SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM

TMLAI TRANSACTIONS ON
MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

Volume 6 No. 6
ISSN 2054-7390

Automatic Classification of Program Paths Feasibility Using Active Learning

Moheb R. Girgis, Alaa I. El-Nashar, Asmaa M. Elsify
Department of Computer Science, Faculty of Science, Minia University, El-Minia, Egypt
moheb.girgis@mu.edu.eg; nashar_al@yahoo.com; as_mo_elsify@yahoo.com

ABSTRACT

One of the challenging problems that faces the automated test data generation for path testing is the existence of infeasible paths, where no input data can be found to exercise them. Substantial time and effort may be wasted in trying to generate input data to exercise such paths. This paper proposes an active-learning approach to the automatic feasibility classification of program paths. This approach is based on the hypothesis that certain features of program behavior are stochastic processes that exhibit the Markov property, and that the resultant Markov models of individual program paths can be automatically clustered into effective predictors of path feasibility. To this end, the paper presents a technique that represents program paths as Markov models, and a clustering algorithm for Markov models that aggregates them into an effective path feasibility classifier. In this approach, the classifier is a map from program path statistics, namely, edge, branch, or definition-use profiles, to a label for the path, "feasible" or "infeasible". The presented technique employs the bootstrapping active learning strategy, where the classifier is trained incrementally on a series of labeled instances, to extend its scope of training to be able to succeed in classifying new paths. The paper also presented the results of the experiments that were conducted to evaluate the effectiveness of the three paths feasibility classifiers built by using the proposed technique, and the bootstrapping technique.

Keywords: Software testing; Infeasible paths detection; Batch learning; Active learning; Paths feasibility classifier; Markov model.

1 Introduction

Software testing is a practical way of obtaining increased confidence in software. Software testing consists of generating test data according to some testing strategy, such as path testing, and then checking the output produced by the test data against the expected results. One of the challenging problems that faces the automated test data generation for path testing is the existence of infeasible paths, where no input data can be found to exercise them. Experimental evidences have shown that a significant amount of infeasible paths are present in complicated programs, and the detection of these infeasible paths is an undecidable question [1]. Substantial time and effort may be wasted in trying to generate input data to exercise such paths. So, timely detecting these infeasible paths cannot only save test resources but also improve test efficiency.

DOI: 10.14738/tmlai.66.5811
Publication Date: 27th December, 2018
URL: http://dx.doi.org/10.14738/tmlai.66.5811

Machine learning techniques, such as classification, have been successfully applied to software engineering problems. Some researchers used the batch learning approach (e.g., [2-8]), in which a fixed quantity of manually labeled training data is collected at the start of the learning process. Other researchers used the active learning approach (e.g., [9-10]), in which the classifier is trained incrementally on a series of labeled data elements. The advantage of active learning is that it can extend the scope of the classifier beyond what batch learning would yield, for the same amount of labeling effort [10].

Any program path has control-flow components, such as branches and edges, and data-flow components, such as definition-use (def-use) chains. These components are features for which aggregate statistical measures can be collected, such as branch profiles, edge profiles, and def-use profiles. The aim of this paper is to explore the use of these features as predictors of path feasibility. In order to achieve this aim, the paper proposes an active-learning approach to the automatic feasibility classification of program paths. This approach is based on the hypothesis that these program features are stochastic processes that exhibit the Markov property, and that the resultant Markov models of individual program paths can be automatically clustered into effective predictors of path feasibility.

The proposed approach includes a technique that represents program paths as Markov models, and a clustering algorithm for Markov models that aggregates them into a path feasibility classifier. In this approach, the classifier is a map from program path statistics, namely, edge, branch, or def-use profiles, to a label for the path, "feasible" or "infeasible". This is based on the idea that such profiles can reflect the patterns that may cause path infeasibility, so the models of two infeasible paths that share these profiles can be similar, i.e. belong to the same class. The proposed approach, initially, employs the batch-learning technique for path feasibility classification, then the technique is combined with the bootstrapping active learning strategy [10], where the classifier is trained incrementally on a series of labeled instances, to improve it.

So far as the authors are aware, no other work has been proposed combining the use of Markov models and batch/active learning in the classification of paths feasibility.

The paper is organized as follows: Section 2 presents a review of the related work in using machine learning in program behavior classification and paths feasibility classification. Section 3 describes the proposed approach for building a path feasibility classifier, which includes modeling program paths using Markov models built from edges, branches, and def-use profiles; training the path feasibility classifier; using the trained classifier; and improving the classifier by using the bootstrapping active learning strategy. Section 4 describes the results of the experiments that have been conducted to evaluate the proposed approach. Section 5 presents the conclusion of the work presented in this paper.

2 Related Work

As the main concern of this work is the automatic feasibility classification of program paths using active-learning, this section reviews examples of the related work in using machine learning in program behavior classification and paths feasibility classification.

Several approaches have been proposed for program behavior classification using machine learning. Examples of these approaches are reviewed below.

Some of the approaches in this area used Markov models to describe the stochastic dynamic behavior of program executions. Whittaker and Poore [11] used Markov chains to model software usage from specifications prior to implementation. Cook and Wolf [12] used Markov models to represent individual executions in their study of automated process discovery from execution traces. They concentrated on transforming Markov models into finite state machines as models of process. Jha et al. [4] used Markov models of event traces as the basis for intrusion detection. They address the problem of scoring events that have not been encountered during training. Bowring et al. [10] proposed an active learning approach to build a classifier of program behaviors. Firstly, they model individual program executions as Markov models built from the profiles of event transitions such as branches. Then, they build clusters of these Markov models, which then together form a classifier tuned to predict specific behavioral characteristics of the considered program, such as “pass” or “fail”.

Another group of approaches focused on failure detection. Dickinson et al. [3] proposed a technique that uses cluster analysis of execution profiles to find failures among the executions induced by a set of potential test cases. They use many feature profiles (e.g., branch decision, method calls) as the basis for cluster formation. Podgurski et al. [6] proposed an approach to fault detection and failure categorization that combines clustering with feature selection, and used multidimensional scaling to visualize the resulting grouping of executions. In both of these approaches, the clusters are formed once using batch learning and then used for subsequent analysis. Brun and Ernst [8] used dynamic invariant detection to extract program properties relevant to revealing faults and then applied batch learning techniques to rank and select these properties. Gross et al. [7] proposed 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. Their models are built once using batch learning. Haran et al. [13-14] proposed techniques for automatically classifying execution data collected in the field. They used statistical learning algorithms to build the classification models. Their techniques build the models by analyzing executions performed in a controlled environment (e.g., test cases run in-house) and then use the models to predict whether execution data produced by a fielded instance were generated by a passing or failing program execution. Lo et al. [15] proposed a technique to classify software behaviors based on past history or runs. With this technique, it is possible to generalize past known errors and mistakes to capture failures and anomalies. Francis et al. [16] proposed two tree-based techniques for classifying reported software failures in order to facilitate prioritizing them and diagnosing their causes. The first technique is based on the use of dendrograms, which are rooted trees used to represent the results of hierarchical cluster analysis. The second technique employs a classification tree constructed to recognize failed executions.

The final group of related work used statistical learning methods to analyze program executions. Harder et al. [17] automatically classify software behavior using an operational differencing technique. Their method extracts formal operational abstractions from statistical summaries of program executions and uses them to automate the augmentation of test suites. Munson and Elbaum [18] postulate that actual executions are the final source of reliability measures. 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. Ammons et al. [5] proposed a mining technique for extracting formal specifications from interaction traces by learning probabilistic finite suffix automata models. 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.

To the best of our knowledge no much work have been done in the area of using machine learning in paths feasibility classification. Baskiotis et al. [19] proposed an adaptive feasible paths sampling mechanism, called EXIST. It proceeds by iteratively generating candidate paths based on the current distribution on the program paths, and updating this distribution after the path has been labelled as feasible or infeasible. Baskiotis and Sebag [20] proposed an active learning algorithm, called S4T (for Structural Sampling for Statistical Software Testing), which samples new feasible paths using some initially available feasible paths.

3 Building Path Feasibility Classifier

Our path feasibility classifier employs Markov models for program paths in predicting the path feasibility. In order to build such models, we considered subset of the features that profile event transitions in program paths. An event transition is a transition from one program entity to another; types of first-order event transitions include branches (source statement to destination statement), method calls (caller to callee), and definition to use (def-use) chains; one type of second-order event transition is branch-to-branch [10]. An event-transition profile is the frequency with which an event transition occurs in a program path.

As demonstrated by Bowring et al. [10], such event-transition features describe stochastic processes that exhibit the Markov property. So, in this work, the ability of Markov models built from them to predict program path feasibility is explored. 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 s1 at time t+1 given that the state at time t was s0.

The relative frequency of an event transition in a program path provides a measure of its probability. For example, an edge in the program control-flow graph (CFG) can be considered as event transition between its source and destination nodes. Thus, these nodes represent states in the Markov model. The transition probability in the Markov model between the source node and the destination node is the relative occurrence frequency, or profile, of the edge in a path. In this work, three event transitions are considered: the edge event transition between an edge source node and its destination node, the branch event transitions between a predicate node and its two destination nodes, and the def-use event transition between a variable definition node and its use node. The frequencies of transitions between these events in program paths will be collected for use in building the Markov models representing these paths.

The proposed approach to build a classifier for path feasibility has two stages. First, individual program paths are modeled as Markov models built from the profiles of event transitions (edges, branches, or def-uses) in these paths. Each program path is represented by one model. Then, an automatic clustering algorithm is used to build clusters of these Markov models, which together form a classifier tuned to predict the feasibility of the paths of the program under test.

Figure 1 shows the structure of the Paths Feasibility Classifier Building System, which implements the proposed approach. It consists of four modules: *Static Analysis Module*, *Paths Generation Module*, *Training Instances Preparation Module*, and *Classifier Training Module*.

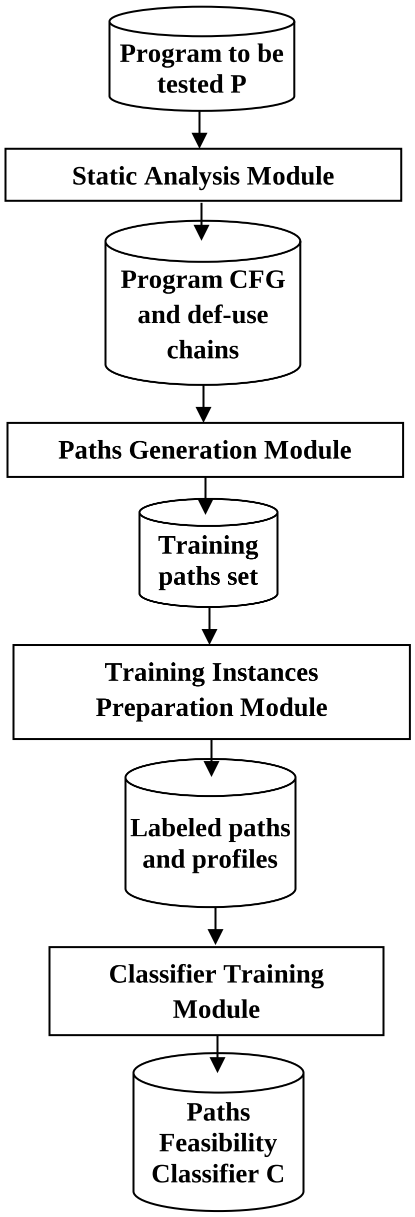

Program to be tested P
Static Analysis Module
Program CFG and def-use chains
Paths Generation Module
Training paths set
Training Instances Preparation Module
Labeled paths and profiles
Classifier Training Module
Paths Feasibility Classifier C

Figure 1. The paths feasibility classifier building system.

Firstly, the program to be tested P is presented to the ***Static Analysis Module***, which produces the CFG elements (lists of edges and branches) and list of def-use chains of P. This information is passed to the ***Paths Generation Module***, which generates the ZOT subset of program paths, which includes program paths that traverse loops zero, one and two times [21]. This subset constitutes the training paths set of P. It is passed to the ***Training Instances Preparation Module***, which computes the event-transition (branches, edges, or def-uses) profiles for each path in the training paths set, and assigns to it a feasibility label using a *Symbolic Execution System* [22]. The feasibility label can be "f" for feasible or "inf" for infeasible. This module produces the training instances of P, where each training instance consists of a path with its event-transition profiles and feasibility label. Finally, the training instances are passed to the ***Classifier Training Module***. This module first groups the training instances by the distinct feasibility labels *f* and *inf*. Then, it converts each training instance in each feasibility group to a Markov model. The module initially uses a batch-learning paradigm to train one classifier per feasibility group. Finally, the module assembles the two feasibility groups of classifiers, C_f and C_{inf} to form the classifier C for P.

The ***Classifier Training Module*** implements the algorithm ***TrainPathFeasibilityClassifier***, shown in Figure 7. Before describing this algorithm, the Markov model building process is explained.

3.1 Building Markov Model

The proposed approach depends on using Markov models to encode the event-transition profiles of program paths. In this subsection, the mapping from program events to the concept of state, which is used in building Markov models, is illustrated for the three types of event-transitions considered in the approach, which are program edges, branches, and def-uses.

The algorithm ***BuildMarkovModel***, shown in Figure 2, is a generic algorithm that constructs a matrix representation of a Markov model from event-transition profiles of each edge, branch, or def-use in a program path.

```
Algorithm BuildMarkovModel(S, D, $fl$)
Input: S = {$s_0$, $s_1$, ..., $s_{n-1}$}, a set of states
          D = $\left\{\left(s_{from_j}, s_{to_j}, profile_j\right) : 0 \le j < |D|\right\}$, a list of ordered triples for
          each event transition and its profile
          $fl$ = a string representing a feasibility label
Output: (M, D, $fl$), a Markov model M, D and $fl$
Begin
1.    M ← new double array [|s|,|s|] initialized to 0
2.    For each ($s_{from}$, $s_{to}$, profile) ∈ D where $s_{from}$, $s_{to}$ ∈ S
3.        m[$s_{from}$, $s_{to}$] ← profile;
4.    End For
5.    For i ← 0 to |s|-1
6.        rowSum = 0;
7.        For j ← 0 to |s|-1
8.            rowSum ← rowSum + m[i,j];
9.        End For
10.       If rowSum > 0
11.           For j←0 to |s|-1
12.               m[i,j] ← m[i,j] / rowSum;
13.           End For
14.       End If
15.   End For
16.   Return (M, D, $fl$);
```

Figure 2. Algorithm BuildMarkovModel to build Markov model for a path.

BuildMarkovModel algorithm takes, for a path *p*, three inputs: S, D, fl, where S is a set of states used to specify the event transitions; D is a list of the event transitions in *p* and their profiles stored as ordered triples, (s_{from}, s_{to}, profile), where s_{from} and s_{to} are the source and destination nodes of an event transition (edge, branch, or def-use), respectively, and *profile* is the occurrence frequency of the event transition in *p*; and fl is the feasibility label for the model. The output (M, D, fl) is a triple consisting of the model M, the profile data D, and the feasibility label fl of the path *p*. In line 1, the matrix M for the model is initialized using the cardinality of S. In lines 2-4, each transition in D that involves states in S is recorded in M. In lines 5-15, each row in the 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.

3.1.1 Building Markov model from edge profiles

The edges are possible transfers of control flow between the nodes of the CFG, e.g. an edge (i, j) corresponds to a possible transfer of control from node i to node j. Each node represents a group of consecutive statements which together constitute a basic block. An edge (i, j) is considered as an event transition in the CFG between the source node i and the destination node j.

Consider the path *p* = 1, 2, 4, 5, 6, 5, 6, 5, 7, 9 in the CFG of the example program, shown in Figure 3. We compute the profiles of each edge by counting how many times the edge occurred in the path. For example, the profile of edge (5-6) is 2. The Markov model, built from the edges and their profiles in *p*, is shown in Figure 4 as a matrix. It models the program states identified by the source and destination nodes of each edge. The transitions are read from row to column. A Markov model built from edge profiles is simply the adjacency matrix of the CFG with each entry equals to the row-normalized profile. For example, in the row of node 5, as the profile of edge (5-6) is 2 and the profile of edge (5-7) is 1, i.e. 3 in total, there are two entries, 2/3 and 1/3, in the cells (5, 6) and (5, 7).

```
              static void Main(string[] args) {
1               int x=double.Parse(Console.ReadLine());
1               int y=double.Parse(Console.ReadLine());
1               int pow;
1                 if(y < 0)
2                   pow = -y;
3                 else
3                   pow = y;
4                 int z = 1;
5                 while (pow != 0) {
6                   z =z * x;
6                   pow = pow – 1;
6                 }
7                 if(y < 0)
8                   z =1/z;
9                 double answer =z + 1;
9               Console.WriteLine(answer);
              }
```

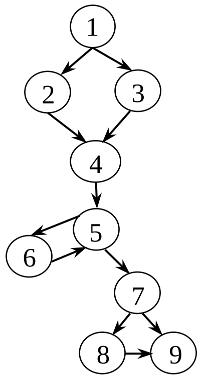


Figure 3. Example program and its CFG.

As mentioned above, the algorithm ***BuildMarkovModel***, shown in Figure 2, can construct a Markov model from edge profiles in a program path. In this case, the input S is the set of CFG nodes; D is a list of the edges in *p* and their profiles stored as ordered triples, (s_{from}, s_{to}, profile), where s_{from} and s_{to} are the source and destination nodes of an edge, respectively, and *profile* is the occurrence frequency of the edge in *p*. For the example path, *p* = 1,2,4,5,6,5,6,5,7,9, the inputs to ***BuildMarkovModel*** are:

- S = {1, 2, 3, 4, 5, 6, 7, 8, 9}
- D = ((1, 2, 1), (2, 4, 1), (4, 5, 1), (5, 6, 2), (6, 5, 2), (5, 7, 1), (7, 9, 1))
- fl = “inf”

In this case, the output component M is the Markov model shown in Figure 4.

	1	2	3	4	5	6	7	8	9
1	0	1/1	0	0	0	0	0	0	0
2	0	0	0	1/1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1/1	0	0	0	0
5	0	0	0	0	0	2/3	1/3	0	0
6	0	0	0	0	2/2	0	0	0	0
7	0	0	0	0	0	0	0	0	1/1
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

Figure 4. Markov model for the edge profiles of the example path *p* = 1,2,4,5,6,5,6,5,7,9.

3.1.2 Building Markov model from branch profiles

Next, we illustrate the process of building Markov model from branches profiles. Here, a branch means an out edge of a predicate node. A branch (i, j) of a predicate node i is considered as an event transition in the CFG between the source node i and the destination node j.

In the CFG of the example program, shown in Figure 3, there are three predicate nodes, 1, 5, and 7. Again, consider the path *p* = 1, 2, 4, 5, 6, 5, 6, 5, 7, 9 in this CFG. We compute the profiles of each branch of each predicate node by counting how many times the branch has occurred in the path. For example, the profile for branch (1-2) of predicate node 1 is 1. The matrix representation of the Markov model, built from the branches and their profiles in *p*, is shown in Figure 5. It models the program states identified by the source and destination nodes of each branch of each predicate node. Each row shows the row-normalized profiles of the branches of a predicate node. For example, in the row of predicate node 5, as the profile of branch (5-6) is 2 and the profile of branch (5-7) is 1, i.e. 3 in total, there are two entries, 2/3 and 1/3, in the cells (5, 6) and (5, 7), respectively.

The algorithm ***BuildMarkovModel***, shown in Figure 2, can also construct a Markov model from branch profiles in a program path. In this case, the input S is the set of predicate nodes in the CFG; D is a list of the branches in *p* and their profiles stored as ordered triples, (s_{from}, s_{to}, profile), where s_{from} and s_{to} are the source and destination nodes of a branch, respectively, and *profile* is the occurrence frequency of the branch in *p*. For the example path, *p* = 1,2,4,5,6,5,6,5,7,9, the inputs to ***BuildMarkovModel*** are:

- S = {1, 5, 7}
- D = ((1, 2, 1), (5, 6, 2), (5, 7, 1), (7, 9, 1))
- fl = "inf"

In this case, the output component M is the Markov model shown in Figure 5.

	1	2	3	4	5	6	7	8	9
1	0	1/1	0	0	0	0	0	0	0
5	0	0	0	0	0	2/3	1/3	0	0
7	0	0	0	0	0	0	0	0	1/1

Figure 5. Markov model for the branch profiles of the example path *p* = 1,2,4,5,6,5,6,5,7,9.

3.1.3 Building Markov model from def-use profiles

Finally, we illustrate the process of building Markov model from def-use chain profiles. A def-use chain of a variable is a path from the definition to the use of the variable without any intervening redefinitions. Here, we consider the def-use chains required to fulfill the all-uses criterion, which is one of the most demanding in the family of data flow criteria described by Rapps and Weyuker [23]. A def-use chain <d, u> of a variable v is considered as an event transition in the CFG between the source (def) node d and the destination (use) node u.

Table 1 shows the list of def-use chains required to fulfill the all-uses criterion for the example program shown in Figure 3. In this list, each def-use chain (path) is represented by: a def-node (a node containing a def of a variable); a use-node (a node containing a use of that variable); and the set of nodes that must not be included in that path (i.e., nodes containing other defs of that variable). These nodes are called ***killing nodes*** [21]. The value (-1) is used in the killing node column to indicate that the def-use path has no killing nodes. A path is said to cover a def-use chain if it has a subpath that starts at the def-node and ends at the use-node of the def-use chain and does not pass through its killing nodes [21].

In the example program, there are 8 defs: (x,1), (y,1), (pow,2), (pow,3), (z,4), (z,6), (pow,6), and (z,8). Again, consider the path *p* = 1, 2, 4, 5, 6, 5, 6, 5, 7, 9 in the CFG, shown in Figure 3. We compute the profiles of each def-use chain by counting how many times a def-clear subpath from the def-node to the use-node in the chain has occurred in the given path. For example, as the variable y is defined in node 1 and used in nodes 2 and 7, the corresponding def-use chains are: <(y,1), 2> and <(y,1), 7>. Both of these two def-use chains have profile 1, as each occurred once in *p*. The matrix representation of the Markov model built from the def-use chains and their profiles in *p* is shown in Figure 6. It models the program states identified by a def node of each variable as the source and its use node as the destination. Each row shows the row-normalized profiles of the def-use chains of a variable def. For example, in the row of def (pow,6), as the profile of def-use chain <(pow,6), 5> is 2 and the profile of def-use chain <(pow,6), 6> is 1, i.e. 3 in total, there are two entries, 2/3 and 1/3, in the cells ((pow,6), 5) and ((pow,6), 6), respectively.

Table 1. List of def-use chains of the example program.

Variable	Def-node	Use-node	Killing Nodes
Y	1	2	-1
Y	1	3	-1
X	1	6	-1
pow	2	5	3
pow	2	6	3
pow	3	5	2
pow	3	6	2
Z	4	6	8
Z	6	6	4, 8
pow	6	5	2, 3
pow	6	6	2, 3
Y	1	7	-1
Z	4	8	6
Z	6	8	4
Z	4	9	6, 8
Z	6	9	4, 8
Z	8	9	4, 6

The algorithm ***BuildMarkovModel***, shown in Figure 2, can also construct a Markov model from def-use chain profiles in a program path. In this case, the input S is the set of variable defs in the program; D is a list of the def-use chains in *p* and their profiles stored as ordered triples, (s_{from}, s_{to}, profile), where s_{from} and s_{to} are the source and destination nodes of a def-use chain, respectively, and profile is the occurrence frequency of the def-use chain in *p*. For the example path, *p* = 1,2,4,5,6,5,6,5,7,9, the inputs to BuildMarkovModel are:

- S = {(x,1), (y,1), (pow,2), (pow,3), (z,4), (z,6), (pow,6), (z,8)}
- D = (((y,1), 2, 1), ((y,1), 7, 1), ((x,1), 6, 2), ((pow,2), 5, 1), ((pow,2), 6, 1), ((z,4), 6, 1) , ((z,6), 6, 1), ((z,6), 9, 1), ((pow,6), 5, 2), ((pow,6), 6, 1))
- fl = "inf"

In this case, the output component M is the Markov model shown in Figure 6.

	1	2	3	4	5	6	7	8	9
y,1	0	**1/2**	0	0	0	0	**1/2**	0	0
x,1	0	0	0	0	0	**2/2**	0	0	0
pow,2	0	0	0	0	**1/2**	**1/2**	0	0	0
pow,3	0	0	0	0	0	0	0	0	0
z,4	0	0	0	0	0	**1/1**	0	0	0
z,6	0	0	0	0	0	**1/2**	0	0	**1/2**
pow,6	0	0	0	0	**2/3**	**1/3**	0	0	0
z,8	0	0	0	0	0	0	0	0	0

Figure 6. Markov model for the def-use profiles of the example path *p* = 1,2,4,5,6,5,6,5,7,9.

3.2 Training the Path Feasibility Classifier

Our approach is to train a path feasibility classifier using the Markov models, which are constructed from the path profiles of the training paths set, as training instances. We use a training technique that is based on an established technique known as *agglomerative hierarchical clustering* [24]. 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. Each merged cluster is also a Markov model. For example, in the first iteration, the merged model is built by combining the path profiles that form the basis for the two models being merged. Then, from this merged profile, ***BuildMarkovModel*** generates a Markov model that represents the new cluster.

The specification of the similarity function is typically done heuristically according to the application domain. In our approach, we use a simple comparison technique called *Hamming distance* to compare two Markov models, as in [10]. To compute the Hamming distance, each of the Markov models is transferred to a binary representation where a 1 is entered for all values (normalized profiles) above a certain threshold, and a 0 is entered otherwise. The threshold value, called *similarity threshold* (*SimTh*), is determined experimentally. The comparison function is called ***ModelSim()*** and is provided as an input to ***TrainPathFeasibilityClassifier*** algorithm shown in Figure 7. The binary transformation of the models is done only temporarily by *ModelSim()* in order to compute the Hamming distance.

```
Algorithm TrainPathFeasibilityClassifier(S, T, ModelSim)
Input: S = {$s_0$, s, …, $s_{n-1}$}, a set of states including a final or exit state,
       T = (($p_i$, $D_i$, $fl_i$), …), a list of ordered triples, where $p_i$ is a path in the training paths set, Di
       = $\left\{\left(s_{from_j}, s_{to_j}, \text{profile}_j\right) : 0 \le j < |D|\right\}$, a list of the event transitions in $p_i$ and their
       profiles, 0 ≤ i < |Training Instances|, and feasibility label $fl_i$ ∊ {"f", "inf"},
       ModelSim, a function to compute the similarity of two Markov models
Output: Path feasibility classifier C = {($M_i$, $D_i$, $fl_i$): $fl_i$ ∊ {"f", "inf"}, 0≤ i< |C|}
Begin
1.   C ← φ;                  // initialize the path feasibility classifier
2.   For each $fl$ ∊ {"f", "inf"}
3.       $C_{fl}$ ← φ;              // initialize the classifier for feasibility $fl$
4.   End For
5.   For each $fl$ ∊ {"f", "inf"}
6.       For each ($p_i$, $D_i$, $fl$) ∊ T, 0 ≤ i < |Training Instances with feasibility $fl$|
7.           $C_{fl}$ ← $C_{fl}$ ∪ BuildModel(S, $D_i$, $fl$);
8.       End For
9.       While |$C_{fl}$| > 2
10.          //agglomerative hierarchical clustering
11.          mid = (maxProfile($C_{fl}$) + minProfile($C_{fl}$)) / 2.0;
12.          SimTh = mid;  // similarity threshold
13.          diff ← φ;        // an empty set to collect models pair-wise differences
14.          For each ($M_i$, $D_i$, $fl$) ∊ $C_{fl}$, 0 ≤ i < |$C_{fl}$|
15.              For each ($M_j$ ,$D_j$ , $fl$) ∊ $C_{fl}$ , i < j < |$C_{fl}$|
16.                  diff ← diff ∪ ModelSim($M_i$, $M_j$, SimTh);
17.              End For
18.          End For
19.          ($M_x$, $M_y$) ← min(diff);   // Select the two closest models
20.          $D_{merged}$ ←$D_x$ ∪ $D_y$;
21.          $M_{merged}$ ← BuildModel(S, $D_{merged}$, $fl$);
22.          $C_{fl}$ ← ($C_{fl}$ – $M_x$ – $M_y$) ∪ $M_{merged}$;
23.      End While
24.      C ← C ∪ $C_{fl}$;       // add feasibility $fl$'s models to C
25.  End For
26.  Return C;
```

Figure 7. Algorithm *TrainPathFeasibilityClassifier* to train Path Feasibility Classifier.

The ***TrainPathFeasibilityClassifier*** algorithm, shown in Figure 7, trains a classifier from models generated by ***BuildMarkovModel***. The algorithm firstly groups the models of the training instances by their feasibility labels, then applies the agglomerative hierarchical clustering to the models of the training instances that have same feasibility label forming a feasibility classifier specific for this label. Here, we will have two classifiers, one for label 'f' and one for label 'inf'. Finally, it forms the final classifier as the union of the clustered models from each of these two specific feasibility classifiers.

TrainPathFeasibilityClassifier has three inputs: S, T, and *ModelSim()*, where S is a set of states that are used to identify the event transitions when ***BuildMarkovModel*** is called; T is a list of triples, each containing a path in the training paths set, a data structure D as defined in ***BuildMarkovModel***, and a feasibility label fl ∊ {"f", "inf"}; *ModelSim()* takes two Markov models as arguments and returns a real number that is the computed similarity measure between the two models.

In line 1, an empty path feasibility classifier C is initialized. In lines 2-4, an empty classifier C_{fl} is initialized for each path feasibility fl. Line 5 begins the processing for each fl. In lines 6-8, the classifier C_{fl} is populated with models built by applying ***BuildMarkovModel*** to each training instance that has feasibility fl. In lines 9-23 the models in each C_{fl} are clustered to reduce their population by merging similar and redundant models, using *ModelSim*, as described above. Line 9 establishes the stopping criterion as two models in each cluster C_{fl}. In Lines 11-12, the *similarity threshold* (*SimTh*) is set to the middle value between the maximum and minimum profile values for all models in C_{fl}. Line 13 initializes an empty set *diff* to collect models pair-wise differences. In lines 14-18, *ModelSim* is used to calculate the pair-wise differences and accumulate them in *diff*. In lines 19-21, the two closest models, M_x and M_y, are identified from *diff*, then merged, by calling ***BuildMoekovModel*** with the union of the corresponding profile sets D_x and D_y. In line 22, models M_x and M_y are replaced by the new merged model in C_{fl}. In line 24, the final clustered models in C_{fl} are added to the classifier C. After the two feasibility groups have been processed, the final classifier C is returned. This classifier is composed of two groups of Markov models, each representing a cluster of paths with same feasibility. Note that the models in each cluster are built from the profiles of all the training instances contributing to the cluster.

3.3 Using the Path Feasibility Classifier

Initially the training paths set includes paths that fulfil some test coverage criteria for the program under test. Our aim is to augment this initial set with new paths that cover more program components. The feasibility of these new paths needs to be checked to eliminate infeasible paths among them to reduce the effort of trying to find test data for them. We can use our classifier to do this job.

Figure 8 depicts the ***Path Feasibility Classification Module***, which uses the trained classifier C to classify a path *p* of program P that is not in the training set. As shown in the figure, this module accepts as input the classifier C and the path to be classified *p*, and computes its event transitions profiles, then it passes C and the list of the event transitions in *p* and their profiles, D, to the algorithm ***ClassifyPath***, shown in Figure 9, which reports the feasibility label of *p*.

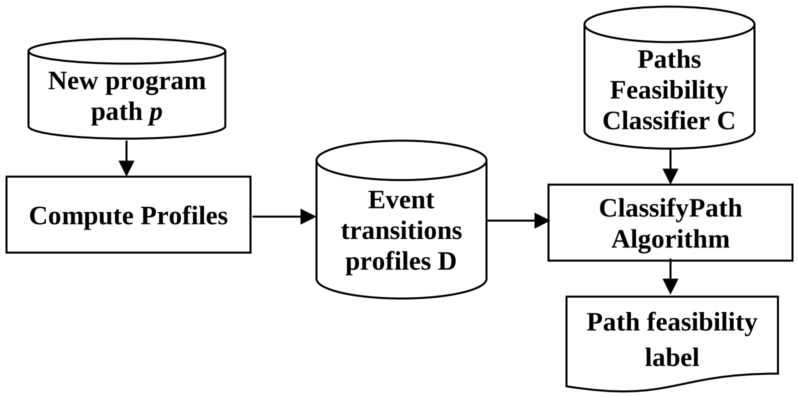


Figure 8. The Path Feasibility Classification Module.

As shown in Figure 9, in lines 1-9 of the algorithm ***ClassifyPath***, each model in C rates *p* with a probability score by applying the algorithm ***ComputeProbability***, shown in Figure 10, and in lines 10-14, the model in

C with the highest probability score for *p* provides the feasibility label for *p*. Note that, the algorithm assigns the given path the label "unknown", whenever the calculated probability is below some threshold *TH*, which means the classifier has failed to label the given path.

```
Algorithm ClassifyPath(C, D)
Input:   C, Path Feasibility Classifier, a set of Markov models:
            {(Mi, Di, $fl_i$): $fl_i$ ϵ {"f", "inf"}, 0 ≤ i < |C|}
         D, list of the event transitions: $\left\{\left(s_{from_j}, s_{to_j}, profile_j\right) : 0 \le j < |D|\right\}$
            in the path p to be classified and their profiles
Output: $fl$, Feasibility label of p
Begin
1.  maxProb = ComputeProbability(M0, D);
2.  index = 0;
3.  For each (Mi, Di, $fl_i$) ϵ C, 1 ≤ i < |C|
4.      prob = ComputeProbability(Mi, D);
5.      If prop > maxProb
6.          maxProb = prop;
7.          index = i;
8.      End If
9.  End For
10. If maxProb < TH)
11.     $fl$ = "unknown";
12. Else
13.     $fl = fl_{index}$;
14. End If
15. return $fl$;
```

Figure 9. Algorithm *ClassifyPath* that determines the feasibility label of a given path using the path feasibility classifier C.

```
Algorithm ComputeProbability(M, D)
Input:   M, matrix of a Markov model
         D, list of the event transitions: $\left\{\left(s_{from_j}, s_{to_j}, profile_j\right) : 0 \le j < |D|\right\}$
            in the path p to be classified and their profiles
Output: $\wp$, probability score assigned to p by M
Begin
1.  $\wp$ = 1;
2.  For each $\left(s_{from_j}, s_{to_j}, profile_j\right)$ ∊ D, 0 $\le j < |D|$
3.      If M[$s_{from_j}$, $s_{to_j}$] ≠ 0
4.          $\wp$ = $\wp$ * M[$s_{from_j}$, $s_{to_j}$] ^ $profile_j$;
5.      End If
6.  End For
7.  return $\wp$;
End
```

Figure 10. Algorithm *ComputeProbability* that computes the probability score assigned to a given path by a Markov model in the classifier C

The algorithm ***ComputeProbability*** computes the probability score with which a Markov model M in the classifier C rates the given path *p*. The probability score is the probability that the model M could produce the sequence of event-transitions in path *p* [10]. To illustrate how ***ComputeProbability*** works, consider another path *p* = 1, 2, 4, 5, 6, 5, 6, 5, 7, 8, 9 in the CFG, shown in Figure 3. The algorithm accepts a Markov model M and the list of the event transitions in *p* and their profiles D. By passing the Markov model M, shown in Figure 4, and the list of event transitions (edges) in *p*:

D = ((1, 2, 1), (2, 4, 1), (4, 5, 1), (5, 6, 2), (6, 5, 2), (5, 7, 1), (7, 8, 1), (8, 9, 1)),

to the algorithm, it calculates the probability score $\wp$ that M gives to p as follows:

$\wp$ = M[1,2]^1 * M[2,4]^1 * M[4,5]^1 * M[5,6]^2 * M[6,5]^2 * M[5,7]^1

= 1.0^1 * 1.0^1 * 1.0^1 * 0.667^2 * 1.0^2 * 0.333^1 = 0.148148

Note that M[7,8] and M[8,9] were not included in the calculation, as each of them has a value 0.

3.4 Improving the Classifier

Initially in ***TrainPathFeasibilityClassifier***, we have used the batch-learning technique. In addition to this technique, there is another effective learning technique for training classifiers, which is active learning [9]. Active learning techniques employ an interactive or query-based approach that can be used to control the costs of training classifiers. Bowring et al. [10] have used a type of active learning called *bootstrapping*. Their application of bootstrapping first uses the classifier to score new program executions and 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.

Now, we explain how to combine our technique for building path feasibility classifier with the bootstrapping active learning strategy to extend the scope of training our classifier to be able to succeed in classifying new paths.

In the bootstrapping process, we select as candidates from the processed set of paths only those paths that remain unknown to the classifier. The feasibility of these candidates is evaluated by an oracle. The oracle in our case is a symbolic execution system with inequality solver [21]. The candidates are then used as new training instances for training C. The classifier C is retrained and refined at certain intervals using the augmented training instances set. The iteration of the bootstrapping process stops when the rate of detection of paths with unknown feasibility falls below some threshold.

4 Experiments

This section presents the results of the experiments that we have conducted to evaluate the effectiveness of the three proposed paths feasibility classifiers: *Branches Model*, *Edges Model*, and *Def-Use Model*, which are based on edges, branches, and def-uses profiles, respectively.

In the experiments, we have applied the three classifiers to 5 sample programs, and used the accuracy metric for evaluating the effectiveness of classification results. The accuracy of a classification model on a test set is defined as:

$$\text{Accuracy} = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}},$$

where a correct classification means that the learned model predicts the same class as the original class of the test case.

4.1 Experiments with Batch Learning

Firstly, we evaluated the three proposed paths feasibility classifiers using only batch learning. For each of the 5 sample programs, we repeated the following steps:

- Randomly select training set.
- Build a classifier from the training set.
- Evaluate the classifiers.

In order to study the effect of the size of the training set on the accuracy classification results, we have conducted the experiments with different percentages of the data set as training data, where each one is referred to as *training data percentage* (*tdp*). For example, tdp = 75% means partitioning the data set into two sets of 75% for training and 25% for test. The tdp values used in the experiments were 25%, 50%, and 75%.

Also, in order to evaluate the effect of the *similarity threshold* (*SimTh*) used in the function *ModelSim()*, we initially set *SimTh* to the middle value (mid) between the maximum and minimum profile values for all models in a cluster C_{fl}, as shown in Figure 7 (Lines 11-12), then we used variations of this value ($\text{mid} \pm 0.2$).

Tables 1-3 show the classification accuracy results for the three proposed paths feasibility classifiers with different tdp and SimTh values. In each row of these tables, the shaded cells indicate the higher accuracies obtained for each sample program.

Regarding the variations in tdp values, the results indicate that the Branches and Edges models gave higher accuracy with tdp =75%, but the Def-Use Model gave higher accuracy with tdp =25%. Regarding the variations in SimTh values, for the Branches and Def-Use Models, the results indicate that the variations have little effect on the accuracy of the classifiers, but for the Edges Model, they have no effect at all.

Figures 11(a)-(c), 12(a)-(c), and 13(a)-(c) show comparisons between the classification accuracy results for the Branches, Edges, and Def-Use Models, respectively, with different tdp values, for each SimTh value (mid, mid+0.2 and mid-0.2).

Figures 14(a)-(c) show comparisons between the average classification accuracy results for the three models with different tdp values, for each SimTh value (mid, mid+0.2 and mid-0.2, respectively). These figures showed that, on average, the Branches Model has the higher accuracy in predicting the feasibility of program paths, followed by the Edges Model, while the Def-Use Model has the lower accuracy. This indicates that the control flow profiles, such as branches and edges, are more useful in predicting the feasibility of program paths than the data flow profiles, such as def-use profiles.

4.2 Experiments with Bootstrapping Active Learning

Secondly, we conducted simple experiment to evaluate the application of bootstrapping to refine the classifiers built using the batch learning technique. In this experiment, we selected one of the paths that were classified as "unknown", labeled it, and added it to the training set for the classifier, then retrained the classifier using the augmented set of training instances. Table 4 shows the improvement rate in the classifier accuracy after augmenting the training set with just one correctly labeled unknown path. It should be noted that, the table shows only the results for Prog#2 and Prog#3 with Def-Use Model, and Prog#5 with Edges and Def-Use Models, as those the cases where the batch learning classification results included paths with label "unknown".

5 Conclusion

The aim of this paper is to explore the use of program features, such as edges, branches, and def-uses profiles, as predictors of path feasibility. To achieve this aim, the paper presented a proposed active-learning approach to the automatic feasibility classification of program paths. In this approach, program paths were represented as Markov models, then these models were aggregated into an effective path feasibility classifier, which is a map from program paths statistics, namely, edge, branch, or def-use profiles, to a label for the path, “feasible” or “infeasible”. Three paths feasibility classifiers: Branches Model, Edges Model, and Def-Use Model, were built, based on edges, branches, and def-uses profiles, respectively. The proposed approach employs the bootstrapping active learning strategy, where each classifier is trained incrementally on a series of labeled instances to extend its scope of training to be able to succeed in classifying new paths.

The paper also presented the results of the experiments that were conducted to evaluate the effectiveness of the three paths feasibility classifiers built by using the proposed approach. In these experiments, the effect of the training set size and the similarity threshold (SimTh) values on the classification results were studied by using different training data percentages (tdps) and different SimTh values, during the training of the proposed paths feasibility classifiers.

Table 1. A comparison between the classification accuracy results for the Branches Model with different tdp and SimTh values.

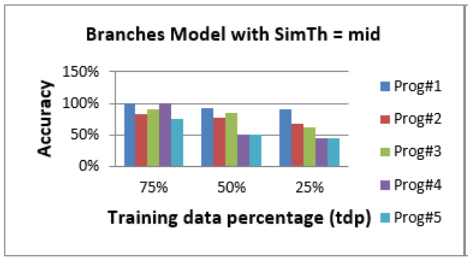
Program #	No. of Paths	Training data percentage (tdp)								
		75%			50%			25%		
		Similarity Threshold (SimTh)								
		mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2
Prog#1	108	100%	85.19%	100%	92.59%	100%	100%	90.12%	90.12%	90.12%
Prog#2	26	83.33%	83.33%	83.33%	76.92%	76.92%	76.92%	68.42%	68.42%	68.42%
Prog#3	43	90.91%	81.82%	63.64%	85.71%	71.43%	71.43%	62.5%	46.88%	46.88%
Prog#4	12	100%	100%	100%	50%	50%	50%	44.44%	44.44%	44.44%
Prog#5	32	75%	75%	87.5%	50%	56.25%	50%	45.38%	62.5%	50%

Table 2. A comparison between the classification accuracy results for the Edges Model with different tdp and SimTh values.

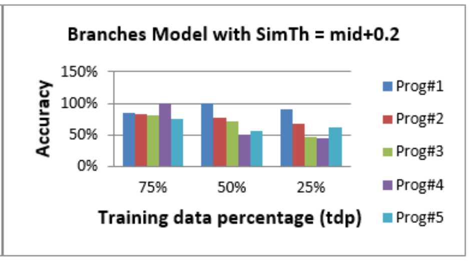
Program #	No. of Paths	Training data percentage (tdp)								
		75%			50%			25%		
		Similarity Threshold (SimTh)								
		mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2
Prog#1	108	100%	100%	100%	48.15%	48.15%	48.15%	60.49%	60.49%	60.49%
Prog#2	26	83.33%	83.33%	83.33%	69.23%	69.23%	69.23%	57.89%	57.89%	57.89%
Prog#3	43	90.91%	90.91%	90.91%	85.71%	85.71%	85.71%	50%	50%	50%
Prog#4	12	33.33%	33.33%	33.33%	50%	50%	50%	55.56%	55.56%	55.56%
Prog#5	32	75%	75%	75%	56.25%	56.25%	56.25%	66.67%	66.67%	66.67%

Table 3. A comparison between the classification accuracy results for the Def-Use Model with different tdp and SimTh values.

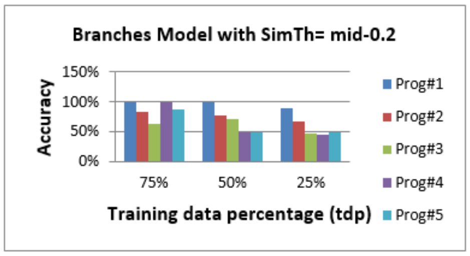
Program #	No. of Paths	Training data percentage (tdp)								
		75%			50%			25%		
		Similarity Threshold (SimTh)								
		mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2	mid	mid+0.2	mid-0.2
Prog#1	108	66.67%	66.67%	85.19%	33.3%	33.3%	48.15%	66.67%	66.67%	79.01%
Prog#2	26	83.33%	50%	33.33%	46.15%	46.15%	69.23%	63.16%	63.16%	68.42%
Prog#3	43	72.73%	81.82%	72.73%	90.48%	85.71%	71.43%	53.13%	53.13%	31.25%
Prog#4	12	33.33%	33.33%	33.33%	50%	50%	50%	55.56%	55.56%	55.56%
Prog#5	32	50%	50%	37.5%	50%	50%	43.75%	58.33%	54.17%	66.67%



(a)

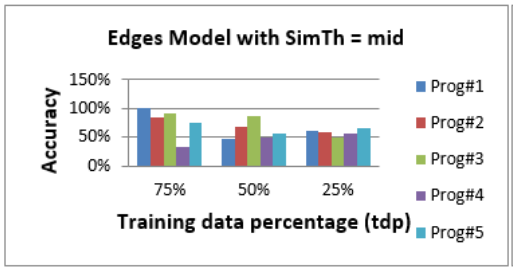


(b)

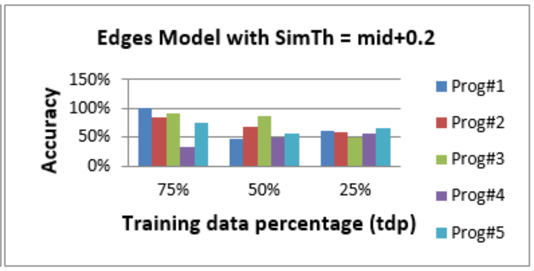


(c)

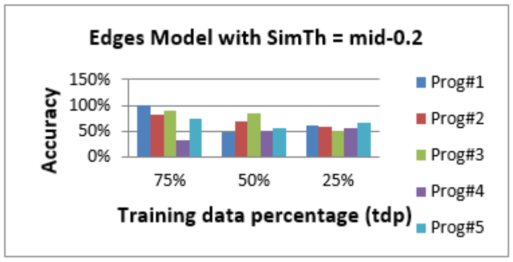
Figure 11. Comparisons between the classification accuracy results for the Branches Model with 3 different tdp values, and (a) SimTh = mid, (b) mid+0.2 and (c) mid-0.2



(a)

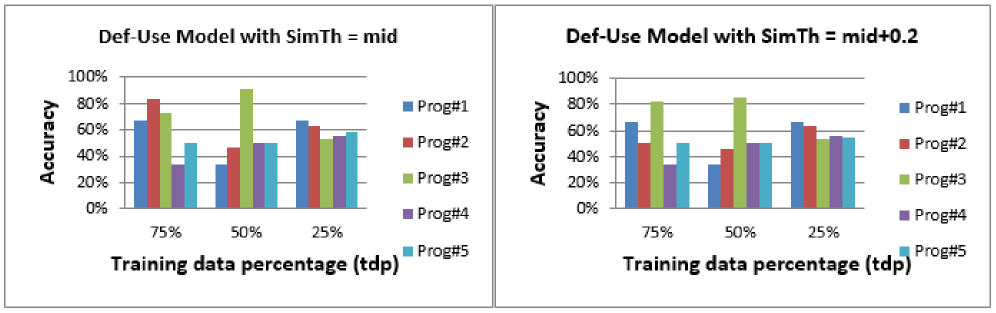


(b)

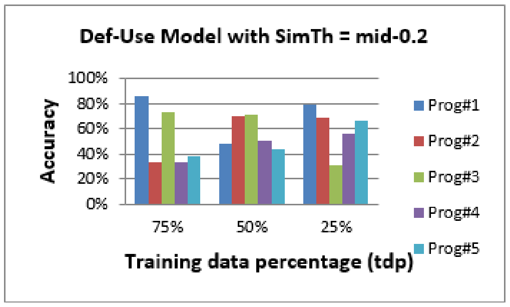


(c)

Figure 12. Comparisons between the classification accuracy results for the Edges Model with 3 different tdp values, and (a) SimTh = mid, (b) mid+0.2 and (c) mid-0.2.

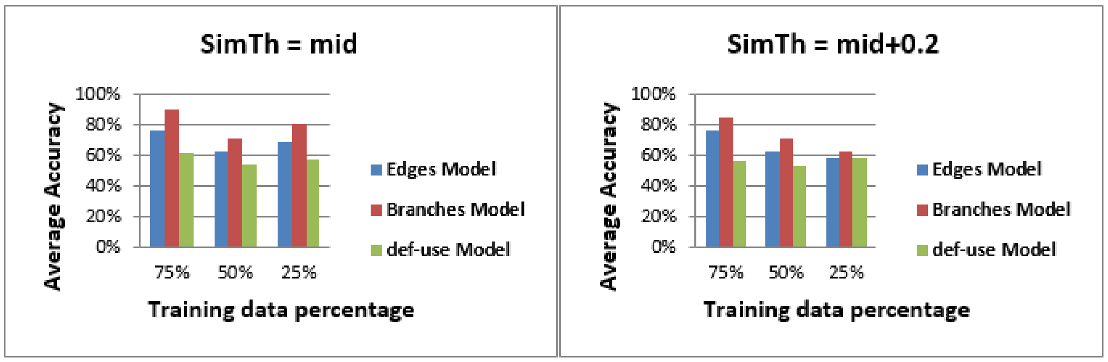


(a) (b)

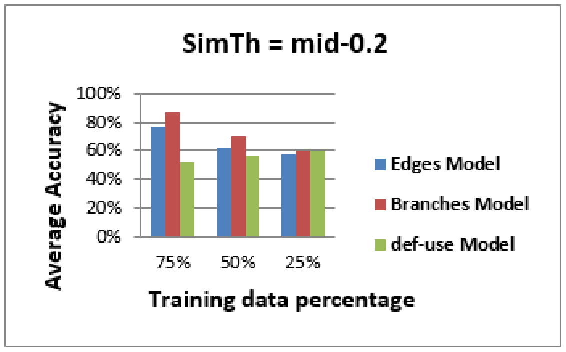


(c)

Figure 13. Comparisons between the classification accuracy results for the Def-Use Model with 3 different tdp values, and (a) SimTh = mid, (b) mid+0.2 and (c) mid-0.2.



(a) (b)



(c)

Figure 14. Comparisons between the average accuracy results for the three models with 3 different tdp values, and (a) SimTh = mid, (b) mid+0.2 and (c) mid-0.2.

Table 4. A comparison between the classification accuracy results obtained by applying the batch learning technique, and then bootstrapping.

Program #	Model	Accuracy		Improvement rate
		Batch Learning	Bootstrapping	
Prog#2	Def-Use	46.15%	66.67%	44.46%
Prog#3	Def-Use	31.25%	51.62%	65.18%
Prog#5	Edges	56.52%	63.64%	12.6%
Prog#5	Def-Use	47.83%	50%	4.54%

Regarding the variations in tdp values, the results indicated that the Branches and Edges models gave higher accuracy with large tdps, while the Def-Use Model gave higher accuracy with small tdps. Regarding the variations in SimTh values, the results indicated that, for the Branches and Def-Use Models, the variations have little effect on the accuracy of the classifiers, but for the Edges Model, they have no effect at all.

The experiments showed also that, on average, the Branches model has the higher accuracy in predicting the feasibility of program paths, followed by the Edges model, while the Def-Use Model has the lower accuracy. This indicates that the control flow profiles, such as branches and edges, are more useful in predicting the feasibility of program paths than the data flow profiles, such as def-use profiles.

In addition, we conducted simple experiment to evaluate the potential of bootstrapping to refine the classifiers built using the batch learning technique. The results of this experiment indicated that bootstrapping improves the classifier accuracy.

The work presented in this paper demonstrated that modeling certain event transitions as Markov processes produces effective predictors of paths feasibility that can be automatically clustered into paths feasibility classifiers. Also, it demonstrated how the bootstrapping active-learning technique can be employed to incrementally and efficiently improve these classifiers.

In the experiments, we have applied the proposed classifiers to small size programs, as a preliminary study. We intend to apply them to more realistic size programs. Also, we intend to study the use program features, other than branches, edges and def-uses, in building paths feasibility classifiers, such as paths of length two, Decision-to-Decision (DD) paths, and function calls. In addition, we intend to investigate how to combine these features to build more effective paths feasibility classifiers.

REFERENCES

[1]. Hedley, D., M. A. Hennell, The cause and effects of infeasible paths in computer programs, In the Proceedings of the 8th International Conference on Software Engineering, London, England, 1985, pp. 259-266.

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

[3]. Dickinson, W., 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), May 2001 p. 339–348.

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

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

[6]. Podgurski, A., 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), May 2003, p. 465–474.

[7]. Gross, K. C., 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), May 2003, p. 24–26.

[8]. Brun, Y. and M. D. Ernst, *Finding latent code errors via machine learning over program executions*, In Proceedings of the 26th International Conference on Software Engineering, 28 May 2004, Edinburgh, UK.

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

[10]. Bowring, J. F., J. M. Rehg, and M. J. Harrold, *Active learning for automatic classification of software behavior*, ACM SIGSOFT Software Engineering Notes, July 2004.

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

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

[13]. Haran, M., A. Karr, A. Orso, A. Porter, and A. Sanil, *Applying Classification Techniques to Remotely Collected Program Execution Data*, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, September 5–9, 2005, Lisbon, Portugal, p. 146-155.

[14]. Haran, M., A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche, *Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks*, IEEE Transactions on Software Engineering, May 2007, 33(5): p. 287-304.

[15]. Lo, D., H. Cheng, J. Han, S. Khoo, and C. Sun, *Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach*, ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), Research Collection School of Information Systems, 2009.

[16]. Francis, P., D. Leon, M. Minch, A. Podgurski, *Tree-Based Methods for Classifying Software Failures*, 15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 Nov. 2004, Saint-Malo, Bretagne, France.

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

[18]. Munson, J. C. 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]. Baskiotis, N., M. Sebag, M. Gaudel, S. Gouraud, *A Machine Learning Approach for Statistical Software Testing*, Twentieth International Joint Conference on Artificial Intelligence, Jan 2007, Hyderabad, India.

[20]. Baskiotis, N. and M. Sebag, *Structural Statistical Software Testing with Active Learning in a Graph*, Proceedings of the 17th international conference on Inductive logic programming, ILP'07, Corvallis, OR, USA, June 19 - 21, 2007, p. 49-62.

[21]. Girgis, M. R., *Using Symbolic Execution and Data Flow Criteria to Aid Test Data Selection*, Software Testing, Verification and Reliability, Vol. 3, p. 101-112, 1993.

[22]. Girgis, M. R., *An experimental evaluation of a symbolic execution system*, Software Engineering Journal, Vol. 7, No. 4, p. 285-290, 1992.

[23]. Rapps, S. and E. J. Weyuker, *Selecting software test data using data flow information*, IEEE Transactions on Software Engineering, 1985, 11(4): p. 367-375.

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