versions of these programs were created, and the original versions of the programs were used as "oracles" to determine when the modified programs failed.

The following procedure was followed in the first two series of experiments, with each of the original subject programs: (1) An *operational profile* was defined, to characterize a plausible pattern of operational usage. (2) A set of inputs to the program was created in accordance with the operational profile. (3) The subject program was executed on the inputs created in step 2, and the outputs were recorded. (4) If the subject program contained no defects, one was introduced, either by hand or with the aid of a mutation generator. (5) The defective version of the program was executed on the inputs created in step 2 to produce execution profiles. (6) The resulting profiles were clustered, using agglomerative hierarchical clustering with each of the dissimilarity metrics described in Section 3 and several different cluster counts. (7) Each of the sampling strategies described in Section 3 were applied to the clusterings produced in step 6, and the results were evaluated.

In the third series of experiments, the subject program (the GCC compiler) was known to contain bugs, so none were introduced into it. This program was executed on an existing test suite with automated output checks. Thus only steps 5 - 7 above were necessary.

4.2. Subject programs

Several Java™ programs were used as subjects in the first series of experiments. These programs were obtained from public domain web sites. The defective versions of all but one of these programs were hand generated, by making operator or constant-value substitutions. The subject programs were a word count program (*wc*), a directory listing program (*ls*), a regular expression parser (*rex*), a regular expression finder (*egrep*) and a Java pretty-printer (*JSFormat*) [2,8]. Their error rates ranged from 0.33% to 29%. Their sizes ranged from 2964 to 6210 lines, and they were executed on between 1633 and 4636 inputs. Function caller/callee profiles were used in the first series of experiments; these were generated by the profiler built into the Sun Java™ virtual machine.

In the second series of experiments, a mutation generator was used to insert various defects into a program, resulting in several defective versions. The subject program was *wc*, the Java™ implementation of the UNIX word count program. A mutation generator was used to create the defective versions for this series. To create this generator, we adapted a set of mutant operators considered by Offutt *et al* [13] for use with Ada™ programs. The operator categories include operator substitution, constant substitution, variable name component substitution, primary expression substitution, statement dropping, variable initializer dropping, insertion of `break` and `continue` statements in loops, `case` statement

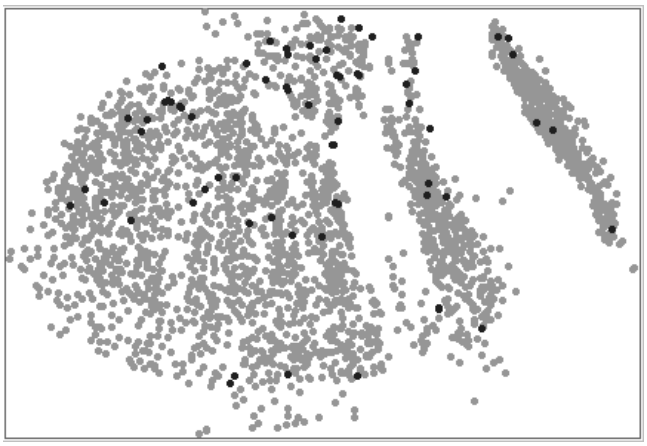

Figure 1. Multidimensional scaling plot of binary dissimilarities for GCC. Points represent executions. Failures are indicated in black.

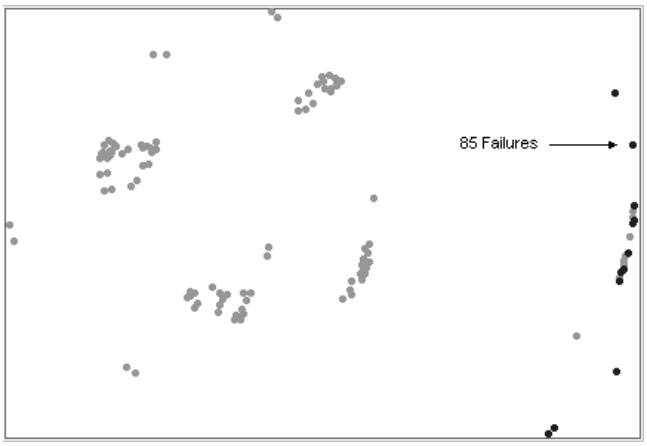

Figure 2. Multidimensional scaling plot of binary dissimilarities for a *wc* mutant. Points represent executions. Failures are indicated in black.

scrambling, replacing statements with a `return` statement, and swapping `do` and `while` statements. Note that traditional mutation testing generates mutants to measure the adequacy of a set of test data. We generated mutants to create a collection of program variants that includes a range of possible defects. Each defective version was the result of a single mutation.

We eliminated any defective version that failed on substantially more than 5% of the input set. This threshold was chosen to represent the quality of programs likely to be seen in operational testing. Any version that did not fail at all was also eliminated.

A total of 34 different defective versions of this program that met the above constraints were generated out of a 674 defective versions created by the mutation generator. All 34 of these programs were run on the same test inputs for a total of 1757 runs each. Note that is a different set of data from that used on the *wc* program in the first series. The number of failures ranged from 1 (.06%) to 110 (6.26%). The average number of failures per version was 87.6, or 4.98%. There were three versions that produced only 1 failure. Function caller/callee profiles were used in this series of experiments; these were generated with the Sun Java™ profiler.

A subset of the mutant programs used in the second series of experiments all failed on the same inputs. The source code of these programs was examined. They were found to have different defects, affecting different aspects the computation of one output (the total character count for multiple file input). The different mutants generally yielded different execution profiles.

The subject program for the third series of experiments was the C compiler of the GNU Compiler Collection (GCC) version 2.95.2 [7]. No defects were introduced. The inputs for these experiments comprised part of the automated regression test suite maintained by the GCC team (GCC.C-Torture). This test suite is available to the public via anonymous CVS. Developers are encouraged to add a test case to the suite whenever they find a bug in the compiler. Note that GCC 2.95.2 was released on October 24, 1999; we used the version of the test suite available as of July 10, 2000, which includes tests for some bugs present in 2.95.2.

The GCC test suite contains different types of tests. The most important of these are: (1) tests to check that valid code is not rejected; (2) tests to check that illegal code is rejected; and (3) tests to check whether certain code compiles and also executes correctly. In our experiments we considered only failures of the third type of test. This is because for the first two test types, whether a test is passed or failed is decided by whether error handling code was called, which is definitely reflected in the profiles. On execution tests, whether a test passes or fails is decided by whether the object code executes correctly (we ignored inputs that didn't compile at all). By disregarding failures of the first two types of tests, we gave cluster filtering a harder task.

Because of the size of the program, we used simple function call profiles, as opposed to caller/callee profiles. The GNU call-coverage profiler, *gcov*, was used to gather these call counts. The rest of the analysis was the same as for the other two sets of programs.

There were 2768 test runs, with a total of 2225 different functions being called. There were only 69 failed executions.

4.3. Results

4.3.1. Distribution of Failures. An important issue when evaluating the utility of cluster filtering is whether failures are distributed in a random fashion. Our experiments indicate that they are not. For example, consider Figure 1, a multidimensional scaling plot of the GCC data, calculated by using the iterative majorization algorithm described in

Table 1. Percentage of failures found in the half of the population contained in the smallest clusters

Cluster %	Count	Binary	SD	Histogram	Proportional	Linear regression	Count-binary	Proportional-binary
1.0	59.46	87.44	77.35	80.33	84.78	57.92	59.46	90.04
2.5	64.95	56.93	80.84	83.91	90.94	64.67	64.95	67.01
5.0	69.67	55.27	90.36	90.41	90.10	77.61	69.66	56.21
10.0	71.09	53.08	90.64	94.52	92.69	77.66	70.60	56.92
15.0	74.37	52.80	94.42	93.32	92.48	75.43	74.43	55.19
20.0	73.07	55.70	93.35	91.48	92.48	74.11	73.08	59.75
25.0	72.03	62.20	94.18	89.59	93.02	73.78	72.03	61.14
30.0	71.73	59.51	88.86	84.92	90.60	73.90	71.68	64.57

[1]. Each point represents an execution, and the distances between the points approximate the dissimilarities produced by the binary metric. It can be seen from this figure that failures in the execution population tend to cluster together.

The principal assumption of cluster filtering is that a significant number of failures will be isolated in clusters of small size. To test this assumption we looked at the distribution of failures over cluster sizes. For each subject program variation we looked at the sub-population of executions that were in the smallest clusters. The percentage of all failures contained in that sub-population was computed from the actual distribution of failures.

Table 1 shows, for different dissimilarity metrics and cluster counts, the average percentage of all failures that are in the half of the execution population contained in the smallest clusters. These averages are taken over all subject program variations. If failures were uniformly distributed, we would expect each average to be about 50%. The data shows that the percentage of failures found in the smallest clusters is significantly higher than 50%. Similar results were seen for the quarter of the execution population contained in the smallest clusters.

4.3.2. Likelihood of Finding Failures. If cluster filtering is to be useful we must be confident that it will find at least some of the failures induced by a set of potential test cases. We therefore investigated the probability that cluster filtering will find no failures at all. For each subject program variation we computed the probability of finding no failures for simple random sampling and for one-per-cluster sampling. Note that the probability for adaptive sampling is the same as the probability for one-per-cluster sampling since the adaptive stage is only applied if an erroneous run is detected.

Table 2 shows the average probabilities, over all subject programs, of finding no failures with simple random sampling and with one-per-cluster sampling, for each cluster count and dissimilarity metric. For most of these combinations one-per-cluster sampling outperformed simple random sampling. In fact, for many of the subject programs, the probability of finding no failures fell to 0. For example, consider the data displayed in Figure 2. In this data set, there are 85 failures represented by one point in the display. Cluster analysis isolated these 85 failures into a cluster with no successes. When one-per-cluster sampling selects an execution from this cluster, it has to select a failure. This type of situation is common across different subject programs.

4.3.3. Efficiency in Finding Failures. Since the failures in the execution populations for our subject programs are concentrated in the smaller clusters formed by cluster analysis (for all dissimilarity metrics), sampling strategies that favor small clusters should be able to find multiple failures more efficiently than random sampling. To verify this, we examined the percentage of failures found with different sampling strategies. For every subject program variation, we applied simple random sampling, one-per-cluster sampling, and adaptive sampling to the output of the clustering step. For each strategy, we computed the expected number of failures found as a percentage of the

Table 2. Average probability of finding no failures for all subject programs.

Cluster %	Random sample	Count	Binary	SD	Histogram	Proportional	Linear regression	Count-binary	Proportional-binary
1.0	0.42	0.31	0.18	0.22	0.18	0.16	0.32	0.31	0.14
2.5	0.19	0.25	0.11	0.10	0.11	0.09	0.17	0.25	0.07
5.0	0.10	0.15	0.08	0.08	0.08	0.06	0.10	0.15	0.05
10.0	0.07	0.08	0.07	0.06	0.04	0.04	0.08	0.08	0.03
15.0	0.06	0.06	0.07	0.04	0.04	0.04	0.06	0.06	0.02
20.0	0.05	0.06	0.06	0.02	0.04	0.02	0.05	0.06	0.02
25.0	0.05	0.05	0.04	0.02	0.04	0.02	0.05	0.05	0.02
30.0	0.05	0.05	0.04	0.02	0.03	0.00	0.05	0.05	0.02

total number of failed executions. For adaptive sampling, we also computed the expected sample size as a percentage of the population size.

For simple random sampling, the expected percentage of failures found is equal to the sample size expressed as a percentage of the population size. Table 3 shows, for each cluster count and dissimilarity metric, the average percentage of failures found with one-per-cluster sampling, taken over all the Java™ program variations. In every case this percentage is substantially higher than the percentage of executions sampled. The efficiency ratio (*% failures found*) / (*% executions sampled*) ranges from 2.5 to 5.1 for the most efficient dissimilarity metric, SD, and from 1.4 to 3.0 for the least efficient metric, binary. The average efficiency ratio over all dissimilarity metrics and cluster count percentages is 3.03.

With the GCC executions, one-per-cluster sampling proved less efficient. For most dissimilarity metrics the expected percentage of failures found was less for one-per-cluster sampling than for simple random sampling. Table 4 shows the average of this percentage over all dissimilarity metrics, for both GCC and the Java™ program variations and for all cluster counts. The average efficiency ratio over all programs, cluster counts, and dissimilarity metrics was 0.91, or slightly worse than simple random sampling.

Adaptive sampling was more efficient for finding failures than simple random sampling or one-per-cluster sampling, for all program variations and most dissimilarity metrics. Table 5 shows, for each dissimilarity metric and cluster count, the average percentage of failures found with adaptive sampling, taken over all the Java™ program variations. In every case this average is larger than the corresponding averages for simple random sampling and one-per-cluster sampling. The efficiency ratio for the SD metric ranged from 2.9 to 6.5. For the binary metric this ratio ranged from 2.6 to 7.7. The average efficiency over all cluster counts and all dissimilarity metrics was 4.23. Table 4 shows, for GCC and the Java™ program variations and for all cluster counts, the average percentage of failures found with adaptive sampling, taken over all dissimilarity metrics. The average efficiency ratio over all cluster counts and all dissimilarity metrics was 1.77. The adaptive strategy seems to find clusters of erroneous runs that were not completely seen by one-per-cluster sampling. For example, consider Figure 2. In this data set, one-per-cluster sampling would only select one of the failures indicated by an arrow in the display, while adaptive sampling select all 85 of them. Thus, adaptive sampling helps when failures cluster together.

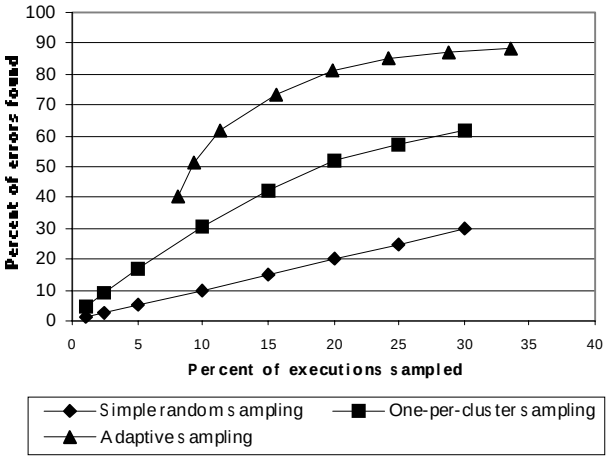

Figure 3. percentage of failures found vs. sample size as a percentage of population size, averaged over all Java™ program variations and dissimilarity metrics.

Table 4 shows, for GCC and all Java™ program variations and for all cluster counts, the average expected sample size with adaptive sampling, taken over all dissimilarity metrics and expressed as a percentage of the population. These averages are all slightly larger than the initial sample size taken as a percentage of the population. This is to be expected since additional sampling takes place when failures are found in the initial sample. The fact that the increase in sample size is small suggests that the failures in the initial sample were found in relatively small clusters.

Figure 3 illustrates the improvement in efficiency for different sampling strategies. It contains a plot of the expected percentage of failures found vs. the expected sample size expressed as a percentage of the population size. The data is the average of results over the Java™

Table 3. Percentage of failures found using one-per-cluster sampling in all Java™ program variations

Cluster %	Count	Binary	SD	Histogram	Proportional	Linear regression	Count-binary	Proportional-binary
1.0	4.48	2.95	5.11	5.82	4.14	4.13	4.48	3.33
2.5	9.73	5.83	10.66	10.90	9.37	10.68	9.73	7.42
5.0	17.29	11.60	18.54	19.14	15.71	20.34	17.29	15.32
10.0	31.92	21.19	31.27	31.51	29.76	39.73	31.91	28.26
15.0	48.04	25.41	45.32	44.91	42.63	53.12	48.04	32.57
20.0	54.55	29.84	66.25	51.66	66.13	58.34	54.55	36.34
25.0	59.56	37.24	71.15	55.63	71.03	62.76	59.56	41.12
30.0	63.03	41.82	75.53	61.49	76.98	65.35	63.03	46.01

program variations and over all dissimilarity metrics. It clearly illustrates the distinct improvement in efficiency of one-per-cluster sampling over simple random sampling and of adaptive sampling over one-per-cluster sampling. The other notable feature is the flattening of the curve when approximately 80% of the errors are found. After that point not much improvement is seen with additional testing effort. That seems to come at a sampling percentage of approximately 20%.

4.3.4. Other Sampling Strategies. We also examined the effectiveness of small cluster sampling and *n*-per-cluster sampling. The results for small cluster sampling are similar to the results for one-per-cluster sampling when the sample size for small cluster sampling is set equal to the number of clusters. On the average, small cluster sampling found 1% to 2% more failures for a given dissimilarity metric and cluster count than did one-per-cluster sampling. Similar results were obtained with *n*-per-cluster sampling. The effect of increasing the number of executions selected from each cluster is comparable to that of increasing the number of clusters and then selecting one execution from each.

4.3.5. Dissimilarity Metrics and Cluster Counts. In general, dissimilarity metrics that emphasize unusual occurrences performed better than count-based metrics. Examples of the former type of metric include the binary, SD, proportional, histogram, and proportional-binary metrics. Count-based metrics include the count, linear regression, and count-binary metrics. The metrics that isolated the most failures into small clusters (Table 1) were the SD, proportional, and histogram metrics. The proportional metric was most effective at finding one or more failures. This metric required a 30% sampling rate to guarantee success in every case. The proportional-binary metric was nearly as effective. With a 10% sampling rate, it found at least one failure in 44 out of 46 cases. The metrics that were the most efficient with the adaptive sampling strategy were the binary, SD, proportional, histogram, and proportional-binary metrics (Table 5). They were far more efficient than the count-based metrics at low sampling rates and they neared their maximums at lower sampling rates than did the count-based metrics.

Several pertinent observations about cluster counts can be made. With low cluster counts some failures were dispersed into larger clusters of mostly successful executions while others tended to form large clusters of mostly failures (e.g. Figure 2). The latter effect was particularly beneficial for adaptive sampling while the former effect kept a large fraction of failures from being discovered. With high cluster counts (greater than 30% of the population size), the successful executions were partitioned into many clusters. There were more small clusters with only successful executions, so it required larger samples to find the erroneous executions.

4.3.6. Summary and Discussion. Our results suggest the following conclusions:

- Cluster filtering of executions is significantly more effective for finding failures than simple random sampling.
- Adaptive sampling is substantially more efficient for finding multiple failures than one-per-cluster sampling, *n*-per-cluster sampling, and small cluster sampling.
- Dissimilarity metrics that give extra weight to unusual profile features are most effective.

The main threat to the validity of these conclusions is the limited number and types of subject programs used in our experiments. All but one of these were text processing programs, and all but one (GCC) were of small to moderate size. Another issue is the fact that all but two of the programs had a single defect that was deliberately introduced, either by hand or with a mutation generator. To confirm or generalize our results, much more empirical research is needed.

The inputs to the Java™ programs used in our experiments were selected to approximate operational usage. However, it will be important to evaluate cluster filtering with operational inputs captured in the field. The fact that cluster filtering was somewhat less effective when

Table 4. Average percentages of failures found and executions sampled for 1-per-cluster and adaptive sampling

	1-per-cluster sampling		Adaptive sampling			
	Java™	GCC	Java™		GCC	
Cluster %	% failures found	% failures found	% sampled	% failures found	% sampled	% failures found
1.0	4.30	1.12	8.17	40.26	3.44	5.55
2.5	9.29	2.85	9.35	51.59	4.91	10.85
5.0	16.90	5.52	11.39	61.89	7.34	16.55
10.0	30.69	8.86	15.57	73.31	12.24	22.29
15.0	42.50	11.50	19.85	81.11	17.20	29.61
20.0	52.20	13.76	24.22	84.81	22.13	32.58
25.0	57.26	19.32	28.84	86.84	27.01	42.42
30.0	61.66	23.54	33.50	88.54	31.89	48.45

Table 5. Average percentage of failures found using adaptive sampling with the Java™ program variations

Cluster %	Count	Binary	SD	Histogram	Proportional	Linear regression	Count-binary	Proportional-binary
1.0	17.20	58.70	39.43	44.71	56.52	15.40	17.20	62.96
2.5	22.76	65.19	52.91	58.26	72.46	26.23	22.76	72.07
5.0	29.82	73.78	68.78	64.11	74.45	47.60	29.82	79.02
10.0	46.06	78.39	79.12	81.79	80.64	64.47	45.84	84.08
15.0	67.36	79.70	85.61	83.37	84.38	70.51	67.44	85.26
20.0	74.93	80.51	89.11	84.37	88.98	75.04	74.93	85.97
25.0	78.38	83.33	90.29	85.50	90.11	78.61	78.38	86.97
30.0	80.68	84.18	91.53	87.56	93.53	81.11	80.68	87.78

applied to the GCC test suite suggests the possibility that cluster filtering will be less effective in general with synthetic test cases than with operational inputs, e.g., because there is less regularity in the former.

It is important to note that even the most effective types of cluster filtering did not usually find all of the failures in an execution population. While cluster filtering can be used to improve reliability and perhaps to gauge how much debugging a program requires, it certainly cannot be used to certify the correctness of a program, although related methods can be used to estimate program reliability [15].

The significance of the expected number of failures found as a measure of the effectiveness of cluster filtering merits some discussion. Adaptive sampling performed well by this measure, because failures were often clustered together. One reason failures may cluster together is that they have the same cause. This raises the issue of whether anything is gained by finding multiple failures with the same cause. In *Debugging Applications* [18], Robbins addresses this issue:

> *By duplicating the bug from multiple paths, you have a much better sense of the data and boundary conditions that are causing the problem.* [p. 16]

Finding multiple beta-test executions that fail because of the same defect also provides an indication of how often that defect affects users in the field.

5. Related work

The use of cluster filtering in observation-based testing is related to *subdomain testing* (also called *partition testing*). However, subdomain testing involves conceptually subdividing a program's entire *input domain*, which is typically infinite, rather than partitioning a finite set of actual executions. Having specified subdomains of the input domain, it is still necessary to construct or generate test cases from them, which is often difficult and which cannot be automated, in general. Several authors have evaluated subdomain testing in ways that are similar to how we have evaluated cluster filtering [3, 5, 6, 9, 21].

Podgurski, *et al* examine the use of cluster analysis for improving the accuracy/efficiency of software reliability estimation [15,16]. In this application, program executions are captured in the field and later replayed and profiled. The profiles are then clustered to obtain a stratified sampling design for estimating reliability. Podgurski, *et al* report experiments in which cluster analysis of branch-traversal profiles permitted the failure frequency of several programs to be estimated with stratified random sampling more accurately (with lower variance) than it could be with simple random sampling.

Leon, *et al* describe several applications of multivariate visualization techniques in observation-based testing and present a case study suggesting that such techniques can reveal features of execution populations that are relevant to finding failures [12]. Steven, *et al* describe the design of a tool called *jRapture* for capturing, replaying, and profiling Java™ program executions, which is intended for use in observation-based testing.

Elbaum, *et al* [4] and Rothermel, *et al* [19] describe a variant of observation-based testing that involves *prioritizing* test cases for regression testing, and they empirically evaluate test-case prioritization techniques. Reps, *et al* investigate the use of a type of execution profile called a *path spectrum* for discovering Year 2000 problems and related issues, and they propose several other applications of path spectra to software maintenance and testing [17]. Harrold, *et al* evaluated several types of program spectra (profiles) empirically, to determine how well they indicate the occurrence of execution failures [10]. They observed that failures were likely to be indicated by differences in complete-path spectra, path-count spectra, and branch-count spectra. They also observed that differences in such spectra are more likely to indicate failures than are differences in execution-trace spectra. Pavlopoulou and Young describe how monitoring *residual test coverage* in software that is deployed or undergoing beta testing can be used to validate the thoroughness of testing in the development environment, and they describe a prototype system that monitors residual statement coverage in Java™ programs [14].

6. Conclusion

We have described a technique called *cluster filtering*

for selecting a subset of set of potential test cases to evaluate for conformance to requirements. Cluster filtering involves profiling the executions induced by the original test cases and then applying automatic cluster analysis to the profiles. Executions are then selected from some or all of the resulting clusters. We reported the results of an experimental evaluation of cluster filtering, in which alternative dissimilarity metrics, cluster counts, and sampling strategies were employed. The results suggest that cluster filtering is more effective than simple random sampling for identifying failures in populations of operational executions, with adaptive sampling from clusters being the most effective sampling strategy. The results also suggest that dissimilarity metrics that give extra weight to unusual profile features are most effective. Further empirical evaluation is needed to confirm our results.

7. References

1. Borg, I. and Groenen, P. *Modern Multidimensional Scaling: Theory and Applications*, Springer, 1997.

2. Davidson, T. jstyle.JSFormatter, http://www.bigfoot .com/~davidsont/jstyle, 1998.

3. Duran, J.W. and Ntafos, S.C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10, 4 (July 1984), 438-444.

4. Elbaum, S., Malishevsky, A.G., and Rothermel, G. Prioritizing test cases for regression testing. *Proceedings of the 2000 International Symposium on Software Testing and Analysis* (Portland, OR, August 2000), 102-112.

5. Frankl, P.G. and Weyuker, E.J. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering* 19, 3 (March 1993), 202-213.

6. Frankl, P.G. and Weyuker, E.J. Provable improvements on branch testing. *IEEE Transactions on Software Engineering* 19, 10 (October, 1993), 962-975.

7. GCC. *The GCC Home Page*, http://www.gnu.org /software/gcc/gcc.html, Free Software Foundation, 2000.

8. GNU Java Software. http://www.gnu.org/software /java/java-software.html, Free Software Foundation, 2000.

9. Hamlet, D. and Taylor, R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering* 16, 12 (December 1990), 1402-1411.

10. Harrold, M.J., Rothermel, G., Wu, R., and Yi, L. An empirical investigation of program spectra. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Montreal, Canada, June 1998), 83-90.

11. Jain, A.K. and Dubes, R.C. *Algorithms for Clustering Data*, Prentice Hall, 1988.

12. Leon, D., Podgurski, A., and White, L.J. Multivariate visualization in observation-based testing. *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland, June 2000), ACM Press, 116-125.

13. Offutt, J., Voas, J., and Payne, J. Mutation Operators for Ada, Technical Report ISSE-TR-96-09, George Mason University, October, 1996.

14. Pavlopoulou, C. and Young, M. Residual test coverage monitoring. *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, CA, May 1999), ACM Press, 277-284.

15. Podgurski, A., Masri, W., McCleese, Y., Wolff, F.G., and Yang, C. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology* 8, 9 (July, 1999), 263-283.

16. Podgurski, A. and Yang, C. Partition testing, stratified sampling, and cluster analysis. *Proceedings of the First ACM Symposium on Foundations of Software Engineering* (Los Angeles, CA, December 1993), ACM Press, 169-181.

17. Reps, T., Ball, T., Das, M., and Larus, J. The use of program profiling for software maintenance with applications to the Year 2000 Problem. *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, September 1997), ACM Press, 432-449.

18. Robbins, J. *Debugging Applications*, Microsoft Press, 2000.

19. Rothermel, G., Untch, R., Chu, C., and Harrold, M.J. Test-case prioritization: an empirical study. *Proceedings of the International Conference on Software Maintenance* (August, 1999), 179-188.

20. Steven, J., Chandra, P., Fleck, B., and Podgurski, A. jRapture: a capture/replay tool for observation-based testing. *Proceedings of the 2000 International Symposium on Software Testing and Analysis* (Portland, Oregon, August 2000).

21. Weyuker, E.J. and Jeng, B. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering* 17, 7 (July 1991), 703-711.