Tree-Based Methods for Classifying Software Failures

Patrick Francis, David Leon, Melinda Minch, Andy Podgurski
Electrical Engineering & Computer Science Department
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106 USA
+1 216 368 4231, +1 216 368 6884
paf9@cwru.edu, dzl@cwru.edu, melinda@cwru.edu, andy@eecs.cwru.edu

Abstract

Recent research has addressed the problem of providing automated assistance to software developers in classifying reported instances of software failures so that failures with the same cause are grouped together. In this paper, two new tree-based techniques are presented for refining an initial classification of failures. One of these techniques is based on the use of dendrograms, which are rooted trees used to represent the results of hierarchical cluster analysis. The second technique employs a classification tree constructed to recognize failed executions. With both techniques, the tree representation is used to guide the refinement process. We also report the results of experimentally evaluating these techniques on several subject programs.

1. Introduction

The problem of classifying instances of software failures (failed executions) according to their causes arises in two common situations: (1) when a large number of failures are reported by users of deployed software and (2) when a large number of failures are induced by executing a synthetic test suite. In both cases, it is likely that many of the failures fall into a relatively small number of groups, each consisting of failures caused by the same software defect (fault). To facilitate corrective maintenance, it is desirable to identify these groups *before* the causes of the failures are diagnosed, if possible, because they indicate the number of defects that are responsible for the failures, how frequently each of these defects causes failures, and which failures are relevant to diagnosing specific defects. Failures caused by a particular defect sometimes manifest so distinctively that it is easy to determine that they have the same cause by observing program output. Many other failures cannot be classified so easily, however. Our recent research suggests that the problem of classifying failures according to their causes can be simplified by applying certain multivariate data analysis/data mining techniques to *execution profiles* [27]. This approach requires that three types of information about executions be recorded and analyzed: execution profiles reflecting the causes of the failures; *auditing information* that can be used to confirm reported failures; and *diagnostic information* that can be used in determining their causes.

In [27], we presented a semi-automated strategy for classifying software failures, which entails the application of both supervised and unsupervised pattern classification techniques[1] and multivariate visualization techniques to execution profiles. Experimental evaluation of this strategy with failures of three large subject programs suggested that it is effective for grouping failures with the same cause(s). The strategy calls for manual investigation of certain failures to confirm or, if necessary, to refine the initial classification. The results may indicate that certain groups in the initial classification should be split or that other groups should be merged together. However, the strategy described in [27] provides only limited guidance about *how* groups should be split or merged.

In this paper, we present two new *tree-based* techniques for refining an initial classification of failures. The first of these techniques is based on the use of *dendrograms* [16], which are tree-like diagrams used to represent the results of *hierarchical cluster analysis*. When applied to a set of *n* objects that are related by a *similarity* or *dissimilarity* metric, hierarchical clustering algorithms produce a clustering with *k* clusters for each *k* between 1 and *n*. *Agglomerative* algorithms do this placing each object in a singleton cluster and merging clusters iteratively, with the two most similar clusters

[1] *Supervised* pattern classification techniques require a training set with positive and negative instances of a pattern; *unsupervised* techniques do not.

being merged at each step. *Divisive* clustering algorithms start with a single cluster containing all objects and then split clusters iteratively, with the least homogeneous cluster being split at each step. A dendrogram represents the sequence of split or merge operations carried out by a hierarchical clustering algorithm. (See Figure 1.) Our dendrogram-based technique for refining failure classifications uses a dendrogram to decide how non-homogeneous clusters should be split into two or more sub-clusters and to decide which clusters should be considered for merging. In Section 3, we report the results of experimentally evaluating this technique with failures of four subject programs. These suggest that the strategy is effective for improving the quality of an initial classification of failures.

The second technique that we present for refining an initial failure classification relies on generating a *classification tree* [3] to recognize failed executions. A classification tree is a type of pattern classifier that takes the form of a binary decision tree. (See Figure 4.) Each internal node in the tree is labeled with a relational expression that compares a numeric attribute/feature of the object being classified to a constant *splitting value*. Each leaf is labeled to indicate whether it represents a positive or negative instance of the class of interest (e.g., failed execution). An object is classified by traversing the tree from the root to a leaf. At each step of the traversal prior to reaching a leaf, the expression at the current node is evaluated. The left branch is taken if the expression evaluates to *true*, and the right branch is taken if it evaluates to *false*. A classification tree is constructed algorithmically using a training set containing positive and negative instances of the class of interest.

We show in Section 4 that a classification tree can be constructed to successfully recognize failed program executions, using a training set containing profiles of both failed and successful executions. The decision nodes in the resulting tree test the values of profile features (elements). When such a tree is used to classify a set of other executions consisting entirely of failures, it implicitly clusters the failures based on the leaves they are associated with. Moreover, this clustering can be extended to a *hierarchical* clustering by associating a cluster with each internal node of the classification tree, consisting of the union of the clusters associated with its two subtrees. This hierarchical clustering can be used to refine an initial clustering in much the same way that dendrograms can be used for this purpose. In Section 4, we present experimental evidence that failures that are clustered together by classification trees often have the same cause(s) and that refinement of this clustering is unnecessary. Finally, it is natural to wonder whether examining the decisions on the path from the root of a classification tree to a node representing a set of failures is helpful for *diagnosing* the cause(s) of the failures. In Section 4, we present evidence that this practice is not particularly helpful, due, for example, to spurious correlations.

2. Basic classification strategy

In this section we outline the basic classification strategy described in [27]. If m failures are observed over some period during which the software is executed n times in total, it is likely that these failures are due to a substantially smaller number k of distinct defects. Let $F = \{f_1, f_2, ..., f_m\}$ be the set of reported failures. For simplicity, assume that all reported failures are actual ones and that each failure is caused by just one defect. Then F can be partitioned into $k < m$ subsets $F_1, F_2, ..., F_k$ such that all of the failures in F_i are caused by the same defect d_i for $1 \leq i \leq k$. We call this partition the *true failure classification*. The basic strategy for approximating the true failure classification has four phases [27]:

1. The software is instrumented to collect and transmit to the developer either execution profiles or captured executions, and it is then deployed.
2. Execution profiles corresponding to reported failures are combined with a random sample of profiles of operational executions for which no failures were reported. This set of profiles is then analyzed to select a subset of all profile features[2] (a projection) to use in grouping related failures. The feature selection strategy is to:
    - *a.* Generate candidate feature-sets and use each one to create and train a pattern classifier to *distinguish failures from successful executions.*
    - b. Select the features of the classifier that performs best overall.
3. The profiles of reported failures are analyzed using cluster analysis and/or multivariate visualization techniques, in order to *group together failures whose profiles are similar with respect to the features selected in phase (2).*
4. The resulting classification of failures into groups is explored in order to *confirm it or, if necessary, refine it.*

The result of approximating the true failure classification using this strategy is a partition $C = \{G_1, G_2, ..., G_p\}$ of F. We call C the approximate failure classification. For it to be useful, all or most of the groups G_i should contain all or mostly failures with

[2] By a *feature* of an execution profile we mean an attribute or element of it. For example, a function call profile contains an execution count for each function in a program, and each count is a feature of the profile.

closely related causes.

Note that the strategy just described and the refinement techniques discussed in this paper are intended for classifying software failures in general and not just program crashes and aborts.

3. Refinement using dendrograms

In this section we describe in more detail our technique for using dendrograms to refine an initial failure clustering, discuss implementing it with the help of the Hierarchical Clustering Explorer tool, and describe the results of evaluating the technique experimentally with failures of several subject programs.

3.1. Attributes of dendrogram clusters

Each cluster in a dendrogram comprises a subtree of the dendrogram. The height of a subtree indicates its similarity to other subtrees – the more similar two executions or clusters are to each other, the further from the root their first common ancestor is. Each subtree or cluster has several attributes that are significant for our refinement technique. Each cluster in the dendrogram is composed of failures with one or more causes. A cluster's *largest causal group* is the largest set of failures within a cluster that have the same cause. These failures may be scattered throughout the cluster, or concentrated in one area of the cluster. Ideally, all of the executions in a cluster will belong to the largest causal group, in which case the cluster is considered *homogenous*.

Since each cluster consists of a subtree of the dendrogram, clusters can also be considered as entities that contain multiple sub-clusters. A cluster's *largest homogeneous subtree* is the largest homogeneous cluster that could possibly be obtained by recursively splitting the original cluster, as in Figure 1. In other words, it is the set of failures within the cluster that have the same failure type and compose a distinct subtree in the cluster. It is desirable for each cluster to have one large subtree that comprises a majority of its executions, and for this large subtree to contain a subset of the executions in the largest causal group. Ideally, the largest homogeneous subtree of a cluster will include all of the cluster's executions. A cluster can still be useful, however, if a majority of its executions have the same cause as those in the largest homogeneous subtree. If a clustering is too coarse, some clusters may have two or more large homogeneous subtrees of different failure types. Such a cluster should be split at the level where its large homogeneous subtrees are connected, so that these subtrees become siblings as in Figure 1. Typically, these large subtrees connect at the highest level in the cluster.

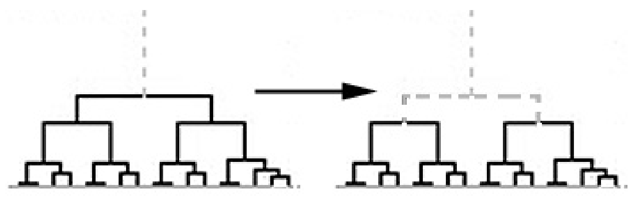

Figure 1: Splitting a cluster

Each cluster in the dendrogram has one *sibling*, which is the cluster to which it is the most closely related; that is, the cluster with which it shares a parent.

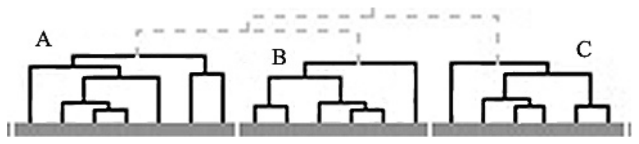

Figure 2: A and B are siblings, and the subtree that contains both A and B is C's sibling.

If a clustering is too fine, siblings may be clusters containing failures with the same causes. Such siblings should be merged at the level of their parent, as illustrated in Figure 3, in order to form a cluster that has one largest homogeneous subtree. Recursively merging clusters would be appropriate in a situation like the one in Figure 2, if clusters A, B and C all have failures with the same causes. We have found that in practice, this situation arises more often when using classification trees, but generally does not occur in dendrograms. One might also consider whether cluster C contains failures with the same causes as those in clusters A or B, but not both. We have found in our experiments that this is usually not the case. The clusters within the group comprising A and B are more closely related to each other than they are to the cluster C, and we should not expect the failures in A or B to have causes in common with the failures in cluster C.

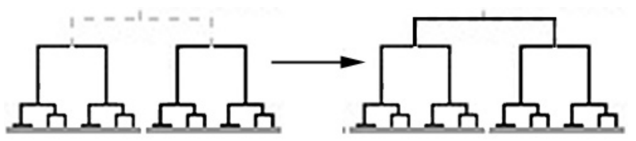

Figure 3: Merging two clusters

3.2. Refinement strategy

The strategy that we have for refining an initial classification of failures using dendrograms has three phases:

1. Select the number of clusters into which the dendrogram will be divided, using a method such as the Calinski-Harabasz [6] metric.
2. Examine the individual clusters for homogeneity by choosing the two executions in the cluster with

maximally dissimilar profiles, and determining whether these two executions have the same cause. If the selected executions have the same or related causes, it is likely that all of the other failures in the cluster do as well. If the selected executions do not have the same or related causes, the cluster is not homogeneous, and should be split. [3]
3. If neither the cluster nor its sibling is split by step 2, merge them if the failures that were examined have the same cause.

Clusters that have been generated from merging or splitting operations should be analyzed in the same way, which allows for the recursive splitting or merging of clusters.

3.3. Subject programs and test suites

We used four subject programs for this study: the *GCC* compiler for C [10], the *Javac* [19] and *Jikes* [20] Java compilers, and *Jtidy* [22], a Java-based HTML syntax checker and pretty-printer. These programs were chosen for several reasons: they can be executed repeatedly with a script; source code a number of versions is available, and test inputs are readily available. Failures for GCC, Javac and Jikes were detected by using self-validating test suites. For Jtidy, we used HTML and XML files gathered off of the web as operational inputs.

Version 2.95.2 (Debian GNU/Linux) of the GCC compiler for C was used. It was executed on a subset of the regression test suite for GCC 3.0.2, which included tests for defects still present in version 2.95.2. We used only the tests which execute compiled code in order to check for miscompilations. GCC was executed on 3333 tests and failed 136 times. Version 1.15 of Jikes and Javac build 1.3.1_02-b02 were executed on the Jacks test suite (as of 2/15/02) [15], which tests adherence to the Java Language Specification [17]. Jikes was executed on 3149 tests and failed 225 times; Javac was executed on 3140 tests and failed 233 times. Note that the Jacks test suite contains tests that are specific to the Jikes and Javac compilers. Version 3 of Jtidy was executed on 7990 HTML and XML files collected from the Internet, and failed 308 times. Inputs for Jtidy were gathered by retrieving random files from Google Groups [11] with a web crawler.

GCC and Jikes, which are written in C and C++, respectively, were profiled using the GNU test coverage profiler Gcov, which is distributed with GCC. To profile Javac and Jtidy, both of which are written in Java, we used two different profilers we implemented ourselves. The Javac profiler uses the Java Virtual Machine Profiling Interface [18]. The Jtidy profiler instruments the code using the Byte Code Engineering Library [5].

The failures for the GCC, Javac, and Jikes data sets were manually classified in [27]. We identified 26 defects for GCC, 67 defects for Javac, and 107 defects for Jikes. For Jtidy, we examined the project's bug database to find examples of defects (with known fixes) that were still present in version 3 and we selected five of them for study. We created an *oracle* version in which these defects are fixed, but a *failure checker* for each defect was added. These failure checkers detect the triggering conditions for each defect and report whether they are satisfied during execution. Using this oracle, we determined which defects, if any, would be triggered by each test case when executed on the original version of the program. Unlike the other subject programs, some of Jtidy's executions failed due to a combination of different defects. Nine such combinations were observed. When computing statistics, we consider each combination to be a unique class of failures.

3.4. Experimental results

In order to confirm that the strategy outlined in Section 3.2 works, we applied it to dendrograms (Figures 5 – 8 at the end of this paper) created from the data sets in Section 3.3. These dendrograms were generated by using the Hierarchical Clustering Explorer [13] on the failed executions of the subject programs. HCE was used with the Unweighted Pair Group Method with Arithmetic Mean as the clustering algorithm and Euclidean distance with normalized data as the dissimilarity metric. For GCC, Javac and Jikes, we used the features selected by step 2 of the strategy outlined in Section 2; Jtidy was small enough to make feature selection unnecessary.

Seven metrics were used to evaluate the strategy both before and after applying the changes detailed by step 3 in Section 3.2:

- Size of each cluster's largest causal group, as a percentage of the cluster's type.
- Size of each cluster's largest homogeneous subtree, as a percentage of the cluster's type.
- Percentage of clusters that are homogeneous.
- Overall percentage of executions that were in a homogeneous cluster.
- Average *completeness* of clusters in the dendrogram. The completeness of a cluster is, for the failure types in the cluster, the percentage of the total failures with those failure types that it contains. If a cluster contains a failure with a certain cause, it must also contain every other failure with the same cause to be considered 100% complete.

[3] Splitting a cluster is only effective if doing so will improve the clustering; for example, if a split would simply turn a cluster with *n* executions into a singleton and a cluster of size *n-1*, with no largest homogeneous subtree, it should not be performed.

- Number of singleton clusters
- Percentage of *singleton failures* that are correctly classified as singletons. An execution is a singleton failure if it is the only one to fail because of its particular cause

Singleton clusters are discarded in the analyses of homogeneity and completeness, as all singleton clusters are homogeneous. Several clusters that were appropriate candidates for splitting could have been split twice or more; we restricted our splitting to twice, at most.

The results of the experiment are outlined in Table 1. The first section of the table describes the entire dendrogram for each subject program before and after splitting and merging operations take place. It includes the number of clusters in the dendrogram as well as the seven metrics described earlier in this section. The second section of the table contains measurements from only those clusters that were split, and the clusters resulting from the splits. All clusters that were split were heterogeneous before splitting took place.

GCC. The initial dendrogram for GCC produced 27 clusters, 85% of which were homogeneous. All clusters whose least similar failures had the same cause were homogeneous. We found four clusters that should be split according to our heuristics, and four pairs of clusters that should be merged. As Table 1 shows, the operations performed on the GCC dendrogram improved the overall quality of the clustering, most notably in the clusters obtained by splitting one of the original clusters. All of the clusters that were merged with their siblings remained homogeneous. 75% of the clusters resulting from a merge operation had siblings composed of two or more clusters as in Figure 2. The GCC data set contains only one singleton failure and it is classified correctly. Though the original clustering of the dendrogram generated too many singleton clusters, none of them were merged. Also, no new singleton clusters were generated by the splitting the original clusters.

Javac. The initial dendrogram for Javac produced 34 clusters, 65% of which were homogeneous. There was one heterogeneous cluster whose least similar failures had the same cause. We found no pairs of clusters to merge, and 9 candidates for splitting. The results for Javac indicate the need for a clustering that is finer than the original one; there were no clusters to be merged, and increasing the number of clusters increased all measures of homogeneity in the dendrogram.

Jikes. The initial dendrogram for Jikes produced 33 clusters, 48% of which were homogeneous. All clusters whose least similar failures had the same cause were homogeneous. We found no pairs of clusters to merge, and 13 clusters to split. Like Javac, the results for Jikes indicate a need for a finer clustering, though almost all of the splits performed resulted in one or more singleton clusters. Normally, a large number of singleton clusters indicates a clustering that is too fine, but a third of the failures in the Jikes data set are singleton failures. For this reason, valid splitting operations will create many singletons, and many splitting operations must be performed to produce clusters that are homogeneous and contain failures that are related to each other.

Jtidy. The initial dendrogram for Jtidy produced 8

Table 1: Experimental results for four subject programs

	GCC		Javac		Jikes		Jtidy	
All Clusters	Before	After	Before	After	Before	After	Before	After
Number of non-singleton clusters	27	28	34	38	33	35	8	11
Number of singleton clusters	13	13	24	31	22	34	6	8
Correctly classified singleton failures	100%	100%	61%	91%	24%	39%	0%	0%
Homogeneous clusters	85%	93%	65%	89%	48%	63%	38%	27%
Average cluster homogeneity	93%	96%	83%	96%	70%	81%	62%	72%
Average failures in largest homogeneous subtree	92%	95%	82%	94%	66%	76%	48%	65%
Average completeness of clusters	63%	73%	82%	82%	88%	85%	20%	14%
Executions in a homogeneous cluster	65%	84%	54%	68%	27%	45%	4%	4%
Split Clusters								
Homogeneous clusters	0%	78%	0%	80%	0%	41%	0%	0%
Average cluster homogeneity	54%	89%	58%	91%	51%	72%	46%	61%
Average failures in largest homogeneous subtree	51%	86%	56%	85%	46%	63%	23%	52%
Average completeness of clusters	73%	51%	75%	76%	88%	88%	25%	15%

clusters, 38% of which were homogeneous. We found no pairs of clusters to merge, and 5 clusters to split. As in the GCC data set, the Jtidy data set contains only one singleton failure, but it was placed into the second largest cluster. The Jtidy data set contained the largest number of failures, and the smallest number of different failure causes. The number of failures associated with each cause ranged from 1 to 79. The original clustering put the rarest failures together in small homogeneous clusters, and put the most common failures together in large heterogeneous clusters. This resulted in clusters whose overall quality wasn't affected by one or two splitting operations.

4. Refinement using classification trees

In this section, we describe how software failures can be clustered using a classification tree and how the tree can be used to refine this classification. We also describe experimental results from applying this technique to our subject programs.

A classification tree is a type of pattern classifier that takes the form of a binary decision tree. (See Figure 4.) Each internal node in the tree is labeled with a relational expression that compares a numeric attribute/feature of the object being classified to a constant *splitting value*. Furthermore, each leaf of the tree is labeled with a *predicted value.* This value indicates which instance of the class of interest the leaf represents. In our data, we have two classes: success and failure. In Figure 4, for example, the value at each leaf gives the probability that an execution in that leaf is a failure.

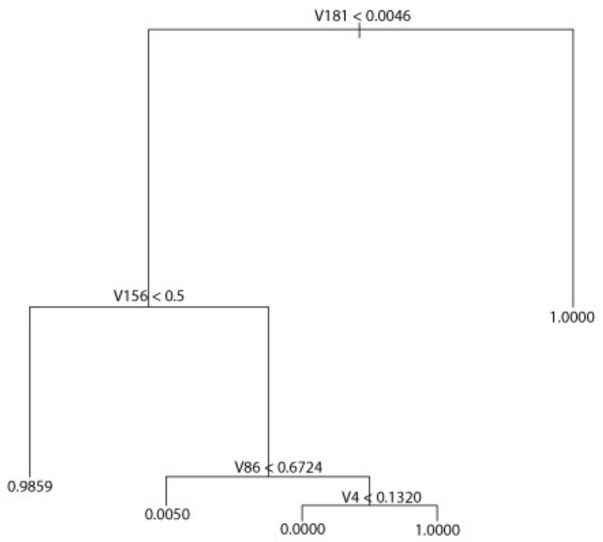

Figure 4: An example CART tree based on the Jtidy data set

An object is classified by traversing the tree from the root to a leaf. At each step of the traversal prior to reaching a leaf, the expression at the current node is evaluated. The left branch is taken if the expression evaluates to *true*, and the right branch is taken if it evaluates to *false*. When the object reaches a leaf, the predicted value of that leaf is taken as the predicted class for that object.

To create our classification trees we used the CART (Classification And Regression Tree) algorithm [3]. A brief summary of this algorithm is as follows:

Consider a training sample

$$L = \{(\mathbf{x}_1, j_1), \ldots, (\mathbf{x}_N, j_N)\}$$

where each $\mathbf{x}_i$ represents an execution profile, and j_i is the result (success/failure) associated with it.

- The deviance of a node $t \subseteq L$ is defined as: $d(t) = \frac{1}{N_t}\sum_{i=1}^{N_t}\left(j_i - \bar{j}(t)\right)^2$, where N_t is the size of t, and $\bar{j}(t)$ is the average value of j in t.
- Each node t is split into two children t_R and t_L. The split is chosen which maximizes the reduction in deviance. That is, from the set of possible splits S, the optimal split s^* is found by: $$s^* = \arg\min_{s \in S}\left(d(t) - \frac{N_{t_R}}{N_t}d(t_R) - \frac{N_{t_L}}{N_t}d(t_L)\right)$$
- A node is declared a *leaf node* if $d(t) \leq \beta$, for some threshold β.
- The predicted value for a leaf is the average value of *j* among executions in that leaf

With classification trees, we investigate three primary areas. First, we examine how useful these trees are as failure classifiers. Second, we evaluate the clustering the trees produce when used to classify failed executions. We also apply our refinement strategy to these clusterings. Finally, we investigate whether the sequence of decisions leading to a cluster provides useful diagnostic information for the failures in that cluster.

4.1. Experimental Methodology

For each of the subject programs described in section 3.4, we split the data into two disjoint subsets. 75% of the data was randomly selected as a *training set*, with the remaining 25% forming a *testing set*. To produce our classification trees, we used the implementation of CART available in the R statistics package [29]. For each program, we trained a CART tree on the training set and evaluated its performance using the testing set. The trees were trained to create many branches. Nodes with as few as two observations were split if they were heterogeneous. A node was declared a leaf when its deviance was less than 0.001. We found that these settings produced trees with enough branches (and therefore leaves) to classify well, without overfitting the training data.

4.1.1. Failure Classification. Each leaf of the CART tree is labeled with a *predicted value* which indicated the class associated with that leaf, in this case success or failure. If the majority of training executions in a leaf were successes, then this leaf will predict success, and vice versa.

4.1.2. Clustering. When a CART tree is used to classify objects, each object is assigned to a specific leaf of the tree. Since each leaf is associated with a different series of decisions based on profile features, multiple leaves predicting failures do so for different reasons. This seems to indicate different properties leading to failures. Therefore, we hypothesize that leaves in a CART tree group together failures with the same cause. We consider each of these leaves to be a cluster. To measure the quality of this clustering, we used the metrics of homogeneity and completeness as described in Section 3.

To refine the tree's clustering, we also use the merging strategy described in Section 3.2. Unlike dendrograms, though, CART trees do not support splitting of clusters. Since each cluster is a leaf of the tree, there are no subtrees to examine. It is often possible to identify clusters that contain only failures, but of mixed causes. These clusters should be split. However, since the cluster already contains only failures, the CART algorithm will consider it "perfect" and never split it. Because of this, we do not split the CART clusters and instead perform only merges.[4]

Also, since the tree is generated with the intent to separate failures from successes, there are often many leaves that contain only successes. When refining the clustering in order to better cluster the failures, these "success nodes" can be pruned off. However, this pruned tree can only be used for clustering failures, and not for general failure classification.

4.1.3. Diagnostic Information. We wish to investigate whether the decisions used to reach a node provide any useful diagnostic information for the executions within that node. Each decision expression relates a feature of the data set to a constant splitting value. Since the features used in our data represent the number of times a given function is called, we examine these functions to see if they point to the locations of the defects.

[4] It may be possible to apply an unsupervised clustering algorithm (e.g. k-means) to the members of the cluster in order to determine a good sub-clustering. However, we do not explore this possibility in this paper.

4.2. Experimental Results

Table 2: Performance of CART on Each Program's Testing Set

Data set	*Number of Failures*	*Correctly Predicted Failures*	*Number of Successes*	*Correctly Predicted Successes*
GCC	136	86.67%	3197	99.88%
Javac	233	77.59%	2907	97.39%
Jikes	225	67.31%	2924	97.14%
Jtidy	308	88.88%	7682	99.79%

4.2.1. Using CART Trees as Failure Classifiers. We examined the CART tree for each of the subject programs' training sets, and measured its performance on the program to measure its success at identifying failures. Our results are shown in Table 2. These results show that CART trees perform well as failure classifiers. The trees perform much better at predicting successes, but this is expected, given the comparatively small numbers of failures in each data set. However, the more important performance measure is the tree's effectiveness at predicting failures. Since the aim is to identify and fix all failures, it is preferred to have successes mispredicted as failures, rather than have failed executions ignored because they were mispredicted as successes. The trees generated in our experiments perform well at this, correctly identifying failures at least 67% of the time for all data sets.

4.2.2. Using CART Trees for Clustering. Using the metrics of homogeneity and completeness as defined above, we examined the clusterings produced by the trees for each data set. These results are summarized in Tables 3 and 4. Also, since singleton clusters are trivially homogeneous, we excluded these from the measurements and examined them separately.

Table 3: Homogeneity of CART clusters

Data set	*Percentage of non-singleton clusters with homogeneity greater than or equal to:*		
	.6	.8	1.0
GCC	90.48%	76.20%	71.43%
Javac	64.00%	56.00%	48.00%
Jikes	22.22%	13.89%	13.89%
Jtidy	90.91%	90.91%	72.73%

Table 4: Completeness of CART clusters

Data set	***Percentage of non-singleton clusters with completeness greater than or equal to:***		
	.6	.8	1.0
GCC	85.71%	66.67%	61.90%
Javac	56.00%	44.00%	28.00%
Jikes	61.11%	50.00%	38.89%
Jtidy	18.18%	18.18%	9.09%

In the cases of GCC and Jtidy, the clusters produced by the CART trees are fairly homogeneous. In both cases, at least 71% of the non-singleton clusters are completely homogeneous. Javac and Jikes do not perform as well, with 48% and 13.89% homogeneous non-singleton clusters, respectively. With Javac, however, there are a number of clusters that are only slightly heterogeneous. For example, 56% of clusters have homogeneity greater than 0.8, and 64% have homogeneity greater than 64%. The tree does a fair job, but the clusters are not completely homogeneous.

The completeness shows similar results. The GCC clusters perform reasonably well, with 61.9% of clusters complete. Coupled with its high homogeneity, this shows that the GCC clustering is a very good one. Jtidy has a very low number of complete clusters. However, the clusters are very homogeneous. This indicates that the clusters have been over-split and need to be merged. Javac and Jikes do not have very good completeness scores. This is primarily because only the completeness of non-singleton clusters is considered.

The GCC clustering contains no singletons and Jtidy has only four. On the other hand, the clusterings for both Javac and Jikes include a substantial number of singletons. Javac produced 43 clusters, 18 of which were singletons. Jikes produced 52 clusters, 16 of which were singletons. However, the majority of these were properly grouped. That is, there was only one failure of a given type, and it was properly clustered by itself. In Javac, 55.56% of these singletons were "correct", and in Jikes 75% were correct. From this aspect, the clustering for these data sets is much better than the statistics indicate.

The "correct" singletons discussed above are an example of a "perfect" cluster. These are clusters which are both 100% homogeneous and 100% complete. With GCC, fully 52.38% of the clusters are perfect, while Javac and Jikes have a respectable 32.56% and 25.00% of clusters that are perfect, respectively.

It should be noted that Jtidy is a bit different from the other subject programs. It is the only one where we have failure checkers that definitively report the presence or absence of a defect for each execution. These failure checkers look for the conditions known to trigger each defect. However, as a consequence of this, an execution may have more than one failure type associated with it. This can make it unclear as to what a "good" clustering looks like. For example, the largest cluster the tree creates contains 160 executions. Classifying each combination of defects as a different failure type, this cluster looks very heterogeneous. The largest causal group accounts for only 34% of the cluster. However, each execution in the cluster triggers the "newNode" defect, possibly in combination with some others. From this point of view, the cluster is completely homogeneous. This observation increases the percentage of homogeneous Jtidy clusters to 86.67%.

After applying our refinement strategy as outlined in Section 4.1.2, we found that only a small number of clusters qualified for merging. Javac contained two pairs of clusters to merge, and Jtidy had three pairs. Both GCC and Jikes did not have any clusters to merge. This seems to indicate that only minor improvements can be made through the merging of clusters. In other words, the hierarchical structure of the tree is not particularly useful for refining the clustering.

4.2.3. Using Tree Nodes for Diagnosing Failures. It is natural to ask whether the sequence of decision nodes leading to a cluster provides useful diagnostic value for debugging the failures contained in that cluster. After all, each decision is based upon the frequency with which a specific function is called. It seems reasonable that the collection of these functions should inform the developer of the location in the code that the defect resides. In our experiments, though, we found that this is simply not the case. In the case of Jtidy, where the exact locations of the defects are known, the functions selected for decisions do not point to these locations. Although the selected functions may bear some relationship to the true location of the defect, it is generally too indirect to be helpful.

Furthermore, some of the decisions in the tree may be based upon spurious correlations among the executions. A good example of this occurs with GCC. The top decision node splits the data based on whether or not there was a function call in the code. While it is true that all of the members of this cluster are failures, and none of them call functions, this is not related to the cause of the defect. Because of spurious correlations like this, we conclude that the decision nodes are not useful for diagnosing the causes of failures.

5. Related work

Several previous papers have addressed issues closely related to failure classification. Agrawal, *et al* describe the χ*Slice* tool, which analyzes system tests to facilitate location of defects [1]. χ*Slice* visually highlights differences between the *execution slice* of a test that induces a failure and the slice of a test that does not. Reps, *et al* investigate the use of a type of execution profile called a *path spectrum* for discovering Year 2000 problems and other kinds of defects [30]. Their approach involves varying one element of a program's input between executions and analyzing the resulting spectral differences to identify paths along which control diverges. Jones, *et al* describe a tool for defect localization called *Tarantula*, which uses color to visually map the participation of each statement in a program in the outcome of executing the program on a test suite [21].

Podgurski, *et al* used cluster analysis of profiles and stratified random sampling to improve the accuracy of software reliability estimates [28]. Leon, *et al* describe several applications of multivariate visualization in *observation-based* (software) *testing*, including analyzing synthetic test suites, filtering operational tests and regression tests, comparing test suites, and assessing bug reports [24]. Dickinson, *et al* present a technique called *cluster filtering* for filtering test cases [8][9]. This technique involves clustering profiles of test executions and sampling from the resulting clusters. They present experimental evidence that cluster filtering is effective for finding failures when unusual executions are favored for selection. Note that the aforementioned work on observation-based testing differs from the work reported here in three main respects:

- The goal of the previous work was to identify possible failures in set of mostly successful executions. The goal of the current work is to identify groups of failures with closely related causes among a set of reported failures.
- The previous work did not involve user feedback; the current work depends upon failure reports from users.
- The previous work applied unsupervised pattern classification techniques to complete program profiles. The current work uses supervised pattern classification techniques to select relevant profile features prior to clustering or visualization.

Hildebrandt and Zeller describe a *delta debugging* algorithm that generalizes and simplifies failure-inducing input to produce a *minimal test case* that causes a failure [14]. Their algorithm, which can be viewed as a feature-selection algorithm, is applicable to failure classification in the case that failure-causing inputs reported by different users simplify to the same minimal failure-causing input. Note that Hildebrandt and Zeller's approach requires an automated means of detecting whether a simplified input causes the same kind of failure as the original input. In [33], Zeller describes another form of delta debugging that isolates the variables and values relevant to a failure by systematically narrowing the state difference between a passing run and a failing run.

Chen *et al* present a dynamic analysis methodology for partially automating problem determination in large, dynamic Internet services, which involves course-grained tagging of client requests and the use of cluster analysis to identify groups of components that tend to be used together in failed requests but not in successful ones [4]. They also describe a framework called *Pinpoint* that implements the methodology on the J2EE platform.

Liblit *et al* present a technique for isolating "deterministic" bugs by starting with an initial set of predicates characterizing a program's state at various execution points and then eliminating irrelevant predicates using a set of elimination strategies that are applied to a mixture of successful and unsuccessful runs [7]. They also present a technique for isolating "non-deterministic" bugs that employs a logistic regression classifier. In [34], the same authors describe a single classification utility function that integrates multiple debugging heuristics and can penalize false positives and false negatives differentially. Liblit *et al*'s approach corresponds roughly to the first two phases of our basic classification strategy (see Section 2). Unlike our approach, theirs does not distinguish between failures with different causes and attempt to group them accordingly.

Microsoft Corporation has developed a tool called *ARCADE* that attempts to automatically classify crashes and hangs reported from the field into buckets, each corresponding (ideally) to a unique defect [31]. Crashes/hangs are sorted into buckets based on the contents of *minidumps* produced by the Watson failure-reporting mechanism [26]. Relatively few minidump fields are used in classification. However, failures that do not cause a crash or hang are not reported and classified at present.

Other related work addresses the problem of correlating events or alarms generated by a distributed system or an intrusion detection system, for the purpose of fault localization. Gruschke proposes an event correlation system that groups events according to information in a knowledge base derived from a dependency graph of a distributed system [12]. Yemini *et al* detail their *Distributed Event Correlation System (DECS),* which defines the elements of a system, the relationships between them, and the possible events that can be raised in a codebook that represents the dependency graph formed by the relations between events [32]. Bouloutas, *et al* describe a general framework for designing a fault localization system that uses dependency graphs and heuristic algorithms for alarm correlation [2]. Brown, *et al* describe *Active Dependency Discovery (ADD),* which verifies dependency graphs of a distributed system by systematically perturbing the system and measuring the cascading effects of perturbations in the system [4]. Analysis of these effects results in information about how statistically likely a node is likely to affect other points on the network when it experiences a failure. Finally, Julisch and Dacier demonstrated that a form of *conceptual clustering* is effective for grouping similar alarms from intrusion detection systems (IDS) [23].

6. Conclusions

We have presented two new tree-based techniques for refining an initial classification of software failures based on execution profiles. One of these techniques uses dendrograms to guide the refinement process; the other employs classification trees to classify failures initially and to guide refinement of this classification. Experimental results were presented suggesting that both techniques are effective for grouping together failures with the same or similar causes, although our results suggest that refinement of the grouping induced by a classification tree may not be beneficial. Our results also suggest that examination of the individual decisions made by a classification tree used to group failures is not helpful for diagnosing the cause of the failures. Significant additional experimental work with a wide variety of subject programs and failure types is needed to confirm these results.

We found that the majority of clusters whose least similar failures had the same cause were indeed homogeneous. Splitting and merging clusters appropriately had a significant positive effect on both the overall homogeneity of each cluster and the separation of failures with different causes.

For dendrograms, the largest homogeneous subtree in each cluster was contained in the cluster's largest causal group, both before and after splitting and merging operations took place. Also, for those data sets that had more than one singleton failure, correct classification of singletons by a dendrogram improved after splitting operations.

Future exploration of using dendrograms for clustering refinement will involve more exhaustive use of heuristics involving the siblings of clusters and the way the height of their parents affects whether to merge or split them. We also envision writing a tool for visualizing dendrograms that is specifically made for software testing research.

For classification trees, future work will involve exploration of different methods of refining the trees' implicit clusterings, possibly through incorporating traditional clustering techniques.

7. References

[1] Agrawal, H., Horgan, J.J., London, S., and Wong, W.E. Fault location using execution slices and dataflow tests. 6th IEEE Intl. Symp. on Software Reliability Engineering (Toulouse, France, October 1995), 143-151.

[2] Bouloutas, A. T., Calo, S., and Finkel, A. Alarm correlation and fault identification in communication networks. IEEE Transactions on Communication 42, 2/3/4 (1994), 523-533.

[3] Breiman, L., J. Friedman, R. Olshen, and C. Stone, 1984. *Classification and Regression Trees,* New York: Chapman and Hall.

[4] Brown, A., Kar, G., and Keller, A. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. 7th IFIP/IEEE Intl. Symposium on Integrated Network Management (Seattle, WA, May 2001).

[5] Byte Code Engineering Library, http://jakarta.apache.org/bcel/, Apache Software Foundation, 2002 - 2004.

[6] Calinski, R.B. and Harabasz, J. A dendrite method for cluster analysis. Communications in Statistics 3, 1-27.

[7] Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: problem determination in large, dynamic Internet services. 2002 International Conference on Dependable Systems and Networks (Washington, D.C., June 2002).

[8] Dickinson, W., Leon, D., and Podgurski, A. Finding failures by cluster analysis of execution profiles. 23rd Intl. Conf. on Software Engineering (Toronto, May 2001), 339-348.

[9] Dickinson, W., Leon, D., and Podgurski, A. Pursuing failure: the distribution of program failures in a profile space. 10th European Software Engineering Conf. and 9th ACM SIGSOFT Symp. on the Foundations of Software Engineering (Vienna, September 2001), 246-255.

[10] GCC. The GCC Home Page, www.gnu.org/software/gcc/gcc.html, Free Software Foundation, 2004.

[11] Google Groups, http://groups.google.com/, Google, Inc., 2004.

[12] Gruschke, B. A new approach for event correlation based on dependency draphs. 5th Workshop of the OpenView University Association: OVUA'98, Rennes, France, (April, 1998).

[13] Hierarchical Clustering Explorer 2.0, http://www.cs.umd.edu/hcil/hce/hce2.html, Human-Computer Interaction Lab, University of Maryland, 2004.

[14] Hildebrandt, R. and Zeller, A. Simplifying failure-inducing input. 2000 Intl. Symp. on Software Testing and Analysis (Portland, August 2000), 135-145.

[15] Jacks, International Business Machines Corporation, Jacks Project, www.ibm.com/developerworks/oss/cvs/jacks/, 2002.

[16] Jain, A.K. and Dubes, R.C. Algorithms for Clustering Data, Prentice Hall, 1988.

[17] Java Language Specification, Sun Microsystems, Inc., java.sun.com/docs/books/jls/second_edition/html/j.title.doc .html, 2000.

[18] JavaTM Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html, 2001.

[19] Javac, Sun Microsystems Inc., Java™ 2 Platform, Standard Edition, java.sun.com/j2se/1.3/, 1995 – 2002.

[20] Jikes, IBM developerWorks, www-124.ibm.com/ developerworks/opensource/jikes/, 2002.

[21] Jones, J.A., Harrold, M.J., and Stasko, J. Visualization of test information to assist fault localization. 24th International Conference on Software Engineering (Orlando, May 2002).

[22] Jtidy, http://jtidy.sourceforge.net, World Wide Web Consortium (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), 1998-2000

[23] Julisch, K. and Dacier, M.. Mining intrusion detection alarms for actionable knowledge. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Edmonton, Alberta, July 2002).

[24] Leon, D., Podgurski, A., and White, L.J. Multivariate visualization in observation-based testing. 22nd Intl. Conf. on Software Engineering (Limerick, Ireland, June 2000), ACM Press, 116-125.

[25] Liblit, B., Aiken, A., Zheng, A.X., and Jordan, M.I. Bug isolation via remote program sampling. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (San Diego, June 2003).

[26] Microsoft Corporation. Microsoft Error Reporting: Data Collection Policy. http://watson.microsoft.com/dw/1033/dcp.asp (January, 2003).

[27] Podgurski, A., Leon, D., Francis, P., Minch M., Sun, J., Wang, B. and Masri, W. Automated support for classifying software failure reports. 25th International Conference on Software Engineering (Portland, OR, May 2003).

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

[29] The R Project for Statistical Computing. http://www.r-project.org

[30] Reps, T., Ball, T., Das, M., and Larus, J. The use of program profiling for software maintenance with applications to the Year 2000 Problem. 6th European Software Engineering Conf. and 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering (Zurich, September 1997), 432-449.

[31] Staples, M. and Hudson, H. Presentation at 2003 Microsoft Research Faculty Summit (Bellevue, WA, July 2003), https://faculty.university.microsoft.com/ 2003/uploads/496_115_Lassen_Trustworthiness_ Staples.ppt.

[32] Yemini, S. A., Kliger, S., Mozes, E., Yemini, Y., and Ohsie, D. High speed and robust event correlation. IEEE Communications Magazine (May 1996), 82-90.

[33] Zeller, A. Isolating cause-effect chains from computer programs. ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (Charleston, SC, November 2002).

[34] Zheng, A.X., Jordan, M.I., Liblit, B., and Aiken, A. Statistical debugging of sampled programs. Neural Information Processing Systems (NIPS) 2003 (Vancouver and Whistler, British Columbia, Canada, December 2003).

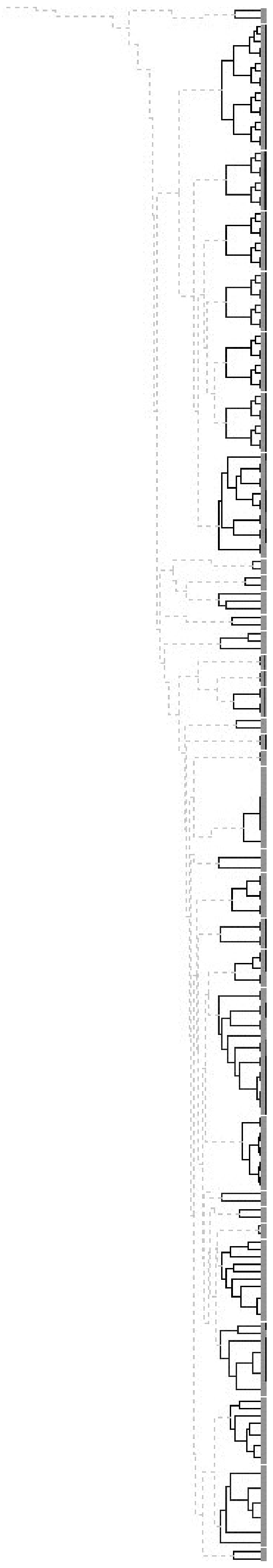

Figure 5: GCC Dendrogram with 27 clusters

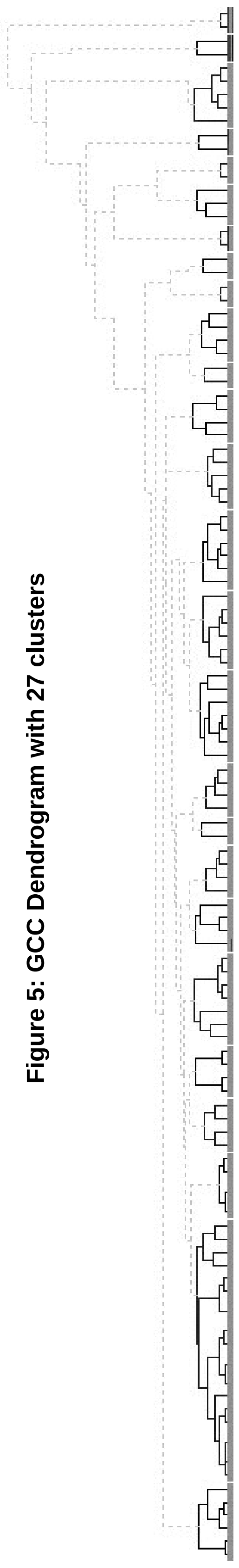

Figure 6: Javac Dendrogram with 34 clusters

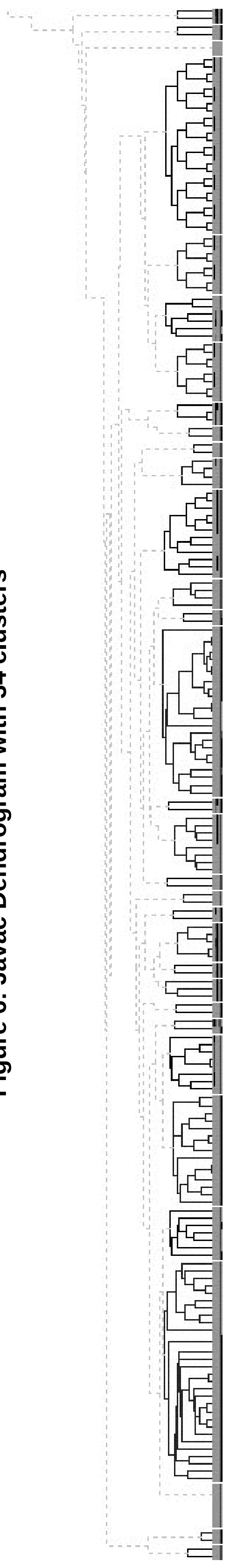

Figure 7: Jikes Dendrogram with 33 clusters

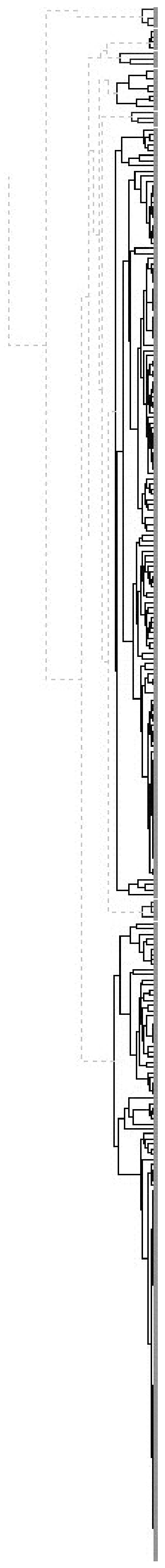

Figure 8: Jtidy Dendrogram with 8 clusters