Advances in Software Testing: Dendrogram-based Methods For Clustering Refinement and Code Coverage Analysis of Software in Operational Use

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

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.

Advances in Software Testing: Dendrogram-based Methods For Clustering Refinement and Code Coverage Analysis of Software in Operational Use

Abstract

by

MELINDA MINCH

Recent research has addressed the problem of providing automated assistance to software developers 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 technique 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. Test suite selection and evaluation is also of recent interest. The second part of this paper contains a comparison of code coverage data from operational executions of several subject programs with code coverage data from the programs' respective test suites. Such a comparison may be used to evaluate the effectiveness of a regression test suite.

Part I

Using Dendrograms to Classify Software Failures

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. 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. 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, which defects cause a failure with the most frequency, and which faults cause the most catastrophic failures.

Research conducted recently by the Software Engineering Lab at Case Western

Reserve University shows that 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 [40]. 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 [40] 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 [40] 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* [23]. 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 [40]. 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 [40]:

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 exeuction 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 only one failure type, and when all of the failures of each type 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 have the same cause, or failure type. 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 have the same failure type 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 good, 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.

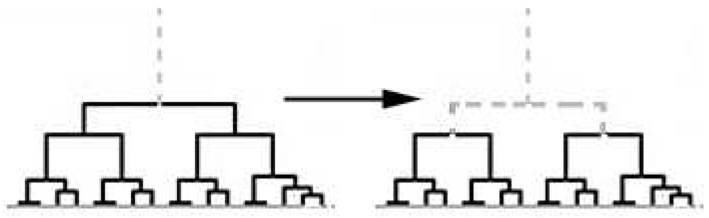

Figure 3.1: Splitting a cluster.

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.

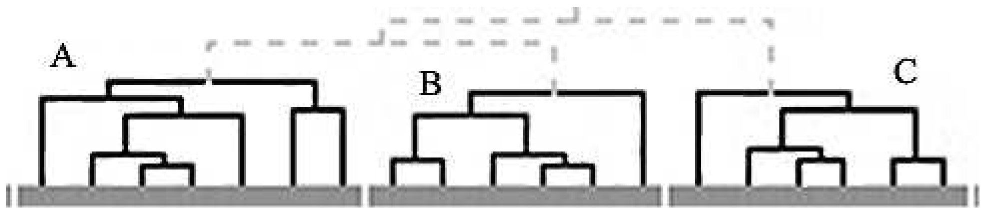

Figure 3.2: Illustration of siblings.

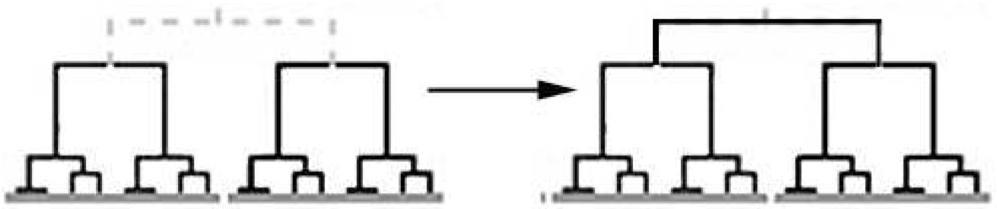

Figure 3.3: Merging two clusters.

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 with the same causes. 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[4] metric.

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 executions have the same cause. If the selected executions have the same or related causes, it is probable that all of the other failures in the cluster do 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 have the same cause.

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) [19], 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 subtrees 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 [15]; the Javac [26] and Jikes [27] Java compilers; and JTidy [29], a Java port of the popular HTMLTidy utility [21]. 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 [22], which tests adherence to the Java Language Specification [24], 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 [17] 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 [25], and the profiler written for JTidy instrumented JTidy's code using the Byte Code Engineering Library[3].

The failures for the GCC, Javac, and Jikes data sets were manually classified in [40]. 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 have the same cause 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 failures with a certain cause that are contained in that cluster. If a cluster contains a failure with a certain cause, it must also contain every other failure with that cause to be considered 100% complete.
- Number of singleton clusters.

- Percentage of *singleton failures*, or failures that are the only failure that have their particular cause, 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 had the same cause 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 had the same cause 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 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 have no causes in common with the causes of 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 of the same cause 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.

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 [10, 11]. 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 [28]. 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 [44]. Podgurski, et al use cluster analysis of profiles and stratified random sampling to improve the accuracy of software reliability estimates [41]. Leon, et al describe several applications of multivariate visualization in observation-based testing, including analyzing synthetic test suites, filtering operational tests and regression tests, comparing test suites, and assessing failure reports [31].

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 [6]. 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 [20]. 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 [46] 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 [45]. ARCADE sorts user reports based on the contents of minidumps produced by the Watson failure reporting mechanism [35]. 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 [32]. 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 [47]. 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 Part I of this thesis was also presented in [13], 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 [31, 40] 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 clustering 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 failure types is needed to confirm the experimental results presented in this study.

Part II

Code Coverage of Operational Executions

Chapter 9

Introduction

The software development cycle is an iterative task; sequential versions of software are created, tested and re-tested. An essential part of this cycle is *regression testing*, or testing software to make sure that no old faults have been re-introduced into the software, and that no new ones have been created during the process of software development. Most large software projects have a regression test suite that is created and updated by developers or testers. Such a test suite is often the only test framework for a project, and it might be run at regular intervals, or whenever new code is added to the project. Since regression test suites tend to accumulate many tests over time, the cost of executing an entire test suite is often non-trivial. For this reason, the problem of test suite selection and minimization is at the forefront of software engineering research- it is important to choose a set of tests that will maximize the number of errors discovered using the minimum amount of time and effort.

Most regression test suites are created artificially; that is, the people in charge of the test suite (or an automated test data generator) write the tests according to some specification and add to them in a similar manner when new defects are found. Some test suites, such as the Jacks test suite for Java compilers [22], even have a specific format that the code for each test must follow. For this reason, most regression test suites consist entirely of *synthetic* test cases, which are usually as concise as possible, designed to test one specific behavior in the software, and reused in cases where it is possible to do so. This is in great contrast with the way that most software is actually used: for much longer durations than the amount of time that it takes to run one synthetic test, with many sequences and combinations of function calls, and on a variety of computing platforms and environments. This means that synthetic regression test suites do not reflect operational usage, and that they are not designed to do so. This is an inherent limitation of synthetic tests. Also, for this reason, synthetic regression test suites are not changed when users find different ways, scenarios and environments in which to use the software under test. The regression test suite doesn't keep up, even as failures arising from these new uses are reported, because it is still geared toward the original, "intended" use of the software, set out in the requirements and specifications documents for the software. Since the reliability of software in the field depends on how it is used, using synthetic tests in isolation is a poor method for finding many of the defects that have the greatest impact on the end user.

The hypothesis of this study is that since synthetic regression test suites often do not reflect operational usage, and are not intended to do so, the behaviors tested by a regression test suite will differ from the behaviors exercised by operational usage, and that this difference will increase over time. Code coverage is used as a metric to compare the scope of the regression test suite with the way in which user operations are distributed. Though it is customary to use the percentage of code coverage as a metric to evaluate regression test suites, the focus of this study is more on *which* pieces of code are covered by the regression test suite, rather than *how many*, and how these results compare to the code coverage data from operational executions.

Chapter 10 explains the the relationship of regression tests to operational data in more depth. Experimental procedure and a description of the subject programs used in the experiments is in Chapter 11. Chapter 12 outlines the experimental results and possible threats to validity.

Chapter 10

Operational usage and regression tests

This chapter provides a more detailed explanation of operational profiles, observation-based testing, and the relationship of those two topics with regression testing methods. In the introduction, the idea was put forth that while synthetic test suites test a specific behavior or function, many software failures "in the wild" are caused by a combination of factors: a sequence of operations performed by the user, executions of a certain duration or containing a precise sequence of function calls, the use of a particular data type alone or in combination with other types, or any number of other things that concise, fine-grained, synthetic tests are not designed to discover.

Taking this idea further, an *operational profile* [37] can be used to characterize the real-world use of software. An operational profile can be thought of as a set of attributes, such as function calls, that are disjoint, and the frequency or likelihood of the occurrence of each of them in any given execution of the software. For example, if a certain function is called in half of the executions of a piece of software, then it has a 50/50 chance of being called in one execution that is random over that program's operational profile distribution. It is impossible to construct an operational profile empirically, since doing so would require knowledge of all possible inputs and user scenarios, past and present. It is, however, a useful concept to keep in mind, and it can be approximated through methods such as *observation-based testing* [31], which uses subject programs that are instrumented to collect data about each program execution, such as function call or basic block counts. The subject programs may be run with input garnered from a number of sources, such as simulated usage scenarios, operational data, or even synthetic test suites.

It stands to reason that a significant percentage the defects that have the highest impact on software reliability for the end user are the defects that will be exposed most often by normal, everyday use of the software. If a regression test suite is to take this into account, a number of the tests in the suite should reflect a random selection of operations over the operational profile distribution. Such testing is often left to the beta-testing stage, but inclusion of such tests into regression test suites would result in a larger amount of automation for finding reliability-affecting defects, and free up beta-testers to focus on finding faults that cannot readily be discovered by automated testing, such as usability issues. Another advantage to making regression tests more reflective of the operational profile would be that more accurate reliability estimates could be made using regression test suites. Using code coverage data to determine which parts of the software are executed by the regression test suite and the frequency with which they are executed, and comparing those statistics with those gathered from operational executions is a way to get a rough estimate of whether the regression test suite reflects operational usage.

Chapter 11

Analyzing the code coverage of regression test suites

This section will discuss the steps taken to obtain measurements of code coverage for regression test suites and operational executions, the procedure for setting up these experiments, and the subject programs used in this study. Experiments were conducted with multiple versions of three subject programs.

Though it would be ideal to examine the code coverage of the test suite for each release version of a piece of software, it is impractical to do so for most software projects, since there are not enough release versions to provide enough data points for a comprehensive study. Therefore, Concurrent Versions System (CVS) [8] snapshots were used for the purpose of obtaining intermediate releases of software for this study.

Basic block counts were collected for the code coverage information, but comparing basic block counts from different versions of software can be problematic and might lead to false conclusions about the data. As software evolves, new code is added, other statements are deleted, and basic blocks may be split into several portions, calling the validity of such analysis into question. This study, therefore, uses the percentage of code covered in each source file as a metric, and only delves into basic block coverage and function call counts when a more fine-grained view of the data is especially helpful.

11.1 Experimental procedure

The following steps were taken for each of the three subject programs:

1. Download a series of CVS snapshots of each subject program, taken at regular intervals, such as weekly or biweekly, over a period in the life cycle of the software. These snapshots are of development (i.e. non-release) versions of the software. The time intervals are chosen according to the availability of CVS snapshots, the age of the subject program, and the pace at which the development of the project proceeds.

2. Build each snapshot in the same program environment. If a snapshot will not build for any reason, it is discarded.

3. Run the appropriate version(s) of a test suite for the subject program on each snapshot.

4. Gather large amounts of real-world operational input for each subject program. Ideally, such operational input would would reflect the operational profile for the software. Since execution profiles for a large number of real-world executions of the subject program is not available, it is not possible to approximate the operational profile for each subject program in that way. It should suffice, however, to have a large amount of diverse input that is gathered from publicly available sources, and to execute the software on this input.

5. Run each snapshot of the subject program on the operational input, and gather the output from each execution.

6. Use either an independent validator or output from an oracle version of the subject program to determine which subject program executions failed, by examining the program output.

7. Rebuild each snapshot so that code coverage data can be gathered from it, and run each snapshot again using the test suite(s) and the operational input.

11.2 Subject programs and test suites

Three programs were used for this study: the Jikes Java compiler [27]; HTMLTidy [21], which is an HTML and XML validator and pretty-printer; and the Mono C# compiler [36], which is part of the Mono project, an open source implementation of Microsoft's .NET platform [34]. These subject programs were selected because they fit the following criteria: they can be easily executed with a script, they are open-source projects, they have their own bug database or bug-tracking system, there are many incremental versions of each program available as snapshots or in their CVS repositories, and their success or failure during operational tests can be readily determined by examining their output files. The remainder of this section is a discussion of the specifics of each subject program.

Jikes. 51 bi-weekly snapshots of the Jikes compiler from 29 September, 2003 to 4 October, 2004 were downloaded and built under Cygwin [9] using a Dell Optiplex GX150 running Windows 2000, with a Pentium III 866 MHz processor and 512 MB of RAM. Two versions of the Jacks test suite, version jikes-milestone-1_18, and a snapshot of the test suite as of 13 December, 2002 were used as the regression test suite for Jikes. For operational inputs, Java files from open-source Java projects were used. Each Jikes snapshot was run on 4,990 Jacks tests and 8,656 Java files.

All of the Jikes snapshots used the Sun Java runtime, version 1.4.2_06. Operational failures were discovered by running the class files generated by each Jikes snapshot through JustICE, the Java BCEL Bytecode Verifier provided by the Byte Code Engineering Library[3]. The *gcov* tool, which is included with the GCC compiler, was used to measure code coverage for Jikes, and the GCover [16] tool was used to analyze *gcov* output. It should be noted that in order to use *gcov* in conjunction with the Jacks test suite, Jacks had to be modified so that it used a flat directory structure.

HTMLTidy. 100 weekly snapshots of HTMLTidy from between 23 April, 2003 and 13 April, 2005 were downloaded and built on Gentoo Linux using a Dell Optiplex GX150 with a Pentium III 866 MHz processor and 512 MB of RAM. The latest stable version of HTMLTidy, as of 8 April, 2005, was used as an oracle, and the regression test suite that ships with it was used as the regression test suite for this trial. For operational input, 16,373 HTML files were gathered from the Internet using a web crawler, and each HTMLTidy snapshot was run on each of these. Operational failures were discovered by running the oracle on the operational input, and using the *diff* utility to compare the HTML output from the oracle to the HTML files generated by each snapshot. Changes in the amount of whitespace and the case of the text in output files were ignored. The code coverage for HTMLTidy was also measured using *gcov* and GCover.

Mono. 39 weekly snapshots of the C# compiler from the Mono project from between 1 July, 2004 and 7 April, 2005 were downloaded and built under Cygwin using a computer running Windows XP, with a 1.13 GHz Athlon processor and 512 MB of RAM. The latest version of the Mono test suite, as of 16 April, 2005, available on the Mono project web site, was used as the regression test suite for the Mono C# compiler. For operational inputs, open-source projects written in C# were used. Since the way in which Mono calculates dependencies between C# classes makes it difficult to perform stand-alone compilation of source files, the *make* utility and the NAnt .NET build tool [38] were used with each Mono snapshot to execute the build and install scripts that were included in the open-source projects. Operational failures were discovered by executing the Portable Executable Verifier tool from Microsoft, PEVerify.exe [39], on the binaries produced by the Mono C# compiler, which are in the Microsoft Portable Executable format. Code coverage for Mono was measured using Clover.NET [7], a code coverage analysis tool for the .NET Framework.

Chapter 12

Experimental results

12.1 Jikes results

The number of regression test failures for Jikes was determined by running the Jacks test suite on each snapshot. The line in Figure 12.1 labeled “older” shows results from the December, 2003 version of Jacks, while the line labeled “recent” shows results from the jikes_milestone-1_18 version of the Jacks test suite, which is newer than all of the Jikes snapshots. The older test suite contains 3,235 tests, and the recent version contains 4,990. The recent test suite fails more often than the older one when the number of failures is low for both suites, and less often where the number of failures is high. This may be suggestive of greater efficiency and orthogonality of tests in the newer version of Jacks; if a test suite is redundant, with many of its tests attempting to reveal the same defect or class of defects, it will reveal fewer failures when such a defect is not present, and many more when it is. Conversely, a more efficient test suite will reveal a larger number of failures with broad coverage, and not show a large number of failed tests for one bug. According to a version history of Jikes, two releases were made during the graph’s plateau: Jikes 1.19 (31 January, 2004) and Jikes 1.20 (18 April, 2004). The last Jikes release before the plateau was made two years earlier. It may be that increased contribution to the project and a larger amount of development in a short time period caused a jump in the number of regression test failures, and their fixes were made in later releases.

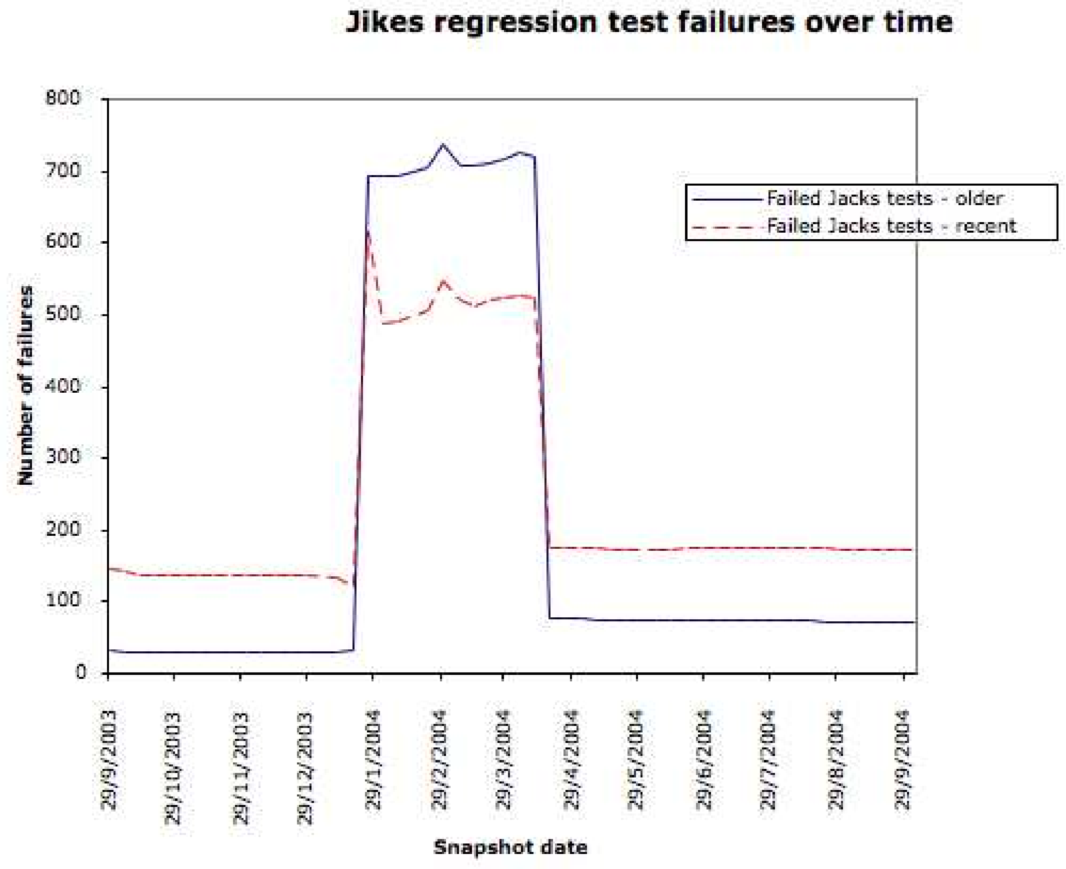

Figure 12.1: Jikes regression test failures over time

Whatever the cause, the drop in failed tests after this plateau is not accompanied by a corresponding drop in the number of operational failures for Jikes, as shown in Figure 12.1. Any Java class file resulting from a Jikes compilation that fails one pass of the JustICE verifier is considered a failed execution for the purpose of this study. The number of operational failures deviates much less from the average number of failed executions across all Jikes snapshots than the number of failed regression tests does. These data seem to indicate that for Jikes, regression test successes are not a good predictor of reliability, and the emphasis of the Jacks test suite is not on functionality that is frequently used by operational executions. This is unsurprising, since Jacks is meant to test adherence to the Java Language Specification, and so it includes many edge cases and rare constructs that would not ordinarily be exercised by real-world use.

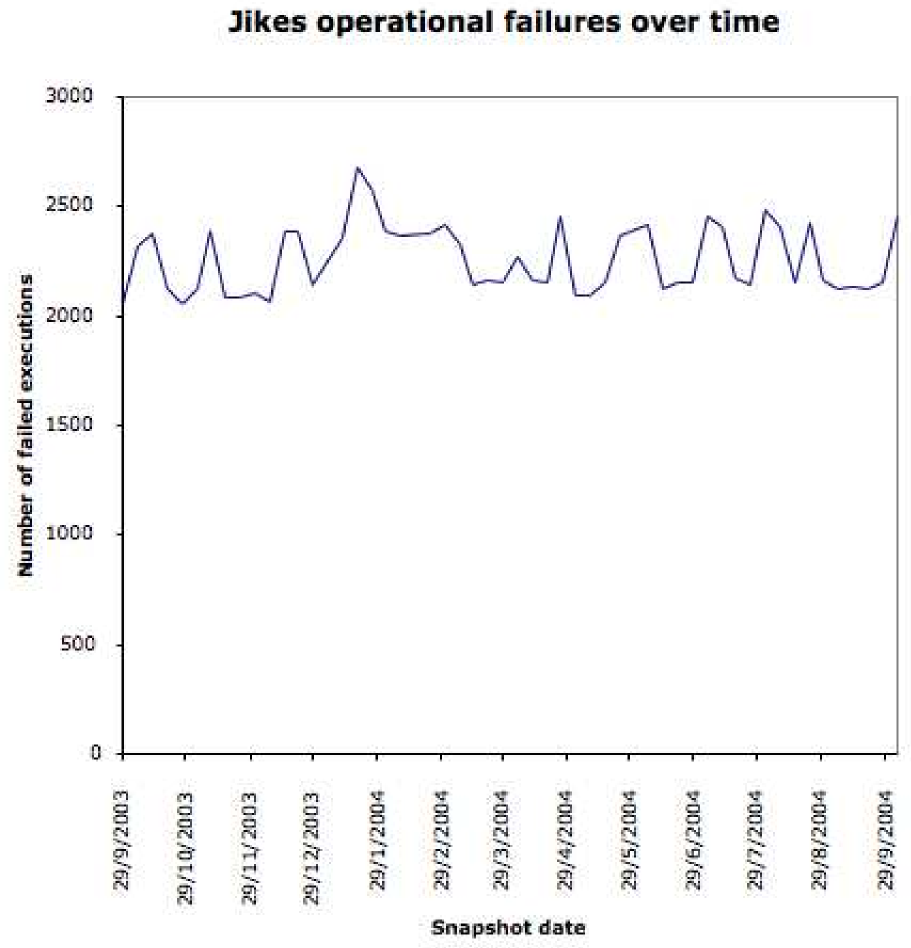

Figure 12.2: Jikes operational execution failures over time

Jikes code coverage results are still pending, as of 9 June, 2005. They will appear in the final version of this thesis.

12.2 HTMLTidy results

The number of regression test failures per snapshot for HTMLTidy was determined by executing the regression test suite that shipped with each snapshot, and comparing its results to the results of the oracle's test suite using *diff.* The oracle used was the release version of HTMLTidy that was available on 15 April, 2005. Results for the HTMLTidy regression test suite consist of the HTML output for each test, and the console output for each test, if any. The line in Figure 12.2 represents the number of output files for the results produced by each snapshot that differ from the results given by the oracle, minus the number of regression tests that were added to the test suite after the snapshot in question was released. The adjusted failures were calculated because *diff* counts files that only exist in one set as a difference- therefore each test that is not present in the test suites of earlier snapshots was counted as a failure, in the raw data. It can be seen from this graph that the number of new defects found by the regression test suite falls, even while the test suite is being changed and augmented.

In contrast, the number of defective operational executions does not change at all- during the entire time period investigated, 5,148 of the 16,373 operational executions from each HTMLTidy snapshot are failures. This means that even though the number of defects revealed in the software decreases over time, 31% of the operational executions of HTMLTidy fail. For HTMLTidy, regression test successes are no guarantee of reliability, and it seems from this initial data that the regression test suite must test functionality that is orthogonal to the features that are normally used in the real world. The test suite for HTMLTidy consists almost entirely of synthetic tests, and the occasional HTML file from the field that elicited a failure in some version of the software.

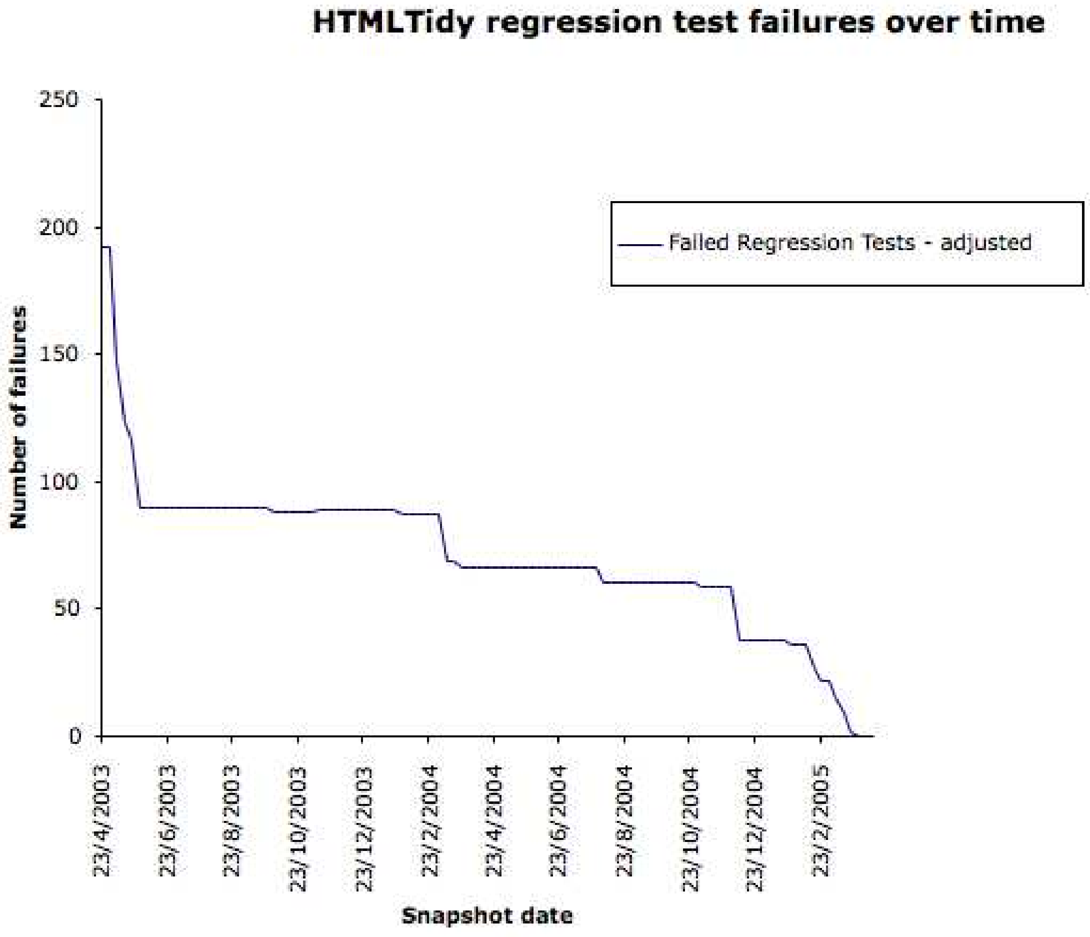

Figure 12.3: HTMLTidy regression test failures over time

Figure 12.2 is a graph of the percentage of code covered by the regression test suite in each of HTMLTidy's C source files. Overall, code coverage by the test suite increases slightly over time. Not surprisingly, jumps in code coverage correspond roughly with drops in the number of regression test failures, particularly near the beginning and end of the snapshot range. The changes in coverage of the files localize.c, clean.c and fileio.c are particularly prominent in this respect. These files contain code for "handl[ing] errors and general messages", "clean[ing] up misuse of presentation markup" and "do[ing] standard I/O" respectively, as indicated by the comments at the beginning of each of those files.

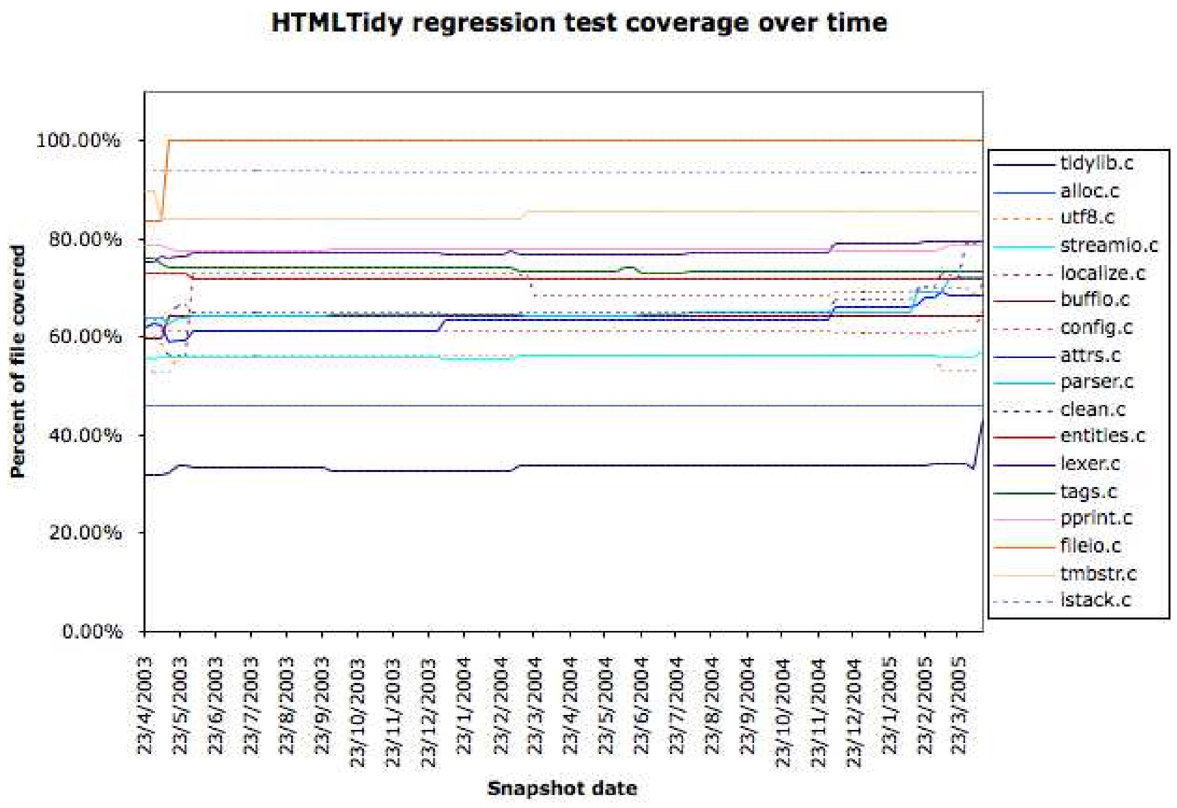

Figure 12.4: HTMLTidy regression test code coverage

Figure 12.2 is a graph of the code coverage for HTMLTidy's operational executions. Code coverage stays roughly constant over time for the operational executions, with a few exceptions at the beginning of the snapshot range. Like the code coverage for the regression tests, the operational executions' coverage for fileio.c and localize.c experiences a jump around 23 May, 2003. There are two possible explanations for such a jump: either more code was covered overall, or the number of executable lines of code was reduced. Further investigation revealed that the case of fileio.c fits the former explanation, and localize.c the latter. Before the coverage jump, there was one function in fileio.c that was never called; a change elsewhere in the program source caused that function to be called in later snapshots. In the case of localize.c, 156 lines of executable code were removed, and no gain in the actual number of lines covered was made. It should be noted that data from the snapshot for 10 Mar, 2004 had to be left out of Figure 12.2, because one of the operational executions caused the instrumented version of the snapshot to enter into an infinite loop. Also not shown in either code coverage graph for HTMLTidy are 4 source files, which had 0% coverage by both the regression tests and the operational executions for all of the HTMLTidy snapshots.

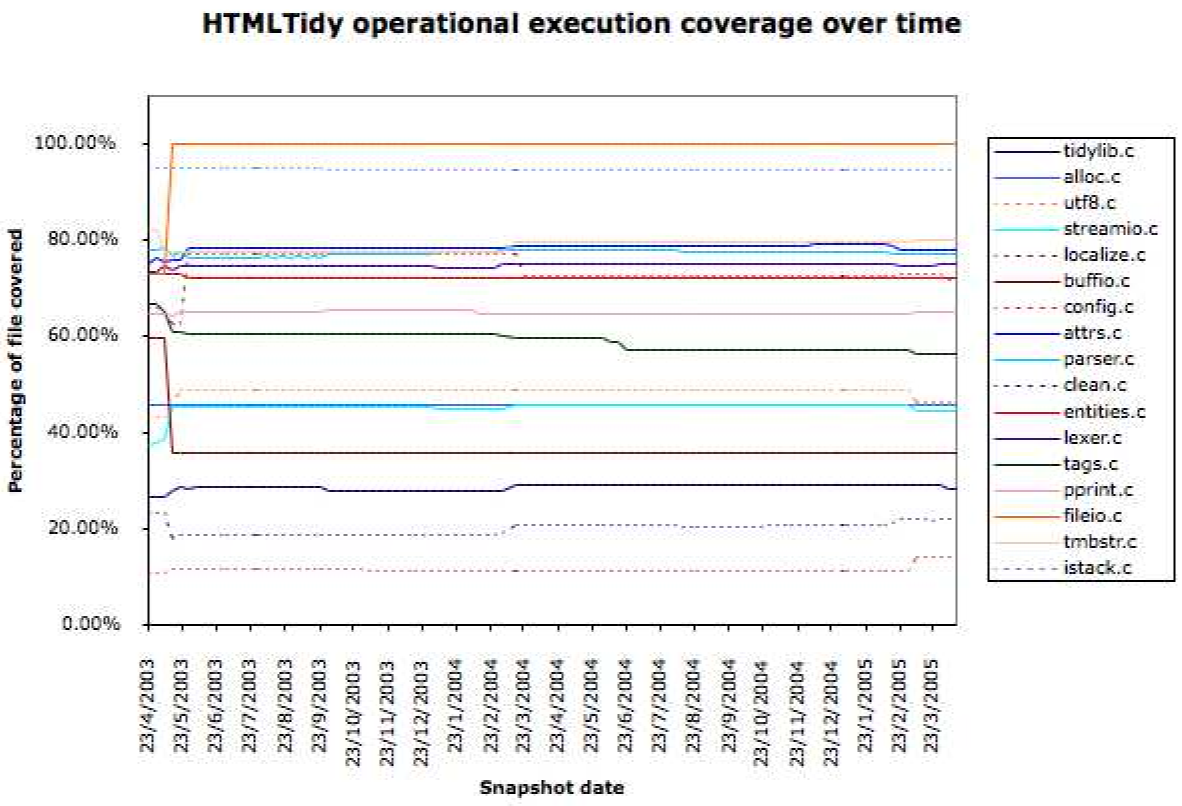

Figure 12.5: HTMLTidy operational execution code coverage

Five files in particular stand out as having large differences between the code coverage by regression tests and the code coverage by operational executions, and their code coverage statistics are detailed in Table 12.1. Two of them, clean.c and config.c, are covered more by the regression tests, while the other three experience more code coverage with the operational executions.

Table 12.1: Code coverage statistics for selected files in HTMLTidy

	regression test coverage			operational test coverage		
	min	max	avg	min	max	avg
clean.c	63.50%	79.10%	66.02%	17.73%	22.13%	20.09%
config.c	60.00%	65.62%	61.17%	10.66%	14.11%	11.51%
attrs.c	59.18%	69.09%	63.43%	75.20%	79.02%	78.50%
localize.c	56.16%	73.06%	69.91%	62.39%	77.17%	73.97%
parser.c	62.66%	72.31%	65.17%	76.44%	78.08%	77.35%

The file config.c contains code to “read the config file and manage config properties”, according to the documentation at its beginning. It could be surmised from these data that the regression tests for HTMLTidy contain a wider variety of input with respect to “dirty” presentation markup and configuration options. In order to investigate this phenomenon further, I compared the operational and regression test code coverage results of the snapshot that had the highest percentage of operational code coverage for each file. Examining clean.c revealed that the bulk of the unexecuted code was contained in 39 functions that were called during execution of the regression test suite, but not for the operational executions. The majority of these missed functions had to do with repairing some kind of HTML formatting flaw, such as closing certain tags and updating deprecated HTML syntax. Similarly, most of the unexecuted code in config.c was in 14 functions that had to do with configuration options passed to HTMLTidy on the command line, or from a configuration file- neither of which were used for the operational executions.

A similar comparison was performed for attrs.c, localize.c, and parser.c, with the code coverage results for the snapshot that had the highest percentage of regression test code coverage for each file. These files contain code to “recognize HTML attributes”, “handle errors and general messages” and parse the HTML input, respectively. I found that the bulk of the unexecuted code in these cases was found in unexercised branch conditions. For attrs.c, 19 branches were identified as having been exercised in the operational executions, but not in the regression tests. Most of these branches were error conditions, and contained a call to a *ReportError* function in HTMLTidy. The *ReportError* function outputs an error message to the console about undesirable or unexpected HTML markup that HTMLTidy has enccountered in the input file. The results for parser.c were similar- most of the code that was covered by the operational executions but not by the regression tests was found in 92 branches, which dealt with error conditions, markup corrections, and special cases such as comments and unexpected HTML elements. Many of these conditions that were left uncovered by the regression tests were well-documented by comments in the code, such that it would be easy to construct a test that elicited them. The results for localize.c were slightly different, with 3 functions and 6 branch conditions overlooked by the regression tests. The missed functions, however, were named *ReleaseDate, FileError* and *ReportEncodingWarning*, and all of the missed branches contained error-handling code. Such results indicated a possible need for more regression tests that are intended to check for such errors and special cases.

In addition to the experiments conducted on the rest of the subject programs, a special version snapshot of HTMLTidy from 7 May, 2003 was created, in which 48 defects that were caused by missing conditions were fixed, but a *failure checker* for each defect was included. These failure checkers detect the conditions under which a failure would be elicited, and report whether they are satisfied during program execution. The missing condition defects were discovered by examining CVS logs for HTMLTidy, and identifying patches that added one or more variables to an existing conditional statement (such as an if statement or the condition for a while loop) or introduced a new if statement. Using this instrumented version of HTMLTidy, I was able to determine which of these missing condition defects would be triggered during the execution of this particular snapshot.

24 of the 48 failure checkers for HTMLTidy were actually tripped by regression test and/or operational executions. As is shown in Table 12.2, the regression test suite elicited a slightly wider variety of defects than the operational executions, while the operational executions exihibited a wider range in failure frequency than the regression test suite. In Table 12.2, the total number of times elicited for each defect is shown, along with the number of times that the basic block containing the defect was executed. The basic block counts shown for each defect are the number of times that the basic block containing the failure checker was called. In other words, the basic block count is the number of times that program execution reached the point of the missing condition defect, and the number of times elicited is the number of times that the missing condition made a difference in the way that the program executed. It should be noted that all of the missing conditions that went unelicited by the regression tests were from the files parser.c, attrs.c, and lexer.c, which contains code that functions as "a lexer for the HTML parser". In contrast, the unelicited missing conditions from the operational executions are located in the files parser.c, lexer.c, tags.c, which contains code to "recognize HTML tags" and tidylib.c, which has HTMLTidy's "internal library definitions".

Defect name	Regression tests		Operational executions	
	Times elicited	Basic block count	Times elicited	Basic block count
1183751	2	2	2	1908
1054566*	6	7	2302	2371
1117013	7	118	2372	16,043
729957	16	40	1060	8743
patch A	1	131	1	16,043
1050673	0	17	2	38,491
735868	1,717	7,194	5,341,161	15,778,012
909576	1	40	0	8743
736924	1	3	1	391
734435	29	29	75,738	75,885
513113	0	1	30	60
994841	59	1,178	552,804	9,033,265
1031493	1	7	0	171,526
1062511	2	2	47,153	43,593
735923	2	41	3	3
736541	6	170	0	25,340
1079820 A	6	7	864	999
1053625	0	0	94	1,518
1098012	1	5	0	7
940741	0	6	37	179,548
996484	144	570	45,105	164,345
w3c	110	118	15,762	16,041
1030944	13	131	0	16,043
623046	2,170	2,383	7,108,872	7,148,028

* Defect 1054566 is always immediately followed by defect 117013 in both the regression tests and the operational executions.

Table 12.2: Missing condition defects elicited in HTMLTidy

In Figure 12.6, the average failure frequency and average basic block frequency per execution for each defect is shown. To obtain the average failure frequency, the total number of times elicited for each defect was divided by the number of executions for each data set. Similarly, the average basic block frequency is the basic block count for each defect divided by the total number of executions. There were 173 regression tests and 16,373 operational input files for this snapshot of HTMLTidy. The frequencies of failures caused by these defects range from once in the entire set of operational executions to 434 times per execution, on average, while the basic block frequency ranges between 0 and 963.

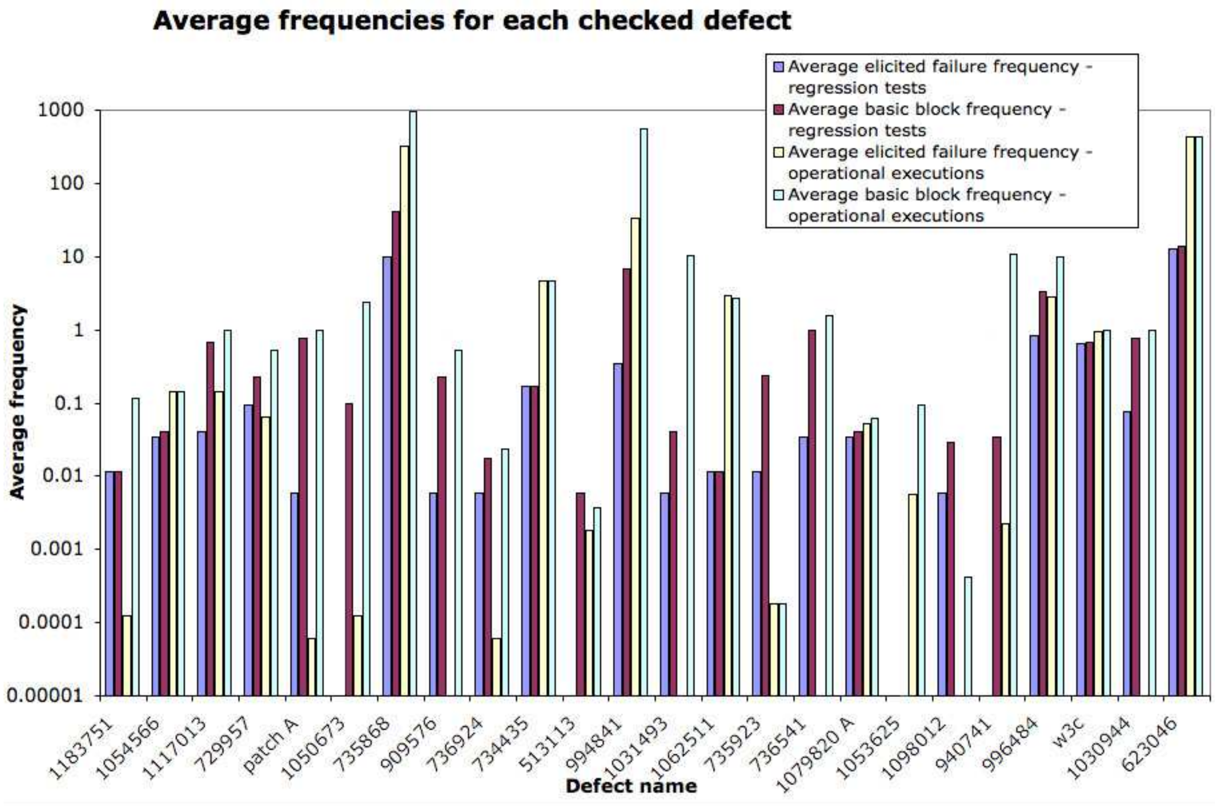

Figure 12.6: Average failure and basic block frequencies for HTMLTidy

12.3 Mono results

Figure 12.3 is the result of executing snapshots of Mono on the regression test suite described in Section 11.2. Individual versions of the test suite for each snapshot were not available. There are three cases for which a Mono regression test is considered to have failed for a particular snapshot:

- The snapshot will compile a test file when the oracle won't.
- The oracle will compile a test file when the snapshot won't.

- The snapshot produces an invalid executable according to PEVerify. Warnings were also counted, since running PEVerify on the oracle's test suite output produced no warnings.

There are 136 tests in the regression test suite for the Mono C# compiler, and the latest release within the snapshot date range, Mono version 1.0.6, was used as the oracle. For Mono, none of these tests stop failing during the course of the period examined- if a test fails for the oldest snapshot, it fails for the most recent one. When the number of failed regression tests remains constant between snapshots, the same tests are failing for each. Though it can be seen in Figure 12.3 that the number of failed regression tests for Mono increases over time, the number of valid executables that each snapshot produces stays constant. Since these snapshots were all tested with the same version of the regression test suite, the increases in the number of regression test failures is most easily explained by the introduction of new defects into the software.

For the operational executions in this study, 1,769 C# files are compiled into 127 executables. 59 of these executables are invalid for each snapshot and 63 are invalid for the oracle, according to PEVerify. This is a strong indicator that the number of passed regression tests is a poor predictor of operational reliability, especially given the fact that the Mono oracle passes all of the tests in the regression test suite. It seems, again, that the functionality being tested by the test suite does not reflect the functionality that is used in the field.

Though some of the regression tests are based on bugs discovered "in the wild", no real operational input seems to have been used in the actual tests. Some of the tests contain variable names or code that looks like operational input, but such code is still used in a small, specific test, and has been taken out of its original context. Context, for Mono, is especially important, since nearly all of the operational input that I found for Mono requires multiple files to be listed as arguments on the command line so that they can be compiled together in one execution of Mono. In contrast to this, the regression tests for Mono never have multiple files depending on each other. The closest that the regression test suite comes to doing so is including files from the .NET class libraries, such as System.Data, in the compiler directives.

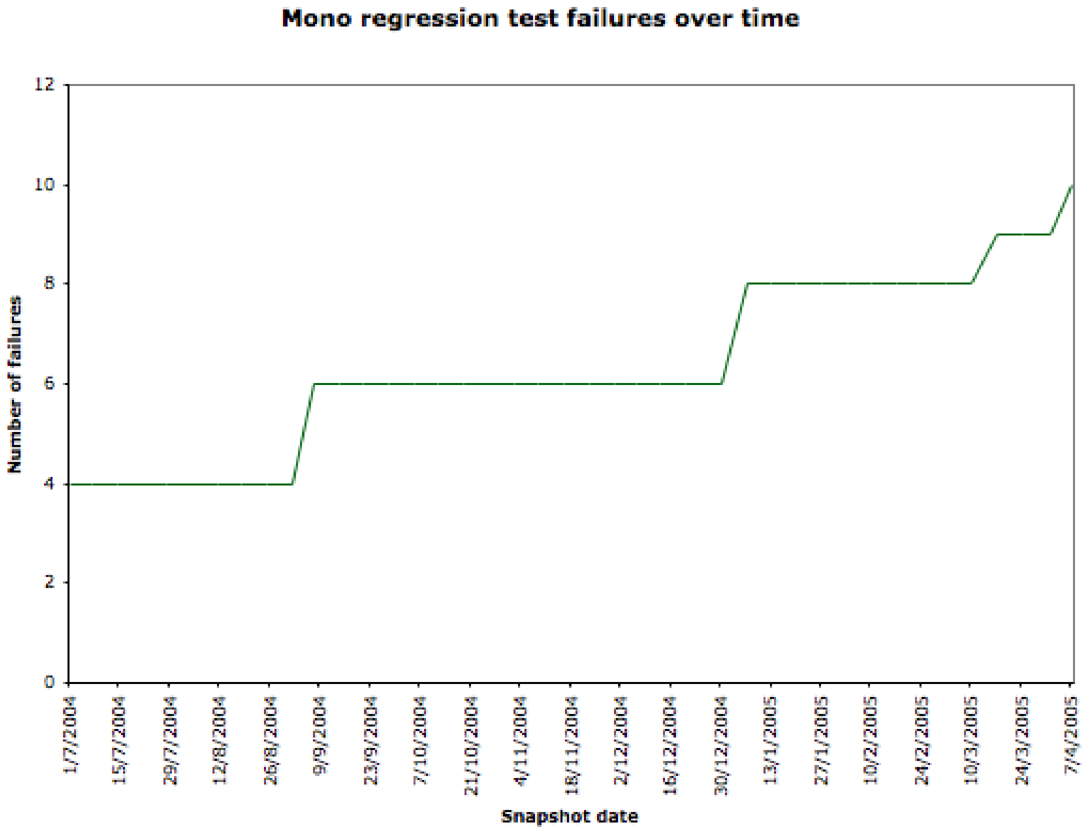

Figure 12.7: Mono regression test failures over time

Code coverage results for Mono are pending as of 9 June, 2005. They will appear in the final version of this thesis.

12.4 Summary of experimental results

A summary of experimental results will appear in the final version of this document.

12.5 Threats to Validity

Though I have attempted to make this study as thorough and correct as possible, some threats to its validity remain, and must be addressed. First, many different pieces of software are used for a variety of purposes in this study. All of them are likely to have defects, any of which could possibly have thrown off the results of this study. Also, the subject programs in this thesis were chosen according to very specific criteria, outlined in Section 11.2. The fact that they were selected in this way, rather than at random, suggests that they may not be representative of software projects as a whole. More experiments should be done with a wide variety of subject programs in order to confirm the results of this study.

There are possible issues with the choice of operational inputs, as well. Since the operational inputs for the compilers were gathered by browsing open-source projects available on the Internet, it is possible that they are not representative of the operational distribution for their subject programs. It may be that a large percentage of code written for those compilers' languages is not publicly available. This problem is not shared by the operational input for HTMLTidy, since most HTML documents do end up on the Internet. Also, because the operational input was gathered recently for this study from sources outside of my control, the date when each piece of input was created cannot be unequivocally determined, since timestamps on files can be changed or falsified. This means that it is likely that many of the subject programs' snapshots are being executed on operational input that is "from the future", as it would have been extremely difficult to make sure that each snapshot had a set of operational input that was comparable to the others in terms of size and behaviors elicited, as well as being from the snapshot date or earlier. If anything, however, this effect should skew the earlier snapshots toward failing more often, and it does not seem to have had any great effect on the number of operational failures in early snapshots.

Finally, it is difficult to compare the number of failed operational runs for Mono with the number of failed operational executions for the other subject programs, because Mono compiles many C# files together to make one executable. This makes the ratio of output to input for Mono's operational executions smaller than that of the other subject programs, which have 1:1 ratios of output to input.

Chapter 13

Future work

In the future, it would be useful to study the effects, costs and benefits over time of refreshing synthetic test suites with operational profile data gathered from end users of the software, beta testers, or a large amount of operational input. The effect on code coverage as well as the effect on the number of failures elicited by the new and old test suites might be studied and compared. Also, code coverage data from operational executions might be used for the purpose of test selection. For example, tests that exercise parts of the code that are consistenly covered by operational executions might be given a higher priority. This approach has the advantage of using existing, freely available tools, and would be a good way of making rough estimates about a program's operational profile. It would be useful to conduct a study that compares regression test selection using operational code coverage with other methods of regression test selection. Finally, the experiments in this study should be repreated with a variety of other subject programs in order to further establish its validity.

Chapter 14

Related Work

In [37], Musa presents the concept of the operational profile, which is a quantitative characterization of how a software system will be used. Musa contends that software reliability will increase when operational profiles are used for software testing, because the most frequently used features of the software will be tested the most thoroughly. Frankl, et al compare operational testing with traditional, or debug testing in [14], and find that for the purposes of demonstrating that software reliability has been achieved, operational testing is superior. In [18] Harder, et al present the operational difference methodology for generating, augmenting and minimizing test suites. This methodology applies techniques that are analogous to code coverge of program statements to semantic program properties, and it describes the operational behavior of the program under test as a formal mathematical statement.

Podgurski and Weyuker show that the practice of reusing regression test cases in order to estimate the reliability of modified software can lead to estimation bias, as dependencies exist between previous test results and software changes [42]. Instead, the reliability should be estimated by updating an estimate of the previous version's reliability. Kim and Porter discuss a method of regression test selection and prioritization that is based on historical execution data and the results of previous regression tests [30]. Chen, et al discuss the overestimation of software reliability by reliability growth models that use code coverage information alone. They improve such estimates by using coverage information in conjunction with a measure of time intervals between failures elicited by a test suite.

Marick notes that gaps in the code coverage of a regression test suite likely indicate the presence of undiscovered missing condition defects in the software under test, and notes that existing code coverage is not uniformly distributed [33]. Raghavan, et al discuss Dex, a semantic-graph differencing tool for examining patches to large software projects [43]. Their work concentrates on finding patches for defects caused by missing conditions, but does not make inferences about the number of missing conditions in the software under test; rather, it attempts to discover the number of changes that would suggest the existence of such defects.

Elbaum, et al address the way in which the changes that come from software maintenance impact the code coverage for a program over time and across successive versions of the software [12]. Though they state that their technique and metrics would hold for any input to the subject programs, synthetically created inputs and existing test suites are used to execute the subject programs of their study. Bishop presents a theory for using test coverage data to estimate the number of residual faults in a software project, and claims that linear extrapolation of code coverage can be used to make a conservative estimate the number of residual faults [2]; however, it is assumed in that study that faults are evenly distributed over the executable code.

Several of the studies detailed in this section use code coverage information, but none examine the code coverage of operational executions, the differences between the code coverage of operational executions and that of regression test suites, or the way that such differences change over time. Much of the prior work investigates various attributes and information to be gleaned from operational executions; however, none examine code coverage data from operational executions.

Chapter 15

Conclusions

A conclusion will appear in the final version of this document.

Chapter 16

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.

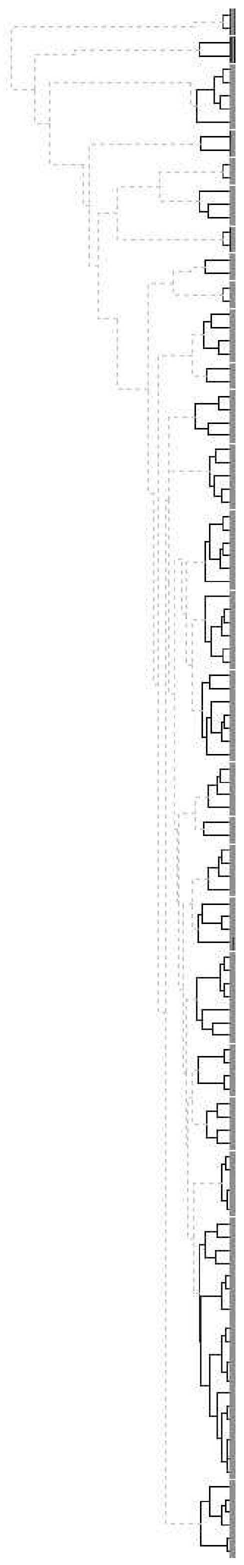

Figure 16.1: Full dendrogram for GCC

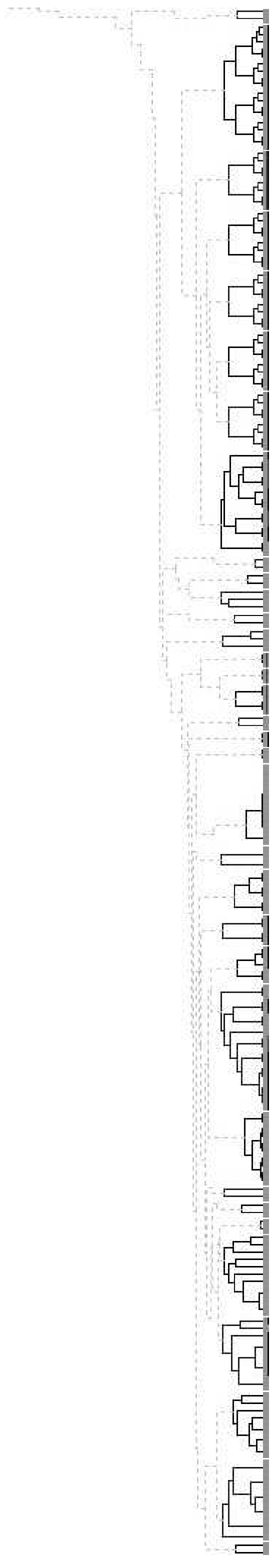

Figure 16.2: Full dendrogram for Javac

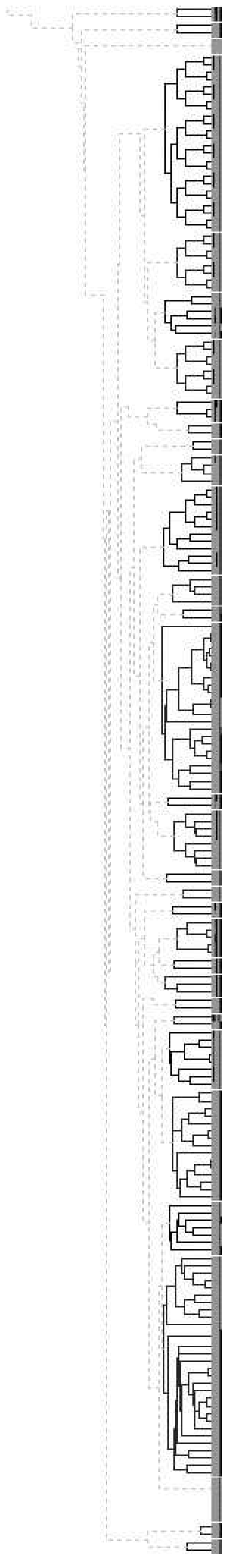

Figure 16.3: Full dendrogram for Jikes

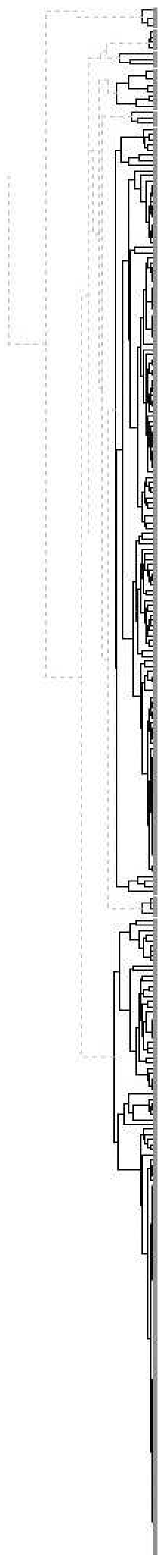

Figure 16.4: Full dendrogram for JTidy

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] Bishop, P. G. Estimating residual faults from code coverage. 21stIntl. Conf. on Computer Safety, Reliability and Security (Catania, Italy, September 2002), 163-174.

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

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

[5] Chen, M.H., Lyu, M.R. and Wong, W.E. An empirical study of the correlation between code coverage and reliability estimation. 3rdIntl. Software Metrics Symposium (Berlin, Germany, March 1996), 133-141.

[6] 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).

[7] Clover.NET. Cenqua Pty Ltd., `http://www.cenqua.com/`, 1999 - 2005.

[8] Concurrent Versions System (CVS), http://www.gnu.org/software/cvs/ . Free Software Foundation Inc., 1998.

[9] Cygwin, http://www.cygwin.com. Red Hat, Inc., 2000 - 2005.

[10] 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.

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

[12] Elbaum, S., Gable, D. and Rothermel, G. The impact of software evolution on code coverage information. Intl. Conf. on Software Maintenance (Florence, Italy, November 2001), 169-179.

[13] 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.

[14] Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L. Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering 24, 8 (August 1998), 586-601.

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

[16] GCover. http://poa.berlios.de/gcover-project/, The POA Team, University of Stuttgart, 2004.

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

[18] Harder, M., Mellen, J. and Ernst, M. D. Improving test suites via operational abstraction. 25thIntl. Conf. on Software Engineering (Portland, OR, May 2003), 60-71.

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

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

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

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

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

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

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

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

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

[28] 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).

[29] 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.

[30] Kim, J. and Porter, A. A history-based test prioritization technique for regression testing in resource constrained environments. 24thIntl. Conf. on Software Engineering (Orlando, FL, May 2002), 119-129.

[31] 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.

[32] 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).

[33] Marick, B. How to misuse code coverage. 16thIntl. Conf. and Exposition on Testing Computer Software (Washington, D.C, June 1999).

[34] Microsoft .NET Platform.`http://www.microsoft.com/net/`, Microsoft Corporation, 2005.

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

[36] Mono. The Mono Project, `http://www.mono-project.com`, 2005.

[37] Musa, J. D. The operational profile in software reliability engineering: an overview. 3rd Int'l Symposium on Software Reliability Engineering (Research Triangle Park, NC, October 1992), 140-154.

[38] NAnt, `http://nant.sourceforge.net/`. Gerry Shaw, 2001 - 2005.

[39] PEVerify Tool, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfpeverifytoolpeverifyexe.asp`. Microsoft Corporation, 2005.

[40] 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).

[41] 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.

[42] Podgurski, A., and Weyuker, E. Re-estimation of software reliability after maintenance. 19th Intl. Conf. on Software Engineering (Boston, MA, May 1997), 79-85.

[43] Raghavan, S., Rohana, R., Leon, D., Podgurski, A. and Augustine, V. Dex: A semantic-graph differencing tool for studying changes in large code bases. 20th IEEE Intl. Conf. on Software Maintenance (Chicago, IL, Sept 2004), 188-197.

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

[45] 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`.

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

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