Active Learning for Automatic Classification of Software Behavior

James F. Bowring
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
bowring@cc.gatech.edu

James M. Rehg
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
rehg@cc.gatech.edu

Mary Jean Harrold
College of Computing
Georgia Institute of
Technology
Atlanta, Georgia 30332-0280
harrold@cc.gatech.edu

ABSTRACT

A program's behavior is ultimately the collection of all its executions. This collection is diverse, unpredictable, and generally unbounded. Thus it is especially suited to statistical analysis and machine learning techniques. The primary focus of this paper is on the automatic classification of program behavior using execution data. Prior work on classifiers for software engineering adopts a classical *batch-learning* approach. In contrast, we explore an *active-learning* paradigm for behavior classification. In active learning, the classifier is trained incrementally on a series of labeled data elements. Secondly, we explore the thesis that certain features of program behavior are stochastic processes that exhibit the Markov property, and that the resultant Markov models of individual program executions can be automatically clustered into effective predictors of program behavior. We present a technique that models program executions as Markov models, and a clustering method for Markov models that aggregates multiple program executions into effective behavior classifiers. We evaluate an application of active learning to the efficient refinement of our classifiers by conducting three empirical studies that explore a scenario illustrating automated test plan augmentation.

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

Software engineers seek to understand software behavior at all stages of development. For example, during requirements analysis they may use formal behavior models, use-case analysis, or rapid prototyping to specify software behavior [3, 16]. After implementation, engineers aim to assess the reliability of software's behavior using testing and analysis. Each of these approaches evaluates behavior relative to a particular development phase, thereby limiting the range of behaviors considered. We believe that a program's behavior is richer and more complex than the characterization provided by these perspectives. Software behavior is ultimately the collection of all its executions, embodying the diversity of all users, inputs, and execution environments for all copies of the program. This collection is unpredictable and generally unbounded, and thus requires a data-driven approach to analysis. Our approach to understanding behavior is to build behavior models from data using statistical machine-learning techniques.

For any program, there exist numerous features for which we can collect aggregate statistical measures, such as branch profiles. If these features are accurate predictors of program behavior, then a broad range of statistical machine-learning techniques [17] could be used to aid analysis tasks such as automated-oracle testing, evaluation of test plans, detection of behavior profiles in deployed software, and reverse engineering.

The focus of this paper is on the automatic classification of program behavior using execution data. In this context, a classifier is a map from execution statistics such as branch profiles to a label for program behavior such as "pass" or "fail". Three issues must be addressed during classifier design: 1) a set of features, 2) a classifier architecture, and 3) a learning technique for training the classifier using labeled data [17]. For example, in recent work by Podgurski and colleagues [20], a classifier based on logistic regression uses composite features from execution profiles to identify failures that may share a common fault.

The application of machine learning techniques such as classification to software engineering problems is relatively new (some representative examples are [1, 4, 7, 14, 11, 15, 20]). All of this prior work adopts a classical *batch-learning* approach, in which a fixed quantity of manually-labeled training data is collected at the start of the learning process. In contrast, the focus of this paper is on an *active-learning* paradigm [5] for behavior classification. In active

learning, the classifier is trained incrementally on a series of labeled data elements. During each iteration of learning, the current classifier is applied to the pool of unlabeled data to predict those elements that would most significantly extend the range of behaviors that can be classified. These selected elements are then labeled and added to the training set for the next round of learning.

The potential advantage of active learning in the software-engineering context is the ability to make more effective use of the limited resources that are available for analyzing and labeling program execution data. If an evolving classifier can make useful predictions about which data items are likely to correspond to new behaviors, then those items can be selected preferentially. As a result, the scope of the model can be extended beyond what a batch method would yield, for the same amount of labeling effort. We have found empirically that active learning can yield classifiers of comparable quality to those produced by batch learning with a significant savings in data-labeling effort. In addition, we have found that the data items selected during active learning can be used to more efficiently extend the scope of test plans. These are important findings because human analysis of program outputs is inherently expensive, and many machine learning techniques depend for their success on copious amounts of labeled data.

A second contribution of this paper is an investigation of whether features derived from Markov models of program execution can adequately characterize a specific set of behaviors, such as those induced by a test plan. Specifically, we explore the thesis that the sequence of method calls and branches in an executing program is a stochastic process that exhibits the Markov property. The Markov property implies that predictions about future program states depend only upon the current state.

Method caller/callee transitions and branches together form the basic set of arcs in interprocedural control-flow graphs [1] (ICFGs). These features are part of the larger feature set used in References [7, 20]. Furthermore, branch profiles have been shown to be good detectors of the presence of faults by other researchers (e.g., [13, 23]). We specify Markov models of program behavior based on these features. Previous works (e.g., [7, 20]) have demonstrated the power of clustering techniques in developing aggregate descriptions of program executions. We present an automated clustering method for Markov models that aggregates multiple program executions. Each resulting cluster is itself a Markov model that is a statistical description of the collection of executions it represents. With this approach, we train classifiers to recognize specific behaviors emitted by an execution without knowledge of inputs or outcomes.

There are two main benefits of our work. First, we show that modeling certain event transitions as Markov processes produces effective predictors of behavior that can be automatically clustered into behavior classifiers. Secondly, we show that these classifiers can be efficiently evolved with the application of active-learning techniques.

---

[1] An *interprocedural control-flow graph* is a directed graph consisting of a control-flow graph for each method plus edges connecting methods to call sites and returns from calls. A *control-flow graph* is a directed graph in which nodes represent statements or basic blocks and edges represent the flow of control.

The contributions of this paper are:
- A technique that automatically clusters Markov models of program executions to build classifiers.
- An application of our technique that efficiently refines our classifiers using active learning (bootstrapping) and a demonstration of the application to an example of automated test plan augmentation.
- A set of empirical studies that demonstrate that the classifiers built by our technique from Markov models are both good predictors of program behavior and good detectors of unknown behaviors.

2. RELATED WORK

The previous work that is closest in spirit and method to our work is that of Podgurski and colleagues [7, 20]. 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 one or two 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 many feature profiles as the basis for cluster formation. We concentrate on two features that summarize event transitions—method caller/callee profiles and branch profiles. We show the utility of Markov models based on these profiles as predictors of program behavior. In Podgurski et al. [20], 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 once using batch learning and then 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 [24]. 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 [15]. 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
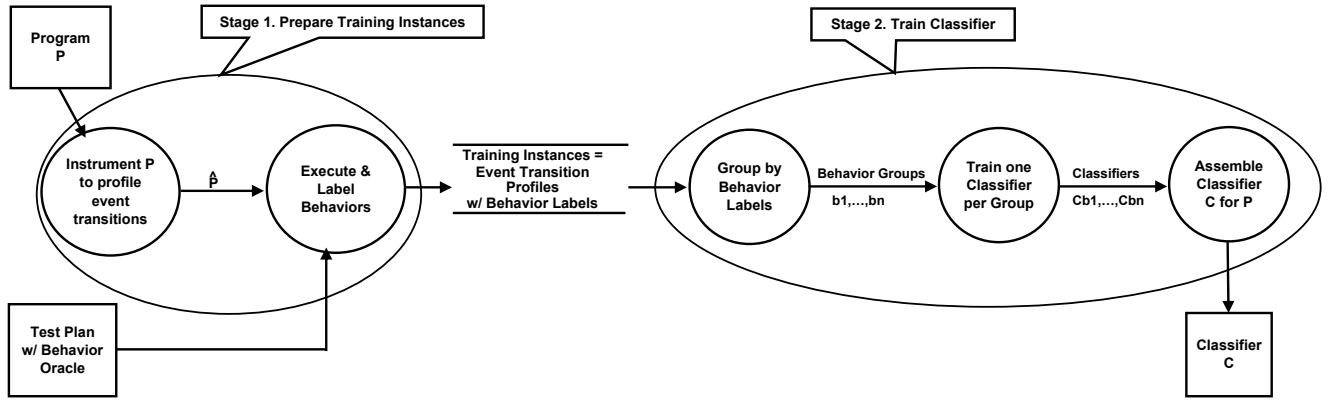
Figure 1: Building Classifier: Stage 1 - Prepare Training Instances; Stage 2 - Train Classifier.

differencing technique [12]. 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 [4]. 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.

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, where we do and consequently we use Markov statistics of events instead of multivariate estimates to model program behavior. Their models are built once use batch learning whereas we demonstrate the advantages of active learning.

Munson and Elbaum posit that actual executions are the final source of reliability measures [18]. 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 transition probabilities into the failure state. We focus on behavior classification for programs that may not have a well-defined failure state. Ammons, Bodik, and Larus describe specification mining, a technique for extracting formal specifications from interaction traces by learning probabilistic finite suffix automata models [1]. Their technique recognizes the stochastic nature of executions, but it focuses on extracting invariants of behavior rather than mappings from execution event statistics to behavior classes.

3. MODELING SOFTWARE BEHAVIOR

Our goal is to build simple models of program behavior that will reliably summarize and predict behavior. To do this, we focus on the subset of features that profile event transitions in program executions. An *event transition* is a transition from one program entity to another; types of *1st-order* event transitions include branches (source statement to sink statement), method calls (caller to callee), and definition-use pairs (definition to use); one type of *2nd-order* event transition is branch-to-branch. An *event-transition profile* is the frequency with which an event transition occurred during an execution.

We show that these event-transition features describe stochastic processes that exhibit the Markov property by building Markov models from them that effectively predict program behavior. The Markov property provides that the probability distribution of future states of a process depends only upon 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. The relative frequency of an event transition during a program execution provides a measure of its probability. For instance, a branch is an event transition in the control-flow graph[2] (CFG) of a program between a source node and a sink node. The source and sink nodes are states in the Markov model. 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 probabilities in the Markov model between this source node and the two sink nodes are the relative execution frequencies, or profiles, of the branches.

In general, the state variables could represent other events such as variable definitions and variable uses. The state variables could also represent more complex event sets such as paths of length two or more between events. Paths of length two are referred to as *2nd-order* event transitions. The profiling mechanism would collect frequency data for transitions between these events for use in the Markov models. An area of future research is to examine the trade-offs between the costs of collecting higher-order event-transition profiles and the benefits provided by their predictive abilities.

Our technique (Figure 1) builds a classifier for software behavior in two stages. Initially, we model individual program executions as Markov models built from the profiles of event transitions such as branches. Each of these mod-

---

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

els thus represents one instance of the program's behavior. The technique then uses an automatic clustering algorithm to build clusters of these Markov models, which then together form a classifier tuned to predict specific behavioral characteristics of the considered program.

Figure 1 shows a data-flow diagram of our technique. In the data-flow diagram, the boxes represent external entities, (i.e., inputs and outputs), and the circles represent processes. The arrows depict the flow of data and the parallel horizontal lines in the center are a database. Reading the diagram from left to right, the technique takes as inputs a subject program P, its test plan, and its behavior oracle, and outputs a Classifier C. P's *test plan* contains test cases that detail inputs and expected outcomes. The *behavior oracle* evaluates an execution e_k of P induced by test case t_k and outputs a behavior label b_k, such as, but not restricted to, "pass" or "fail."

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

In Stage 2, Train Classifier, the technique first groups the training instances 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. The technique initially uses a batch-learning paradigm to train one classifier C_{b_k} per behavior group b_k. Finally, the technique assembles the behavior group classifiers, $C_{b_1}, \ldots, C_{b_n}$ to assemble the classifier C for P. Before exploring the algorithm TRAINCLASSIFIER, shown in Figure 4, we discuss Markov model building.

3.1 Building Markov Models

Central to our technique is the use of Markov models to encode the event-transition profiles produced by $\hat{P}$. The transformation is a straightforward mathematical process, but it is important to understand the mapping from program events to the concept of state that we use in building the Markov models.

To illustrate, consider the CFG in Figure 2. The branch profiles for an arbitrary example execution e_1 are shown in the labels for each branch. For example, branch $B1$'s profile denotes that it was exercised nine times. The Markov model built from this execution is also shown in Figure 2 as a matrix. It models the program states identified by the source and sink nodes of each branch. The transitions are read from row to column. A Markov model built from branch profiles is simply the adjacency matrix of the CFG with each entry equal to the row-normalized profile. For instance, node $p1$ is the source of two transitions, branches $B1$ and $B2$, which occur a total of ten times.

As an example implementation of the transformation, we present algorithm BUILDMODEL, shown in Figure 3. BUILDMODEL constructs a matrix representation of a Markov model from event-transition profiles. Note that as the number of states specified in the Markov model increases, efficiency will dictate the use of a more sparse representation than that of conventional matrices.
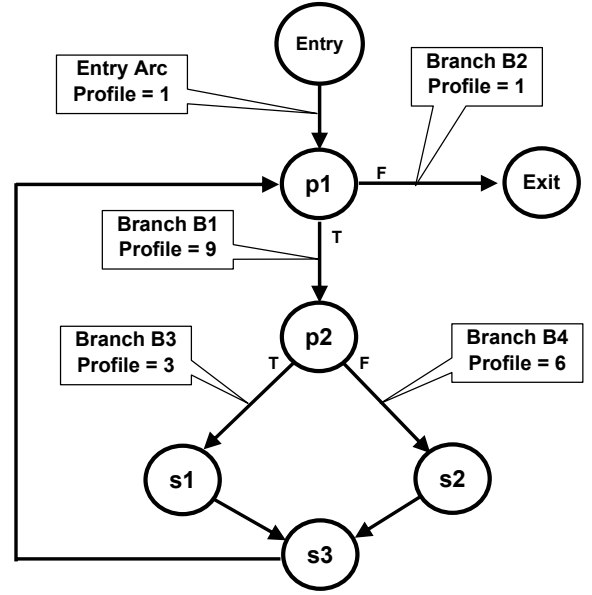
Markov Model of Branch Profiles

	Entry	p1	p2	s1	s2	Exit
Entry	0	1 / 1	0	0	0	0
p1	0	0	9 / 10	0	0	1 / 10
p2	0	0	0	3 / 9	6 / 9	0
s1	0	1 / 1	0	0	0	0
s2	0	1 / 1	0	0	0	0
Exit	0	0	0	0	0	1 / 1

Figure 2: Representing branch profiles as a Markov model.

BUILDMODEL has three inputs: S, D, b. S is a set of states or events used to specify the event transitions. D contains the event transitions and their profiles stored as ordered triples, each describing a transition from a state s_{from} to a state s_{to} with the corresponding profile: $(s_{from}, s_{to}, profile)$. b is the behavior label for the model. The output (M, D, b) is a triple of the model, the profile data, and the behavior label. In line 1, the matrix M for the model is initialized using the cardinality of S. In lines 2-3, each transition in D that involves states in S is recorded in M. In lines 4-8 each row in matrix M is normalized by dividing each element in the row by the sum of the elements in the row, unless the sum is zero.

For the execution e_1 shown in Figure 2, the inputs to BUILDMODEL for $1st$-order event transitions are:

- $S = \{Entry, p1, p2, s1, s2, exit\}$
- $D = ((Entry, p1, 1), (p1, Exit, 1), \ldots, (p2, s2, 6))$
- $b = $ "pass"

In this case, the output component M is the Markov model shown in Figure 2.

3.2 Training the Classifier

Our approach is to train a behavior classifier using as training instances the Markov models that we have constructed from the execution profiles. The training process we use is an adaptation of an established technique known

Algorithm BUILDMODEL(S, D, b)
Input: $S = \{s_0, s_1, \ldots, s_n\}$, a set of states,
         including a final or exit state,
         $D = ((s_{from}, s_{to}, profile), \ldots)$, a list of
         ordered triples for each transition and its profile
         $b =$ a string representing a behavior label
Output:(M, D, b), a Markov model, D and b

  (1) $M \leftarrow new\ float\ Array[|S|, |S|]$, initialized to 0
  (2)   **foreach** $\{s_{from},\ s_{to},\ profile\} \in D$, where $s \in S$
  (3) $M[s_{from},\ s_{to}] \leftarrow M[s_{from},\ s_{to}] + profile$
  (4)   **for** $i \leftarrow 0$ **to** $(|S| - 1)$
  (5) $rowSum \leftarrow \sum_{j=0}^{|S|-1} M[i, j]$
  (6)   **if** $rowSum > 0$
  (7)   **for** $j \leftarrow 0$ **to** $(|S| - 1)$
  (8) $M[i, j] \leftarrow M[i, j]/rowSum$
  (9) $return\ (M, D, b)$

Figure 3: Algorithm to build model.

as *agglomerative hierarchical clustering* [9]. With this technique, initially each training instance is 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, and the technique repeats. The stopping condition is either a desired number of clusters or some valuation of the quality of the remaining clusters.

The specification of the similarity function is typically done heuristically according to the application domain. In our case, we need to specify how two Markov models can be compared. In the empirical studies presented in this paper, we choose a very simple comparison, which is the Hamming distance[3] between the models. To compute the Hamming distance, each of the two Markov models is first mapped to a binary representation where a 1 is entered for all values above a certain threshold, and a 0 is entered otherwise. The threshold value is determined experimentally. We label the comparison function SIM and supply it as an input to our algorithm TRAINCLASSIFIER shown in Figure 4 and discussed below. The binary transformation of the models is done only temporarily by SIM in order to compute the Hamming distance.

Each merged cluster is also a Markov model. For example, in the first iteration, the merged model is built by combining the execution profiles that form the basis for the two models being merged. The profiles are combined by first forming their union, and then summing the values of redundant entries. Thus, if two executions were identical in their profiles, the result of combining the two profiles would be one profile with each entry containing a value twice that of the corresponding entry in the single execution. Referring to Figure 2, the merged profile for two copies of this execution would be:

- $D = ((Entry, p1, 2), (p1, Exit, 2), \ldots, (p2, s2, 12))$

Then from this merged profile, BUILDMODEL generates a Markov model that represents the new cluster.

For the stopping criterion, we have developed a simple heuristic that evaluates the stability of the standard deviation of the set of similarity measures that obtains at any given iteration. When the standard deviation begins to

---
[3]The *Hamming distance* between two binary numbers is the count of bit positions in which they differ.

Algorithm TRAINCLASSIFIER(S, T, SIM)
Input: $S = \{s_0, s_1, \ldots, s_n\}$, a set of states,
         including a final or exit state,
         $T = ((testcase_i, D_i, b_k), \ldots)$, a list of ordered triples ,
         where $D = ((s_{from}, s_{to}, profile), \ldots)$,
         and $b =$ a string representing a behavior label,
     SIM, a function to compute the similarity of
         two Markov models
Output: C, a set of Markov models, initially $\emptyset$

  (1)   **foreach** $(testcase_i, D_i, b_k) \in T$,
  (2) $0 < i <= |D|, 0 < k <= \#\ behaviors$
  (3) $C_{b_k} \leftarrow \emptyset$, initialize Classifier for behavior k
  (4)   **foreach** $C_{b_k}, 0 < k <= \#\ behaviors$
  (5)   **foreach** $(testcase_i, D_i, b_k) \in T$
  (6) $C_{b_k} \leftarrow C_{b_k} \cup BuildModel(S, D_i, b_k)$
  (7) $Deltas \leftarrow \emptyset$, an empty set to collect pair-wise deltas
  (8) $Stats \leftarrow new\ Array[|C_{b_k}|]$, cluster statistics
  (9)   **while** $|C_{b_k}| > 2$
  (10)   //agglomerative hierarchical clustering
  (11)   **foreach** $(M_i, D_i, b_k) \in C_{b_k}, 0 < i < |C_{b_k}|$
  (12)   **foreach** $(M_j, D_j, b_k) \in C_{b_k}, i < j <= |C_{b_k}|$
  (13) $Deltas \leftarrow Deltas \cup SIM(M_i, M_j)$
  (14) $Stats[|C_{b_k}|] \leftarrow StandardDeviation(Deltas)$
  (15)   **if** KNEE$(Stats)$ **then break**
  (16)   **else**
  (17) $(M_x, M_y) \leftarrow Min(Deltas)$
  (18) $D_{merged} \leftarrow D_x \cup D_y$
  (19) $M_{merged} \leftarrow$ BUILDMODEL(S, D_{merged}, b_k)
  (20) $C_{b_k} \leftarrow (C_{b_k} - M_x - M_y) \cup M_{merged}$
  (21) $C \leftarrow C \cup C_{b_k}$, add behavior k's models to C
  (22) $return\ C$

Figure 4: Algorithm to train classifier.

change markedly from the pattern established by the previous iterations of the clustering technique, the clustering is stopped. In the event that the standard deviation does not change markedly, we stop with two clusters. An area of future research is to evaluate how each formed cluster might map to specific sub-behaviors of the behavior under consideration and this research in turn may guide the refinement of the stopping criterion.

Our algorithm leverages our a priori knowledge of the specific behaviors under study in order to direct the classifier's learning process. As described above in Stage 2, Train Classifier, the training instances are segregated by their behavior labels. This allows us to first use agglomerative hierarchical clustering for the training instances representing each labeled behavior. Then we form the final classifier as the union of the constituent clusters from each of these behavior-specific classifiers. For example, the classifier learned for behavior "pass" might be composed of three clusters. Since the classifier for "pass" trained only on instances with behavior "pass," these three clusters contain no examples of behavior "fail." If, on the other hand, the training were done on all training instances of both behaviors, the clusters might contain examples of both behaviors. In this way, we direct the classifier to learn to discriminate between behaviors for which we initially possess the labels.

Algorithm TRAINCLASSIFIER, shown in Figure 4, trains a classifier from models generated by BUILDMODEL. TRAINCLASSIFIER has three inputs: S, T, SIM. S is a set of states that are used to identify the event transitions when BUILDMODEL is called. T is a list of triples, each containing a test-case index, a data structure D as defined in BUILD-

MODEL, and a behavior label b_k. SIM takes two Markov models as arguments and returns a real number that is the computed difference between the models.

In lines 1-3, an empty classifier C_{b_k} is initialized for each discrete behavior b_k found in T. Line 4 begins the processing for each b_k. In lines 5-6, the classifier C_{b_k} is populated with models built by applying BUILDMODEL to each training instance exhibiting b_k. Lines 7-8 initialize $Deltas$ and $Stats$, which are explained below.

The remainder of the algorithm clusters the models in each C_{b_k} to reduce their population and to merge similar and redundant models, using SIM, as described above. Line 9 establishes the default stopping criterion as two clusters. In lines 11-14, SIM is used to calculate the pair-wise differences and accumulate them in $Deltas$ at line 13. At each iteration, the algorithm calculates the standard deviation for the values in $Deltas$ and stores it in $Stats[|C_{b_k}|]$ at line 14. Because the cardinality of C_{b_k} decreases by one per iteration, it serves as an index into $Stats[]$.

The stopping criterion based on the quality of the standard deviations is implemented by the function KNEE. KNEE checks the set of standard deviations accumulating in $Stats[]$ at line 15 to determine the rate of change in the slope of a line fit to the values in $Stats[]$. KNEE is detecting a "knee-shape," or bend, in the graph of the standard deviations in $Stats[]$. This knee detection could be done by hand using a graph plotting program, for instance, but KNEE provides an automatic detector for use in our empirical evaluations. KNEE detects a "knee" when the sum of standard error (SSE) in a linear regression of the data points in $Stats[]$ increases by a factor of ten or more from its value in the previous iteration.

If a knee is detected, the clustering stops for that behavior group and the models in C_{b_k} are added to C, the final classifier. In the absence of a knee, the process stops with one model, per the constraint in line 9. Otherwise, the two closest models M_1 and M_2 are merged in lines 17-20, by calling BUILDMODEL with the union of the corresponding profile sets D_1 and D_2. Note that the clusters are formed into new models, each of which contains the profiles of all the training instances contributing to the cluster.

At line 21, the clustered models in C_{b_k} are added to C, the classifier. After all the behavior groups have been processed, the final C is returned. As discussed previously, this classifier is composed of several Markov models, each representing a cluster of similar executions.

3.3 Using the Classifier

Now we can use the classifier C to label executions of $\hat{P}$ that were not in the training set. We want C to classify the behavior of a given execution e_k of $\hat{P}$ and report its particular behavior label. To do so, each of the constituent models of C rates e_k with a probability score. The model in C with the highest probability score for e_k provides the behavior label for e_k.

The *probability score* is the probability that the model M could produce the sequence of event- or state- transitions in the execution e_k. As an example, refer to Figure 2, and consider another execution e_2 of the program with the following execution trace of branches, including the entry arc: $\{Entry\ Arc, B_1, B_3, B_1, B_3, B_2\}$. To calculate the probability score that the Markov model M in Figure 2 produced e_2, we compute the product of the successive probabilities of the transitions in M: $P = P(Entry\ Arc) * P(B_1) * P(B_3) * P(B_1) * P(B_3) * P(B_2)$. Thus $P = 1.0 * 0.9 * 0.333 * 0.9 * 0.333 * 0.1 = 0.008982$. The profile of each of branches B_1 and B_3 in our trace is two. The probability can be directly calculated using the profiles as exponents: $P = 1.0^1 * 0.9^2 * 0.333^2 * 0.1^1$. In general, we qualify this calculation by specifying that the model report that an execution has an *unknown* behavior whenever the calculated probability is below some threshold.

Note that probabilities calculated by multiplication can become very small. To overcome this difficulty, 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 relative ordering of the results.

3.4 Bootstrapping the Classifier

There are a number of learning strategies for training classifiers in addition to the batch-learning technique (used by other researchers and initially in TRAINCLASSIFIER) [9]. We concentrate on a type of active learning [5] called bootstrapping. Our application of *bootstrapping* first uses the classifier to score new executions and to then collect only those executions that remain unknown. These unknown executions are considered candidates that represent new behaviors and therefore each is evaluated, given a behavior label, and identified as a new training instance for the classifier. The classifier is retrained using the expanded set of training instances.

Active learning techniques employ an interactive or query-based approach that can be used to control the costs of training classifiers. Thus these techniques are especially well-suited to environments where the scope of the data is not yet fully known, as is generally the case with software behaviors. While an initial set of behaviors may be known from the software testing process, for instance, exhaustive testing is usually not practical, if even possible. As a consequence, software engineers do not have advance knowledge of specific behaviors that have yet to be observed. When these new behaviors do occur, an active learning environment provides for qualitative feedback from an oracle, in this case the test evaluator, that will allow for refinement of the classifier to include the new behavior. However, new instances of behavior that are recognized by the classifier do not need to be considered, and this is the source of the cost savings. Thus, active learning techniques provide a way to design classifiers that are better suited for the ongoing detection of behaviors than those classifiers built once from an initial batch of data. The inputs from the oracle alter the variance and bias in the sample from that which a batch learner would encounter.

We describe our application of bootstrapping to classifier training in the scenario presented in the next section. We then use this scenario to inform our empirical studies in the subsequent section.

4. SCENARIO

This scenario serves to illustrate how a developer Dev can combine our technique for classifying behaviors with an active learning strategy to reduce the cost of extending the scope of an existing test plan.

Dev has designed and implemented a version of a program P. Dev has developed a test plan to use with P and plans to
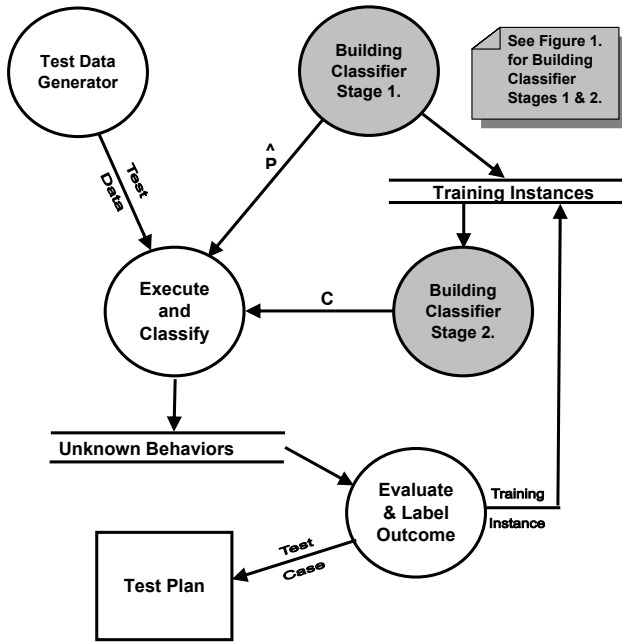
Figure 5: Automating test case selection.

expand it to test future releases of P. Dev is also interested in measuring the quality of the test plan in terms of its coverage of P's potential behaviors. Test plan development and testing are expensive and developers often release software that has been tested and accepted only for some core functionality [19, 21]. By using a measure of test plan quality, Dev can estimate some of the risks involved in releasing P with the current test plan.

The goal of creating new test cases for a test plan is to test additional behaviors. The design of a test case involves selecting test data that will induce new behavior and then evaluating the outcome of executing P with the test data. The measure of quality of the test plan is expressed indirectly by its rate of growth as new behaviors are discovered. In this sense, Dev will find it difficult to add test cases for new behaviors to a high-quality test plan. In general, Dev wants to augment the test plan for P with new test cases but seeks a way to reduce the cost of doing so. Dev may even have an automated test data generator for P, but still relies on test engineers to evaluate each execution.

We provide a solution to Dev that combines our technique for building classifiers with an application of bootstrapping that dynamically refines the classifiers. Figure 5 is a dataflow diagram of our application. In the diagram, Stage 1 of our technique from Figure 1 produces the instrumented $\hat{P}$ and the set of training instances from the initial test plan. Then Stage 2 of our technique from Figure 1 produces the classifier C.

Our application then executes $\hat{P}$ with test data generated on behalf of Dev and classifies each execution using C. Then, for the bootstrapping process, our application selects as candidates from the processed set of executions only those executions that remain unknown to the classifier. These candidates are presented to Dev for evaluation and subsequently given a behavior label. The candidates are then stored as new training instances in the database used

for training C. The corresponding test case for each candidate is added to the test plan. Our application retrains and refines C at certain intervals using the augmented database of training instances. The iteration of the bootstrapping process stops when the rate of detection of unknown executions falls below some threshold set by Dev.

Our application provides at least two economic benefits to Dev. First, our application builds a classifier for less cost than one built using batch learning. The simplest form of batch learning in this testing context is the conventional method of accumulating test cases, each of which must be evaluated by hand. The training of a classifier using batch learning improves the behavior detection rate over a simple look-up, but does not reduce the cost of evaluating new test cases. Each new test case must be evaluated regardless of whether it induces a new behavior or not. On the other hand, active learning combined with a robust classifier does provide a cost savings. This is because with our application only those new test cases that induce unknown behaviors are evaluated and added to the test plan. The test plan therefore grows only with test cases that extend its coverage of the program's behaviors. Secondly, Dev can use the rate at which unknown executions are produced as a relative measure of quality describing the test plan. When revisions are made to the program P, Dev has a target value for the rate from the previous version. By comparing rates, Dev can estimate the risks associated with the current test plan and make an informed business decision based on the estimate. We explore these economic benefits in our empirical Study 3 (Section 5.5).

5. EMPIRICAL STUDIES

To validate our technique and to explore its use in the described scenario, we performed three empirical studies.

5.1 Infrastructure

The subject for our studies is a C program, SPACE, that is an interpreter for an antenna array definition language written for the European Space Agency. SPACE consists of 136 procedures and 6,200 LOC, and it has 33 versions, each containing one fault discovered during development, and a set of 13,585 tests cases. In these studies, we chose as the event-transition features for our models both method caller/callee transitions and branches. While branches occur inside methods, method calls provide an accounting of intra-program transitions. We used the ARISTOTLE analysis system [2] to instrument each of fifteen randomly chosen versions of SPACE to profile these transitions, executed each version with all test cases, and stored the results in a relational database. We built a tool, ARGO, using C#, that implements our algorithms, technique, and application.

5.2 Empirical Method

Our empirical method evaluates the classifiers built by our technique and by our application of bootstrapping. For these studies, we chose the two behavior labels "pass" and "fail". The method has four steps using our database of profiles of branches and method-call event transitions for SPACE:

1. Select a version of SPACE
2. Select a set of test cases for training
3. Build a classifier.
4. Evaluate the classifier on the remaining test cases.

201

For these evaluations, we use the standard classification rate to measure the quality of the classifier. We also define a metric: Classifier Precision. *Classifier Precision* is the ratio of the number of unknown executions to the number of classifications attempted. Classifier Precision measures the ability of the classifier to recognize behaviors with which it is presented. The classification rate measures the quality of behavior detection. As an example, suppose C scores a total of 100 executions and correctly classifies 80 and incorrectly classifies 2, leaving 18 unknown. Then Classifier Precision= $\frac{18}{100} = 0.18$ and classification rate= $\frac{80}{100-18} = 0.976$. It is possible for a classifier to recognize all executions incorrectly, yielding maximum precision and a zero classification rate. Note that the classification rate improves as it approaches 1.0 while Classifier Precision improves as it approaches 0.0.

Our technique requires a definition for the similarity function SIM. SIM has two inputs: M_1 and M_2, the two models to be compared. We developed our definition of SIM for SPACE by trial and error. We discovered that agglomerative clustering using a *binary metric* [7], i.e. the Hamming distance, performed well. SIM manipulates the models but does not permanently alter them. SIM converts each transition probability to a binary value: 1 above some threshold, and 0 otherwise. We found the best empirical results by using the transition probabilities for the method calls only as an effective threshold for calculating the Hamming distance. SIM thus follows these steps:

1. Set all entries for branch transitions to 0, keeping only method transitions
2. Set all non-zero entries to 1
3. Calculate the binary matrix difference

$$SIM(M_1, M_2) = \sum_{i=1}^{k} \sum_{j=1}^{k} |M_{1_{ij}} - M_{2_{ij}}|$$

Once the final model clusters for the classifier are defined, then the models are reconstructed using all of the event-transition profile information to populate the transition probabilities. Again, we found by experience that these more complex representations were much better predictors of behavior than the models used for the clustering itself.

5.3 Study 1: Batch Learning

The goal of our first study was to evaluate our technique for training classifiers using only batch learning. For each of the 15 selected versions of SPACE, we repeated the following process 10 times:

1. Select random training sets of sizes 100, 150, 200, 250, 300, and 350.
2. Build a classifier from each training set.
3. Evaluate the classifiers.

Figure 6 summarizes the results for all classifiers evaluated. The graph's horizontal axis represents the size of the training set and the vertical axis represents Classifier Precision. In the graph, a box-plot summarizes the distribution of results for classifiers built using each training set size. The top and bottom of the box represent the third and first quartiles respectively, while the additional horizontal line in each box locates the median. For instance, at training set size 200, the median is 0.298, the first quartile is 0.278, and the third quartile is 0.332. The "whiskers" above and below the box mark the extent of $1.5 * IQR$, where IQR is the inter-quartile range (i.e., the vertical dimension of the box).
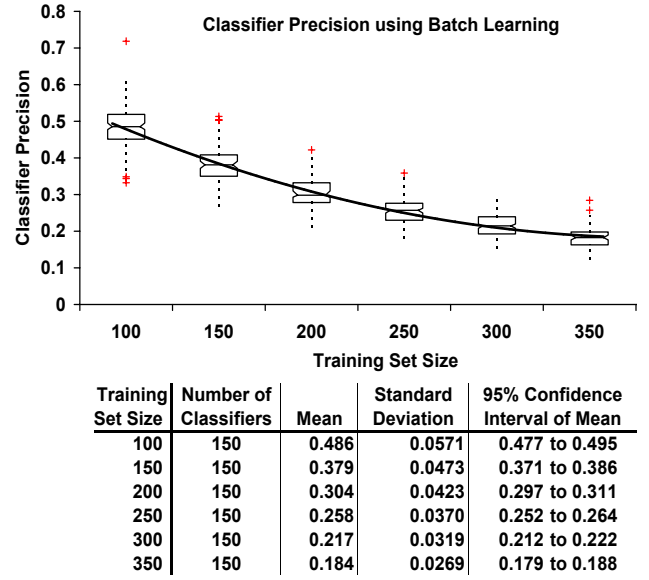

Training Set Size	Number of Classifiers	Mean	Standard Deviation	95% Confidence Interval of Mean
100	150	0.486	0.0571	0.477 to 0.495
150	150	0.379	0.0473	0.371 to 0.386
200	150	0.304	0.0423	0.297 to 0.311
250	150	0.258	0.0370	0.252 to 0.264
300	150	0.217	0.0319	0.212 to 0.222
350	150	0.184	0.0269	0.179 to 0.188

Figure 6: Study 1: Classifier Precision for batch learning.

Training Set Size	Number of Classifiers	Mean	Standard Deviation	95% Confidence Interval of Mean
100	150	0.975	0.0448	0.968 to 0.983
150	150	0.977	0.0406	0.971 to 0.984
200	150	0.977	0.0353	0.971 to 0.983
250	150	0.979	0.0326	0.973 to 0.984
300	150	0.978	0.0342	0.973 to 0.984
350	150	0.977	0.0319	0.972 to 0.982

Figure 7: Study 1: Classification rate.

The individual points above and below the whiskers (shown using a "+") are the outliers. The trend-line fit through the medians shown in the graph is quadratic with a coefficient of determination, $R^2 = 0.984$.

The table in Figure 6 summarizes the parametric statistics. For each training set size listed in the left column, the table shows the number of classifiers, the mean, standard deviation, and the 95% confidence interval of the mean. As the size of the training set increases, Classifier Precision improves, until at a size of 350, the mean is 0.184. This mean represents 2435 unknown test cases (i.e., $0.184 * (13585 - 350)$). Likewise, the standard deviation at 350 represents 356 test cases (i.e., $0.0269 * (13585 - 350)$).

The quadratic trend-line and the decreasing variation in the distribution of Classifier Precision values with increasing training set size suggest that the rate of improvement will continue to decrease. This result is a property of both SPACE and the distribution of behaviors in its test suite. The results also indicate that the classifier model is able to learn and continuously improve with these data, albeit at a slowing rate.

The table in Figure 7 summarizes the parametric statistics for the classification rate. The means are uniformly high. It is possible that larger training set sizes could reveal a trend for the classification rate, but given its current high value, there is not much room for improvement. The mean value for training set size 350 is 0.977, which represents 304 misclassified behaviors (i.e., $(1 - 0.977) * (13585 - 350)$).
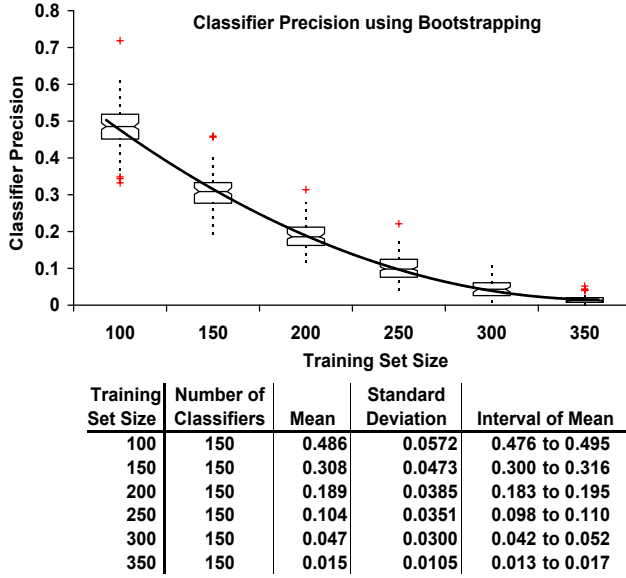
Classifier Precision using Bootstrapping

Training Set Size	Number of Classifiers	Mean	Standard Deviation	Interval of Mean
100	150	0.486	0.0572	0.476 to 0.495
150	150	0.308	0.0473	0.300 to 0.316
200	150	0.189	0.0385	0.183 to 0.195
250	150	0.104	0.0351	0.098 to 0.110
300	150	0.047	0.0300	0.042 to 0.052
350	150	0.015	0.0105	0.013 to 0.017

Figure 8: Study 2: Classifier Precision for bootstrapping.

Training Set Size	Number of Classifiers	Mean	Standard Deviation	95% Confidence Interval of Mean
100	150	0.976	0.0449	0.968 to 0.983
150	150	0.974	0.0435	0.967 to 0.981
200	150	0.975	0.0416	0.968 to 0.982
250	150	0.976	0.0399	0.970 to 0.983
300	150	0.977	0.0353	0.971 to 0.983
350	150	0.976	0.0321	0.971 to 0.982

Figure 9: Study 2: Classification rate.

5.4 Study 2: Bootstrapping the Classifiers

The goal of our second study was to evaluate our application of bootstrapping to refining the classifiers built by our technique. In order to compare our results with those of Study 1, we chose an approach to bootstrapping that gives the classifier new executions until the number of unknown executions reaches a threshold. In this study, we set the threshold at 50. We then labeled these 50 executions using a database lookup, and added them to the training set for the classifier. Finally, we retrained the classifier using the augmented set of training instances, and repeated the search for unknown executions. Thus, beginning with a training set size of 100, the increments of 50 parallel those in Study 1. In each instance of this study, the exact initial training set of size 100 used in Study 1 is used again.

Figure 8 summarizes the results for all the classifiers evaluated with a graph and table similar to those in Figure 6. Here, the variation in the distribution of Classifier Precision values also decreases with increasing training set size. The Classifier Precision approaches 0 at training set size 350, where the mean of 0.015 represents 198 unknown executions (i.e., $0.015*(13585-350)$). Here, the quadratic trend-line ($R^2 = 0.989$) fit to the medians in the graph is asymptotic to a Classifier Precision of 0. This result is a property of both SPACE and the distribution of behaviors in its test suite. The results also indicate, as in Study 1, that the classifier model is able to learn and continuously improve with these data, albeit at a slowing rate.

The table in Figure 9 summarizes the parametric statistics for the classification rate. As in Study 1, the means are uniformly high. It is possible that larger training set sizes could reveal a trend for the classification rate, but given its current high value, there is not much room for improvement. The mean value for training set size 350 is 0.976, which represents 317 wrongly classified behaviors (i.e., $(1-0.976)*(13585-350)$). This mean value is slightly less than that in Study 1, but a comparison is not appropriate, again because of the small size of the training set.

5.5 Study 3: Batching vs. Bootstrapping

The goal of our third study was to compare the rates of growth in Classifier Precision between batch learning and bootstrapping. It is this comparison that motivates our presented scenario. The comparison of the results of Study 1 and Study 2 are shown in Figure 10. The dotted curve shows the means from Study 1 and the solid curve shows the means from Study 2. The two sets of classifiers were initialized with the same training set at size 100.

As the training set size increases, so does the gain in Classifier Precision of bootstrapping over batch learning. As an example, consider size 200, where the difference is 1525 executions (i.e., $(0.304-0.189)*(13585-200)$). Imagine that developer Dev in our scenario, using SPACE as P, built an initial classifier C with the 100 test cases in P's test plan. Now for an additional investment in evaluations of 100 more executions, giving a training set size of 200, the classifier in Study 1 yields, on average, 4069 unknown executions (i.e., $0.304*(13585-200)$). The corresponding classifier from Study 2 yields, on average, 2529 unknown executions (i.e., $0.189*(13585-200)$). The benefit of the application of bootstrapping over batch learning is 1540 classified executions or 38% for the same investment in evaluating 100 executions. From the graph, it is clear that the rate of improvement continues to increase through training set size 350.

The second economic benefit for Dev is an estimation of the risks involved in releasing P with its current test plan. By simply using our technique to build a classifier from the test plan and then measuring the rate of unknown executions it produces from additional test data, Dev can rate the quality of the test plan. For instance, if after a fixed time, no new test data produce unknown executions, then Dev has confidence in the test plan. On the other hand, a high rate of detection of unknown behaviors, expressed as Classifier Precision, signals a risk to deployment.
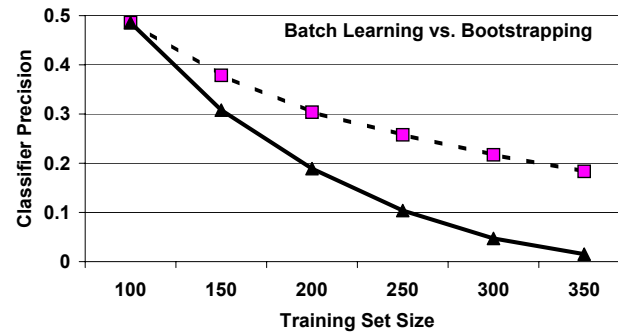

Figure 10: Study 3: Comparison of learning techniques.

6. DISCUSSION, CONCLUSIONS, AND FUTURE WORK

We have presented our technique for the automated modeling and classification of software behaviors based on the equivalence of Markov models to event-transition profiles in program executions. We have presented an application of our technique that efficiently refines our classifiers using bootstrapping, and we illustrated with a scenario how our application could reduce the costs and help to quantify the risks of software testing, development, and deployment.

We performed three empirical studies that validate both the technique and the application as well as support the presented scenario. However, there are several threats to the validity of our results. These threats arise because we used only fifteen versions of one medium-sized program and its finite set of $13,585$ test cases. However, SPACE is a commercial program and the versions contain actual faults found during development. Furthermore, the specific structure of SPACE may be uniquely suited to our technique.

Nevertheless, our empirical studies of fifteen versions of SPACE demonstrated that the application of our technique is effective for building and training behavior classifiers for SPACE. The work suggests a number of research questions and additional applications for future work.

First, we have systematically explored one feature set here and we will investigate others from the class of features measuring event transitions. This work prepares us to investigate other individual features that may exhibit the Markov property, such as variable definition and use pairs. We will also investigate the predictive power of higher-order event transitions. For each case, we will evaluate the costs and trade-offs involved in leveraging the explored feature for modeling behaviors. As additional features are explored, we will also examine how combinations of features affect the predictive power of the generated models.

Second, we discovered that agglomerative hierarchical clustering was an effective technique for building our classifiers. We will investigate additional clustering algorithms and similarity metrics. We will evaluate how each formed cluster might map to specific sub-behaviors as a guide to refining the stopping criteria during clustering.

Third, whereas our empirical studies demonstrate the effectiveness of the behavior labels "pass" and "fail," we saw that the classifiers for each of these behaviors were composed of several models. This suggests that we may be able to automatically identify more fine-grained behaviors, including sub-behaviors inside individual modules. We are interested in the relationship between the Markov models for specific behaviors and their representation in the requirements and specifications for a program. If there is a demonstrable relation, it may lead to techniques for evaluating the quality of a test plan or to tools to aid reverse engineering.

Fourth, our empirical studies show that for our subject, bootstrapping improves the rate at which the classifier learns behaviors. We will investigate other machine learning techniques and their possible application to the training of behavior classifiers. Of particular interest is determining the best set of event transitions or, as Podgurksi and colleagues suggest, the best set of features [20] with which to train classifiers. We will also examine how these techniques might scale to large programs.

Fifth, our empirical studies show the effectiveness of our application for classifying the behavior of our subject. We need to determine how the application will perform for other programs. We will formulate and explore additional applications of our techniques, such as the detection of behavioral profiles in deployed software, anomaly and intrusion detection, and testing non-testable programs. We will also explore ways to provide for programs to be self-aware of their behaviors by having access to models of behavior.

Finally, we plan to explore the use of hidden Markov models (HMMs) [9] and simplicial mixtures of Markov models [10] to extend our behavior-modeling technique. In a *hidden Markov model*, the state variable in the Markov chain is assumed to be unobservable (i.e. cannot be measured directly). However, at each time instant the hidden state emits an observation variable which can be measured. The HMM specifies a statistical relationship between the hidden state and the observation, in addition to the usual transition probabilities for the hidden state. Once the parameters of the model have been learned from data, powerful inference techniques, such as the Viterbi algorithm [22], can be used to estimate the most likely sequence of hidden states given a sequence of observations. Learning in the HMM case uses the Expectation-Maximization technique [9], which is more complex than standard parameter estimation for a Markov chain. However, as a consequence of their widespread use in automatic speech recognition, there is an established body of practice for training HMMs which can be leveraged for our application.

HMMs are interesting from two perspectives. The first is that they can describe more complex probability distributions than simple Markov chains. When branch transitions are taken as the states in a Markov chain, then their statistical distribution follows standard first- or second-order dependencies. However, if these transitions are taken as the observations in an HMM with a first- or second-order hidden-state dependency, marginalization of the hidden state induces much more complex distributions over the branches. In particular, the branch distribution does not have to obey the Markov property. In this context, the hidden states could correspond to higher-level categories of transitions. The second potential benefit of HMMs is their ability to couple multiple aspects of program execution together in one statistical model. For example, advanced HMM models such as factorial HMMs, input-output HMMs, and conditional random fields [8] support more complex coupling between execution events and associated data such as program inputs and outcomes. These advanced tools may enable more powerful models of program behaviors.

Simplicial mixtures of Markov models are another recent generalization of basic Markov chains which have demonstrated an ability to capture common behavioral patterns in event streams. As data from program executions are also event streams, there is a real possibility of adding this technique to our family of statistical methods.

Acknowledgements

7. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 4–16, January 2002.

[2] Aristotle Research Group. ARISTOTLE: Software engineering tools, 2002. `http://www.cc.gatech.edu/aristotle/`.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Boston, 1998.

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering.*

[5] D. A. Cohn, L. Atlas, and R. E. Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.

[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] T. G. Dietterich. Machine learning for sequential data: A review. In T. Caelli, editor, *Structural, Syntactic, and Statistical Pattern Recognition*, volume 2396 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2002.

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

[10] M. Girolami and A. Kaban. Simplicial mixtures of markov chains: Distributed modelling of dynamic user profiles. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[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] 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.

[13] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verifications, and Reliability*, 10(3), September 2000.

[14] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *Software Engineering*, 21(3):181–199, 1995.

[15] 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.

[16] J. A. Kowal. *Behavior Models: Specifying User's Expectations*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[18] 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.

[19] J. Musa. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. McGraw-Hill, New York, 1999.

[20] 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.

[21] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, Reading, Mass., 1999.

[22] L. Rabiner and B. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, New Jersey, 1993.

[23] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM Software Engineering Notes*, 22(6):432–439, November 1997.

[24] 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.