Software Behavior: Automatic Classification and its Applications

James F. Bowring James M. Rehg Mary Jean Harrold

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{bowring, rehg, 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. We explore the thesis that $1st$- and $2nd$-order Markov models of event-transitions are 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, yielding a statistical description of program behaviors. With this approach, we can train classifiers to recognize specific behaviors emitted by an execution without knowledge of inputs or outcomes. 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. We present a set of potential research questions and applications that our work suggests.

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, 15]. After implementation, engineers aim to assess the reliability of software's behavior using testing and dynamic analysis. A program's behavior is ultimately the collection of all its executions. This collection is diverse, unpredictable, and generally unbounded. Thus it is potentially suited to statistical analysis and machine learning techniques.

One challenge is to find productive applications of statistical analysis techniques to leverage this collection of executions for the solution of software engineering problems. A basic question is whether aggregate statistical measures of program execution, such as branch profiles, are accurate predictors of program behavior. If they are, then a broad range of statistical machine learning techniques could be used in dynamic analysis tasks, including automated-oracle testing, the evaluation of test plans, the detection of behavior profiles in deployed software, and reverse engineering tasks.

Many researchers have explored these questions (e.g., [1, 4, 6, 7, 10, 11, 12, 13, 14, 16, 18, 20, 21]). For example, Dickinson, Leon, and Podgurski demonstrate the advantage of automated clustering of execution profiles over random selection for finding failures [7], and Podgurski and colleagues show the efficacy of automated classification of failure reports [18]. As another example, Harder, Mellen, and Ernst use their operational difference technique to automatically extract abstractions of software behavior from statistical summaries of program executions [11].

Other researchers have employed a stochastic approach to behavior classification that uses the sequences of events in program executions. For example, Munson and Elbaum posit that event-transitions in actual executions are the final source of reliability measures [16]. This research demonstrates the ability of stochastic models to summarize the dynamic aspects of program executions, but does not address their use in directly classifying behaviors.

We explore the thesis that $1st$- and $2nd$-order Markov models of event-transitions are effective predictors of program behavior. Branch profiles, for example, have been used extensively in dynamic analysis (e.g., [12, 20]) and are equivalent to $1st$-order Markov models. $2nd$-order Markov models can encode branch-to-branch transitions and have the potential for greater predictive power at a modest additional cost.

Previous works such as [7, 18] have demonstrated the power of clustering techniques in developing aggregate descriptions of program executions. In this paper, we present a clustering method for Markov models that aggregates multi-

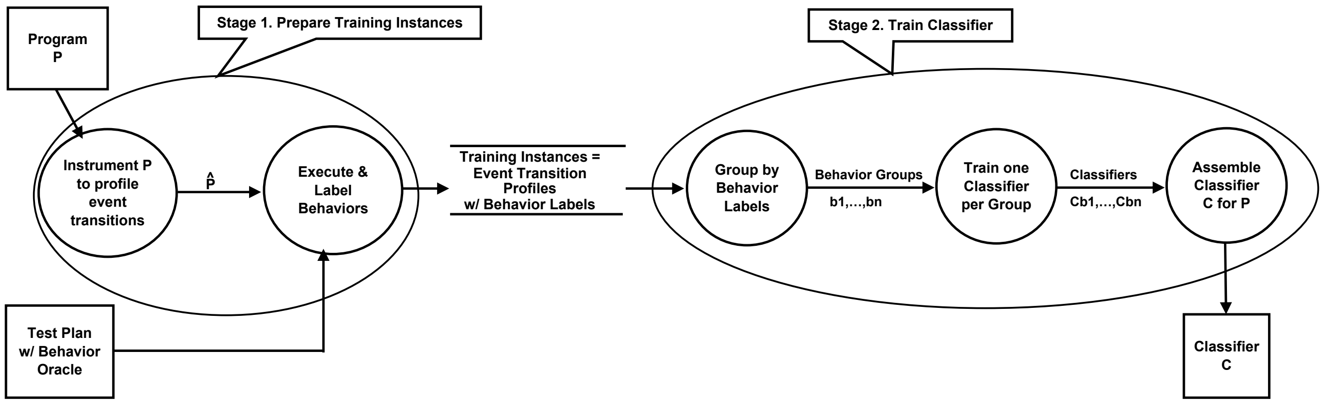

Program P
Stage 1. Prepare Training Instances
Instrument P to profile event transitions
$\hat{P}$
Execute & Label Behaviors
Test Plan w/ Behavior Oracle
Training Instances = Event Transition Profiles w/ Behavior Labels
Stage 2. Train Classifier
Group by Behavior Labels
Behavior Groups
b1,...,bn
Train one Classifier per Group
Classifiers
Cb1,...,Cbn
Assemble Classifier C for P
Classifier C

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

ple program executions, yielding a statistical description of program behaviors. With this approach, we can train classifiers to recognize specific behaviors emitted by an execution without knowledge of inputs or outcomes. In particular, these classifiers do not require a distinct failure state to detect failure.

A key question is whether these statistical models can characterize a set of behaviors, such as those induced by a test plan. If they can, then a useful application would be the automatic discovery of new behaviors that were not captured in the test plan. In our application, a classifier identifies program executions with unknown behaviors. These executions are then evaluated and labeled so that they can be used to refine the classifier. This process is known as bootstrapping, and is an example of a class of techniques known as active learning [5]. The ability to identify executions whose behavior is recognized allows us to then characterize a set of executions by evaluating only the subset containing unknown behaviors.

The contributions of this paper are:

- A technique that models program executions as Markov models and automatically clusters them to build classifiers.
- An application of our technique that efficiently refines our classifiers using active learning (bootstrapping) and a demonstration of its advantages with an example of automated test plan augmentation.
- A set of empirical studies that demonstrate that Markov models are good predictors of program behavior and that by clustering Markov models we can train classifiers to recognize unknown behaviors.

2. Software Behavior Classification

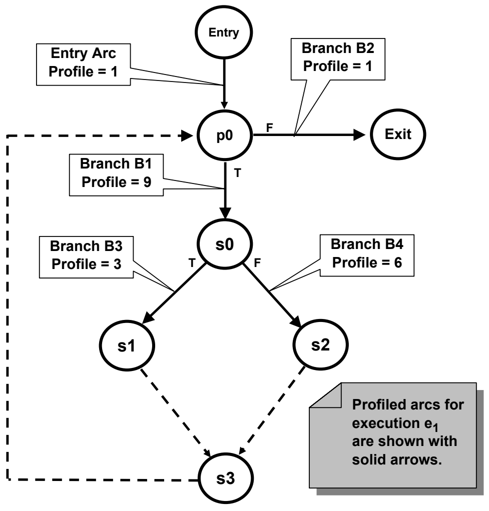
Our technique builds a software behavior classifier in two stages. We describe the basic flow of the technique and then we detail and illustrate the algorithms.

2.1. Overview

Figure 1 shows a data-flow diagram of our technique, which 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. 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. 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 profile 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 group to a $1st$-order discrete-time Markov model (hereafter referred to as a Markov model). Markov models can encode any order event-transitions, such as our $1st$ and $2nd$-order event-transitions. The technique uses a machine-learning paradigm, called passive training, to train one classifier C_{b_k} per behavior group b_k. (In *passive training*, the classifier trains on all instances with which it is presented.) Finally, the technique assembles the behavior group classifiers, $C_{b_1}, \ldots, C_{b_n}$ to assemble the classifier C for P. Be-



Markov Model of Branches
(1st-order event-transitions)

	Entry	p0	s0	s1	s2	Exit
Entry	0	1 / 1	0	0	0	0
p0	0	0	9 / 10	0	0	1 / 10
s0	0	0	0	3 / 9	6 / 9	0
s1	0	0	0	0	0	0
s2	0	0	0	0	0	0
Exit	0	0	0	0	0	1

Markov Model of Branch-to-Branch
(2nd-order event-transitions)

	Entry Arc	B1	B2	B3	B4	Exit
EntryArc	0	9 / 10	1 / 10	0	0	0
B1	0	0	0	3 / 9	6 / 9	0
B2	0	0	0	0	0	1 / 1
B3	0	3 / 3	0	0	0	0
B4	0	5 / 6	1 / 6	0	0	0
Exit	0	0	0	0	0	1

Figure 2. Markov models from profiles.

fore exploring the algorithm TRAINCLASSIFIER, shown in Figure 4, we discuss Markov model building.

2.2. Building Markov Models

Central to our technique is the use of Markov models to encode the event-transition profiles produced by $\hat{P}$. *Markov models* capture 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 event-transition profiles of an execution represent the probability of being in state or event s_1 at time t+1 given that the state or event at time t was s_0. As an example, consider a control-flow graph[1] (CFG). The arcs of a CFG are event-transitions (e.g., branches, loop backedges, or connections to join points). If a node has two branches leaving it and each has an associated profile (or frequency) from an execution e of $\hat{P}$, then the probability that each branch was taken in e is a function of their relative frequencies.

To illustrate, consider Figure 2 and its CFG. The branch profiles for an execution e_1 are shown in the labels for each branch. For example, branch $B1$'s profile denotes that it was exercised nine times. A Markov model built from branch profiles is simply the adjacency matrix of the CFG with the the profiles as entries and then normalized by row, as shown in the "Markov Model of Branches" in Figure 2. A Markov model of one possibility for the $2nd$-order branch-to-branch transitions is also shown in Figure 2. For example, in the "Markov Model of Branch-to-Branch", $B4$ is followed by $B1$ five times and by $B2$ one time for a total of six branch-to-branch transitions emanating from $B4$.

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

```
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) $M \leftarrow new\ Array[|S|, |S|]$
(2) foreach $\{s_{from}, s_{to}, profile\} \in D,\ each\ s \in S$
(3)     $M[i, j] \leftarrow M[i, j] + profile$
(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.

We present algorithm BUILDMODEL, shown in Figure 3 as an example implementation of this transformation. BUILDMODEL constructs a behavior model M as a matrix representation of a Markov model from execution profiles.

BUILDMODEL has three inputs: S, D, b. S is a set of states that identify 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 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 then:

- $S = \{Entry, p0, s0, s1, s2, exit\}$
- $D = ((Entry, p0, 1), (p0, Exit, 1), \ldots, (s0, s2, 6))$
- $b =$ "pass", for instance

The inputs to BUILDMODEL for $2nd$-order event-transitions are:

- $S = \{EntryArc, B1, B2, B3, B4, exit\}$
- $D = ((EntryArc, B1, 9), \ldots, (B4, B2, 1))$
- $b =$ "pass", for instance

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

```
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)    $Deltas \leftarrow \emptyset$, 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 [9]
  (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)       $C \leftarrow (C - M_x - M_y) \cup M_{merged}$
  (21)   $C \leftarrow C \cup C_{b_k}$, add behavior k's models to C
  (22) $return\ C$
```

Figure 4. Algorithm to train classifier.

2.3. Training the Classifier

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 BUILDMODEL, and a behavior label b_k. SIM is a function that can be tailored to the specific program and behavior set under study. SIM takes two Markov models as arguments and returns a real number that is the computed difference between the models. We define our SIM in our empirical study (Section 4.2).

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 $Stats$, 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. The approach is an adaptation of agglomerative hierarchical clustering [9]. Clustering proceeds as follows. The algorithm uses SIM to calculate the pair-wise difference for all pairs of models in C_{b_k} and selects the pair of models with the smallest difference. This pair of models is merged, reducing the cardinality of C_{b_k} by one. The process repeats with a stopping criterion.

In lines (11-14), SIM is used to calculate these pairwise 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 $Stats[]$. 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 them. This knee detection could be done by hand, but for the purposes of our empirical evaluations, we detect 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 between iterations.

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.

2.4. Using the Classifier

We can use the classifier C to label new executions of $\hat{P}$. To classify a new execution e_k of $\hat{P}$, each of the constituent models of C rates e_k with a probability score. The model with the highest probability score provides the behavior label for e_k.

The *probability score* is the probability that the model M produced 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: $P = 1.0^1 * 0.9^2 * 0.333^2 * 0.1^1$. A model will return a probability score of zero, meaning *unknown*, if any event-transition in the execution it is scoring has a zero probability in the model.

Note that since probabilities calculated by multiplication become very small, for ease of computation we use the standard transformation to the sum of the logarithms of each probability.

2.5. Bootstrapping the Classifier

There are a number of learning strategies for training classifiers in addition to the passive learning technique (used 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 collect only those executions that remain unknown. Then these unknown executions are considered candidates representing new behaviors and 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.

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

3. Scenario

In this section we present a scenario that illustrates the use of our technique to aid a developer in extending the scope of an existing test plan.

A developer Dev has designed and implemented a version of a program P. Dev has developed a test plan to use with P and plans to 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 requirements. Test plan development and testing are expensive and often developers release software that has been tested and accepted only for some core functionality [17, 19]. 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. 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 has an automated test data generator for P, but still relies on employees to evaluate each execution.

We provide a solution to Dev as an application of bootstrapping to refine the classifier built with our technique. Figure 5 is a data-flow diagram of our application. In the

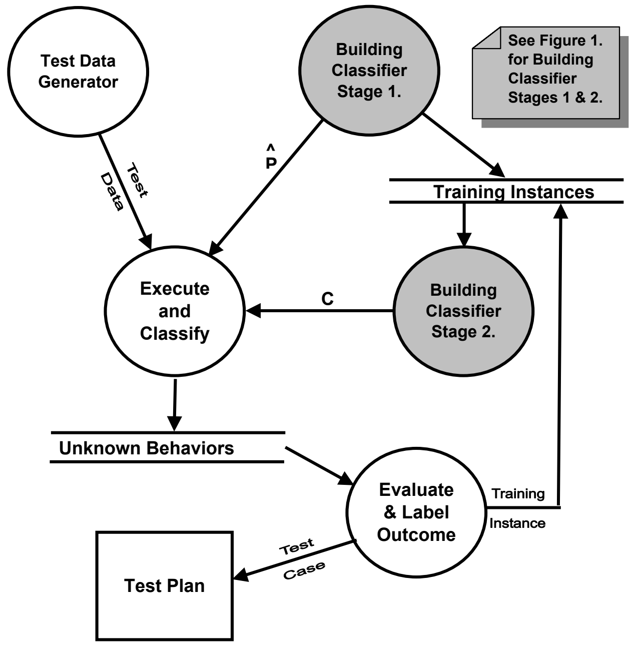


Figure 5. Automating test case selection.

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. Stage 2 of our technique from Figure 1 produces the classifier C. Our application executes $\hat{P}$ with test data produced by the test data generator and classifies each execution using C. For the bootstrapping process, our application selects as candidates only those executions that remain unknown. These candidates are evaluated and given a behavior label. They are then stored as new training instances for refining C. The corresponding test case for each candidate is generated and added to the test plan. Our application retrains C at certain intervals using the augmented set of training instances, stopping when the rate of detection of unknown executions falls below some threshold.

Our application provides at least two economic benefits to Dev. First, by guaging the rate at which unknown executions are produced, Dev can estimate the risks associated with the current test plan. Secondly, using our application, Dev can improve the efficiency with which new test cases for new behaviors are generated compared to simply evaluating all new executions. We explore this gain in efficiency in our empirical Study 3 (Section 4.5).

4. Empirical Studies

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

4.1. 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 both $1st$-order method calls and $2nd$-order branch-to-branch transitions as the event-transitions for our models. 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.

4.2. 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 branch-to-branch 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.

For these evaluations, we define two metrics: Classifier Precision and Classifier Safety. *Classifier Precision* is the ratio of the number of unknown executions to the number of classifications attempted. *Classifier Safety* is the ratio of the number of executions correctly classified to the difference between the number of classifications attempted and the number of unknown executions. Classifier Precision measures the ability of the classifier to recognize behaviors with which it is presented. Classifier safety measures the behavior detection rate. 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 Classifier Safety= $\frac{80}{100-18} = 0.976$. Note that it is possible for a classifier to recognize all executions incorrectly, yielding maximum precision and zero safety.

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. SIM manipulates the models but does not permanently alter them. We developed our definition of SIM for SPACE by trial and error, discovering that agglomerative clustering using a *binary metric* [7] performed better when the $2nd$-order branch-to-branch event-transitions were excluded. It follows these steps:

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

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

4.3. Study 1: Evaluating the Classifiers

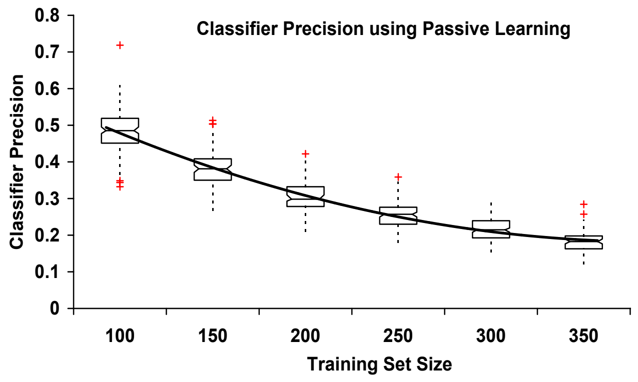
The goal of our first study was to evaluate our technique for passively training classifiers in terms of Classifier Precision and Classifier Safety. For each of the 15 selected versions of SPACE, we repeated the following process 10 times:

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). 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.

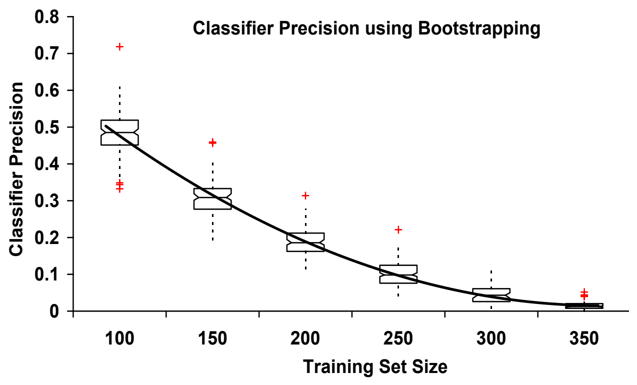
The table in Figure 6 summarizes the parametric statistics. For each training set size listed in the left column, the table shows the number of classifiers, the mean, standard deviation, and the 95% confidence interval of the mean. As the size of the training set increases, Classifier Precision improves, 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



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.



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.

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 Classifier Safety. The means are uniformly high. It is possible that larger training set sizes could reveal a trend for Classifier Safety, 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 wrongly classified behaviors (i.e., $(1 - 0.977) * (13585 - 350)$).

4.4. 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 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 statis-

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: Classifier Safety.

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: Classifier Safety.

tics for Classifier Safety. As in Study 1, the means are uniformly high. It is possible that larger training set sizes could reveal a trend for Classifier Safety, 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.

4.5. Study 3: Passive Learning vs. Bootstrapping

The goal of our third study was to compare the rates of growth in Classifier Precision between passive 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.

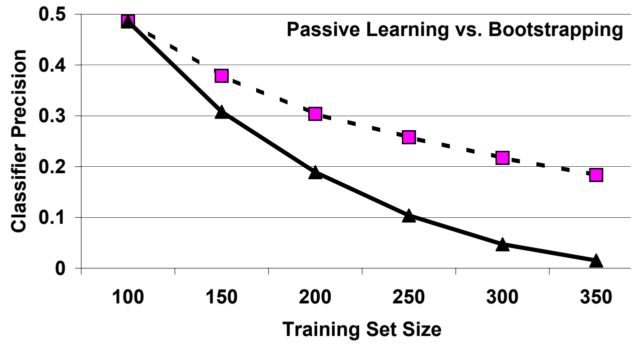


Figure 10. Study 3: Comparison of learning techniques.

As the training set size increases, so does the gain in Classifier Precision of bootstrapping over passive 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 passive 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.

5. Related Work

The previous work that is closest in spirit and method to this paper is that of Podgurski and colleagues [18, 7]. This 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 is the central role of Markov models in our approach and our use of active learning techniques to improve the efficiency of behavior modeling.

Dickinson, Leon, and Podgurski demonstrate the advantage of automated clustering of execution profiles over random selection for finding failures [7]. They use various profiles, including branch profiles, as the basis for cluster formation. We focus on the utility of $2nd$-order Markov models as predictors of program behavior. In Podgurski et al. [18], clustering is combined with feature selection, and multidimensional scaling is used to visualize the resulting grouping of executions. In both of these works, the clustering methods are passive in the sense that clusters are formed from a batch of data 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 [21]. 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 [14]. They address the problem of scoring events that have not been encountered during training, when 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 [11]. 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. 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 [10]. In comparison, we use Markov statistics of events to model program behavior.

Munson and Elbaum posit that actual executions are the final source of reliability measures [16]. 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.

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 $1st$- and $2nd$-order 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 discovered that agglomerative hierarchical clustering was sensitive to the granularity of the similarity metric used. We will investigate ways to tune this and other metrics suggested by reference [7] as well as explore additional clustering algorithms.

Second, we found that $1st$- and $2nd$-order event-transitions were powerful for the modeling and classification of behaviors. We plan to explore models of order 3 and higher to determine the most effective granularity for modeling dependencies. We will investigate uses of these models to detect sub-behaviors such as inside individual modules, and to detect more abstract behaviors such as those modeled by operational profiling.

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. 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 [18] with which to train classifiers.

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 (HMM) to extend our behavior modeling technique. An HMM augments a standard Markov model with a set of variables known as the observations. The state of the Markov model becomes a hidden variable, accessible only through the observations emitted by the Markov model. HMMs are interesting from two perspectives. First, simple

$1st$- or $2nd$- order Markov models over the hidden state in an HMM induce more complex distributions over the set of observations through marginalization. If the observations are event-transitions, then the hidden states could correspond to higher-level categories of transitions. In comparison to the Markov models used in this paper, significantly more complex distributions over events could be modeled while retaining the attractive complexity properties of our current technique. Second, advanced HMM models such as 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 predictions about program behaviors.

Acknowledgements

This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 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. William Ribarsky provided key insights and support. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided `Space` and many of its test cases. Gregg Rothermel provided additional test cases and advice on experimental protocols.

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] 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.

[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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.