Dendrogram-Based Methods For Clustering Refinement

by

Melinda Minch

Submitted in partial fulfullment of the requirements

For the degree of Master of Science

Thesis Advisor: Dr. Andy Podgurski

Department of Electrical Engineering and Computer Science

Case Western Reserve University

August, 2005

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Melinda Minch

candidate for the Master's degree.

(signed) _____
(Chair of the Committee)

_____

_____

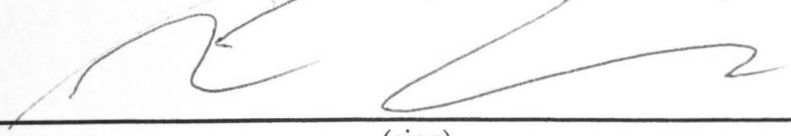
_____

_____

_____

(Date) 20 June 2005

I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.



(sign)

To R.C., who has come here, and also gets to go back again.

Contents

List of Tables

List of Figures

Many thanks to Debbie and Gary Minch, Eldan Goldenberg, Vinay Augustine and Pat Francis.

Dendrogram-Based Methods For Clustering Refinement

Abstract

by

MELINDA MINCH

Recent research has addressed the problem of providing automated techniques for classifying instances of software failures, so that failures with the same cause are grouped together. In the first part of this paper, a new method is presented for refining an initial classification of failures. This technique is based on the use of dendrograms, which are rooted trees that are used to represent the results of hierarchical cluster analysis. I also report the results of experimentally evaluating these techniques on several subject programs.

Chapter 1

Introduction

Software testing often involves making sense of and drawing the largest possible number of inferences from large amounts of information: the results of thousands of unit tests, scores of calculated metrics, and feedback taken from thousands, if not millions, of users. The problem of *failure classification* is one such scenario. Testers may need to classify large numbers of software failures that have been reported by users of released software in the field, or gathered from executing a test suite on software that is still being developed. Research conducted recently by the Software Engineering Lab at Case Western Reserve University shows that whatever the source of these failures, it is likely that many of them are caused by the same set of software defects, and that failures with the same or related causes can be grouped together [24]. By determining these groups before the causes of the failures are known, software testers can determine which failures were caused by the same defects, the number of defects that are represented by the failures and which failures can be used to diagnose a specific defect. Grouping failures in such a manner can also aid in the prioritization of defect resolution by showing which faults cause the most failures and which defects cause a failure with the most frequency.

Automated failure classification can be performed using data mining, pattern classification and multivariate data analysis techniques to analyze the *execution profiles* of successful and failed software runs [24]. Such a procedure requires two types of information other than the execution profiles: *auditing information*, such as bug reports, that can be used to confirm failure reports; and *diagnostic information*, such as stack traces, that can be used to find the causes of the software failures. Empirical studies of automated failure classification show that it is effective for clustering failures that share the same cause. After the failures have been clustered, the strategy outlined in [24] employs manual inspection of selected failures in order to confirm the effectiveness of the clustering, or to refine it. According to certain heuristics, clusters may be split into two or more new clusters, or merged together.

The technique presented in [24] does not include a definitive way to select an appropriate number of clusters into which the set of failures should be divided, nor does it explicitly describe how clusters should be split or merged in order to refine the classification. The number of defects in each program under test is not known before clustering happens, so the number of clusters that the set of failures should be divided into is decided by making an informed guess. For these reasons, a method of refining the number of clusters in a set of software failures after it has been classified would be useful.

Such a method can be constructed with the use of a *dendrogram* [14]. Strictly defined, a dendrogram is any treelike structure used to represent a hierarchy. For the purpose of this work, a dendrogram is a graph in the form of a tree that is used to indicate the similarity of each of its elements. Each individual element in the dendrogram is represented with a leaf of the tree. The height from the root of the tree to a particular subtree in the dendrogram indicates the similarity of the elements in that subtree to the elements in other subtrees. If two subtrees are

very much alike, they will join at a level that is close to the leaves of the tree; if two subtrees are dissimilar, they will join closer to the root of the tree. By creating dendrograms that represent the similarity of execution profiles and examining the way in which the profiles are arranged in subtrees, the similarity of clusters to each other and to other groups of executions may be evaluated. This process can aid in refining an existing clustering of execution profiles, or in developing a good clustering for failed executions.

The failure classification strategy that is used to create the original clusterings in this study is described in Chapter 2. Chapter 3 contains an explanation of the ways in which dendrograms can be used for clustering refinement. A description of the experimental procedure and the software used in the experiments is in Chapter 4, and Chapter 5 outlines the experimental results. Finally, the Appendix contains images of the dendrograms created for each of the subject programs.

Chapter 2

Basic Classification Strategy

This chapter delineates the basic failure classification strategy used in [24]. If m failures are reported by users or discovered using a test suite during a finite time period, then it is likely that these failures are due to a substantially smaller number k of distinct defects. To simplify matters for the purpose of this explanation, it is assumed that each failure is caused by only one defect, that defects are distinct from each other, and that all failures are accurately reported. If $F = \{f_1, f_2, \ldots, f_m\}$ is the set of reported failures, then F can be partitioned into $k < m$ subsets $F_1, F_2, \ldots, F_k$ such that all of the failures in F_i are caused by the same defect d_i for $1 \leq i \leq k$. This partitioning is called the *true failure classification*. The basic failure classification strategy is used to approximate the true failure classification, as follows [24]:

1. The software under test is instrumented to collect either execution profiles or captured executions, and transmit them to the developer. It is then deployed to users or tested with some kind of test suite.

2. The execution profiles of reported failures are combined with a random sample of execution profiles from successful executions of the program. These are taken from profiles of operational executions for which no failures were

reported. This set of successful and unsuccessful execution profiles is then analyzed to select a subset of all profile *features* to use in grouping related failures. A feature of an execution profile is a single element of the profile; if an execution profile consists of function call counts, for example, each feature would be the number of times a certain function is called during that particular execution of the software under test, and the execution profile would have as many features as there are functions in the program. 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 perform best overall.

3. The execution profiles are grouped together according to the similarity of their pared-down feature sets, using a similarity metric such as Euclidean distance or Manhattan distance. This is accomplished using mutlivariate visualization techniques and cluster analysis.

4. The clustering of failed executions that results is examined in order to confirm it, or if necessary, refine it.

The clustering created by this process is a partition $C = \{G_1, G_2, \ldots, G_n\}$ of F, which is known as the *approximate failure classification*. For C to be useful in software testing, all or most of the groups should consist mostly or entirely of failures with the same or closely related causes. Such a clustering can be refined further using the techniques outlined in this paper.

Chapter 3

Attributes of Dendrogram Clusters

This chapter describes in more detail the properties of dendrograms that are useful for clustering refinement, and how to analyze and apply them to techniques for refining a clustering of software failures. A clustering of execution failures is the most useful when each cluster contains failures that were all caused by the same defect, and when all of the failures elicited for one defect are contained in one distinct cluster. The attributes of dendrogram clusters that are explored in this section are all relevant to refining a clustering toward this end.

Dendrograms are made up of subtrees, and those subtrees, in turn, have subtrees nested within them. Each cluster of execution profiles in a dendogram comprises a subtree of the dendrogram, and each subtree or cluster has several attributes that can be examined and used in the refinement technique. By examining the way in which executions are arranged in clusters and subtrees, their similarity to each other and to other clusters may be evaluated. The height of any subtree in a dendrogram 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.

Every cluster in the dendrogram is composed of failures with one or more

causes; Section 4.2 contains more detail about determining the cause of a failed execution. A cluster's *largest causal group* is the largest set of failures within a cluster that were caused by the same defect. 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 will have subtrees within them as well. A cluster's *largest homogeneous subtree* is the largest set of failures within the cluster that were caused by the same defect and compose a distinct subtree in the cluster. It is desirable for each cluster to have one large homogeneous 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, meaning that the cluster is completely homogeneous. A cluster is still desirable, however, if a majority of its executions have the same cause as those in the largest homogeneous subtree. If a set of execution profiles is clustered too coarsely, some clusters may have two or more large homogeneous subtrees of different failure types. Such clusters should be split at the level where their two largest homogeneous subtrees are connected, so that these subtrees become *siblings* as in Figure 3.1. Typically, these large subtrees have their closest common ancestor at the highest level in the cluster that contains them.

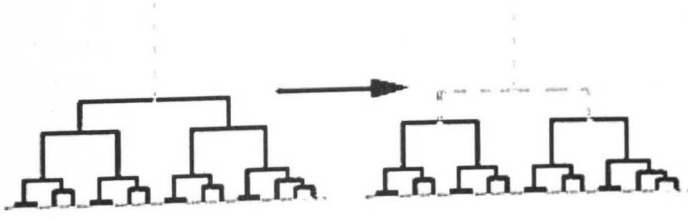


Figure 3.1: Splitting a cluster.

Each cluster in the dendrogram has one sibling, which is the subtree to which it is the most closely related; that is, the subtree with which it shares a parent. Such a subtree may contain one or more clusters. Figure 3.2 shows an example of a sibling that contains multiple clusters: A and B are siblings, and the subtree that contains both A and B is C's sibling.

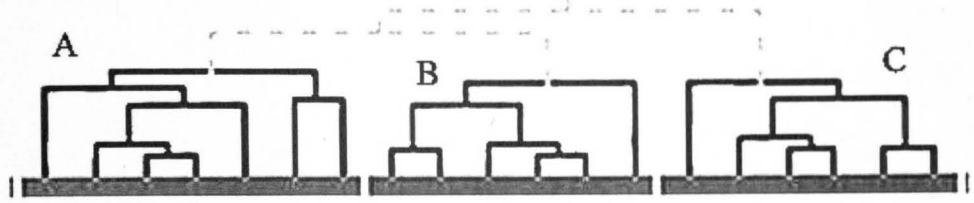


Figure 3.2: Illustration of siblings.

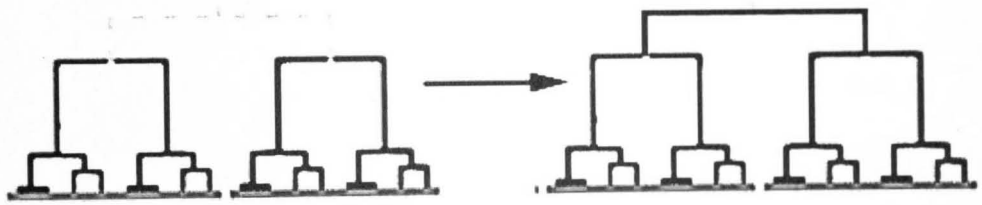


Figure 3.3: Merging two clusters.

During this study, I observed that if a cluster's sibling contains a group of clusters, it is usually not similar to other clusters in a way that is significant for failure classification. In other words, the clusters within the group are more closely related to each other than they are to the cluster that is a sibling of the group. In such a case, one should not expect the failures in the lone sibling cluster to have causes in common with the failures in the group of clusters. In addition, I saw that if the clustering for a set of execution profiles is too finely grained, siblings may be individual clusters containing failures that were caused by the same defect. The parent of such siblings may be at a low level in the dendrogram in comparison to the level at which other clusters join. Such siblings should be merged at the

level of their parent. Merging clusters in such a manner means that the clusters' large homogeneous subtrees are merged, resulting in a cluster that has one largest homogeneous subtree.

Chapter 4

Evaluating a clustering with the dendrogram

The strategy for using dendrograms to refine an initial classification of failures has three phases:

1. Select the number of clusters into which the dendrogram will be divided, using a method such as the Calinski-Harabasz[3] metric, which is a measure of the quality of a clustering:

$$CH(c) = \frac{B/(c-1)}{W/(n-c)}$$

where c is the number of clusters, n is the total number of executions in the data set, B is the *between-cluster* sum of squared distances according to the chosen distance metric, W is the *within-cluster* sum of squared distances from the cluster centroids.

2. Examine the individual clusters for homogeneity by choosing the two executions in each cluster with *maximally dissimilar profiles* according to the chosen similarity metric, and determining whether these two failures have the same

cause. If the selected failures were caused by the same defect, it is probable that all of the other failures in the cluster were as well.

3. Choose appropriate candidates for splitting and merging of clusters, according to the properties outlined in the previous section.

   (a) Split a cluster if it is found to be non-homogenous, and one or both of the resulting clusters would be a non-singleton or contain a largest homogeneous subtree.

   (b) Merge two clusters if they are homogeneous, are siblings, and if their failures were caused by the same defect.

The clusters with the largest amount of internal dissimilarity should be examined first. Internal dissimilarity may be measured using the two maximally dissimilar profiles, or by calculating the average similarity between all individual execution profiles in the cluster. If the clusters with high internal dissimilarity are homogenous, it is reasonable to assume that the others are as well, though it is still useful to examine clusters with more internal similarity. If a cluster resulting from a splitting operation is still an appropriate candidate for splitting, it may be advantageous to split the new cluster as well. Doing so has the effect of splitting the original cluster twice.

4.1 The Hierarchical Clustering Explorer

Hierarchical Clustering Explorer 2.0 (HCE) [10], a tool from the University of Maryland, was used to generate the dendrograms featured in this thesis. HCE allows the user to classify multidimensional data into an arbitrary number of clusters after a hierarchical clustering algorithm is applied. This is done by selecting the minimum similarity that executions must have in order to be clustered together.

A low minimum similarity will result in fewer clusters and a short distance from the root of the tree to the root of each cluster; likewise, a high minimum similarity will result in many small or singleton clusters, with roots close to the leaves of the dendrogram. HCE also provides several other features to aid in cluster evaluation, including labels for each execution profile, an in-depth view of each execution's feature set, a detail cutoff that gives a "big picture" view of the clusters, and a "zoom" bar.

The following steps are necessary to generate and analyze a dendrogram of execution profiles taken from one of the subject programs listed in Section 4.2 using Hierarchical Clustering Explorer:

1. Import a tab-delimited file containing a set of execution profiles into HCE.
2. De-select the option that says to normalize.
3. Choose *UPGMA (Unweighted Pair Group Method with Arithmetic Mean)* as the clustering algorithm and Euclidean distance as the similarity metric.
4. If the imported file includes a column indicating whether each execution was a failure, uncheck that column in the "Check columns to cluster" box.
5. Select the number of clusters to divide the dendrogram into by dragging the *minimum similarity bar* to the appropriate height in the dendrogram. The number and placement of the clusters generated is determined by the sub-trees below that level of the dendrogram.
6. Evaluate the goodness of the clustering with the techniques outlined in the next section.

4.2 Subject programs and test suites

Four subject programs were used for this study: the GCC compiler for C [8]; the Javac [17] and Jikes [18] Java compilers; and JTidy [20], a Java port of the popular HTMLTidy utility [12]. These programs were chosen for several reasons: they can be executed repeatedly with a script; source code for a number of versions is available; and failures for GCC, Javac and Jikes can be detected by instrumenting the compiler's code or by using self-validating test suites. Unfortunately, failure reports from ordinary users of these subject programs were unavailable. The classification strategy outlined in Chapter 2 was applied to failures detected by self-validating tests of the compilers, and to failures arising from operational inputs for JTidy.

Version 2.95.2 (Debian GNU/Linux) of the GCC compiler for C was used. Only the C compiler proper was profiled, by executing it on a subset of the regression test suite for GCC that consists of tests which actually execute compiled code. This set of tests came from the test suite shipped with GCC 3.0.2, which includes tests for defects still present in version 2.95.2. GCC was executed on 3,333 tests, and failed 136 times. The Jacks test suite, as of 15 February, 2002 [13], which tests adherence to the Java Language Specification [15], was used to test the Java compilers. Version 1.15 of Jikes was executed on 3,149 tests and failed 225 times; Javac build 1.3.1_02-b02 was executed on 3,140 tests and failed 233 times. It should be noted that the Jacks test suite contains tests that are specific to the Jikes and Javac compilers, and that tests specific to one compiler are only executed when that compiler is tested by Jacks. Version 3 of JTidy was executed on 7,990 HTML and XML files that were amassed from the Internet, and failed 308 times. Input for JTidy was gathered by retrieving random files from Google Groups [9] 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, two different profilers implemented by members of the Software Engineering Lab at CWRU were used. The Javac profiler used the Java Virtual Machine Profiler Interface [16], and the profiler written for JTidy instrumented JTidy's code using the Byte Code Engineering Library[2].

The failures for the GCC, Javac, and Jikes data sets were manually classified in [24]. 26 defects were identified for GCC, 67 defects were identified for Javac, and 107 defects were identified for Jikes. For JTidy, the project's bug database was examined in order to find examples of defects with known fixes that were still present in version 3, and five of those defects were selected for study. An oracle version of JTidy which includes both fixes and *failure checkers* for these defects was created. These failure checkers detect the triggering conditions for each defect and report when those conditions are satisfied during execution. Through the use of this oracle, it could be determined whether any of these five defects were triggered by the set of operational data under version 3. Unlike the other subject programs, JTidy sometimes failed due to a combination of two or more of the studied defects. Nine such combinations of defects were observed.

Chapter 5

Experimental results

In order to confirm that the strategy detailed in Chapter 4 works, dendrograms were created for the four subject programs, and they were used to evaluate their corresponding clustering of failed executions. Seven metrics were used to evaluate the strategy both before and after applying the changes detailed by step 3:

- Average percentage of failures in each cluster that were caused by the same defect as the failures in the cluster's largest homogeneous subtree.
- Average percentage of each cluster's failures that are actually contained in its largest homogeneous subtree.
- Percentage of clusters in the dendrogram that are homogeneous clusters.
- Overall percentage of executions that are in a homogeneous cluster.
- Average *completeness* of clusters in the dendrogram- the completeness of a cluster is the percentage of the total number of failures that were caused by a specific defect that are contained in that cluster. If a cluster contains a failure that was caused by a certain fault, it must also contain every other failure caused by that fault to be considered 100% complete.

- Number of singleton clusters.
- Percentage of *singleton failures*, or failures that are the only failure in the data set that were caused by their particular defect, that are correctly classified by the dendrogram as singletons.

Since the proposed use of dendrograms is for refining existing clusterings, the initial number of clusters in each dendrogram was chosen using the classification methods outlined in Chapter 2. If the chosen number of clusters could not be shown with the Hierarchical Clustering Explorer due to limitations in the software, the next largest number of clusters was used. Singleton clusters are discarded in the analyses of homogeneity and completeness, as all singleton clusters are homogeneous by definition and singletons are not counted as clusters by HCE. Several clusters in these dendrograms that were appropriate candidates for splitting could have been split twice or more; the number of split operations was restricted to two per original cluster for the purpose of this study, due to time limitations. Images of the dendrograms for each subject program can be found in the Appendix.

The results of the experiment are outlined in Table 5.1. The first section of the table describes the entire dendrogram for each subject program both before and after the splitting and merging operations took place. It includes the number of clusters in the dendrogram as well as the 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.

Table 5.1: 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%

* The data set contained only one singleton failure.

5.1 GCC results

The initial dendrogram for GCC produced 27 clusters, 85% of which were homogeneous. All of the clusters whose two maximally dissimilar failures were caused by the same defect were homogeneous. Four clusters were found that should be split according to the heuristics in Chapter 4, and four pairs of clusters that should

be merged were also found. As Table 5.1 shows, the operations performed on the GCC dendrogram improved the overall quality of the clustering, most notably in the new clusters resulting from a splitting operation. All of the new clusters that resulted from two homogeneous clusters being merged with their siblings remained homogeneous. 75% of those resulting clusters had subtrees containing groups of clusters as their siblings, which means that these resulting clusters were both dissimilar from their siblings and homogeneous: marks of a useful clustering. The data from Table 5.1 makes the dendrogram for GCC appear to do very well with respect to classifying singleton failures, but the GCC data set contains only one such singleton in reality, meaning that too many singleton clusters were generated by the dendrogram. It is worth noting, however, that no new singleton clusters were generated by splitting the original clusters.

5.2 Javac results

The initial dendrogram for Javac produced 34 clusters, 65% of which were homogeneous. There was one heterogeneous cluster whose two maximally dissimilar failures had the same cause. No pairs of clusters were found appropriate to merge, though there were 9 candidates for splitting in the initial Javac dendrogram. The results for Javac indicate the need for a clustering that is more fine-grained than the original one, as there were no clusters to be merged, and an increase in the number of clusters as a result of split operations caused an increase in all measures of homogeneity for the dendrogram.

5.3 Jikes results

The initial dendrogram for Jikes produced 33 clusters, 48% of which were homogeneous. All of the clusters whose two maximally dissimilar failures had the same cause were homogeneous. There were no appropriate pairs of clusters to merge, and there were 13 clusters that were appropriate candidates for splitting found in the Jikes dendrogram. Like Javac, the results for Jikes indicate a need for a finer clustering, though almost all of the splitting operations resulted in one or more singleton clusters. In fact, a third of the failures in the Jikes data set are singletons, so splitting operations will create many singleton clusters that are valid. For this reason, a large number of splitting operations must be performed to produce clusters in the Jikes dendrogram that are homogeneous and distinct.

5.4 JTidy results

The initial dendrogram for JTidy produced 8 clusters, 38% of which were homogeneous. There were no pairs of clusters to merge, and there were 5 clusters that were appropriate candidates for splitting. The data in Table 5.1 makes the JTidy dendrogram appear to do badly at classifying singleton failures, but the JTidy data set contains only one singleton failure. The JTidy data set also contains both the largest number of failures and the smallest number of different failure causes of the data sets from the four subject programs. The number of failures associated with each cause range from 1 to 79. The original clustering for JTidy put the rarest failures together in small homogeneous clusters, and put the most common failures together in large heterogeneous clusters. This resulted in clusters for which overall quality was not affected by one or two splitting operations. The JTidy clustering might be improved through a repeated sequence of splitting, re-examining, and eventually merging clusters so that the failures are arranged more appropriately.

5.5 Summary of experimental results

In general, 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, the majority of clusters whose least similar failures were caused by the same defect were indeed homogeneous. The best initial dendrogram clustering made by HCE seems to have been the one for GCC, followed by Javac, Jikes and finally JTidy. The appropriate use of splitting and merging operations had a significant positive effect on both the overall homogeneity of each cluster and the separation of failures with different causes into different clusters. For those data sets that had more than one singleton failure, the correct classification of singletons improved after splitting operations.

Chapter 6

Future work

Further exploration of using dendrograms for clustering refinement may include more exhaustive use of the following heuristics to refine clusterings:

- If the parent of a cluster and its sibling is at a high level in the dendrogram relative to its children, then it is likely that its children are dissimilar enough to merit remaining separated, instead of being combined into one large cluster. Similarly, having two subtrees within a cluster which has its root at a much higher level than the roots of the subtrees is an indication that the subtrees may be good candidates for splitting.

- A cluster's sibling might also be a subtree that contains a group of clusters, or one cluster that is composed primarily of failures that were not caused by any of the same faults as the failures in the lone sibling cluster. Again, this indicates that the siblings are dissimilar enough to merit being separate clusters.

- If a cluster contains only one large homogeneous subtree, and that large subtree makes up a majority of the cluster, this indicates that the cluster is homogeneous, and should not be split into two or more clusters.

- A majority of points in the cluster that are not contained in the largest homogeneous subtree may be failures caused by the same defect as those in the largest subtree, or have a cause in common with it. Again, this indicates homogeneity.

Another potentially useful project would be to create a piece of software that has functionality that is similar to that of the Hierarchical Clustering Explorer, but that is made specifically for use in software testing research. Though HCE was invaluable for the purpose of this study, it is intended for use in the field of bioinformatics, and the limitations and functionality of the software reflect this. Such a tool might exclude some of the functionality that went unused in HCE, and include some additional features:

- The user could be allowed to specify the number of clusters before clustering is done, if desired.
- The "zoom" and "stretch" capabilities of HCE might be extended.
- Different information could be displayed when the user clicks on a cluster, such as a listing of the cluster's two least similar executions, nearest neighbors and size.
- Individual clusters might be numbered or labeled.

It may also be useful to assign for each subject program a maximum dissimilarity. This would be the maximum dissimilarity, according to the metric used by the hierarchical clustering algorithm to create an initial clustering, that the two least similar executions in any homogenous cluster are allowed to have after a clustering has been refined using the strategy in Chapter 4. This would allow a more automated refinement of future clusterings of each subject program.

An in-depth exploration of whether long sequences of splitting and merging clusters further improves failure classifications would also be worthwhile. Such a study would concentrate the utility of such a technique for refining clusterings of data sets with large numbers of singletons or data sets that include both very rare and extremely common failures, such as Jikes and JTidy respectively. To this end, a heuristic for limiting the number of splits should be defined, and its effectivity tested in such a study.

Chapter 7

Related Work

Several studies have addressed other facets of failure classification, or issues that are related. Dickinson, et al. present a technique called cluster filtering for selecting test cases [5, 6]. Cluster filtering involves clustering profiles of test executions and sampling from the resulting clusters, and the authors present evidence that it is effective for finding software failures when unusual executions are favored for selection. Agrawal, et al. discuss the χSlice tool, which analyzes tests to facilitate finding the location of defects [1]. χSlice visually highlights differences between execution slices of tests that induce failures, and the slices of tests that do not. Jones, et al. describe a tool for defect localization called Tarantula, which uses color to map the way in which each statement in a program leads to the outcome of executing the program on a test suite [19]. Reps, et al. explore the use of a type of execution profile called a path spectrum for discovering Year 2000 problems and other kinds of software faults; this 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 [26]. Podgurski, et al. use cluster analysis of profiles and stratified random sampling to improve the accuracy of software reliability estimates [25]. Leon, et al. describe several applications of mul-

tivariate visualization in observation-based testing, including analyzing synthetic test suites, filtering operational tests and regression tests, comparing test suites, and assessing failure reports [21].

This work on failure classification and observation-based testing differs from the work reported in this study in that it does not involve failure reports from users or any kind of user feedback, it applies unsupervised pattern classification techniques to complete execution profiles- that is, it does not use feature selection, and it attempts to pick out failed executions from a group of otherwise successful ones. In contrast, the current work relies on users submitting failure reports, it uses supervised pattern classification techniques for feature selection before clustering the execution profiles, and it attempts to identify groups of failures that have closely related causes in a set of failed executions.

Chen, et al. show a dynamic analysis technique for partially automating problem determination in large Internet services, which involves course-grained tagging of client requests, and they implement this technique under the J2EE platform with a framework called Pinpoint [4]. Hildebrandt and Zeller present a delta debugging algorithm that generalize program inputs that elicit failures in order to produce a minimal test case that causes a failure [11]. The delta debugging algorithm, which can be viewed as a feature selection algorithm, can be used for failure classification in the case that failure-causing inputs reported by different users simplify to the same test case. This approach requires an automated means of determining whether the minimal test cases produced cause the same type of failure as the original input. Zeller relates another form of delta debugging in [28] that picks out the programming constructs, such as fields and their values, that are specific to a failure by systematically narrowing the state difference between passing and failing executions.

Microsoft Corporation has developed a tool called ARCADE that attempts to

automatically classify crashes and hangs reported from the field into buckets, each of which is supposed to correspond to a unique defect [27]. ARCADE sorts user reports based on the contents of minidumps produced by the Watson failure reporting mechanism [23]. This approach is limited to failures that cause a program to crash or hang. Liblit, et al. discuss a method for isolating deterministic bugs by starting with a set of predicates that describe a program's state at various points during an execution, and then eliminating irrelevant predicates using a set of strategies that are applied to both successful and unsuccessful execution [22]. They also present a technique for isolating nondeterministic bugs using a logistic regression classifier, and 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 the basic classification strategy outlined in Chapter 2. Theirs, however, does not distinguish between failures with different causes and attempt to group them accordingly.

Finally, the work contained in this thesis was also presented in [7], by Francis, et al..

Chapter 8

Conclusions

I have presented a new dendrogram-based technique for refining an initial classification of failed software execution profiles. The experimental results in Chapter 5 suggest that such a technique is effective for grouping together failures with the same or similar causes. In general, employing the changes suggested by the criteria outlined in this paper resulted in a positive effect on the desirable aspects of clusters. These attributes include cluster homogeneity, the placement of failures into appropriate clusters, the completeness of clusters, and the correct classification of singleton failures. It was also noted that in most cases, if a cluster's two maximally dissimilar profiles are homogeneous, then the entire cluster is likely to be homogeneous.

The technique presented is useful in the area of failure classification, since current failure classification methods do not have a definitive way to determine the number of clusters into which a set of program executions should be divided, and any refinement in the number of clusters must be done after classification is complete. This study also corroborates earlier hypotheses [21, 24] that execution profiles of failures with the same cause will be more similar to each other than they are to other execution profiles. Future exploration of the use of dendrograms for clus-

tering refinement would involve more use of heuristics that examine attributes of the clusters' homogeneity and siblings, as well as the manufacture of dendrogram-viewing software that is intended for use in software engineering research. Additional experimental work with a wider variety of subject programs and software defects is needed to confirm the experimental results presented in this study.

Chapter 9

Appendix

Images of the dendrograms for each subject program

The images in this appendix are rotated in order to fit on the page. Each of these dendrograms is the original one for its subject program; that is, no splits or merges have been applied yet. The solid lines indicate subtrees inside clusters into which the dendrograms have been divided; the dotted lines are subtrees that are larger than the clustering.

Figure 9.1: Full dendrogram for GCC

Figure 9.2: Full dendrogram for Javac

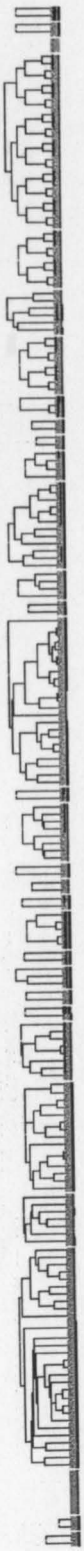


Figure 9.3: Full dendrogram for Jikes

Figure 9.4: Full dendrogram for JTidy

Bibliography

[1] Agrawal, H., Horgan, J.R., 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] Byte Code Engineering Library, `http://jakarta.apache.org/bcel/`, Apache Software Foundation, 2002 - 2004.

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

[4] 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).

[5] Dickinson, W., Leon, D., and Podgurski, A. Finding failures by cluster analysis of execution profiles. 23rd Intl. Conf. on Software Engineering (Toronto, Ontario, May 2001), 339348.

[6] 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.

[7] Francis, P., Leon, D., Minch, M., and Podgurski, A. Tree-based methods for classifying software failures. 15th IEEE Intl. Symposium on Software Reliability Engineering (Saint-Malo, Bretagne, France, November 2004), 451-462.

[8] GCC. The GCC Home Page, http://gcc.gnu.org/, Free Software Foundation Inc., 2005.

[9] Google Groups. http://groups-beta.google.com/, Google, Inc., 2005.

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

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

[12] HTMLTidy. http://tidy.sourceforge.net, World Wide Web Consortium (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University), 1998 - 2003.

[13] Jacks. The Mauve Project, http://sources.redhat.com/mauve/, 2005.

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

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

[16] Java Virtual Machine Profiler Interface (JVMPI). Sun Microsystems Inc., http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/, 2002.

[17] Javac, Sun Microsystems Inc., Java^{TM} 2 Platform, Standard Edition 1.3, `http://java.sun.com/j2se/1.3/index.jsp`, 1994 - 2005.

[18] Jikes. IBM developerWorks, `http://jikes.sourceforge.net/`, 2005.

[19] 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).

[20] 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.

[21] 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.

[22] 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).

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

[24] 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).

[25] 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.

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

[27] 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`.

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

This digital copy was produced by the Case Western Reserve University Archives in 2016.

Original documents from the University Archives were scanned at 300 ppi in black and white or grayscale or color. Blank pages were not scanned. The images were OCR'd using Adobe Acrobat 9.0.

Please send questions or comments to
University Archives
Case Western Reserve University
archives@case.edu
216-368-3320

Warning Concerning Copyright Restrictions

The Copyright Law of the United States governs the making of photocopies or other reproductions of copyrighted material. Under certain conditions specified in the Law, libraries and archives are authorized to furnish a photocopy of other reproduction. One of these specified conditions is that the photocopy or other reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a user makes a request for, or later uses, a photocopy or other reproduction for purposes in excess of "Fair Use," that user may be liable for copyright infringement