Improving the Classification of Software Behaviors using Ensembles of Control-Flow and Data-Flow Classifiers

James F. Bowring, Mary Jean Harrold, and James M. Rehg
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{bowring | harrold | rehg}@cc.gatech.edu

ABSTRACT

One approach to the automatic classification of program behaviors is to view these behaviors as the collection of all the program's executions. Many features of these executions, such as branch profiles, can be measured, and if these features accurately predict behavior, we can build automatic behavior classifiers from them using statistical machine-learning techniques. Two key problems in the development of useful classifiers are (1) the costs of collecting and modeling data and (2) the adaptation of classifiers to new or unknown behaviors. We address the first problem by concentrating on the properties and costs of individual features and the second problem by using the active-learning paradigm. In this paper, we present our technique for modeling a data-flow feature as a stochastic process exhibiting the Markov property. We introduce the novel concept of *databins* to summarize, as Markov models, the transitions of values for selected variables. We show by empirical studies that databin-based classifiers are effective. We also describe ensembles of classifiers and how they can leverage their components to improve classification rates. We show by empirical studies that ensembles of control-flow and data-flow based classifiers can be more effective than either component classifier.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; G.3 [Mathematics of Computing]: Probability and Statistics; I.2.6 [Artificial Intelligence]: Learning

General Terms: Measurement, Reliability, Experimentation, Verification

Keywords: Software testing, software behavior, machine learning, Markov models

1. INTRODUCTION

The automatic detection and classification of software behaviors is an important component of many software development and maintenance activities. For example, to improve the quality of software after its release, developers can monitor the deployed software and use the results of the monitoring to determine automatically the frequency and properties of the software behaviors, including failures [7, 20]. For another example, to facilitate autonomic computing, developers can model software systems as self-regulating biological systems, which requires automated behavior detection to support system self-awareness [1, 10, 15].

One approach to analyzing behaviors is to view a program's behaviors as the collection of all its executions. There are many features of these executions, such as branch profiles, for which we can collect statistical measures. If these features are accurate predictors of program behavior, we can build models and automatic behavior classifiers from them using a broad range of statistical machine-learning techniques.

Recent research in software engineering has resulted in approaches that use established procedures to build predictive models from statistical measures of program executions (e.g., [5, 7, 11, 16, 21]). These approaches model large sets of features and train classifiers using *batch-learning*, wherein the machine-learning process depends on a fixed quantity of manually-labeled training data. These approaches do not, however, address two key problems in the development of useful classifiers for software engineering applications. The first problem concerns the costs of modeling features and collecting data. Creating models requires analysis of the properties of the considered features, which can result in significant space and computational costs. Program instrumentation, which is the most common way to collect data from executions, adds execution overhead to a program in terms of both memory and running time. Because feature modeling and data collection efforts are proportional to the number of features monitored, identifying the most effective features can provide significant savings. The second problem concerns the need for classifiers to adapt to and recognize new behaviors of a program. New behaviors may appear as the usage of a program matures and evolves and as it is subjected to new inputs. Classifiers built once, such as at program release, by batch learning on a known distribution of behaviors cannot adapt so as to correctly classify new behaviors. An approach that accommodates new behaviors can result in a resilient classifier that is useful for the lifetime of a program.

To address the first problem with existing techniques, we are concentrating on identifying the properties and costs of individual features. In previous work, we showed that pro-

files of individual control-flow features can form the basis for effective classifiers [3]. This focus on individual features contrasts with related work (e.g., [21]) that uses randomly selected sets of features. Specifically, we demonstrated that branch-profile and method-call-profile features can be modeled as stochastic processes that exhibit the *Markov property*, which implies that predictions about future states depend only on the current state. Because Markov models are "history-less," they are efficient summarizers of events. We developed an automated clustering technique for Markov models of a single feature that aggregates multiple program executions into effective classifiers. By investigating the predictive power of individual features, we seek to understand the costs of using each feature.

To address the second problem with existing techniques, we are using *active learning*, where the classifier trains incrementally on a series of labeled data elements that summarize a feature of program executions. During each iteration of active learning, the current classifier selects from the pool of unlabeled executions those that exhibit unknown behaviors. A test engineer, or an oracle, labels these selected executions for inclusion in the training set for the next round of learning. In this way, the classifier adapts by extending the scope of behaviors that it recognizes. We have demonstrated that active learning can make better use of the resources available for analyzing and labeling program execution data than does batch learning [3].

In our previous work, we concentrated on control-flow features, and built classifiers that correctly classified new executions between 50% and 90% of the time. Although our classifiers performed well in most cases, there were cases in which the control-flow features were poor predictors. To see this, consider two versions of a program that exhibit identical control-flow for all or most of their inputs, but produce conflicting outputs. A simple example of this is a straight-line program P, with no conditional statements, that calculates the square of a number x. If one version of P calculates x^2 using the multiplication operator, as in "$x * x$," and another version of P mistakenly uses the addition operator, as in "$x + x$," both versions will exhibit identical control-flow behavior under all inputs. However, the two versions will produce the same output only when the input is either 0 or 2. In all other cases, the data-flow behaviors of the two versions of P differ.

To capture the data-flow behaviors of programs for use in building classifiers, we developed the novel concept of databins as a way to summarize as Markov models the transitions of values for selected variables. A *databin* for a variable during one execution is one of a small number of percentile ranges of the values of that variable that acts as a state in a state-transition matrix containing each transition's relative frequency or profile. To provide a classifier that is sensitive to different features of a program's execution, we also developed an *ensemble* classifier that combines the two complementary features—branch profiles and databin profiles.

In this paper, we introduce our classification technique based on databin profiles. We show, using empirical studies, that behavior classifiers based on databin profiles can be effective and that active learning is less expensive than batch learning for training databin-based classifiers. We also present ensemble classifiers that are composed of two classifiers built from different features. In this paper, we show empirically that ensemble classifiers based on branch profiles and databin profiles can outperform their component classifiers. The improvement realized with an ensemble classifier is consistent with results in machine learning, and arises when two good classifiers view a set of data from different but complementary perspectives [12]. Thus, our finding of improved classification rates for ensembles suggests that control-flow and data-flow features of program executions capture divergent statistical views of behaviors.

The main contributions of this paper are:

1. The introduction of databin transitions as a stochastic representation of data-flow in a program;
2. The description of an ensemble classifier that combines control-flow and data-flow based classifiers;
3. Empirical studies that demonstrate that
   (a) active learning of databin-profile classifiers is more cost effective than with batch learning, as we have previously shown for branch-profile classifiers and
   (b) branch-profile based classifiers are generally more effective than databin-profile based classifiers
   (c) ensembles of classifiers based on branch-profiles and databin-profiles can outperform classifiers based on either feature individually.

2. BUILDING BEHAVIOR CLASSIFIERS

In this section we overview our previous work in utilizing Markov models of state-transition profiles to build behavior classifiers. We also review our approach to reducing the costs associated with classifier building.

2.1 Modeling a Feature's State Transitions

In our work, we seek to isolate and evaluate individual features, and to do so, we concentrate on the subset of features that profile state transitions in program executions. We define a *state transition* for a program feature as a transition from one program entity to another. Types of control-flow state transitions include branches (source statement to sink statement), method calls (caller to callee), and branch-to-branch. Types of data-flow state transitions include definition-use transitions; we discuss data-flow transitions in more detail in Section 3.

In previous work, we showed that control-flow state-transition features can describe stochastic processes that exhibit the Markov property by building Markov models from them that effectively predict program behaviors. The Markov property provides that the probability distribution of future states of a process depends only on the current state. Thus, a *Markov model* captures the time-independent probability of being in state s_1 at time $t + 1$ given that the state at time t was s_0. In this modeling context, the states of the Markov model are the program execution states that define the feature of interest. For example, the Markov model states for a branch are the state transitions between source nodes and sink nodes representing the branch in the control-flow graph[1](CFG) of a program. The relative frequency of a state transition during a program execution provides a measure of its probability. For a source node that is a predicate there are two branches, "true" and "false," each representing a transition to a different sink node. The transition

[1]A *control-flow graph* is a directed graph in which nodes represent statements or basic blocks and edges represent the flow of control between the statements or basic blocks.

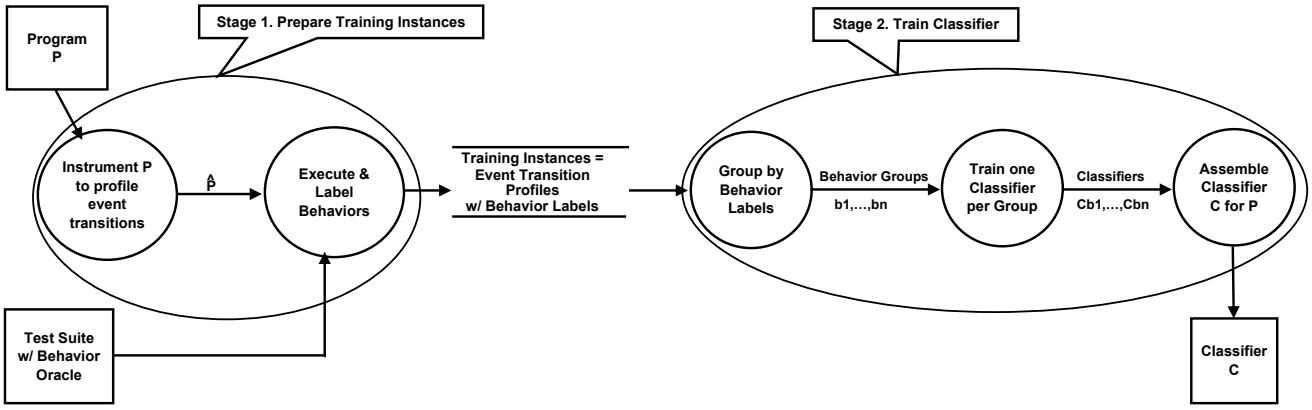

Figure 1: Technique: Stage 1 - Prepare Training Instances; Stage 2 - Train Classifier.

probabilities between this source node state and each of the two sink nodes states, in the Markov model, are the relative execution frequencies, or profiles, of the respective branches.

2.2 Training Classifiers

We developed a two-stage technique for training behavior classifiers from the profile data of a state-transition feature of a program collected during multiple executions. We demonstrated its use in building classifiers from branch-profile data [3], but the technique is generally applicable to any state-transition feature. Figure 1 shows a dataflow diagram[2] of our technique. The technique takes as inputs a subject program P, a test suite with a behavior oracle, and outputs a Classifier C. P's *Test Suite* contains test inputs. The *Behavior Oracle* outputs a behavior label b_k, such as "pass" or "fail," for each execution e_k of P induced by test input t_k.

In Stage 1, Prepare Training Instances, the technique instruments P to get $\hat{P}$ so that as $\hat{P}$ executes, it records state-transition profiles for the chosen feature. For each execution e_k of $\hat{P}$ with test input t_k, the behavior oracle labels e_k. This produces a *training instance*—consisting of e_k's state-transition profiles and its behavior label—that is stored in a database of *Training Instances*.

In Stage 2, Train Classifier, the technique first groups the training instances from the database by the distinct behavior labels $b_1, \ldots, b_n$ generated by the behavior oracle. For example, if the behavior labels are "pass" and "fail," the result is two behavior groups. Then, the technique converts each training instance in each behavior group to a Markov model, as discussed above. The technique uses an automatic clustering algorithm to train one classifier C_{b_k} per behavior group b_k. The clustering algorithm is an adaptation of an established procedure known as *agglomerative hierarchical clustering* [9]. Using this procedure, each training instance is initially considered to be a cluster of size one. The technique proceeds iteratively by finding the two clusters that are nearest to each other according to some similarity function. These two clusters are then merged into one cluster, and the procedure repeats. The stopping criterion is either a desired number of clusters or some valuation of the quality of the remaining clusters. Finally, our technique assembles the group classifiers, $C_{b_1}, \ldots, C_{b_n}$ into the classifier C for P.

[2]In a *dataflow diagram*, boxes represent external entities, (i.e., inputs and outputs), circles represent processes, arrows depict the flow of data, and parallel horizontal lines in the center represent a database.

2.3 Using the Classifier

We use classifier C to label executions of $\hat{P}$ that were not in the training set. The new execution is modeled from the collected profile data. Each constituent clustered model of C rates the execution with a probability score. The model in C with the highest probability score for the new execution provides the behavior label. For example, if there were three clusters representing behavior label "pass" and two clusters representing behavior label "fail," the scoring process would produce five probabilities or votes. The cluster with the highest relative probability wins the vote and labels the execution.

The *probability score*, PS, is the probability that the model M could produce the sequence of state transitions in the subject execution. As an example, consider an execution of a program with the following trace of branches:

$$\{Branch_1, Branch_3, Branch_1, Branch_3, Branch_2\}.$$

Suppose we have a cluster model M with the following probabilities, $Pr()$, for these three branches:

$$Pr(Branch_1) = 0.9,\ Pr(Branch_2) = 0.1,\ Pr(Branch_3) = 0.333.$$

To calculate the probability score PS that the Markov model M produced the trace, we compute the product of the successive probabilities of the trace using the probabilities in M:

$$PS = Pr(Branch_1) * Pr(Branch_3) * Pr(Branch_1) * Pr(Branch_3) * Pr(Branch_2).$$
$$PS = 0.9 * 0.333 * 0.9 * 0.333 * 0.1 = 0.008982.$$

For active learning, we specify that the classifier report a subject execution as *unknown* whenever this calculated probability is below some heuristically-determined threshold. We detail this heuristic in Section 5 (Empirical Studies).

Note that probabilities calculated by multiplication can become very small. To overcome this, we use a standard technique with Markov models of converting the probabilities to their negative natural logarithms and then summing them. This transformation preserves the ordering of the results.

2.4 Active Learning and Cost Reduction

Active learning can provide cost savings in the construction of classifiers over the batch-learning approach. In *active learning*, the predictions of the current classifier inform the decision about which data item to process next during classifier training. This filtering process reduces the manual effort required to label new data. Active learning is especially useful when the scope of the data is not fully known, which is generally the case with software behaviors. For example, in our technique, we can begin with a small test suite to produce a small training set and then use the resultant classifier to classify new executions induced by additional test inputs. Our classifier will either label each new execution's behavior or report it as unknown. When these unknown behaviors do occur, the active learning paradigm provides for qualitative feedback from an oracle. The oracle in this case is a test engineer who evaluates the execution and then manually labels the behavior. After the oracle labels the execution, that execution becomes a new training instance for the classifier. The new, refined classifier can now recognize the new behavior.

The cost savings in this process are directly proportional to the number of executions that must be evaluated and labeled by a test engineer. Conventionally, test engineers evaluate every test execution and label it. Using active learning, we start with a small test suite and then we evaluate, label, and incorporate into the model only those additional executions that the classifier deems as new or unknown. At the limit, active learning may require that every new execution be evaluated, thereby providing no savings over the conventional process. However, we have demonstrated in previous work that active learning can reduce evaluation costs over batch learning while also yielding a better classifier [3].

3. DATA-FLOW FEATURES

To extend the range of features for our classifiers, and based on our success in modeling control-flow features as stochastic processes, we sought to model data-flows as stochastic processes. We can define a state transition for a data-flow feature as a transition from one variable state to another. Types of data-flow state transitions include definition-use pairs and transitions between values or value ranges of variables.

We found experimentally that traditional data-flow transitions such as definition-use pairs were quite similar in structure to those we have seen with control-flow transitions. As an alternative, we elected to model the transitions in the flow of data values as a way to extract a distinct feature from an execution.

In modeling data-flows as state transitions of the values of variables, we encountered three main problems. First, most programs use a wide range of variables and types, each of which may be instantiated numerous times. Tracking the values of these variables would be at best expensive and at worst intractable. Second, a single variable's range of values might differ between executions. For example, suppose one execution of a program calculates the mean of values in $[0, \ldots, 1]$ and another execution calculates the mean of values in $[10^3, \ldots, 10^5]$. Lack of normalization of this difference of ranges for each variable complicates the collection and modeling of the value ranges. Third, the number of states for a single variable could become intractable if each distinct value were to be considered as a state. For example, a loop variable that iterates through all 8-bit integers would produce an unwieldy number of states. Clearly, a method for controlling the size of this state space is required.

Transition Matrix

databin	1	2	3
1	2	1	1
2	1	1	1
3	1	1	1

Markov Model

databin	1	2	3
1	0.5	0.25	0.25
2	0.33	0.33	0.33
3	0.33	0.33	0.33

Figure 2: Databin example.

To solve the first problem, we needed to reduce the number of variables while preserving the predictive power of our behavior model. After exploring the variable space of a number of our subjects, we arrived at the following simplifying heuristic to reduce the number of considered variables:

1. Consider every use of each field of structures and classes as the use of a single variable.
2. Consider all elements of an array as instances of a single variable if not considered by the previous restriction.
3. Ignore constants.
4. Ignore variables that behave as constants.
5. Ignore variables local to any method.
6. Ignore pointer variables.
7. Ignore booleans and variables that take only two values
8. Consider only the first character of any string variable

3.1 Databin Models

To solve the second and third problems, we created a representation that partitions the range of values into a fixed number of bins (this approach is similar to that used in histograms). These bins, which we call *databins*, are percentiles of the range for a given variable, where each percentile becomes a state in a transition matrix. The use of databins solves the problems of normalization and of defining a tractable set of states for a variable. For example, consider a databin count of 3. In this case, the range of a variable's values will be partitioned or *binned* into three percentile ranges: the lower third, the middle third, and the upper third. If the range of a variable *var* during an execution is $[0, \ldots, 8]$, then Databin-1 represents $[0, 1, 2]$; Databin-2 represents $[3, 4, 5]$; and Databin-3 represents $[6, 7, 8]$. If we consider individual databins as equivalence classes for the values of *var* across executions, this binning into percentiles of the range provides the required normalization. For instance, if the range of *var* during another execution were $[0, \ldots, 23]$, then the three databins would be $[0, \ldots, 7]$, $[8, \ldots, 15]$, and $[16, \ldots, 23]$.

During a program's execution, we measure state-transition profiles for databins. From these we build stochastic models of a variable's behaviors. Consider our variable *var* and three databins. Suppose we collect the following sequence of values for *var* during an execution: $0, 1, 2, 6, 1, 4, 8, 8, 5, 5, 1$. The range is $[0, \ldots, 8]$, and the three databins are $[0, 1, 2]$, $[3, 4, 5]$, and $[6, 7, 8]$. We transform this sequence of values to a sequence of databins, using the integers $1, 2, 3$ to represent the three databins: $1, 1, 1, 3, 1, 2, 3, 3, 2, 2, 1$. We convert this sequence of databins to a transition matrix of size 3x3 by traversing it and counting transitions. Note that for a transition matrix, the starting state is a row label, and the

ending state is a column label. By normalizing the transition matrix, we obtain a Markov model representation. The two matrices are shown in Figure 2.

3.2 Databin Classifiers

For the branch-profile feature, we model each execution of the program as a single transition matrix and equivalent Markov model. For databins, we model each execution as a set of databin transition matrices, with one matrix for each variable. Thus we use as the component models only those Markov models that form the diagonal of the larger Markov model that represents all possible transitions, as shown abstractly in Figure 3. In the figure, each box along the diagonal is a Markov model similar to the one shown in Figure 2, representing a summary of the transitions between percentile ranges for one variable. In this paper, we do not consider the potential transitions between the values of different variables, depicted as empty cells in Figure 3.

Referring to our technique shown in Figure 1, the clustering process requires a similarity metric to compare two databin models. We calculate similarity as the sum of the pair-wise similarity measures between each of the Markov models in the set of variable models. We previously demonstrated that a useful heuristic for the similarity of two Markov models is their Hamming distance when a threshold transforms very low values to zeroes [3]. We also require a procedure to merge two databin models. Again, we proceed by defining the merged model as the set of merged Markov models, where the models of matching variables are merged.

With these modifications, our technique for building classifiers (Figure 1) will also build a classifier from databin profiles. Note that the classifier C produced by our technique is a collection of clusters, each of which is itself a databin model created by the successive mergings of the clustering process. To use this classifier to label a new subject execution, we proceed in a fashion similar to that detailed in Section 2. First we calculate the probabilities that each of the component clusters in the classifier produced the subject execution, as represented by its databin profiles. These probabilities are calculated as the product of the probabilities that each variable model in the cluster's set of variables produced the corresponding databin profiles for the same variable in the subject execution. For example, suppose that we have a program with two variables $v1$ and $v2$, and that we have created a classifier C, using our technique. One of the clusters of C is a databin model consisting of two Markov models, one for each of $v1$ and $v2$. Further suppose that these models are both identical to the one shown on the right in Figure 2. We ask this databin model to score a new execution that has the following databin profiles of the form ($databin_{from}$, $databin_{to}$, $profile$):

$v1$: (1,1,3); (2,3,1) $v2$: (1,2,2); (3,2,1)

The probability score, PS, is calculated as the product of the probability that the databin model's $v1$ produced $v1$ in the execution times the similar calculation for $v2$:

$$PS = Pr(v1) * Pr(v2) = [(0.5^3) * (0.33^1)] * [(0.25^2) * (0.33^1)] = 0.00085078.$$

The cluster in C with the highest relative probability above a given threshold wins the right to label the execution's behavior. If all the probabilities fall below the threshold, then the classifier reports that the behavior is unknown.

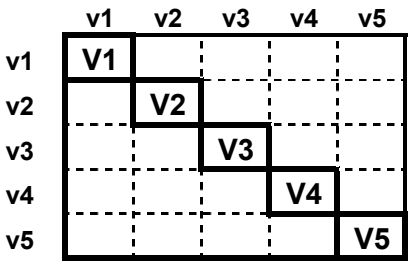

Figure 3: Abstracted Markov model for five variables each with a databin transition matrix.

4. ENSEMBLES

Our technique produces classifiers from the state-transition profiles of a single feature, such as branch profiles or databin profiles. These classifiers label new executions based on statistical summaries of the particular feature. Consider a program P and suppose that we use our technique to build two classifiers for P—one based on branch profiles, C_{branch}, and the other based on databin profiles, $C_{databin}$. Additionally, suppose that we produce a set of new executions for which we collect both branch profiles and databin profiles. We are interested in the extent to which C_{branch} and $C_{databin}$ agree on their classifications. If there is a divergence in agreement, we would like to know if we can leverage it to produce a better classifier by somehow combining C_{branch} and $C_{databin}$.

One approach is to combine two or more feature-based classifiers into an ensemble classifier. The machine-learning community uses *ensemble classifiers* in a wide variety of applications. Hansen and Salamon show that the necessary and sufficient conditions for ensemble classifiers to outperform their component classifiers are that the components each have classification error rates less than 0.5 and that they have diverging classifications of the same test data [12]. Ensemble classifiers fall into two general categories: those that manipulate the data directly during training, and those that bring together existing classifiers. The first category includes *bagging* and *boosting*, which train iteratively on weighted data sets [4, 24]. The second category includes weighted voting algorithms [18].

In this paper, we explore using binary feature ensembles of classifiers: one based on control-flow profiles and the other based on data-flow profiles. As a result, we can use a simple weighted voting scheme to produce the ensemble's classification. The two component classifiers report their respective labels, including "unknown," and their confidence in these labels. We have previously defined the "unknown" label to mean that the probability associated with the label is below a threshold. This is a special case of our confidence measure, which is a normalized distance from this threshold. The voting scheme consists of comparing the labels reported by the two classifiers. If the two classifiers agree, the common label is used to label the execution. If they disagree, the classifier with the higher confidence wins the vote and its label is used to label the execution. We have developed a heuristic for defining the threshold after each training session. The classifier classifies its own training set of data and produces a range of probabilities, calculated as described in Section 2. The threshold is the mean of these values.

Subject	*LOC*	branches	variables	test cases
flex	7447	2538	60	533
sed	6102	2560	12	1293
space	5420	1228	25	13585
print_tokens	481	133	7	4072
replace	415	150	32	5542
schedule	227	74	15	2650

Table 1: Table of subject programs.

5. EMPIRICAL STUDIES

The goals of our empirical studies are to evaluate the performance of databin-based classifiers, and then to compare them with branch-based classifiers directly and as cooperating members of ensemble classifiers. As discussed earlier, we have previously shown the efficacy of control-flow based classifiers and the advantage of using active learning to build them [3]. To reach our goals, we pose the following research questions:

Q1: How does active learning compare with batch learning for databin-based classifiers?

Q2: How do ensemble classifiers composed of both a control-flow and a data-flow classifier perform under active-learning?

5.1 Measure of Dependent Variable

Our measure of the quality of a classifier is the classification rate, which is the only dependent variable used in these evaluations. The *classification rate* is defined as

$$\frac{\text{number of test instances correctly classified}}{\text{total number of test instances classified}}.$$

Our classifiers provide a confidence measure with each classification. If the confidence level is below a heuristically-determined threshold value, then the classification result is set aside and the classifier reports instead that the given test instance is unknown. This threshold is set each time the classifier trains: the classifier classifies its own training set and computes the parametric statistics for the probabilities produced. The threshold is the mean of these probabilities. In calculating the classification rate, those test instances that are reported as unknown are considered to be incorrectly classified. For example, suppose classifier C scores a total of 100 executions and correctly classifies 80 and reports that 5 are unknown. Then the classification rate for C is $\frac{80}{100} = 0.8$.

5.2 Subject Programs and Infrastructure

As subjects for our studies we used six C programs—*flex* and *sed* [8], *space* [23, 25], and three of the Siemens test subjects (*print_tokens, replace*, and *schedule*) [14, 22] —as listed in Table 1. The table shows averages across versions for each subject for lines of code (LOC[3]), number of branches, number of databin variables used in our studies, and number of test cases available.

Each subject has a variety of versions with actual or injected faults and a binary-valued fault matrix labeling the behaviors of the versions when exercised with the available test cases. For these studies we considered only the behavior labels "pass" and "fail," as provided by the fault matrices. We selected a subset of the available versions that had a better than ten percent proportion of failing to passing test cases, thereby increasing the likelihood that random selection of test cases for a subject would include those that failed.

[3]Excludes comments, blanks, standalone "[," "]," "(," or ")"

5.2.1 Databin-profile Feature

For databins, we determined empirically that a bin count of 5 provided a better rate of classification than bin counts greater or less than 5, such as 3 or 7. Thus, for each modeled variable, we used a transition array of size 5x5. For a given subject in Stage 1 of our technique (Figure 1), we executed an instrumented version of the subject and collected the databin-profile data. We used Daikon's front end for C, Kvasir, that instruments and executes C programs using the DWARF-2 debugging format.[4] Kvasir produces trace files of the values assigned to variables, which we post-processed to reduce the number of variables as described in Section 3, and to calculate the state-transition profiles between pairs of the five databins for each variable. We used our tool ARGO to construct and serialize a set of data instances from the models for each execution, including the behavior label of "pass" or "fail." ARGO, written in C#, implements our technique for constructing classifiers and for conducting classification experiments on subject programs.

5.2.2 Branch-profile Feature

In our previous work, we found that the three control-flow features we studied—branch profiles, method-call profiles, and branch-to-branch profiles—yielded classifiers with similar classification rates. However, classifiers built from the branch-profile feature produced the highest classification rate on average, and were the least expensive in terms of both modeling and collecting data. Thus, to answer **Q2**, we chose the branch profile as the representative control-flow feature based on its performance for these subjects.

To collect data for the branch-profile feature of a given subject, we again followed Stage 1 of our technique (Figure 1), and executed an instrumented version of the subject and collected the branch-profile data. We used the ARISTOTLE analysis system[5] to instrument each subject to profile branches. We used our tool ARGO to construct and serialize a set of data instances from the models for each execution, including the behavior label of "pass" or "fail."

5.3 Empirical Method

To facilitate the use of databin-profile and branch-profile features together in an ensemble, we used the same set of training instances and thus the same set of test instances at each iteration for both classifiers. A *training instance* and a *test instance* both refer to data collected from a single execution. A classifier trains on the former and evaluates the latter. Both training and test instances are drawn from a general pool of *data instances*. For each version of each subject, our empirical method proceeds as follows for fifteen repetitions:

1. **Partition the data instances.** From the set of data instances, select at random 250 instances to be the pool of training instances. The remaining data instances become the set of test instances.

[4]http://pag.csail.mit.edu/daikon

[5]http://www.cc.gatech.edu/aristotle/

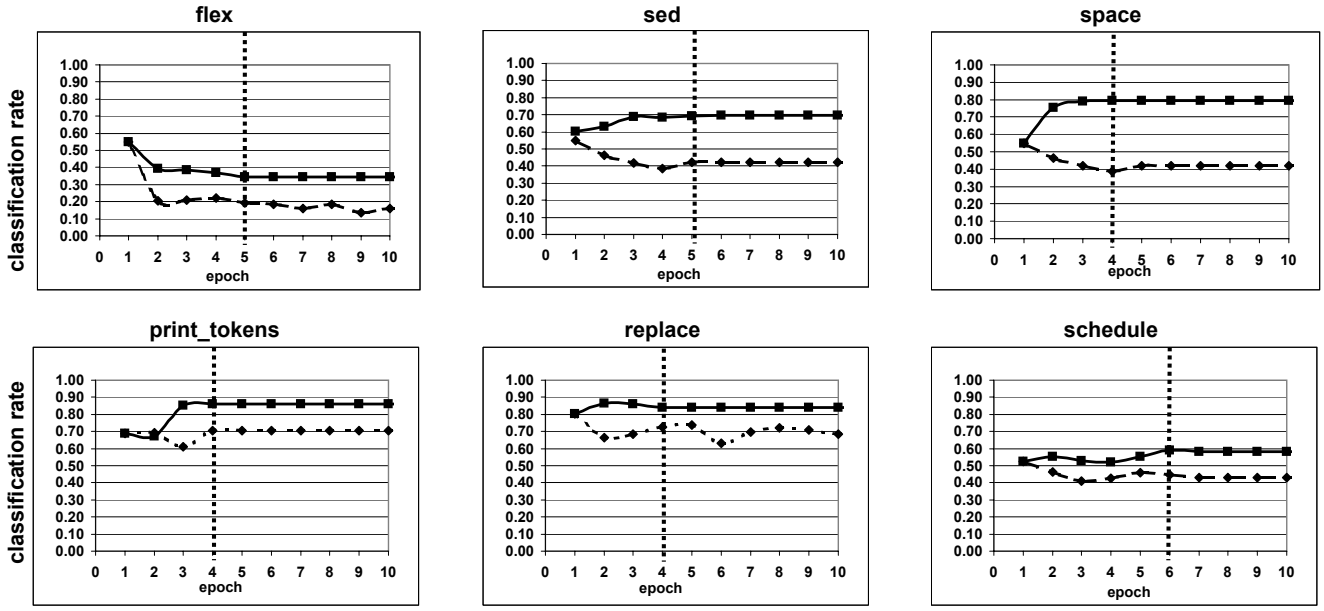

Figure 4: Active (solid line) vs. batch (dotted line) learning for databin-profile feature.

2. **Initialize batch and active learning.** Select 25 training instances with the label "pass" at random from the pool of training instances and construct two classifiers, $C_{Batch} = C_{Active}$. Classify the set of test instances and report the results.
3. **Batch Learning.** Repeat nine times: Select 25 training instances from the pool of training instances and incorporate them into C_{Batch}; Classify the set of test instances and report the results.
4. **Active Learning.** Repeat nine times: Select at most 25 training instances from the pool of training instances that the current *Active* classifier labels as unknown; Incorporate them into C_{Active}; Classify the set of test instances and report the results.

The evaluation of the ensemble classifiers did not involve additional classifier building. Rather, at each iteration of step 4 in the preceding protocol the ensemble classifier was formed from the branch-profile and databin-profile classifiers. Then the classification process was the voting mechanism described in Section 4.

5.4 Results

In this section, we present the results of our investigations into the two research questions.

5.4.1 Research Question 1

Question **Q1** explores the classification rates of databin-based classifiers under batch and active learning. The results are shown in Figure 4 as an individual graph for each of the six subjects. Within each graph, the horizontal axis shows the ten training *epochs*[6] corresponding to the increments of 25 training instances used for the batch learning. Within each graph, the vertical axis shows the classification rate, the dependent measure, which ranges from 0.0 to 1.0. The dotted vertical line on each graph denotes the epoch at which the active learner stopped finding new behaviors in the pool of training instances. Two lines (plots) are shown on each graph—the solid line for the classification rate of the active learner and the dashed line for the classification rate of the batch learner. The results show that across all subjects, the use of active learning improves the rate of classification over that of batch learning for the databin-profile based classifiers. Also note that the batch learners at epoch ten have incorporated all 250 training instances, whereas the active learners stopped acquiring new training instances in the vicinity of five epochs (see vertical dotted lines), or 125 training instances, depending on the subject.

For all subjects, the initial classifier, which trained only on the label "pass," has a classification rate above 0.5. However, for *flex, sed, space*, and *schedule*, the classification rate of the batch learning classifier falls below 0.5 (i.e. that which random choice could achieve) with succeeding epochs.

[6]An epoch represents an iteration of the training process.

5.4.2 Research Question 2

Question **Q2** compares the classification rates of branch-profile and databin-profile classifiers built using active learning and then explores the classification rates of ensemble classifiers composed of one of each. Each component classifier is built using active learning. The results are shown in Figure 5 as an individual graph for each of the six subjects. Within each graph, the horizontal axis shows the ten training epochs. Within each graph, the vertical axis shows the classification rate, the dependent measure, which for these results ranges between 0.3 and 1.0. Three lines (plots) are shown on each graph—the dashed line for the branch-profile classifier, the dotted line for the databin-profile classifier, and the solid line for the ensemble classifier.

The graphs show that in all cases, except for *flex*, both of the component classifiers begin at a classification rate above 0.5 and this corresponds to an error rate below 0.5, which is one of the necessary and sufficient properties for successful ensembles (see Section 4.) Given this property, it follows

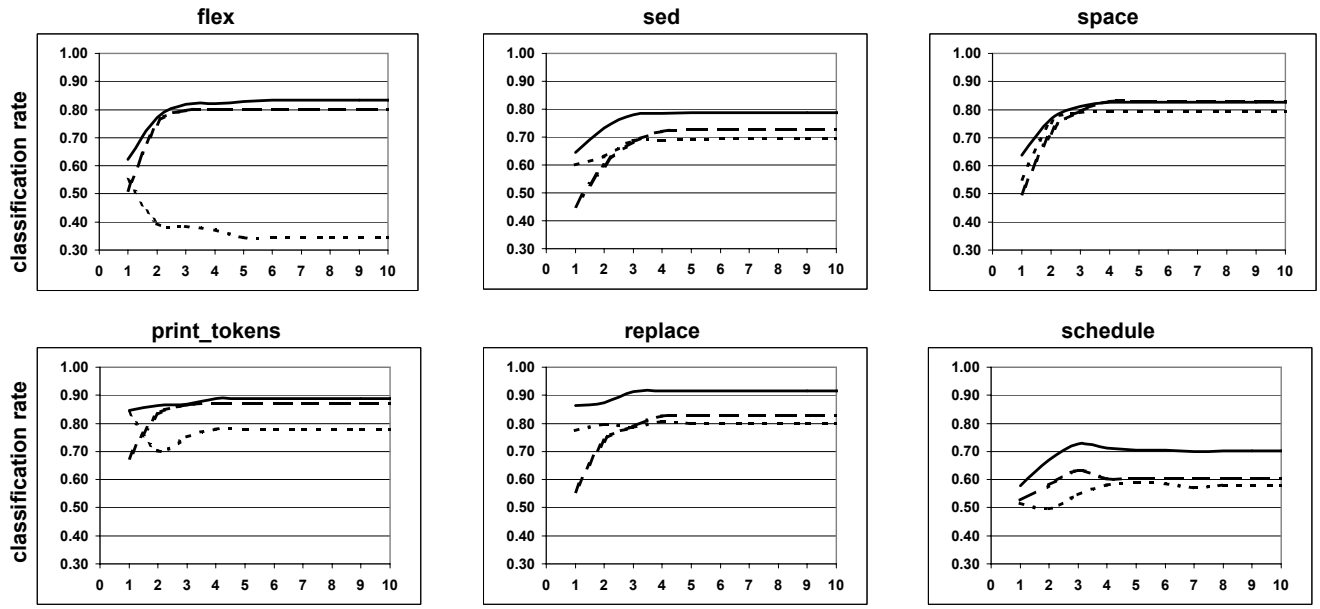

Figure 5: Ensemble classifiers (solid line) of branch-profile (dashed line) and databin-profile (dotted line) classifiers.

that any improvement in the ensemble over the two component classifiers represents the divergence of their respective classifications. In the case of *flex, space* and *print_tokens*, for instance, the contribution of databin profiles is negligible, as the ensemble does only marginally better than the branch-profile classifier. In the case of *sed, replace* and *schedule*, for instance, there is an approximate 10% improvement of the ensemble over either component classifier. The graphs also show that in all cases, branch profiles do better than databin profiles. In those cases where the ensemble does produce an improvement, we can infer that the databin classifier is divergent from the branch classifier. This means that the databin classifier contributes new information to the ensemble.

6. RELATED WORK

The previous work that is closest in spirit and method to our work is that of Podgurski and colleagues [7, 21]. Their work uses clustering techniques to build statistical models from program executions and applies them to the tasks of fault detection and failure categorization. The two primary differences between our technique and this previous work are the central role of Markov models in our approach and our use of active-learning techniques to improve the efficiency of behavior modeling. An additional difference is that we explore the utility of using individual control-flow and data-flow features instead of a large set of features.

Dickinson, Leon, and Podgurski demonstrate the advantage of automated clustering of execution profiles over random selection for finding failures [7]. They use the profiles of many different features as the basis for cluster formation. We concentrate on three features that summarize event transitions—branch profiles, method-call profiles and databin profiles. We show the utility of Markov models built from these feature profiles as predictors of program behavior. In Podgurski et al. [21], clustering is combined with feature selection, and multidimensional scaling is used to visualize the resulting grouping of executions. In both of these works, the clusters are formed just once using batch learning and then the clusters are used for subsequent analysis. In contrast, we explore an active learning technique that interleaves clustering with evaluation for greater efficiency.

Another group of related papers share our approach of using Markov models to describe the stochastic dynamic behavior of program executions. Whittaker and Poore use Markov chains to model software usage from specifications prior to implementation [26]. In contrast, we use Markov models to describe the statistical distribution of transitions measured from executing programs. Cook and Wolf confirm the power of Markov models as encoders of individual executions in their study of automated process discovery from execution traces [6]. They concentrate on transforming Markov models into finite state machines as models of process. In comparison, our technique uses Markov models to directly classify program behaviors. Jha, Tan, and Maxion use Markov models of event traces as the basis for intrusion detection [16]. They address the problem of scoring events that have not been encountered during training, whereas we focus on the role of clustering techniques in developing accurate classifiers.

The final category of related work uses a wide range of alternative statistical learning methods to analyze program executions. Although the models and methods in these works differ substantially from ours in detail, we share a common goal of developing useful characterizations of aggregate program behaviors. Harder, Mellen, and Ernst automatically classify software behavior using an operational differencing technique [13]. Their method extracts formal operational abstractions from statistical summaries of program executions and uses them to automate the augmentation of test suites. In comparison, our modeling of program behavior is based exclusively on the Markov statistics

of events. Brun and Ernst use dynamic invariant detection to extract program properties relevant to revealing faults and then apply batch learning techniques to rank and select these properties [5]. However, the properties they select are themselves formed from a large number of disparate features and the authors' focus is only on fault-localization and not on program behavior in general. In contrast, we seek to isolate the features critical to describing various behaviors. Additionally, we apply active learning to the construction of the classifiers in contrast to the batch learning used by the authors. Lin and Ernst propose batch learning of execution data to build mode controllers for multi-mode programs [17]. Again, the authors use a large feature space in contrast to our use of specific stochastic features as well as batch learning, where we use active learning.

Gross and colleagues propose the Software Dependability Framework, which monitors running programs, collects statistics, and, using multivariate state estimation, automatically builds models for use in predicting failures during execution [11]. This framework does not discriminate among features as we do. Also, we use Markov statistics of event-transitions instead of multivariate estimates to model program behaviors. Their models are built once using batch learning whereas we leverage active learning.

Munson and Elbaum posit that actual executions are the final source of reliability measures [19]. They model program executions as transitions between program modules, with an additional terminal state to represent failure. They focus on reliability estimation by modeling the probability of transitions to this failure state. We focus on behavior classification for programs that may not have a well-defined failure state.

7. DISCUSSION, CONCLUSIONS, AND FUTURE WORK

In this paper, we have extended our work on the automated modeling and classification of software behaviors and presented databin profiles, a new data-flow state-transition feature of program executions. We showed that databin profiles can be modeled as a stochastic process that exhibits the Markov property. We add databin profiles to our collection of features that can provide effective summaries of behaviors, which includes branch profiles and method-call profiles. One goal of our work is to discover and document the individual features of program executions that have the best predictive power in known code constructs.

In this paper, we also presented a set of empirical studies that indicate the efficiency and effectiveness of our approach for a set of subjects. These studies validate the usefulness of databin modeling and demonstrate that, as with our previous work with control-flow features, active learning is more efficient and effective than the conventional batch learning for databin profiles. These studies also that show that, for our subject programs, the branch-profile feature has up to 10% more predictive power than the databin profiles. However, the competitive performance of the databin classifier suggests that data-flow features may be as important as control-flow features in characterizing program behavior. Additionally, our empirical studies show that ensemble classifiers composed of two classifiers, one based on branch profiles and the other based on databin profiles, can yield a classification rate superior to or as good as either component classifier. This finding of improved classification rates for ensembles suggests that control-flow and data-flow features of program executions can capture divergent statistical views of behaviors for some subjects.

Although our empirical studies provide evidence of the potential usefulness of our approach, there are several threats to the validity of our results. Threats to the external validity of an experiment limit our ability to generalize from our results. The primary threats to external validity for these studies arise because we have considered only versions of six C programs. We cannot claim that these results generalize to other programs. In particular, we cannot generalize as to the effectiveness of either control-flow or data-flow features as predictors of behavior. Future work will include the evaluation of these features for other subjects.

Threats to the internal validity of an experiment occur when there are unknown causal relationships between the independent and dependent variables. In these studies, one threat to internal validity is the finite number of test cases for each subject. The collective properties of the test suites may affect our classification rates. A second threat is that we used only one clustering algorithm and that we set thresholds heuristically during the classification process. Threshold levels are key to our detection of unknown behaviors. A third threat arises from our definition of databins, where we constrained all variables to five databins. Nevertheless, the work suggests a number of research questions and additional areas for future work.

First, in this paper, the cost of the improvement in the ensemble classifier is the added cost of the databin-profile classifier, which must be compared to the cost advantage of the improvement in classification rate. These costs are subject- and task-specific and need to be evaluated on a case-by-case basis. Because databin profiles can be effective predictors, as shown in the graphs in Figure 4, we will explore additional techniques to reduce the costs of modeling and data collection for databins. In these studies, we collected traces of each variable during execution, and then processed the trace to define and profile the databin transitions. One way to reduce costs, for example, is to make informed estimates of the databins for a variable after a few initial executions, and then perform profiling on-line, without the need for recording a trace.

Second, the results for *flex* demonstrate that databin profiles are not always useful. This may be due to the particular design of databin profiles. Other data-flow features may have better predictive power in some cases. We will investigate other formulations of databins, including, for instance, tuning the number of databins per variable. We will extend our research to include transition profiles of conventional data-flows such as definition-use pairs.

Finally, we have shown that ensemble classifiers can combine branch-profile and databin-profile classifiers to advantage. We will investigate more closely the mapping between program structures and these features to learn whether specific code structures are better suited to a particular feature. We will investigate whether we can extend the ensemble construct to include features extracted in a more granular fashion from parts of the program. If so, we could leverage the advantages of both, while eliminating the costs of unnecessary data collection. Once the local features are specified, we could apply the techniques of Software Tomography [2] to distribute the data collection and monitoring tasks more efficiently.

Acknowledgements

This work was supported in part by National Science Foundation awards under CCR-0096321, CCR-0205422, and SBE-0123532 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission, and by the Office of Naval Research through a National Defense Science and Engineering Graduate (NDSEG) Fellowship. Gaurav Sharma assisted with implementing databins.

8. REFERENCES

[1] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 9–14, 2002.

[2] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 2–9, November 2002.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 195–205, 2004.

[4] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[5] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.

[6] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, pages 73–82, January 1999.

[7] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 339–348, May 2001.

[8] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 66–70, August 2004.

[9] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., New York, 2001.

[10] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32, 2002.

[11] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. Votta. Proactive system maintenance using software telemetry. In *Proceedings of the 1st International Conference on Remote Analysis and Measurement of Software Systems (RAMSS'03)*, pages 24–26, May 2003.

[12] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(10):993–1001, 1990.

[13] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25rd International Conference on Software Engineering (ICSE'03)*, pages 60–71, May 2003.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[15] IBM Research. Autonomic computing, 2004. `http://www.research.ibm.com/autonomic/`.

[16] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 206–219, June 2001.

[17] L. Lin and M. D. Ernst. Improving adaptability via program steering. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 206–216, Boston, MA, USA, July 12–14, 2004.

[18] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 256–261, 1989.

[19] J. C. Munson and S. Elbaum. Software reliability as a function of user execution patterns. In *Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences*, January 1999.

[20] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 65–69, July 2002.

[21] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25rd International Conference on Software Engineering (ICSE'03)*, pages 465–474, May 2003.

[22] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[23] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[24] R. E. Schapire. Theoretical views of boosting and applications. In *Algorithmic Learning Theory, 10th International Conference, ALT '99, Tokyo, Japan, December 1999, Proceedings*, volume 1720, pages 13–25. Springer, 1999.

[25] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53, July 1998.

[26] J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, January 1996.