Automated Support for Classifying Software Failure Reports

Andy Podgurski, David Leon, Patrick Francis,
Wes Masri, Melinda Minch
Electrical Engineering & Computer Science Dept.
Case Western Reserve University
Cleveland, OH 44106
1-216-368-6884
andy@eecs.cwru.edu, dzl@po.cwru.edu,
paf9@po.cwru.edu, qds1@hotmail.com
mlm24@po.cwru.edu

Jiayang Sun, Bin Wang
Statistics Department
Case Western Reserve University
Cleveland, OH 44106
1-216-368-0630
jiayang@sun.cwru.edu, bwang@laplace.cwru.edu

Abstract

This paper proposes automated support for classifying reported software failures in order to facilitate prioritizing them and diagnosing their causes. A classification strategy is presented that involves the use of supervised and unsupervised pattern classification and multivariate visualization. These techniques are applied to profiles of failed executions in order to group together failures with the same or similar causes. The resulting classification is then used to assess the frequency and severity of failures caused by particular defects and to help diagnose those defects. The results of applying the proposed classification strategy to failures of three large subject programs are reported. These results indicate that the strategy can be effective.

1. Introduction

Some recent software products such as Netscape *Communicator*, *Mozilla*, and Microsoft *Visual Studio.NET* have the ability to detect certain of their own runtime failures and, with the user's permission, report these to the software's developer via the Internet. A transmitted failure report includes information characterizing the state of the software at the time the failure was detected, which is intended to assist developers in diagnosing the failure's cause. Some applications, including the *Visual Studio.NET* beta version, have a feature that allows a user to transmit a failure report (bug report) to the developer whenever they believe the application has behaved incorrectly, that is, even if the application did not detect a failure itself. The report typically contains the user's characterization of the failure and may also contain information about the application state. Such automated support for reporting failures and collecting diagnostic information is a significant advance in software development technology. Traditionally, developers have relied upon users to report software failures by email or telephone and to provide detailed information about the conditions under which they occurred so their cause could be diagnosed. Often, however, users are unable to provide adequate information even when they are questioned by support personnel.

Although automated failure reporting and collection of diagnostic information facilitates debugging, it is also likely to exacerbate another problem encountered by software developers: they often receive many more failure reports than they have time to investigate thoroughly. Developers attempt to classify and prioritize the failure reports they receive, so they can address at least the most significant ones. With automated problem reporting, the number of failure reports received by developers seems likely to increase dramatically. If so, manual classification and prioritization of these reports may become infeasible.

This paper proposes automated support for classifying reported software failures so as to facilitate prioritizing them and diagnosing their causes. A classification strategy is presented that involves the use of supervised and unsupervised pattern classification[1] and multivariate visualization. These techniques are applied to execution profiles in order to group together reported failures with closely related causes. Failures are initially

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

0-7695-1877-X/03 $17.00 © 2003 IEEE

classified *before* their cause is investigated manually. Limited manual investigation may then be done to confirm or, if necessary, refine the initial classification. The resulting classification is then used to assess the operational frequency and severity of failures caused by particular defects and to diagnose those defects. We report the results of applying the proposed classification strategy to failures of three large subject programs. These results indicate that the strategy can be effective.

We now outline the remainder of the paper. Section 2 explains how classification of failures facilitates maintenance. Section 3 outlines our strategy for classifying failures. Sections 4-7 describe the phases of the strategy in detail. Section 8 describes our experimental results. Section 9 surveys related work. Finally, conclusions and future work are presented in Section 10.

2. How classification helps

(*Note on terminology: We use the terms "software failure" and "failure" as synonyms for "failed program execution".*)

When software has many users it is common for different users to report failures that are due to the same defect, although this may not be obvious from the users' descriptions of the failure. Thus, if users report m failures 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 $1 \le i \le k$. We call this partition the *true failure classification*. Knowledge about the true failure classification is valuable to software developers for the following reasons:

- k is the number of defects responsible for reported failures.
- $|F_i|/n$ is an estimate of the frequency with which defect d_i causes failures in the field.
- The failures in F_i are the executions that are most relevant to diagnosing the defect d_i and to determining its severity.
- To diagnose and repair d_i, it should usually suffice to investigate at most a few of the failures in F_i in detail.

Although in principle developers can determine the true failure classification exactly by manually diagnosing the cause of each failure f_i, $1 \le i \le m$, this may be impractical, and it largely defeats the purpose of prioritizing reported failures. Instead, we propose using automatic classification and multivariate visualization techniques to *approximate* the true failure classification with much less human effort. The approximation is unlikely to be exact, because of estimation error and because the techniques we employ are based on *correlations* that may or may not indicate *causation*. Nevertheless, we hypothesize that even a rough approximation to the true failure classification can be of great practical value to developers. Moreover, it is possible to refine the initial classification as more information is obtained.

Classifying program *crashes* and *aborts* is an important special case of the failure classification problem that is typically easier to solve than the general case, provided that information about the program state just before each crash or abort, such as a call-stack trace, is available. For example, in postings on the *Mozilla* project [22] the fact that multiple crashes occurred at the same instruction and with the same call stack is used as evidence that the crashes have the same cause.[2] Note that this paper addresses the more difficult general case of failure classification, where a user may not realize that a failure has occurred until well after the defective code that caused it has executed.

3. Classification strategy

The basic strategy we present for approximating the true failure classification has four phases:

1. The software is instrumented to collect and transmit to the developer either execution profiles or captured executions, and it is then deployed. (Captured executions can be replayed offline to obtain whatever kind of profile is desired [27].)
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[3] (a projection) to use in grouping related failures. The feature selection strategy is to:

[2] James Larus of Microsoft Research informed one of the authors (Podgurski) that Microsoft Corporation internally uses automated heuristics to classify crash reports produced by its products.

[3] 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.

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 closely related causes.

Phases (1)–(4) of the classification strategy are described in Sections 4–7, respectively.

4. Applicable forms of profiling

The kind of information that can be used in automatically classifying arbitrary software failures is not limited to the kind of information that is typically considered in debugging, e.g., the value of the program counter, the values of key variables, and the contents of the call stack when a failure occurs. Any kind of execution profile can be used that is potentially relevant to the occurrence of failures and that can be collected from the field without inconveniencing users unduly. This includes both generic and application-specific profiles characterizing such aspects of a program's execution as its control flow, data flow, input values and other variable values, and event sequences. For example, profiles might include execution counts for basic blocks, conditional branches, functions, definition-use chains, or state transitions. Profiles can be augmented with information obtained from users when they reported failures, e.g., by having them complete a form.

Both the causes of failures and their effects are relevant to classifying them, and hence the form of profiling should be chosen to reflect both if possible. Since failures often involve small parts of a large program, profiles should generally be as detailed (fine-grained) as possible, considering profiling overhead and analysis costs. Coarse-grained profiles are unlikely to distinguish between different defects in the same fragment of code.

5. Feature Selection

The second phase in our strategy for approximating the true failure classification involves selecting a subset of all profile features to use in grouping failures. This step is necessary because execution profiles typically have thousands of features, many of which are not relevant to the occurrence of failures. For example, a profile might contain an execution count for each basic block in a large program. We hypothesize that *the profile features that are most relevant to classifying failures according to their causes are the features that are most useful for distinguishing reported failures from successful executions.*

The approach to feature selection used in the experiments reported in Section 8.1.2 is a modification of the probabilistic wrapper method of Liu and Setiono [21].[4] Random sets of features of given size are generated iteratively. Each set of features and one part of the profile data is used to train a classifier. The misclassification rate of each classifier is estimated using another part of the profile data, and the features used by the classifier with the smallest estimated misclassification rate are chosen for use in phase (3) of our classification strategy.

Many types of statistically-based classifiers have been developed by researchers [6][9]. This paper does not address the issue of which of these types of classifiers is best suited to classifying program failures. Its goal is to provide evidence that *some* classifiers are useful for this purpose. Hence, in the experiments reported in Section 8, we employ a widely-used but relatively simple type of classifier: *logistic regression* models. Binary logistic regression is a type of statistical regression in which the dependent variable Y represents one of two possible outcomes or responses, such as failure or success in the case of a program execution [9]. In logistic regression, the expected value $E(Y \mid \mathbf{x})$ of Y given the vector of predictor values $\mathbf{x} = (x_1, x_2, ..., x_p)$ is $\pi(\mathbf{x}) = P(Y = 1 \mid \mathbf{x})$. The conditional probability $\pi(\mathbf{x})$ is modeled by

$$\pi(\mathbf{x}) = \frac{e^{g(\mathbf{x})}}{1+e^{g(\mathbf{x})}}$$

where the *log odds ratio* or *logit* $g(\mathbf{x})$ defined by

$$g(\mathbf{x}) = \ln\left(\frac{\pi}{1-\pi}\right) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

is a linear function of $\mathbf{x}$. Each coefficient represents the change in log odds of the response per unit change in the

[4] In the *wrapper* approach to feature selection, candidate feature sets are evaluated by using them to train classifiers, whose misclassification rates are estimated.

corresponding predictor. When logistic regression is used for classification, the coefficients of $g(\mathbf{x})$ are estimated from a sample of $\mathbf{x}$ and Y values to obtain an estimator $\hat{g}(\mathbf{x})$ for $g(\mathbf{x})$. The outcome for input $\mathbf{x}$ is classified as a 1 if an only if $\hat{g}(\mathbf{x}) > 0$, that is, if and only if the estimated odds of a 1 exceed the estimated odds of a 0.

6. Grouping related failures

We consider two alternative approaches to grouping related failures in phase (3) of our classification strategy. The first approach calls for applying automatic *cluster analysis*[5] to the sub-profiles induced by the profile features selected in phase (2). The second approach involves applying a *multivariate visualization* technique such as multidimensional scaling to the aforementioned sub-profiles to produce a two-dimensional scatter plot display representing the similarity or dissimilarity of the sub-profiles to each other. This display is then inspected and clusters are identified visually.[6]

6.1 Automatic cluster analysis

Ideally, the process of grouping failures according to their likely causes would be fully automated. This suggests applying automated *cluster analysis* [8] to the sub-profiles induced by the profile features selected in phase (2) of our classification strategy. Cluster analysis algorithms identify clusters among a set of objects according to the similarity or dissimilarity of their feature vectors, as measured by a *dissimilarity metric* such as d-dimensional Euclidean distance or Manhattan distance. Roughly speaking, objects that are more similar to one another than to other objects are placed in the same cluster. In order to automatically group failures according to their causes, it is necessary to *estimate* the number of clusters among them. Although many approaches to finding the "best" number of clusters in a population have been proposed (see [8] for examples), the problem is quite difficult, because there are often several "reasonable" ways to cluster the same population. Hence, we have concluded that it is unwise to depend solely on automatic cluster analysis to group reported failures according to their likely causes. We propose instead that cluster analysis be used together with other techniques, such as multivariate visualization.

[5] Cluster analysis is an example of unsupervised learning.

[6] Note that if profiles of successful executions are unavailable clustering can be done based on *all* profile features, at the cost of some precision.

One widely used measure of the goodness of a clustering into c clusters, which we employ in Section 8, is the index due to Calinski and Harabasz [3]:

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

where B is the total *between-cluster* sum of squared distances, W is the total *within-cluster* sum of squared distances from the cluster centroids, and n is the number of objects in the population. To use $CH(c)$, its value is plotted for c = 2, 3, ... n, and local maxima are considered as alternative estimates of the number of clusters.

6.2 Multivariate visualization

Multivariate visualization methods such as *multidimensional scaling* (*MDS*) represent a set of objects characterized by dissimilarity or similarity measurements as points in a low dimensional space such as a two-dimensional display [2]. A two-dimensional display produced with MDS is a kind of scatter plot. The points are positioned so that the distance between each pair of points approximates the dissimilarity between the corresponding objects. An arbitrary dissimilarity matrix can be input to multidimensional scaling, so it can be used with a variety of dissimilarity metrics.

We propose that multidimensional scaling be used to display the sub-profiles induced by the profile features selected in phase (2) of our classification strategy, so that groups of related failures can be identified by visual inspection of the resulting scatter plot. We hypothesize that apparent clusters of points in the display will often correspond to such groups of failures. With visualization, users can judge themselves which failures are most closely related, rather than relying on a fixed clustering criterion as in automatic cluster analysis. A drawback of visualization for this purpose is that in projecting high dimensional data onto just two dimensions, small dissimilarities may be poorly represented in the display. Approaches to addressing this issue are presented in [19]. To better distinguish individual clusters, automatic cluster analysis should be used together with visualization. Clusters found automatically can be highlighted in an MDS display, e.g., by coloring their points or drawing their convex hulls.

We have developed a visualization tool that supports MDS of large sets of execution profiles and provides features useful for classifying failures, including ones for: selecting a group of points in the display and determining the corresponding executions; magnifying regions of the

display; and highlighting specified sets of points such as clusters.

7. Confirming or refining the initial classification

Given an initial approximation to the true failure classification, a software developer might choose to use it "as is" for the purpose of prioritizing reported failures. However, it is prudent to do some additional work to confirm or, if necessary, refine the initial classification. This can be done by:

1. Selecting a few failures (two or more) from each group of failures.
2. Attempting to determine if the failures selected from a group actually have closely related causes, using conventional debugging techniques.
3. Attempting to determine if similar groups contain failures with the same cause.

Step (2) is especially important with clusters that are elongated or are "loose", because such clusters have high internal dissimilarity. If all of the failures selected from a group turn out to have closely related causes, this is further evidence that all or most of the failures in the group do. Otherwise the initial classification should be refined. Step (3) is applicable to neighboring clusters.

One criterion for deciding which failures to select from a group is that ones with *maximally dissimilar profiles* should be chosen. For example, one might select one failure from each end of an elongated cluster. Such failures can be selected automatically or by inspection of a scatter plot display. If the selected failures have the same or similar causes, one might conclude that all of the failures between them in the cluster do also. If they have dissimilar causes, one should seek a good place to split the cluster into two or more pieces.

8. Experimental validation

In order to evaluate the effectiveness of our classification strategy, we implemented its first three phases with three large subject programs. Automatic cluster analysis was used to classify failures, and the resulting clusters were then examined manually. To be thorough, we examined all or most of the failures in each cluster rather than sampling just a few failures from each cluster as described in Section 7.

8.1 Experimental methodology

8.1.1 Subject Programs, Inputs, and Profiles. The three subject programs for this study were all compilers: the *GCC* compiler for C [7] and the *Jikes* [15] and *javac* [14] Java compilers. These programs were chosen for several reasons: they are large; they can be executed repeatedly with a script; source code for a number of versions is available; and self-validating test suites are available for them. Unfortunately, we did not have access to failure reports from ordinary users. Instead, our classification strategy was applied to the failures detected by self-validating tests.

Version 2.95.2 (Debian GNU/Linux) of the *GCC* compiler for C was used. Only the C compiler proper was profiled. The compiler was executed on a subset of the regression test suite for *GCC*, consisting of tests that actually execute compiled code. These came from the test suite shipped with *GCC* 3.0.2, which included tests for defects still present in version 2.95.2. *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) [11], which tests adherence to the Java Language Specification [12]. *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. *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*, which is written in Java, a simple profiler was written by using the Java Virtual Machine Profiling Interface [13]. For each test t of one of the subject programs, the generated profile consisted of a vector of counts, with one count per function in the program. The count for each function f indicated how many times f was executed during test t. Note that *GCC* had 2214 functions, *Jikes* had 3644 functions, and *javac* had 1554 functions.

8.1.2 Feature Selection. Phase (2) of our classification strategy was implemented using the *S-PLUS 6* statistical computing environment [26]. Logistic regression (LR) models were used as classifiers. These were implemented using the *S-PLUS* function *glm*. An *S* language program was written to iteratively generate 400-500 candidate models per data set and to fit and evaluate them. Each model included 500 randomly selected features.[7] The *S* program output the best model

[7] The number of candidate models generated and the number of features used were selected based on preliminary experiments. They represent a tradeoff between classifier performance and total training time.

of a given type. To avoid underestimating the misclassification rate of models, the original set of profiles for each subject program was randomly partitioned into three subsets (*Train*, *TestA*, and *TestB*) comprising 50%, 25%, and 25% of the original set, respectively. The profiles in *Train* were used to train candidate models; those in *TestA* were used to pick the best model (i.e., for model validation); those in *TestB* were used to produce a final estimate of the best model's misclassification rate.

The measure used to pick the best model was the average of the percentage of misclassified failures and the percentage of misclassified successes. This measure gives more weight to the misclassification of failures than does the overall misclassification rate. For each of the data sets (*GCC*, *javac*, and *Jikes*), the final logistic regression model correctly classified at least 72% of failures and at least 91% of successes. It is notable that *S-PLUS* reported that a substantial number of selected features were not used in the fitted logistic regression models, because they were linearly dependent on other features. We did not use those features for clustering or visualization either.

8.1.3 Cluster Analysis. To group failures together automatically, we used the *S-PLUS* cluster analysis algorithm *clara*, which is based on the *k-medoids* clustering criterion [18]. To estimate the number of clusters in the data, the Calinski-Harabasz index $CH(c)$ was plotted for $2 \leq c \leq 50$ and its local maxima were examined.

8.1.4 Visualization. For each subject program, the sub-profiles induced by the features of the best classification model were displayed using the *hierarchical MDS* (*HMDS*) algorithm described in [19]. This algorithm was designed to minimize the error in representing small dissimilarities between execution profiles.

8.1.5 Manual Examination of Failures. In many cases, we were able to diagnose the specific cause of a group of failures. In other cases, this was not possible, but other evidence was found that certain failures had the same cause. The nature of such evidence varied with the subject program. The *GCC* failures were manually classified by exploiting the organization of the *GCC* test suite and information about later versions of the compiler. Each execution test in the *GCC* test suite involves compiling a simple source file and executing the resulting program. For each source file, multiple tests are run, each with a different optimization level as some defects are triggered only at certain optimization levels. Since each source file is designed to reveal a specific defect, we evaluated our automatic classification strategy with respect to whether it grouped together multiple failures corresponding to the same source file. We also checked manually whether different source files triggered the same defect.

Table 1. Comparison of automatic clustering and manual classification for *GCC* data set.

Number of clusters	% size of largest group of failures in cluster with same cause	Total failures (136)
21	100	77 (57%)
1	83	6 (4%)
3	75,75, 71	23 (17%)
1	60	5 (4%)
1	24	25 (18%)

For 12 of the 29 source files associated with *GCC* failures, we were able to identify bug fixes in later versions of the compiler that prevent the failures the files induce, and we verified that each fix worked when applied to the version under test (2.95.2). The remaining *GCC* failures were classified by determining when the corresponding test case stopped failing. For example, if test A is fixed in the CVS version as of January 2000 and test B keeps failing until July 2001, then we have reason to believe that they failed because of different defects. This required checking out, building, and testing enough versions of the compiler to separate most tests, but it was still less time-consuming than finding and porting specific bug fixes. Because of the possibility of regression defects introduced by changes to the compiler, the resulting classification is not certain, but we believe it is a good approximation. Only 3 pairs of source files were found to induce failures caused by the same defect. Thus, the *GCC* failures were apparently caused by 26 different defects.

The automatic clusterings of the *javac* 1.3.1 and *Jikes* 1.15 failures were examined manually in two stages. In the first stage, the failures in each cluster were examined to see if they actually had the same cause. In cases where the failures in a cluster had different causes, we were often able to identify sub-clusters consisting of failures with the same cause. In the second phase, the groups of related failures identified in the first phase were displayed using hierarchical MDS, and overlapping groups were examined manually to determine if the failures they contained actually had the same cause.

In order to determine if different *javac* 1.3.1 failures had the same cause, the following activities were

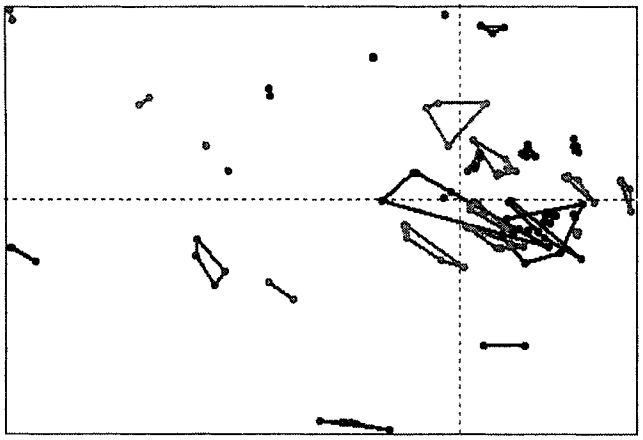

Figure 1. HMDS display of the *GCC* failure profiles after feature selection. Convex hulls indicate results of automatic clustering into 27 clusters.

attempted in the order listed until one of them succeeded, although activity (5) was always conducted:

1. *Debugging javac 1.3.1.* The causes of many of the *javac* 1.3.1 failures were diagnosed using conventional debugging techniques.
2. *Comparing the javac 1.3.1 and javac 1.4 code bases.* A few of the tests that caused *javac* 1.3.1 to fail were found to succeed with *javac* 1.4 due to identified bug fixes.
3. *Examining error codes.* It was found that the input files corresponding to many of the *javac* 1.3.1 failures formed groups: the files in each group were erroneously accepted by *javac* 1.3.1 but were rejected by *javac* 1.4 with the same error code.
4. *Inspecting failure-causing source files.* It was found that many of *javac* 1.3.1 failures were caused by language constructs present in multiple source files.
5. *Checking the association between tests and Java Language Specification sections.* Some of the *javac* 1.3.1 failures were classified based on the fact that they involved tests of a language rule described in a particular *JLS* section.

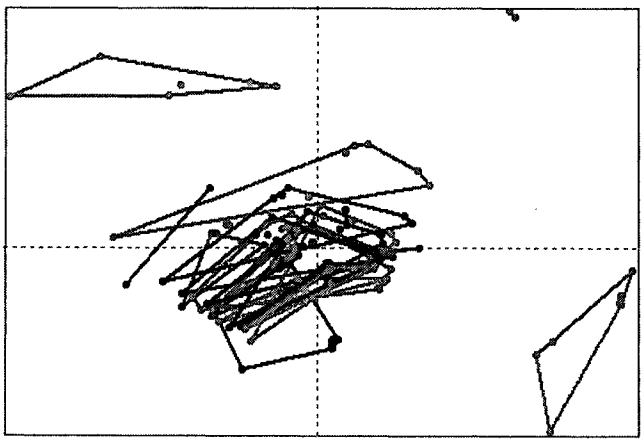

Figure 3. HMDS display of the *GCC* failure profiles before feature selection. Convex hulls indicate failures involving same defect.

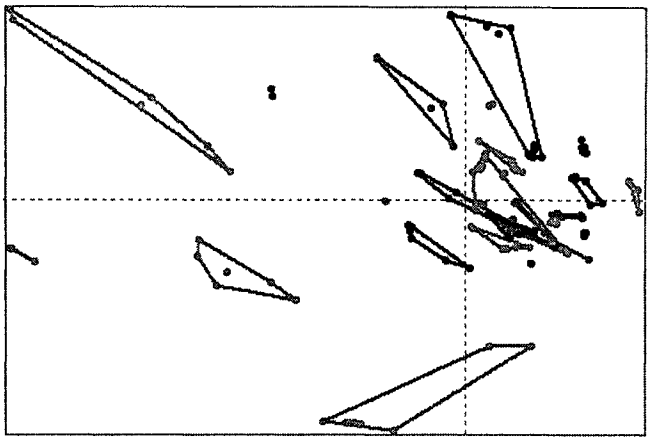

Figure 2. HMDS display of the *GCC* failure profiles after feature selection. Convex hulls indicate failures involving same defect.

To determine if different *Jikes* 1.15 failures had the same cause, the same five activities were attempted, with versions 1.15 and 1.16 of *Jikes* used in place of versions 1.3.1 and 1.4 of *javac*. Most groups of related failures were identified using activity (1) or activity (3).

8.2 Results

8.2.1 GCC. The *GCC* failures were automatically clustered into 27 clusters, as suggested by the Calinski-Harabasz index. The failures in each cluster were analyzed manually to determine the percentage size of the largest subgroup with the same apparent cause. The results are summarized in Table 1. Most of the clusters were comprised of failures caused by the same defect. Five clusters, of sizes between 5 and 8, contained failures caused by two different defects. One cluster, however, contained 25 failures caused by 7 different defects. Overall, in 26 of the 27 automatically generated clusters, the majority of failures appear to have the same cause. Figure 1 shows a hierarchical MDS display of the *GCC* failures, calculated using the profile features selected in phase (2) of our classification strategy. Convex hulls indicate the results of automatic clustering.

Figure 2 is another HMDS display of the *GCC* failures, in which convex hulls are drawn around each

Table 2. Results of manual examination of automatic clustering for *javac* data set.

Number of clusters	% size of largest group of failures in cluster with same cause	Total failures (232)
9	100	70 (30%)
5	88, 85, 85, 85, 83	64 (28%)
4	75, 67, 67, 57	49 (21%)
2	50, 50	20 (9%)
1	17	23 (10%)

Table 3. Results of manual examination of automatic clustering for *Jikes* data set.

Number of clusters	% size of largest group of failures in cluster with same cause	Total failures (225)
12	100	64 (29%)
5	85, 83, 80, 75, 75	41 (18%)
4	70, 67, 67, 56	25 (11%)
8	50, 50, 50, 43, 41, 33, 33, 25	76 (34%)

group of failures that manual analysis indicated were caused by the same defect. In this display, groups of failures corresponding to particular defects are well separated, suggesting that visual classification of failures is likely to be successful. Comparing Figure 1 to Figure 2 confirms that automatic clustering tended to group together failures with the same cause. However, these figures also indicate that automatic clustering erroneously split some groups of failures with the same cause. Of these 26 groups of failures, 5 had their failures split across two clusters, 2 had their failures split across 3 clusters, and one more was split across 4 clusters.

Figure 3 is an HMDS display of the *GCC* failures that was calculated using *all* profile features. As in Figure 2, convex hulls are drawn around each group of failures that manual analysis indicated were caused by the same defect. There is much more overlap of these convex hulls than in Figure 2. It turns out that Figure 3 overemphasizes the effect of optimization levels. For each source file, high optimization tests are placed to the left of the display, while low optimization tests are on the right. The difference between Figures 2 and 3 illustrates the importance of feature selection in our classification strategy.

8.2.2 javac. The *javac* failures were automatically clustered into 26 clusters. The failures in each cluster were examined manually to determine the percentage size of the largest subgroup with the same apparent cause. Table 2 summarizes the results for different groups of clusters comprising 21 clusters. (The remaining 5 clusters comprised 4 singletons and 1 cluster of size 2 whose elements could not be classified decisively.) Overall, in 22 of 26 automatically generated clusters (85%), a majority of failures had the same cause. Note that two large clusters formed by automatic clustering were found to be heterogeneous. Manual analysis revealed that both clusters had several sub-clusters, each consisting of failures with the same apparent cause.

Figure 4 is a hierarchical MDS display of the *javac* 1.3.1 failures. Convex hulls have been drawn around

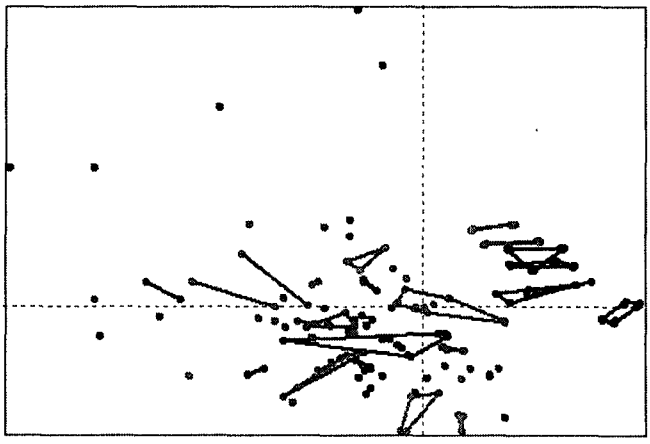

Figure 4. HMDS display of the *javac* failures. Convex hulls indicate the results of manual classification.

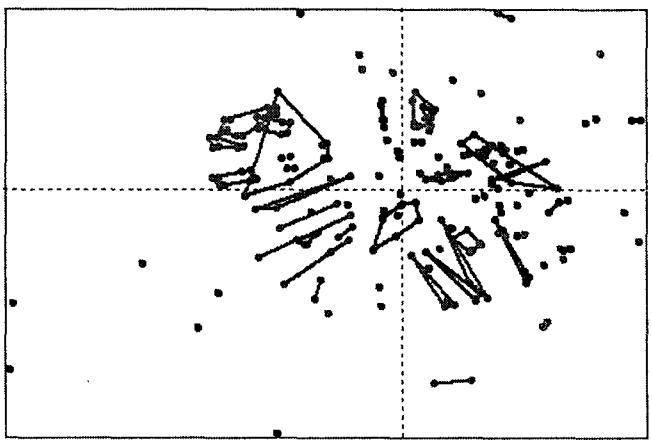

Figure 5. HMDS display of the *Jikes* failures. Convex hulls indicate the results of manual classification.

groups of failures that manual classification indicated have the same cause. Some of the convex hulls are compact and well separated from others, but some are elongated or overlap. Automatic clustering performed well in identifying the groups of failures whose convex hulls were well separated from those of other groups, even when they were elongated. It is unlikely that the elongated clusters would be identified by inspection of the HMDS display, although sub-clusters would. The groups of failures in the upper right quadrant of Figure 4 whose convex hulls overlapped noticeably were subject to further manual analysis. It was found that the failures in the overlapping groups actually had the same cause. Automatic clustering erroneously split those groups.

8.2.3 Jikes. The *Jikes* failures were automatically clustered into 42 clusters. The failures in each cluster were analyzed manually to determine the percentage size of the largest subgroup with the same apparent cause. Table 3 summarizes the results for 29 of the 42 total clusters. (The remaining 13 clusters comprised 9 singletons and 4 clusters whose elements could not be classified decisively.) Overall, in 30 of 42 automatically generated clusters (71%), the majority of failures appear to have the same cause.

Figure 5 is a hierarchical MDS display of the *Jikes* 1.15 failures. Convex hulls have been drawn around groups of failures that manual classification indicated have the same cause. As with the HMDS display of the *javac* failures, some of the convex hulls are compact and well separated from others, but some are elongated or overlap. Automatic clustering again did a good job of clustering the groups of failures whose convex hulls were well separated from those of other groups, even when the convex hulls were elongated. The groups of failures in the top left quadrant of Figure 5 whose convex hulls overlap were subjected to further manual analysis, which revealed that the failures in those groups had the same cause. This was also true of one of the pairs of overlapping groups in the top right quadrant of Figure 5.

8.3 Summary

In most of the clusters created by automatic clustering, a majority of failures appeared to have the same cause. However, automatic clustering created a few large, non-homogeneous clusters with sub-clusters consisting of failures with the same cause. The sub-clusters were evident in the corresponding HMDS displays. In other cases automatic clustering erroneously split groups of failures with the same cause, but the HMDS displays did not provide clear evidence that these groups were homogeneous. Overall, groups of failures with the same cause tended to form fairly cohesive clusters in the HMDS displays. Small, tight clusters in the displays were quite likely to contain failures with the same cause, although they were not always maximal.

8.4 Threats to Validity

All of the subject programs used were compilers. Therefore, it is possible that our results may not generalize to other types of software. Also, hand crafted test inputs were used rather than operational inputs. Test inputs like those in the *GCC* and *Jacks* test suites are generally smaller than typical compiler inputs and so the corresponding profiles may be less "noisy" than profiles of operational inputs would be.[8] Additional evaluation of the classification strategy is necessary, especially with different kinds of software and with actual failure reports from users.

9. RELATED WORK

Several previous papers have addressed issues closely related to failure classification and prioritization. Agrawal, *et al* describe the *χSlice* tool, which analyzes system tests to facilitate location of defects [1]. *χSlice* 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 [25]. 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 [16].

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 [10]. 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.

[8] On the other hand, operational inputs may tend to form more cohesive groups, which would aid our strategy.

Podgurski, *et al* used cluster analysis of profiles and stratified random sampling to improve the accuracy of software reliability estimates [24]. 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 [20]. Dickinson, *et al* present a technique called *cluster filtering* for filtering test cases [4][5]. 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.

Finally, Julisch and Dacier demonstrated that a form of *conceptual clustering* is effective for grouping similar *alarms* from intrusion detection systems (IDS) [17]. We note that IDS alarms are typically generated by matching intrusion *signatures*, hence the attack type is at least partially known. Alarms typically have many fewer features than execution profiles do, and most features of alarms are relevant to classifying them. In many cases, relatively few features of an execution profile are relevant to classifying a particular type of failure or attack. Thus, dimensionality reduction is a much more critical issue in clustering profiles than it is in clustering alarms. On the other hand, our approach may be useful for classifying profiles of intrusions reported by *anomaly detection* systems [23].

10. CONCLUSIONS

The results of applying our classification strategy to *GCC*, *Jikes*, and *javac* suggest that the strategy is effective and scales to large programs, although this must be confirmed with other subject programs of different types. It is especially notable that the strategy performed well with a relatively simple type of classifier and with coarse-grained execution profiles; it may perform even better with more powerful classifiers and more detailed profiles. It is especially important to evaluate our strategy with deployed software and with failure reports from users. Other basic issues to be resolved in future work include the best choices of methods for profiling, classification, clustering, and visualization and the best way to integrate these methods during failure classification. Finally, we note that our approach may be useful for classifying computer intrusions reported by anomaly detection systems.

11. ACKNOWLEDGEMENTS

This work was supported by National Science Foundation award CCR-0098325 to Case Western Reserve University.

12. 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] Borg, I. and Groenen, P. Modern Multidimensional Scaling: Theory and Applications. Springer-Verlag, New York, 1997.

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

[4] 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.

[5] 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), ACM Press, 246-255.

[6] Duda, R.O., Hart, P.E., and Stork, D.G. Pattern Classification, 2nd edition. Wiley, New York, 2001.

[7] GCC. The GCC Home Page, www.gnu.org/software/gcc/gcc.html, Free Software Foundation, 2002.

[8] Gordon, A.D. Classification. Chapman and Hall/CRC, Boca Raton, 1999.

[9] Hastie, T., Tibishirani, R., and Friedman, J. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer-Verlag, New York, 2001.

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

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

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

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

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

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

[16] 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).

[17] 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).

[18] Kaufman, L. and P.J. Rousseeuw. Finding Groups in Data. John Wiley & Sons, New York, 1990.

[19] Leon, D., Podgurski, A., and Dickinson, W. Visualizing distances between executions. Technical Report #02-12, EECS Dept., Case Western Reserve University.

[20] 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.

[21] Liu, H. and Setiono, R. Feature selection and classification: A probabilistic wrapper approach. 9th Intl. Conf. on Industrial and Engineering Applications of AI and ES, 1996, 284-292.

[22] The Mozilla Project, www.mozilla.org.

[23] Noel, S., Wijesekera, D., and Youman, C.. Modern intrusion detection, data mining, and degrees of attack guilt. Applications of Data Mining in Computer Security, edited by D. Barbar'a and S. Jajodia, Kluwer Academic Publishers, 2002.

[24] 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.

[25] 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), ACM Press, 432-449.

[26] S-PLUS 6 statistical software. www.insightful.com.

[27] Steven, J., Chandra, P., Fleck, B., and Podgurski, A. jRapture: a capture/replay tool for observation-based testing. 2000 Intl. Symp. on Software Testing & Analysis (Portland, Oregon, August 2000), ACM Press, 158-167.