Stochastic Modeling in Software Testing

by

Seyedeh Sepideh Emam

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in
Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Using models in order to formalize and abstract the view of a system is a popular approach in different research areas. Deriving behavioral models from software executions is a common approach used in supporting a broad range of software development, maintenance, and verification and validation tasks. Behavioral models are useful tools in understanding how programs work. Although, several inference approaches have been introduced to generate Extended Finite State Automatons from software execution traces, they suffer from accuracy, flexibility and decidability issues.

In this study, we apply a hybrid technique, which uses both Reinforcement Learning and stochastic modelling to generate an Extended Probabilistic Finite State Automaton (called ReHMM) from software traces. Our approach is able to address the problems of inflexibility and un-decidability, reported in other state of the art approaches. Experimental results indicate that ReHMM outperforms other inference algorithms.

Moreover, dynamic specification mining of web applications is a helpful approach in observing program execution and generating a model of program behavior. However, it cannot efficiently support the identification of prevalent navigation patterns from a user's perspective. Inferring a user behavioral model from a history of users' interactions can assist in evaluating the users' "satisfaction level" with the application. Based upon this evaluation, such a model can provide insights into possible design and architectural anomalies and lead to the development of software solutions addressing the users' needs.

In this thesis, we also propose a hybrid approach to fully automate the behavioral model generation and a reward calculation process for user-intensive web applications. Our proposed

solution infers a reward augmented behavioral model by: (1) dynamically generating a set of probabilistic Markovian models from the interaction history; (2) annotating and analyzing the models to verify the quantitative properties; and (3) augmenting the model using a reinforcement learning method to assign reward values to the states of the model.

Additionally, we present a new extended digraph model as the basis of a novel fault-based test case prioritization technique to promote fault-revealing test cases in model-based testing (MBT) procedures. We seek to improve the fault detection rate- a measure of how fast a test suite is able to detect faults during testing – in scenarios such as regression testing. The model is realized using a Reinforcement Learning (RL) and Hidden Markov Model (HMM) based technique, which is able to prioritize test cases for regression testing objectives. We present a method to initialize and train a HMM based upon RL concepts applied to an application's digraph model. The model prioritizes test cases based upon forward probabilities. In addition, we also propose an alternative approach to prioritizing test cases according to the amount of change they cause in applications. To evaluate the effectiveness of the proposed techniques, we perform experiments on Graphical User Interface (GUI)-based applications and compare the results with state-of-the-art test case prioritization approaches. The experimental results show that the proposed technique is able to detect faults early within test runs.

And finally, since the automated test case generation is one of the main challenges in testing mobile applications, and this challenge becomes more complicated when the application under test supports motion-based events, we propose a novel, hidden Markov model (HMM)-based approach to automatically generate movement-based gestures in mobile applications.

An HMM classifier is used to generate movements, which mimic a user's behaviour in interacting with the application's User Interface (UI). We evaluate the proposed technique on three different case studies; the evaluation indicates that the technique not only generates realistic test cases, but also achieves better code coverage when compared to randomly generated test cases.

Preface

Chapter 2 of this thesis is submitted as S. Emam; J. Miller, “Inferring Extended Probabilistic Finite State Automaton Models from Software Executions”. Under the second round of revision at the ACM Transactions on Software Engineering and Methodology (TOSEM), 2017.

Chapter 3 of this thesis is submitted as S.Emam, S.S. Ghaemmaghami, J. Miller, “Inferring Reward Augmented Behavior Models from Log Files in Web Applications” Submitted to ACM Transactions on Internet Technology (TOIT), 2017. I was responsible for developing the idea, producing the models, analyzing the results, and the manuscript composition. J. Miller was the supervisory author and was involved with concept formation and manuscript composition. S.S Ghaemmaghami also contributed in manuscript composition.

Chapter 4 of this thesis has been published as S. Emam; J. Miller, “Test Case Prioritization Using Extended Digraphs” ACM Transactions on Software Engineering and Methodology (TOSEM). 25(1): 6:1-6:41, 2015.

Chapter 5 of this thesis is submitted as S. Emam; J. Miller, “Automated Testing of Motion-based Events in Mobile Application”. Under the second round of revision in the Journal of Software: Evolution and Process, 2017.

This thesis is dedicated to my husband, Ali and my parents, Shirin and Mehdi

For their endless love, support and encouragement.

Acknowledgments

First and foremost I would like to thank my supervisor, professor James Miller for his endless support, patience and encouragement during the years of my Ph.D. It has been an honor to be his Ph.D. student.

I would also like to thank my committee members Dr. Scott Dick, Dr. Marek Reformat, Dr. Eleni Stroulia and Dr. Lin Tan for their thoughtful and constructive comments and feedback.

Finally, I would like to thank my sister Elham and my brother Matin for their love and support.

Table of Contents

List of Tables

List of Figures

1 Introduction

This thesis is the collection of the research papers, which are produced during this PhD program. The list of papers is provided in the preface. In this section we focus on introducing the main goals and approaches, which are defined and developed in this research.

It is worth noting that in the first two papers (Chapters 2 and 3), we proposed new inference techniques to generate probabilistic behavioral models from software execution traces and log files, while in the rest of the thesis (Chapter 4 and 5), we focused on applying the behavioral models in software testing context.

1.1 Inferring Behavioral Models

The application of behavioral models in software analysis and testing activities includes efforts to complement available specification information [1], automate the acquisition of user interaction requirements [2], and the ability to generate test cases to detect program faults [3]–[6]. The large range of applications for behavioral models in software engineering has led to a multitude of researchers proposing several inference techniques. Many of these approaches infer the model in the form of a Finite State Machine (FSA) [7], [8]. While some other approaches augment the FSAs with transition probabilities or constraints, and produce Probabilistic FSAs (PFSAs) or Extended FSAs (EFSAs).

Although, augmenting the behavioral models with transition probabilities helps in generating behavioral models mimicking actual software characteristics, PFSAs are still missing algebraic or universally quantified guards associated with the transition labels and EFSAs are missing the transition probabilities.

On the other hand, specification mining of web applications is a helpful approach in generating a model of system behaviour. Unlike inferring EFSAs and PFSAs, which normally use software execution traces as inputs, behavioral models in web applications can be generated from the history of users' interactions.

Inferring user behavioral models in web applications provides information about hidden user behavioral patterns. Such information helps systems' experts to understand the clients' interests more thoroughly and design the web application in a way that fully addresses users' requirements. User behavioral models are also used in detecting the design anomalies in web applications. Finding the pages playing the role of deadlocks in the user interface design is another main achievement in inferring and analyzing the user-behavioral models [2-7].

1.2 Applying Stochastic Models in Test Case Generation and Prioritization

The complexity and size of software systems are growing; along with the increasing importance of testing and verifying these systems. As a result, many test suites produced during development are reused in a regression-testing mode especially during software maintenance or evolution. Decreasing regression-testing costs while increasing fault detection power are important goals in software testing; these challenges can potentially be addressed by Model-based testing (MBT) techniques [13], [14]. MBT has two phases: (1) the generation of executable test cases; and (2) the execution and evaluation of test cases [15].

However, stochastic modeling approaches are an efficient way to handle Model-based Testing. Different stochastic modelling techniques are developed and applied to address both test case generation and evaluation phases [10], [11], [16]–[18]. Markov chains and Hidden Markov

Models can be used to generate FSAs representing all possible execution traces in software or user-interaction scenarios in the web or mobile applications. Such traces and scenarios can be used later to generate test cases covering different functionalities in the Graphical User Interface (GUI) and the source code. Moreover, stochastic models can be used to select and prioritize test cases, which are more likely to detect faults during regression testing [15], [19], [20].

However, when it comes to testing mobile applications, MBT becomes more challenging due to the ongoing developments in the mobile industry. For instance, inferring models supporting motion-based scenarios and automatically generating executable motion-based gestures (test cases) is a new and challenging area in MBT. Proposing an approach to generate a dynamic behavioural model of a motion-based application and applying that in automatically generating test cases can provide a solution to the problem of testing such applications.

1.3 The Focus of This Research

In this thesis we identify and categorize the major issues and vulnerabilities in the existing model inference approaches. We develop and evaluate a novel hybrid approach using both Reinforcement Learning (RL) and stochastic modeling, which addresses these vulnerabilities and outperforms state-of-the-art approaches in terms of the inference accuracy. We also apply the RL-based modeling approach to generate and prioritize test cases.

1.3.1 Chapter 2: Inferring Extended Probabilistic Finite State Automaton Models from Software

The focus of the chapter 2 of this thesis is on proposing a new approach to generate the Extended Probabilistic Finite State Automaton (EPFSA) from software traces. Our new inference approach (ReHMM) addresses the problems of inflexibility and un-decidability, which had not been

addressed by state-of-the-art techniques. It utilizes both Reinforcement Learning (RL) and stochastic modelling concepts to generate models, which are not only labeled with the method-calls and guards but also are labeled with the transition probabilities. Applying ReHMM on several different software systems indicated that it outperforms other inference algorithms in terms of accuracy.

Chapter 2 of this dissertation is derived from an article submitted for publication:

- S. Emam; J. Miller, "Inferring Extended Probabilistic Finite State Automaton Models from Software Executions". Under revision at the ACM Transactions on Software Engineering and Methodology (TOSEM), 2017.

1.3.2 Chapter 3: Inferring Reward Augmented Behavior Models from Log Files in Web Applications

In chapter 3, we propose a hybrid approach of RL and Markovian process to fully automate the inferring procedure of reward-augmented behavioral models in web applications. In this approach, the states of the inferred behavioral model are automatically augmented with the reward values. Reward values provide information about the behavior of the users in corresponding states (pages) of the web application. Such models are generated using historical user-interaction log files. They are able to provide user behavioral information without instrumenting the source code or manually calculating the reward values. Experimental results indicate our proposed approach has comparable performance to Google Analytics.

Chapter 3 of this dissertation has been submitted for publication:

- S.Emam, S.S. Ghaemmaghami, J. Miller, "Inferring Reward Augmented Behavior Models from Log Files in Web Applications". ACM Transactions on Internet Technology (TOIT), 2017.

1.3.3 Chapter 4: Test Case Prioritization Using Extended Digraphs

A new MBT approach is provided to prioritize GUI-based test cases in chapter 4 of this thesis. This technique uses both RL and HMM concepts to generate an application's digraph model and apply the model in order to prioritize test cases based upon the forward probabilities. It utilizes both forward probabilities and accumulated Q-values to respectively calculate the occurrence likelihood of the sequence of events and the amount of computations triggered by executing such sequences in the GUI-based applications.

The proposed approach is used on Graphical User Interface (GUI)-based applications and the results are compared with state-of-the-art test case prioritization approaches in terms of fault detection rate. The results indicated that our prioritization approach outperforms other techniques (including Random, Best, Worst and Additional code coverage) in terms of APFD.

Chapter 4 of this thesis has been published as:

- S. Emam; J. Miller, " Test Case Prioritization Using Extended Digraphs" ACM Transactions on Software Engineering and Methodology (TOSEM). 25(1): 6:1-6:41, 2015.

1.3.4 Chapter 5: Automated Testing of Motion-based Events in Mobile Application

Finally, in chapter 5, we propose a new MBT approach to generate motion-based test cases in mobile applications. Again the considered technique uses both RL and HMM concepts and automatically generate test cases for mobile applications supporting motion-based events. Such mobile applications generate motion-based events using the data gathered by the accelerometers or by recording the touched points.

When the device is in the motion or its screen is continuously being touched, the probability of receiving unintentional inputs by the mobile application increases. Therefore, providing an approach, which is able to automatically generate test case mimicking actual human user motions would be helpful in not only testing the functionality of the applications but also in early detection of faults.

In this study, we evaluate our test generation technique on three different motion-based mobile applications. The experimental results indicate that the technique is able to generate test cases mimicking actual users' behaviors, while it achieves a better coverage compared to the random test cases.

Chapter 5 of this thesis is submitted for the second round of revision as:

- S. Emam; J. Miller, "Automated Testing of Motion-based Events in Mobile Application". Journal of Software: Evolution and Process, 2017.

2 Inferring Extended Probabilistic Finite State Automaton Models from Software Executions

2.1 Introduction

Most of behavioural model inference techniques are able to infer a model in the form of a Finite State Automaton (FSA). FSA is the model most commonly used by dynamic model inference approaches to demonstrate program behavior and to present information about the execution of event sequences [7], [8].

FSAs are able to provide information about the behavior of software systems using a set of states and transitions. However, because a FSA can only provide a partial view of the software, it does not accurately represent the software's behavior [21], [22]. To address this issue, many researchers have proposed a complementary model containing information about state variables. In the Extended version of a FSA (EFSA), transitions between states are associated with both labels and guards (constraints). A guard represents the conditions, which must hold with respect to the system's data–state variables (for the transition to be available). Transitions of an EFSA can be labeled with algebraic or universally quantified guards. The algebraic guards associated with transitions determine the concrete values that can be assigned to variables, while universally quantified constraints (guards) indicate how data values can reoccur across events.

EFSAs are able to solve many complex learning problems by blending the sequence of events or method calls with the values of the associated parameters. However, they are still missing a significant factor: ***Transition probabilities***. Transition probabilities in FSAs indicate the likelihood of a transition being traversed. Using a model containing the transitions probabilities, it is possible to generate a collection of actions mimicking certain characteristics of the actual

software. For example by utilizing Probabilistic FSAs (PFSAs), it would be possible to calculate the probability of calling a specific method in different states of the software and subsequently detect methods which may be called more frequently than others. Accessing such information helps in solving several software engineering problems:

- Finding hotspots [1] in software executions: virtually many software programs spend the most majority of their time executing a minority of their code. Therefore, by detecting the less-frequent methods in the software system, the compiler can focus the attention of a global native-code optimizer on the hot spots. By avoiding compilation of infrequently executed code, the Hotspot compiler can devote more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. However, less frequent methods can be detected by recognizing the transitions with low likelihood of being traversed in a corresponding PFSA.
- Finding errors and bugs: knowing the probability of a method to be called helps the software testers to focus the testing efforts on the behaviors that are more likely to happen in the software system. This helps in reducing the time, needed for regression testing by eliminating the test cases covering less-frequent functionalities.

In addition, it is proven that the problem of generating a perfect Probabilistic FSA (PFSA) is a decidable problem, whereas perfect learning of a FSA is not decidable. According to research conducted by Gold [23], when the learner has no preference in choosing a path, it is possible that the learner cycles through generating a never ending sequence of examples from the infinite language making the problem of learning a perfect FSA undecidable. However, Ammons et al. [24] illustrated that if the learner is provided with extra information (a probability distribution),

[1] http://www.oracle.com/technetwork/java/whitepaper-135217.html

allowing it to justify choosing a less general automaton over a more general one, the cycling issue can be avoided. In this study, we refer to the issue of a missing probability distribution as the missing ***state-action value*** issue, since it is solved by estimating the value of executing an action in the corresponding state.

In this study, we propose a new solution to address the missing state-action values' issue by representing a new stochastic inference approach called ReHMM.

ReHMM is a novel Extended Probabilistic FSA inference technique that builds the behavioral model incrementally, while walking through software traces from the system. ReHMM takes advantages of both EFSAs and PFSAs by inferring an Extended Probabilistic Finite State Automaton (EPFSA), where its nodes represent the software state and its edges carry method calls, data values, and transition probabilities, adding the state-action values to the model.

ReHMM infers an accurate EFSA from software traces while triggering relevant computations to navigate the model during the inference process. It uses a hybrid technique consisting of stochastic modelling (Hidden Markov models (HMMs)) and reinforcement learning (RL), in particular Q-learning, to infer an "improved" model. This improvement in behavioral-modeling performance is achieved by calculating a new value (called the Q-value) for each transition in the behavioral model and attaching it to the corresponding edge. In other words, Q-learning actively explores the execution space (execution traces) and calculates Q-values incrementally to solve the missing state-action values' issue.

In practice, HMMs are common types of stochastic FSAs, which are used in many sequential pattern processing, information extraction and classification problems (such as speech and handwritten task recognition [25], [26]). In this case, an HMM classification approach is applied, because the inferred models contain an underlying stochastic behavior among the software-

systems' states that is not observable, but affects the observed sequence of events (the execution of method calls). Therefore, as part of the inference algorithm a set of classifiers is generated each of which corresponds to a class label in the trace (e.g. the signature of a method in a trace from a program). Using such a set of traces, the classifier can predict the next event name in the trace (i.e. the name of the next method to be called).

On the other hand, the main reasons for choosing Reinforcement Learning are its strong statistical background, its proven ability in handling a wide range of data, and its ability to re-estimate a Markov model efficiently. Using RL, we are able to estimate the transition probabilities as part of the inference procedure [27].

Therefore, by using an HMM classifier and Q-learning as a hybrid approach, the model would be able to predict the next event name in the trace and subsequently, distinguish between transitions with the same labels but different corresponding Q-values. This also prevents the inference procedure from inappropriate merging of states and transitions.

This study contributes to the research in this area by:

- Utilizing Q-Learning as an incremental learning approach to discover functions, which trigger more computations during software executions, while inferring an EFSA from software executions and calculating the transitions probabilities in the inferred model;
- The adoption of an HMM classifier as a Markovian solution to the problem of generating non-deterministic and inaccurate models;
- Demonstrating the accuracy of the inferred model by performing an empirical evaluation of ReHMM on seven relatively diverse software systems; and

- Comparing the results obtained using the proposed method with those obtained by [28] and [29]. The results show that ReHMM outperforms the current state of the art technique in terms of model accuracy;
- Considering Q-values as a potential indication of the transitions probabilities in designing Extended Probabilistic FSAs.

This chapter is organized as follows. Section 2.2 provides related work and the detailed description of the state of the art approaches, which are used as baselines in this research (MINT data classifier and sk-strings algorithms). Section 2.3 contains research motivations, problem description and application of the proposed approach in software engineering. Section 2.4 contains the definitions and details regarding the inputs and domain of this research. This section also introduces a motivating example, which will be used throughout this chapter (specifically in Section 2.6) to provide a demonstration of the inference steps on a sample system. In Section 2.5, we provide background information and definitions relating to EFSAs, PFSAs, PTA**,** RL, transition functions and HMMs. Section 2.6 includes the design of the proposed techniques by describing the estimation procedure of the RL-based Hidden Markov Model (ReHMM) and inference of Extended PFSA. Section 2.7 describes the evaluation phase, research questions and comparison criteria. Section 2.8 provides the experimental setup, while Section 2.9 includes a discussion on the results, and an analysis of the empirical studies. Section 2.10 discusses the time-complexity analysis of the approach. Section 2.11 looks at the limitations and the threats to the validity of the study. Finally, in Section 2.12, we present our overall conclusions and some thoughts on potential future research.

2.2 Related Work

The challenge of producing FSAs from traces is not a new topic; several research projects have been undertaken which have provided a diverse set of inference approaches and algorithms. Many of these methods are general approaches focusing on the efficient generation of automatons as abstract machines regardless of their application in software engineering [30]–[32]. For example, Biermann and Feldman in [7] present a modified method to synthesize machines from finite subsets of their behavior.

Since, our work builds upon early work in inferring Extended FSAs, in this section, we consider related studies on specification mining in the software engineering context, their improvements in generating stochastic models and their evaluation approaches.

2.2.1 Specification Mining

Many studies have been conducted on automaton-based specification mining [33], [34]. Bartussek et al. [35] presented a new approach to use a set of assertions about input traces and utilize them to generate an abstract specification of different software modules. However, trace-assertions are used in other studies as well to specify software modules [36]–[38]. Additionally, Ammons et al. [24] provided a pioneering approach to mining the specification of method calls extracted from software execution traces. Their proposed method infers a model by observing program executions and concisely summarizing frequent interaction patterns as state machines. The generated FSA is able of capturing both the temporal and data dependencies and learn a probabilistic FSA.

Moreover, Krka et al [39], proposed three novel algorithms CONTRACTOR++, SEKT and TEMI, which combine execution traces with automatically inferred program-state invariants to

mine EFSA-based specifications. It is worth noting that some of execution traces used in this study are the same as the traces used in [18].

In addition to this, there are other studies, which have provided an approach to infer FSAs using graph transformation rules [40] or suggested a method to mine traces for a set of pre-defined micro-patterns and then merge them into a FSA [41]. It is worth noting that that the solutions introduced in these studies are not able to mine specifications into other formats (such as regular expression, Communicating FSAs (CFSAs) [34] or EFSAs.) rather than FSAs.

State of the Art EFSA Inference Approaches: GK-tail [3], kLFA [22], ADABU [42] are other pioneer techniques utilizing data flow information to extract EFSAs. We describe each of these inference techniques in the following paragraphs:

Dallmeier et al. in [42] suggests a technique to construct a state machine that would summarize object behavior. They It generates a model by obtaining a detailed record of the data states at each point of the program execution. It finally builds a model that adds data values to FSA models but these are not transition guards.

Another state of the art EFSA inference method is GK-tail. GK-tail is an extension of the traditional kTail algorithm [7] and an automated inference technique used to generate EFSAs from a set of interaction traces [3]. This technique augments the automata with constraints both on transitions and event sequences. To process traces and infer a behavioral model, GK-tail merges similar traces, derives constraints from the values associated with each event, and then builds an initial EFSA from the set of traces annotated with the constraints. Finally, GK-tail iteratively merges states with the same future of length k. The future of length (k) of a state is the set of event sequences with maximum length (k) that can be triggered from the state. It is worth noting that GK-tail uses Daikon [4] to infer rules for each transition.

Unlike GK-tail, which uses the values already assigned to the attributes, kLFA uses a different approach and focuses on universally quantified constraints representing an occurrence pattern of values across events. Initially, kLFA analyzes the interaction traces and extracts the universally quantified constraints, which indicates how data values reoccur across events [22]. It then rewrites traces by replacing concrete values with symbols representing the discovered patterns. Finally, kLFA infers an EFSA from the rewritten traces which incorporate data flow information [43].

Stochastic FSAs: The challenge of improving FSAs using stochastic procedures and probabilistic values dates back to 1963 when Michael et al. [44] presented the probabilistic FSAs as an stronger version of deterministic automata. Also in the context of software engineering, many studies have focused on conventional Probabilistic FSA generation processes [45], [46]. However, using the probabilistic FSA as a learner to mine the specification of method calls from software executions represents another challenge to this research area, which requires more attention and development. As mentioned earlier, Ammons et al. [24] pioneered to mine specifications using a probabilistic FSAs in 2002. Later, Lo et al. [47], represent an API specification mining architecture called SMArTIC to improve the robustness of the specification mining process by learning PFSAs instead of traditional FSAs. They also provide an approach to structure the mining procedure by filtering and clustering dynamic execution traces as an important piece of their proposed framework. However, the learner block in the SMArTIC framework plays the role of a placeholder, which means that different PFSA specification miners can be placed into this block. Authors have implemented the sk-strings learner [48] to learn the PFSAs in their study.

In this study, we also tried to address some of the potential specification mining issues by taking the advantages of stochastic FSAs.

2.2.2 Evaluation of Inferred models

Another interesting research area in specification mining is the evaluation of miners. Several approaches are provided to evaluate the quality of mined specification. For instance, Lo et al. [49] propose a framework called Quark for empirically assessing the automata generated by different miners. Their method generates traces from a given automaton and then uses these simulated traces to train the specification miner. The original model and the mined one are then used to compute precision, recall and probability similarity (for PFSA learners) measures to evaluate how accurately a miner summarizes the provided traces into a model. The same metrics are used in this study.

Pradel et al [50] also propose another framework to evaluate the entire mining process. Their approach accounts for imprecision and incompleteness in the mined specifications, which can help in detecting the similarities between FSAs. In addition, there are several other approaches proposing different metrics and measures for comparing and evaluating FSAs [3], [51]–[53]. However, none of them provide a specific and customized measure to evaluate the probability similarity (PS) between Probabilistic models.

2.2.3 Baseline Inference Approaches

This section presents an overview of other inference techniques, which are used as baselines in this research.

Many FSA and EFSA inference approaches (such as kTail) fit into the family of "state merging" algorithms. In such algorithms, a PTA (initial FSA) is inferred from a set of traces; then subsequently in the merging step, two states of the PTA are selected to be merged. Several diverse algorithms are provided to improve the performance of the model by modifying the merging step. On the other hand, some new approaches are also developed on top of the established state-merging algorithms to improve the overall performance of the final inferred models by generating more specific PTAs. The data classifier inference techniques can be considered as a class of such approaches, which are able to identify rules and patterns between variables from a set of traces and map these variables to a class label. This procedure helps in predicting the next class of unseen variables and generating the most specific PTA representing the given set of traces. [28], [54] represent a new data classifier inference technique called MINT to address the non-determinism issue in the previous state of the art EFSA inference techniques, such as GK-tail. In this study, MINT is used as a baseline to define our proposed approach (ReHMM). ReHMM also can be considered as a data classifier approach, which is built on top of MINT. Since, MINT uses the same merging strategy as GK-tail's in the state-merging step, it makes more sense to consider and compare ReHMM with MINT instead of GK-tail.

Moreover, MINT provides a proven improvement [29] in inferring rules compared with Daikon [4] – a data-constraint inference tool used in GK-tail. Also, the idea of inferring a set of global rules belonging to the full set of traces represented by Krka et al. [39] and Le et al. [55] is very similar to the method of inferring data models in MINT. Hence, even if the purpose of the models inferred by [39] and MINT are different, the nature of the models are the same, except that MINT provides more flexibility by incorporating data classifier inference techniques. Accordingly, again, we believe that comparing ReHMM with MINT can also demonstrate the

position of ReHMM with regard to other state of the art techniques, as these techniques are either (a) a subset of MINT; and/or (b) have been shown to be outperformed by MINT [29].

MINT Data Classifier Inference Approach: The MINT algorithm consists of the five following steps:

- ***Processing traces to create data traces.*** Given a trace (Tr) which consists of a variable domain V along with a set of transition labels L, a data trace can be defined as the set $TC = \{(e_0, c_0), \dots, (e_n, c_n)\}$, where $e = (l, v)\,, l \in L \; and \; v \in V$, and each variable e is associated with its corresponding class label $c_i \in C$. In the EFSA inference procedure, **class label $\mathbf{c_i}$ commonly represents a categorical outcome such as an event name or a method call**. It is worth noting that extracting class labels does not need any specific knowledge about the software system and can be easily done by parsing the traces and extracting the method, which will be called after each event.
- ***Inference of data classifiers.*** Classifiers are used to predict the next class label in the trace; **a class label is the method call**. The classifier uses a set of inputs (training sets) to simply predicting the next method call. More formally, they use training set to map the variable to their respective classes. Most standard classifiers in the WEKA library[2] can be used to carry out this step [56]. The classifiers used by [28] are: NNGE, Bayes (in this study when we refer to the Bayes algorithm we mean Naïve Bayes) , JRIP, AdaBoost and J48. We applied the same classifiers in this study for the sake of comparison.

[2] http://www.cs.waikato.ac.nz/ml/weka/

- ***Producing the Prefix Tree Acceptor (PTA).*** In this step, a PTA is built by extracting all the prefixes from the traces, where traces with the same prefix will share the same path from the initial state of the FSA until the point where they diverge. A technical definition of a PTA is provided in Section 2.5.1.
- ***Merging.*** Pairs of states are suitable for merging when the number of transitions in their outgoing path (merging score), their labels and their data values are equal. In this situation, both the generated PTA and the data classifiers are used to interactively detect candidates, merge them and then relocate the data values from the source to the target transitions. In addition, G is an optional parameter to represent the *minimum* merging score before a pair of states can be deemed to be equivalent. For instance, if G=1 there must be at least one suffix that is the same for two states to be merged. The suffix contains both labels and data values in the outgoing paths of the candidate states. In this study, we vary G from 0 (which only considers data) to 1 (which not only relies on data but also needs (at least) one common suffix for two states to be merged). Similar merging procedures have been used in other state of the art approaches [3], [22], [42].
- ***Checking the consistency.*** This step checks that the data variables attached during the merging process are consistent with the classifiers. This ensures that the inferred model is deterministic by comparing both the labels and the attached data of the outgoing transitions of the given states, and recursively merges the states that can cause non-determinism. This procedure not only leads to generating a consistent model but it is also beneficial in terms of the computational cost.

In this study, we also infer PFSAs, however we extend the definition of a PFSA beyond its norm (Definition 4). In this work, our PFSA model adds the probabilities into Extended FSAs, so it has the features and benefits of both PFSAs and EFSAs (Definition 9 in Section 2.6).

sk-Strings: In order to compare the inferred PFSA generated using ReHMM with a state of the art probabilistic model in terms of accuracy, we implement the ***sk-strings*** PFSA learner algorithm on the same software systems and traces as another baseline approach. The sk-strings algorithm is an extension of K-tail approach introduced in 1972 for stochastic automata [24], [32], [48]. sk-strings creates a PTA from software execution traces (similar to K-tail) and labels each edge by (1) an event name; and (2) a probability showing how often the edge is traversed. Initial probabilities are distributed equally to transitions from the same source node. Therefore, given a PTA, a node q, the alphabet Σ, a set of final states (the leaves of the PTA) F_c, and a transition function δ, the set of k-strings associated with the node q ($k - string(q)$) is defined to be the set:

$$\{d | d \in \Sigma, |d| = k, \delta(q,d) \subset Q \vee |d| < \delta(q,d) \cap F_c \neq \emptyset\} \qquad (1)$$

In the merging step, two merging candidates can be merged if they are indistinguishable with respect to the top d% of the most probable strings of length k that can be generated from these states [49]. It is worth noting that, k-strings ends at a finish state if d is shorter than the specified string size ($\delta(q,d) \cap F_c$). Raman et al. in [48] provide a set of different heuristics with various degrees of strictness (containing the OR heuristic, the AND heuristic, the LAX heuristic and the STRICT heuristic). In this study, the AND variant of the algorithm is used as per the advice from Lo and Khoo [49]. Algorithms 1 and 2 elaborate upon these implemented procedures [48]. Algorithm provided in Table 2, makes decisions about the equivalency of states using the AND heuristics, then the algorithm provided in Table 1 merges states of the PTA which are equivalent.

Table 1. sk-strings Algorithm

Input: PTA

Output: Generalized Automata

1. begin
2. **For each** state $p = s_0$ **To** s_{N-1} **do**
3. **For** state $q = p$ **To** s_{N-1} **do**
4. **if** $*\ sk - equivalent(p, q, *\ G)$ **then**
5. merge(p,q)
6. $p = s_0$
7. $q = s_{N-1}$
8. **Repeat Until** no new $p, q \in S$ is found
9. end

* sk-equivalence using AND heuristic

* G is the minimum merging score

Table 2. sk-equivalence using AND heuristic

Input: States p and q

Output: TRUE if p and q are sk-equivalent, FALSE otherwise

1. begin
2. $counter = 0;\ i = 0$
3. S_p = (str,prob) pairs output from p
4. S_q = (str,prob) pairs output from q
5. DesendingSort($S_p.probabilities$)
6. DesendingSort($S_q.probabilities$)
7. **for** $i = 1$ **to** num_strings_in(S_p) **do**
8. $cnt = cnt + S_p\ [i].probability$
9. **if** not_acceptable(q, $S_p\ [i].string$,G)
10. **then** return(FALSE)
11. **if** $cnt >= *\ Agreement\%$
12. then cnt = 0
13. **for** $i = 1$ **to** num_strings_in(S_q) **do**
14. $cnt = cnt + S_q\ [i].probability$
15. **if** notacceptable_at(p, $S_q\ [i].string, G$)
16. **then** return(FALSE)
17. **if** $cnt > Agreement\%$
18. **then** return(TRUE)
19. **return**(TRUE)
20. return(FALSE)
21. end

* Agreement% is a global value which equals parameter d to sk-string

In order to be able to provide more meaningful results, we also configured the sk-strings algorithm to consider the minimum merge score G in the merging procedure. This means that the merging candidates are indistinguishable when they are equal in terms of labels, data values and the number of transitions in their outgoing path.

2.3 Research Motivation

The area of inferring EFSAs is wide. Several different approaches have been proposed and implemented to infer Extended FSAs from software executions. However, GK-tail [3] and ADABU [42] are two notable approaches successfully implemented and evaluated on Java programs. Although these techniques are successful in addressing certain tasks, they suffer from two key drawbacks:

1. Lack of flexibility: both of these approaches are tied to a specific form of data-abstraction in order to map concrete values onto abstract rules (more detailed information about these two inference approaches is provided in Section 2.2). A data abstraction technique cannot necessarily be applied on diverse sets of software traces according to the different characteristics of the system in terms of the trace size, event diversity and variable type.
2. Non-determinism: EFSAs inferred by ADABU [42], GK-tail and kLFA [21], [22], [42] represent data flow information through algebraic and universally quantified constraints. However, during the inference procedure, the connection or link between these data constraints and events are ignored, as there are several possible paths to take for a specific data-state. This issue causes non-determinism. As a consequence of non-determinism, the generated model fails to consider the explicit logical relationships between sequences of events and the data. Subsequently, the model may fail in making

similar decisions with regard to accepting or rejecting a single sequence of events on different occasions. For instance, in a situation, where a non-deterministic model is used for generating test cases, the number of required test cases to cover a "reasonable amount" of the code will be significantly increased [57].

Also, in the situation where the detection of the correct behavior of the system is a predefined goal [29], the model must be deterministic and consistent. However, the time required to convert a Non-Deterministic FSA (NFSA) to a Deterministic FSA (DFSA), ($O(2^n)$, where n is the number of nodes) grows exponentially. This demonstrates the significance of applying an inference technique, which is able to detect the non-deterministic transitions and avoid them during the inference process instead of generating an NFSA and then converting it into a DFSA. It is worth noting that our proposed approach (see Section 2.10 for details about the time complexity analysis of ReHMM) automatically avoids non-determinisms within the merging step, it costs less than implementing other algorithms to detect and fix the non-deterministic issues.

2.3.1 Missing State-Action Values

Although Walkinshaw et al. [28], [29] addresses non-determinism and inflexibility issues present in the previous state of the art approaches by providing a new inference technique called MINT, it still suffers from a significant drawback. MINT is unable to address the issue of missing state-action values (transition probabilities) in the inferred EFSA.

In order to address this issue, we have used a stochastic-based approach to define an extended version of the state-action function and subsequently generate an Extended Probabilistic FSA. Probabilistic FSAs provide more control over the trace generation and verification process using

the attached probabilities, so that the collection of generated traces mimics the characteristics of traces collected from actual software executions [24], [49]. For example, methods, which are called during a function execution and have not been met before, are more likely to be traversed in the model immediately after the parent method compared to their siblings. Moreover, the methods, which appear more frequently in software executions, can be represented in the generated traces through appropriate probabilities at different transitions. Therefore, assigning higher probabilities to the children, which are either new or more different (in terms of the functionality that they are executing) compared with the currently traversed states can lead to the generation of more accurate models [24]. In addition, generating PFSAs are more expressive than FSAs, since they provide details on the probabilities of state transitions [47].

On the other hand, it has been proven that perfect learning of a FSA or EFSA from software traces is not a decidable problem; while, it is possible to learn a perfect PFSA from traces. Hence, PFSAs have been used as an intermediate step to learn FSAs [24] or as an independent approach to learn user-behavioral models in several research studies (e.g. [2]).

However, none of these approaches have applied an HMM classifier and Q-learning as a hybrid approach in calculating the transition probabilities (state-action values) and improving the performance of EFSAs.

Walkinshaw et al. [58] represents the state transition function for reverse engineering transitions from the source code in order to identify branches that are responsible for the execution of a specific transition. The state transition function (S) can simply be defined as: $s_1 \xrightarrow{f(x)} s_2,\ (s_1, s_2 \in S)$ which maps the transition onto the source code of method f(x), when f(x) is executed in state s_1 resulting in a transfer to state s_2 [58]. This definition helps us to define a new state-action function, which can be applied in the procedure of inferring EPFSAs from software traces. The

original version of the state transition function suggested by Walkinshaw et al. [58] cannot be directly applied in this study for the following reasons:

1. ***Limitation to Specific Programming Languages.*** In order to use the inferred EPFSA to address the software engineering problems (e.g. generating test cases using software model), the inference procedure should not be limited to any programming language.
2. ***Dependency on the Source Code***. The State transition function relies completely on source code analysis, so cannot be applied in the absence of program source code. While, in the model inference procedure, the execution traces are the only inputs of the EPFSA inference algorithm.
3. ***Independent Analysis.*** This approach is able to find the actual transition functions that transform the data-state variables at each transition. This means that the state transition function, for $s_1 \xrightarrow{f(x)} s_2$ depends only on the execution context represented by state s_1. Therefore, the previously traversed contexts are not considered, while, the relationship between the current state and the previous states plays an essential role in generating accurate EPFSAs from software traces.

Although the state transition function defined in [9] is able to identify path(s) in the source code which govern a transition, it is not designed to be used in the model inference procedure and accordingly cannot be helpful in addressing model inference issues such as improving the accuracy of the model or increasing its capability for software testing purposes (such as inferring a model which can be used to generate test cases with high code coverage level or fault detection rate).

Therefore, in this study, we suggest a new approach to evaluate transitions in the model and assign a probabilistic value to each transition based upon the utility of taking a given action in a specific state. This procedure is used to produce a PFSA with additional information about the values of associated parameters (transition probabilities) in each transition. So, the proposed approach is able to address both EFSAs and state-action functions drawbacks.

2.3.2 Application of ReHMM in Software Engineering

ReHMM, as an EPFSA inference approach has the capability of being applied in several software engineering tasks and activities such as refactoring, requirement engineering and software testing.

Missing a complete and formal requirement specification document is a common problem faced by software engineers. Short delivery time and project scope changes easily lead to forgetting about the specification documents or creating incomplete documents. In order to address this issue, several researches have been conducted to reverse-engineer or mine the software specification using dynamic analysis techniques. For example, QUARK is a framework analysing the quality of generated specifications by producing quality assurance measures on the specifications generated by the miners. In this study, the specification produced using the miners are expressed as automata (PFSAs). Since (1) ReHMM has gone through the same quality assurance procedure (e.g. calculating PS measure- see Section 2.7.1) as QUARK and (2) the empirical results confirm its capability of being used as a high quality PFSA inference technique, ReHMM can also be applied as a specification miner tool, when suitable execution traces are provided as input.

Moreover, PFSAs are successfully used in refactoring and customizing software applications by capturing and analyzing user behaviors. Ghezzi et al. [2] have provided a framework, called BEAR, to mine behavioral models from user traces generated by interacting with modern web applications. BEAR infers the model based upon a set of Markov models generated using historical log files. Applying ReHMM in the same research area can be helpful in maintaining and adapting existing user-intensive Web applications by inferring the models which are not only able to capture user behaviors probabilistically, but are also flexible and deterministic.

In addition to this, Fraser et al. [59] have developed a test assessment and generation tool, called BESTEST, based upon EFSA inference approach proposed by Walkinshaw et al. [29]. BESTEST is designed to determine a finite test set that adequately detects fault prone behavior of software systems. The empirical results demonstrate that test sets with higher behavioral coverage outperform other test cases in terms of fault detection. Accordingly, ReHMM also can be used to infer an accurate EPFSA from test executions to not only identify test sets with high fault detection rate, but also to provide additional information regarding the probability of occurring faults in different software behaviors (a new type of operational profile).

FSAs and EFSAs are also successfully used to address regression-testing tasks – another area of software engineering. Since regression testing happens when a software component is updated or replaced, missing test cases covering the old software system makes the regression testing process challenging. Also, missing a document specifying the software behavior, makes it difficult to identify the newly added behaviors and components of the software which are already tested. In the absence of regression test cases, the only available data is execution traces or server-logs of the previous version. Behavioral models are very helpful in generating, selecting and minimizing regression test sets. Both FSAs and EFSAs [60], [61] are used to demonstrate

the behavioral model of the system using traces (or log files). Subsequently, the model can be used to generate and optimize regression test sets. In this study, we demonstrate that ReHMM provides a richer model than state of the art EFSA and PFSA inference tools, it is expected that richer models can produce more rigorous test sets [29]. Since ReHMM, is able to generate EPFSAs from software execution traces, it also could be applied in addressing regression-testing problems.

The use of benchmarks also has become very common in empirical software engineering research. According to Walkinshaw et al. [62]: “a benchmark consists of a collection of subject data that can be consumed by different techniques, and can be used to draw coherent and valid conclusions about the respective performance of these techniques”. Therefore, the ReHMM inference framework also can be considered as a form of benchmark, since it compares the inferred EPFSA with other models from relevant techniques in terms of well-defined measurements of performance. All of these models are inferred using execution traces from real software systems.

In order to apply ReHMM in addressing real world software engineering problems, no specific experience is required. For example, a software engineer or the system expert needs only to provide the algorithm with sets of test cases of the software under test in the format that can be read by the algorithm. Then the engineer traverses the inferred model to generate an adequate set of test cases.

Now, in order to illustrate the proposed method, we firstly define the inputs and domain of operation for such a system.

2.4 Inputs and Domain

As mentioned earlier, in this study, we propose a new inference approach (ReHMM) to produce Extended Probabilistic FSAs from software executions (traces). Therefore, a collection of traces of a software execution is required as an input of the algorithm. Input traces consist of sequence of events (method calls) and corresponding variable values. It should be noted that in this study we use "class labels", "method-calls" and "sequence of events" interchangeably. The traces used in this study follow the same definition as provided by [35] and should not be confused with the trace definition used in the theory of concurrent systems [63]. However, the encoding process of traces is identical to the definitions provided by [24], [62]. We also presume that the interactions with the system can be illustrated in terms of particular function names and associated parameter values. The following provides a technical definition of the traces that have been utilized in this study. It is worth noting that the existed input traces are processed in an incremental way to generate the model. Moreover, the proposed learning approach can proactively learn from new traces which, allows progressive and automatic analysis of large and complex models as soon as a new version of the software become available.

Definition 1. **Traces.** [29] defines a trace $Tr = \langle e_0, \dots, e_n \rangle$ as a sequence of n trace elements (called events). Each element e_i, maps to a pair (l, v), where l denotes a transition label showing a function or method name, and v is a collection of parameter values for function l.

For instance, (low_water, (28.898501010661718 584.0357656062484 true)) is an event of a trace retrieved from the provided running example (Section 2.4.1) [29], which shows the transition label (l): low-water and parameter values (v): (28.898501010661718 584.0357656062484 true).

Following [29], a positive trace represents a feasible set of events in the system, while a negative trace indicates an impossible behavior of the system.

Therefore, according to the above definition, the input of the inference process has following characteristics:

1. ***Language independency:*** events can be retrieved from interactions with any software systems written in any programming languages. Therefore, the inference process does not rely on the source code analysis, and can be applied on entirely different software systems.
2. ***No need for additional specifications:*** the input traces are self-explanatory and sufficient to infer the model, so there is no need for additional information such as scenario-seeds (interaction skeleton) [24] and a program specification to extract the scenario from the traces. This implies that as the traces are naturally occurring and the process requires no additional artifacts; the process can be implemented at zero cost to a software engineering organization.

Having the traces, ReHMM is able to generate a probabilistic EFSA, which is not only able to address the issues raised in the Research Motivation Section (Section 2.3) and Software Engineering applications (Section 2.3.1), but also outperforms state of the art techniques in terms of following aspects:

1. ***Accuracy***: Empirical evaluations (see Section 2.7 for more details) show that the inferred models using ReHMM outperform the EFSAs generated using other state of the art techniques in terms of accuracy. Here, an accurate model is the one, which is able to correctly accept the positive traces and reject the negative ones.

2. ***Time complexity*** of inferring the model using our proposed method is polynomial and does not exponentially grow by increasing the system size (See 2.10 for more details). This shows that the significant improvement of the model is done without negatively affecting the time complexity of the inference algorithm. This ensures that the approach can be applied to large software systems.

In the next section, in order to motivate this study and also clarify the proposed inference procedure, we consider a small example of the behavior of a mine pump controller [28], [29]. This example will be used as a running example throughout this chapter.

2.4.1 Motivating Example

A pump controller controls the water and methane levels in the mine. The pump is activated or deactivated (based upon the level of the water in the mine) or switched off (when the methane level is too high). The snippet of an initial trace is shown below (each event in the trace is shown in a separate line).

…

trace

turn_on 73.44274560979447 596.7792240239261 false

low_water 28.898501010661718 584.0357656062484 true

switch_pump_off 28.898501010661718 584.0357656062484 true

turn_off 28.898501010661718 584.0357656062484 true

highwater 31.47476437422098 588.4568312662454 false

switch_pump_on 31.47476437422098 588.4568312662454 false

critical 35.04693535326364 603.1076440245823 true

...

The term of “initial trace” is only used to clarify that the trace has not been processed yet and has the same structure as defined in Definition 1. In Section 2.6, we illustrate how data traces are generated from these initial traces.

In this study, we are trying to provide an approach to infer an accurate behavioral model of the system using such traces. In the next section, we provide some technical background and definitions, before describing our proposed approach in detail.

2.5 Technical Background and Definitions

In order to provide a view of the software behavior, EFSAs, PTAs and PFSAs are defined and inferred. PTAs are the initial FSAs, which are generated by taking all prefixes of the input traces as states and generating the smallest FSA, which is a tree. EFSAs demonstrate the behavior of a software system by depicting the relationship between the method calls and the values of the associated parameters. While, PFSAs are FSAs augmented by transition probabilities. In this section, we represent the technical definitions of PTA, EFSA and PFSA.

Additionally, the proposed inference technique in this study is established by combining Reinforcement Learning (RL) and Hidden Markov Model (HMM) concepts to accurately generating an Extended PFSA. In order to walk through the inferring process, a brief technical background about Reinforcement Learning and Hidden Markov Models is also provided. It is worth noting that, approaches to EFSA and PFSA generation, which fall outside of these definitions, are considered to be beyond the scope of this study.

2.5.1 Prefix Tree Acceptor (PTA)

A PTA is a tree-shape automaton, generated as an initial FSA or a simulator model in several inference processes [24], [49], [62], [64]. For instance, as mentioned in Section 2.3, PTAs are generated as an initial form of FSAs in the MINT inference tool [29]. In this research, we also infer a PTA using the same approach that is used in other studies.

Definition 2. **Prefix Tree Acceptor (PTA).** Where Tr is a Trace and TS is a set of Traces, Given $Tr \in TS$, if $Tr = mn$ and $m, n \in TS$, then m is called a prefix. A PTA is a tree-like FSA generated by taking all the possible prefixes in the trace as states and constructing a FSA, which only accepts the traces it is built from. Let TS be the set of Traces from which we build a PTA. $PTA(TS)$ is a FSA that contains a path from the initial state to a final state, for each and every $Tr \in TS$.

2.5.2 Extended Finite State Automaton

Definition 3. **Extended Finite State Automaton (EFSA).** Again according to [29] and [65], an EFSA is a tuple $(S, s_0, F, L, V, \Delta, X)$, where S is a set of states and $s_0 \in S$ indicates the initial state, $F \subset S$ is a set of final states, L is defined as the set of labels ($l \in L$) and V represents the set of parameter values, where $v \in V$ is a concrete value assigned to a variable. V is also known as the memory of the EFSA. Moreover, Δ, the update function, is the function $L \times V \rightarrow V$. Finally, X is the set of transitions taking the form $(s_1, l, v, s_2), where\ s_1, s_2 \in S$. EFSAs explicitly allow transitions from state s_i to state s_i.

Therefore, transitions between states are not only associated with a label $l \in L$, but are also associated with a guard that represents the conditions that must hold with respect to the variables in the EFSA memory.

2.5.3 Probabilistic Finite State Automaton (PFSA)

There are various definitions regarding PFSA in the literature. In this study, we chose a definition, which is sufficiently general to cover most PFSA- related situations [45].

Definition 4. **Probabilistic Finite State Automaton (PFSA).** *A PFSA is a tuple* $D = (S, \Sigma, \delta_D, I_D, Q_D, F_D)$*, where*

- S is a finite set of states;
- Σ is the alphabet;
- $\delta_D \subseteq S \times \Sigma \times S$ is a set of transitions;
- $I_D: S \rightarrow \mathbb{R}^+$ (Initial-state probabilities);
- $Q_D: \delta_D \rightarrow \mathbb{R}^+$ ($q \in Q_D$; Transition probabilities);
- $F_D: S \rightarrow \mathbb{R}^+$ (Final-state probabilities);
- I_D, Q_D and F_D are functions such that:

$$\sum_{s \in S} I_D(s) = 1 \qquad (2)$$

and

$$\forall s \in S, F_D(s) + \sum_{a \in \Sigma, s' \in S} Q_D(s, a, s') = 1 \qquad (3)$$

It is worth noting that probabilities can be null, and hence, functions (I_D, Q_D and F_D) can be considered as total. Again, we limit the scope of the study to inference techniques, which produce results, which correspond to the above definitions of EFSA and PFSA.

2.5.4 Reinforcement Learning (RL)

This section provides an overview of RL and its definition. RL is an area of machine learning which is concerned with the problem of utilizing a software agent[3] to perform actions, which maximize the overall reward [66]. The reward is a number (score), which indicates the immediate utility of an action [67]. The progress of RL algorithms is typically iterative. The agent learns during different iterations by observing the current environment, inferring the environment's state and executing an action. This guides the agent to the next state. In other words, the agent receives the system's state and the reward score associated with the last transition. Then it evaluates the value of the action according to the reward it has gained, and subsequently, selects an action and sends it back to the system. In response, the system makes a transition to a new state; and this cycle is repeated as part of a Markov Decision Process (MDP) [66], [68]. MDPs can be categorized as stochastic extensions of finite automata or Markovian processes which are augmented by actions and rewards so that they consist of actions, transitions and states. In the following paragraphs, some definitions are introduced, which are helpful in demonstrating our proposed approach in the next section. We start by providing the definition of a Markovian system and the reward function to indicate the logic behind the decision making process in our proposed approach.

[3] "A software agent is a persistent, goal-oriented computer program that reacts to its environment and runs without continuous direct supervision to perform some function for an end user or another program" [249]

Definition 5. **Markovian System.** A system can be defined as Markovian if the execution of an action does not depend on previous actions and visited states (i.e. it depends only on the current state and status).

In other words, an MDP contains:

- A finite set of states $S = \{s_1, s_2, \dots, s_N\}$, where N is the number of states;
- A finite set of actions $A = \{a_1, a_2, \dots, a_k\}$, where k is the size of the action space; and
- The transition function $X: S \times A \times S \rightarrow [0,1]$ which computes the probability of reaching the state s' by performing action a in state s and is denoted as $X(s, a, s')$.

Finally, to compare different states and actions during agent and environment interaction, they should be ordered according to the time at which they occur. So s_t denotes the state at time t [68]; according to this definition of a Markovian process, we would have:

$$P(s_{t+1}|s_t, s_{t-1}, s_{t-2}, \dots) = P(s_{t+1}|s_t) = X(s_t, a_t, s_{t+1}) \qquad (4)$$

Definition 6. **Reward Function.** Function $R: S \times A \times S \rightarrow \mathbb{R}$ maps each perceived state (or state-action pair) of the environment to a score (reward), indicating the desirability of that state. A reinforcement learning agent's goal is to maximize the total reward it receives in the long run.

The reward function computes the immediate utility of an action to define the model of the MDP. So a MDP can be denoted by the tuple $\langle S, A, X, R \rangle$, depicting it as a state transition graph [68]. Depending on the definition of the problem we are trying to solve, different heuristics could be suggested to identify the difference between actions and calculate the reward values. In section 2.6 we discuss the heuristics used in this study to define an accurate and customized reward function.

Now, we use the definition of the state transition function as a basis to define the "state-value function". Then, in order to address the missing state-action value issue, the state-value function is used to provide a method to calculate the value of performing an action and to assign Q-values to the model's transitions.

Definition 7. **State Transition Function**. The new state transition function F, for a state transition $s_1 \xrightarrow{F} s_2$ $(s_1, s_2 \in S)$ not only maps the transition to the corresponding method (event) $a \in A$ [58], but also maps it to $\mathbb{R}$; where the value of executing a method call a, in state s_1, is computed using the state-value function $Q: S \times A \rightarrow \mathbb{R}$.

Accordingly, the state-action value function Q is defined in following paragraph.

Definition 8. **State-Value Function.** The Value Function $V^{\pi}(s)$, specifies "how good" it is for the agent to be in a given state. The "how good" notation here is expressed in terms of the future rewards that can be expected. We can define the value of a state under a policy π, formally $V^{\pi}(s)$, as [67]:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\left\{ \sum_{k=0, r_{t+k} \in R}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \qquad (5)$$

Where:

The stochastic policy $\pi: S \times A \rightarrow [0,1]$ is a mapping from each state s and action a to the probability $\pi(s, a)$ by performing an action a when in state s. E_{π} is the expected return earned by following policy π, and the discount factor γ, $0 \leq \gamma < 1$, models the fact that future rewards worth less than an immediate reward. Similarly, the value of performing an action a in state s or state-action value function: $Q: S \times A \rightarrow \mathbb{R}$, can be defined as:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{ \sum_{k=0, r_{t+k} \in R}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \quad (6)$$

We call this function, a "Q-value function" in the rest of this study. The calculated value using this function is also called the state-action value. Almost all RL-based paradigms are based on providing an innovative approach for appropriately estimating the value functions. This has led to the exploration and production of several different estimating methods and techniques. One of the most popular of these is ***Q-Learning*** [69], which is used in this study.

Q-Leaning is a method used to estimate Q-value functions in a *model-free* fashion. In this situation, because of the lack of known transitions and reward models, there is a need for sampling and exploration to learn the required model. Therefore, Q-learning estimates the agent's Q-value function based on an action's Q-value estimation. This process is incrementally evaluated as follows [68]:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q_k(s_t, a) - Q_k(s_t, a_t) \right) \quad (7)$$

Where, α $(0 < \alpha \leq 1)$ is the *learning rate* which determines the extent to which new information can override old information [70]. Because of its proven ability in converging to an optimal policy [71] and estimating the value-functions in free model problems [69], we have used Q-learning to estimate the Q-values in this study. It is also worth noting that Definitions 7 and 8 are used to infer equation 7, which is used to calculate the transition probabilities.

2.5.5 Hidden Markov Model (HMM)

Another important concept that needs to be defined here is a Hidden Markov Model; this model is usually characterized by the following elements [72]:

- N, the number of hidden states in the model, $S = \{s_1, s_2, \dots, s_N\}$.
- M, the number of distinct observation symbols per hidden state, $V = \{v_1, v_2, \dots, v_M\}$.
- The state transition probability distribution $[X]_{ij} = \{x_{ij}\}$, where: $x_{ij} = P(S_{t+1} = s_j | S_t = s_i), 1 \leq \text{i,j} \leq \text{N}$.
- The observation symbol probability distribution in hidden state j, $[Y]_{jk} = \{y_j(v_k)\}$, where $y_j(v_k) = P(O_t = v_k | S_t = s_j), 1 \leq j \leq N, 1 \leq k \leq M$. And
- The initial state distribution $\Pi = \{\pi_i\}$, where $\pi_i = P(S_1 = s_i), 1 \leq \text{i} \leq \text{N}$.

Using the values of N, M, X, Y and Π, the HMM can be used as a generator to create an observation sequence (where T is the number of observations in the sequence): $O = (O_1, O_2, O_3, \dots, O_T)$. We use the notation $\Lambda = (X, Y, \Pi)$ to simply indicate the complete parameter set of the HMM. The trained HMM is used to answer the following question:

- Given the observation sequence $O = (O_1, O_2, O_3, \dots, O_T)$ and an HMM, how to efficiently compute the probability of the observation sequence?

In this study, we learn HMMs for classification purposes (as it will be elaborated upon in Section 2.6), hence we address this question using the Forward Algorithm [72]. This means that the forward algorithm computes the forward probability, $\alpha_k(t)$, as the joint probability of observing the first *t* vectors v_t, $\text{T} = 1, \dots, t,$ while in state *k* at time *t*. Another way to state this would be:

$$\alpha_k(t) = P(v_1, v_2, \dots, v_t, s_t = k | \Lambda) \qquad (8)$$

Where $\alpha_k(t)$ is the probability of observing $v_1, v_2 \ , \dots, v_t$, given that the system is in state *k* at time t. This step is performed as part of the classification process (Figure 1 top right corner). This probability can be calculated by the following recursive formula [73].

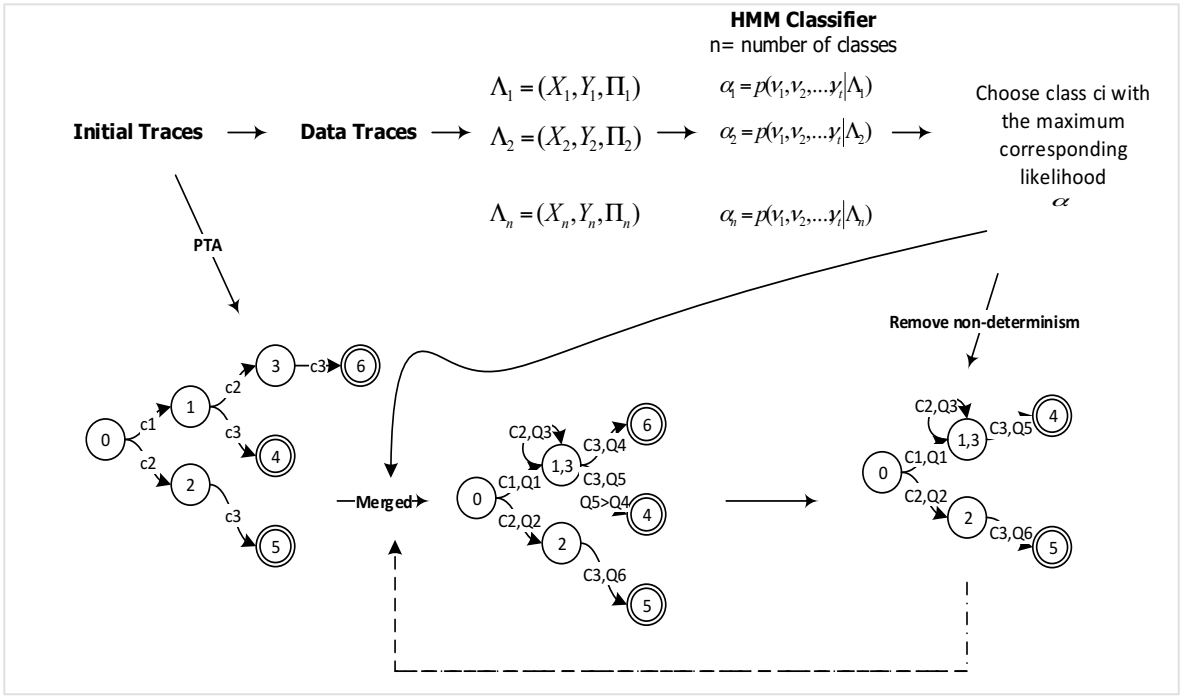


Figure 1. Inference steps for ReHMM approach. HMM classifier calculates the forward probabilities and chooses the next method to be executed based on its corresponding likelihood α. Q-values are also added to the PTA based upon the amount of change the function-execution triggers (Note: edges are labeled with method calls (class labels))

$$\alpha_k(1) = \pi_k b_k(v_1)\,, \quad 1 \le k \le N \qquad (9)$$

$$\alpha_k(t) = \pi_k b_k(v_t) \sum_{l=1}^{N} \alpha_l(t-1) a_{l,k}\ ,$$

$$1 \le k \le N, 1 \le t \le T$$

$$\alpha_k(t) = \sum_{l=1}^{N} \alpha_l(t-1) b_k(v_t) a_{l,k}$$

2.6 ReHMM: An RL-based HMM Inference Approach

Now according to the provided definitions and background, we are able to introduce a new format for FSAs, which incorporates the advantages of both EFSA and PFSA.

Definition 9. **Extended Probabilistic Finite State Automaton (EPFSA).** A EPFSA "W", is a tuple $(S, s_0, L, V, \Delta, \delta_W, I_W, Q)$, where S is a set of states and $s_0 \in S$ indicates the initial state, L is

defined as the set of labels, V represents the collection of variables (memory of the EPFSA), $\delta_W \subseteq S \times \Sigma \times S$ is a set of transitions between the states and $I_A: S \to \mathbb{R}^+$ (Initial-state probabilities). Moreover, Δ, the update function, is the function $L \times V \to V$. Q is the Q-value function (Definition 8), which calculates the Q-value, $q \in [0,1]$, for all model transitions.

It is worth noting that the definition is inspired by the definition of "extended digraphs" from our recent study [27] on software testing. As mentioned earlier, in order to address the problem of the missing state-action values (transition probabilities) along with increasing the accuracy of the inferred model, a new technique is proposed in this study. This approach provides a hybrid technique by mixing Artificial Intelligence (AI) and stochastic-based approaches. The idea of combining HMM and RL concepts to re-estimate and improve a model has been considered as a way to solve several AI problems such as robotic motion prediction [74], speech recognition [75] and natural language generation [76]. Also, [77] suggests a method of handling RL algorithms in partially observable MDPs; and [78] provides a comprehensive study on RL and hidden states. In addition, our recent research [27], confirms the effectiveness of applying this combination in addressing software-testing issues, specifically for test case prioritization processes. All of these studies confirm that HMM and RL concepts can be used as a method to empirically improve the *quality* of an inferred automaton. According to [29], [49] a *high quality* model is the one which is not only able to accept positive traces but also correctly reject negative traces. In addition, the inferred model should be able to retain the probability distribution of the original specification (in case of inferring PFSAs). In this study, when we are referring to the accuracy of the model, we are talking about a measure indicating the quality of the model with respect to both of these criteria. Furthermore, in this study, ReHMM, takes advantage of combining RL and HMM by:

- Predicting the next event name in the trace to identify non-determinism; and

- Adding probabilistic values to the inferred model and incrementally improving the model to generate an accurate EPFSA.

This section presents a step-by-step explanation of proposed ReHMM inference technique along with a motivating example providing an illustration of the inference steps on a sample system. It is also worth noting that, Figure 1 provides an overview of the steps involved in the ReHMM inference approach, while Table 3 and Table 4 provide pseudo code of the same procedure.

Table 3. ReHMM Inference Algorithm

Input: Traces (Initial Execution Traces)

Output: EPFSA (Inferred Extended Probabilistic FSA)

1. *ReHMMinfer* (traces) **begin**
2. DataTraces ⟵ *PrepareDataTrace*(Traces)
3. $(EPFSA, Var^2, Q^3)$ ⟵ *PTA* (Traces)
4. **For each** pair $(s_1, s_2) \in S$ **do**
5. (Λ^1) ⟵ *TrainHMM* (DataTraces)
6. Boolean ⟵ *checkConsistency* (EPFSA, Λ)
7. $(EPFSA', Var', Q')$ ⟵Merge(EPFSA, (s_1, s_2),Var,Q,G)
8. **Repeat** choosing (s_1, s_2)
9. **Until** no new $(s_1, s_2) \in S$ is found
10. end

- *Λ is a ReHMM classifier*
- *Var indicates the data variable values*
- *Q indicates the amount of Q-value for each transitions*

Table 4. Similarity Score Calculator

Input: Method calls in String format (c_1, c_2)

Output: Similarity Score

1. $Similarity\ (c_1, c_2)$ **begin**
2. **if (**$Length.c_1 < Length.c_2$)
3. **then** $Swap\ (c_1, c_2)$
4. BigLength ⟵ $Length.c_1$
5. **Return** bigLength – *ComputeEditDistance*$^*(c_1, c_2)$ / bigLength

** We have implemented the "Levenshtein distance" algorithm to compute the Edit distance in this study*

The procedures of preparing the data traces and generating the PTA are similar to those used in the MINT inference algorithm [65] (Section 2.3). Following paragraph provides a detailed description of the first step of the proposed algorithm: preparing data traces.

Preparing Data Traces: Similar to generating data traces in MINT: Given an initial trace which consists of a variable domain and a set of transition labels, a data trace $TC = \{(e_0, c_0), \dots, (e_n, c_n)\}$ is generated, where $e = (l, v)$, $l \in L \; and \; v \in V$, and each variable e indicates a class label $c_i \in C$.

After producing data traces using the PrepareDataTrace function [28], data traces will be separated into several groups (training sets) based upon their corresponding class labels. In other words, each training-set only contains data traces with the same class label. For example, given an initial trace as below:

Initial trace:

…

trace

highwater 74.2918692932601 570.9631358851515 false

not_critical 48.419183936768924 597.1704678020701 false

switch_pump_on 74.2918692932601 570.9631358851515 false

switch_pump_on 48.419183936768924 597.1704678020701 false

turn_on 74.2918692932601 570.9631358851515 false

critical 46.71352900357 601.7913477049301 true

…

The PrepareDataTrace function returns the following data trace:

Data trace (training set):

…

trace

highwater 74.2918692932601 570.9631358851515 false class: not_critical

switch_pump_on 74.2918692932601 570.9631358851515 false class: switch_pump_on

turn_on 74.2918692932601 570.9631358851515 false class: critical

critical 46.71352900357 601.7913477049301 true class: turn_off

switch_pump_off 46.71352900357 601.7913477049301 true class: not_critical

turn_off 46.71352900357 601.7913477049301 true class: not_critical

…

The PrepareDataTrace function turns the traces into the data traces (training sets) using data values in the events. Then the training sets can now be used to learn a set of HMM classifiers in order to predict the next event name (method) to be called during the inference procedure.

Generating the PTA: At this point, a Prefix Tree Acceptor (PTA) is generated from the initial traces using the PTA function. This function implements the same algorithm as the PTA generation algorithm in [29]. Using the PTA generation algorithm in [29] a tree-shaped state machine is constructed which exactly accepts traces it has been built from. In the generated PTA, traces with the same prefix share the same path from the initial state in the PTA up to the point at which they diverge. However, in our function, transitions of the PTA are not only labeled with both method calls and data variable values, but are also labeled with additional information (*Q-values).* This makes our PTA different from conventional versions [29].

Now, in order to add state-action values (transition probabilities) to the edges, we use Q-learning and estimate the Q-value function. To estimate the Q-value function, we need to define and compute the reward function representing the value of actions first. In many software engineering problems, such as software testing and inferring application behavioral models, the value of actions depends on the computation activated by the action [16]. For example, executing an event with corresponding parameter values, which is very different from previously executed events leads to major changes when compared to executing similar events. This can also be thought of in terms of discovering new behavior (wider exploration) and subsequently new defects (found in the previous unexplored execution space) in the software system. The reward function favours actions that activate new behaviors with major computations and penalizes actions triggering minor changes. In order to identify the difference between actions according to the amount of change they can trigger, different heuristics have been suggested. For example, the method suggested by Mariani et al. [16], assigns high reward values to actions that induce many changes in the abstract GUI state by calculating the difference between corresponding widgets in different states.

In this study, we suggest using $diff_C$ function as a heuristic to calculate reward functions and then Q-values: Given two class labels (method calls), c_1 and $c_2 \in C$, we define the $diff_C$ function that computes the degree of change between method calls. Specifically, $diff_c$ can be defined as: $1 - similarity(c_1, c_2)$, where, $similarity\ (c_1, c_2)$ is computed using the algorithm provided in Table 4.

The problem of computing the similarity between two sequences of characters (Strings) has been addressed in many software engineering areas using edit distance metrics [79], [80]. Such measures calculate the similarity between two sequences by computing the minimum cost of a

series of symbol insertions, deletions or substitutions to transform one string into another one. In this study, we apply the Levenshtein [81] metric to calculate the $similarity\,(c_1, c_2)$. The Levenshtein [81] metric was also used to compute the distance between FSAs in several studies to calculate the similarity between two string inputs [82]. This metric has led to reasonable results in evaluating a mined FSA as compared to reference models. Additionally, in order to demonstrate the reason of choosing Levenshtein distance among existed edit distance measures, we implemented several edit distance metrics as different heuristics in our approach. The results (Table 6) shows that the models inferred using Levenshtein heuristic is performing better compared to the models implementing LCS [83], Hamming [84] and Jaro [85] distances, in terms of the BCR measure (please see section 7.1 for more details on BCR measure).

It is also worth noting that the method names (labels) can only be "captured" as a sequence of Strings. For instance, the following line of a trace: (low_water, (28.898501010661718 584.0357656062484 true)) is actually captured as: "low_water, 28.898501010661718 584.0357656062484 true" and hence the Levenshtein distance is a suitable option in calculating the differences between these Strings. While it may be tempting to believe that the white spacing could be utilized to create four sub-strings etc., it is far from obvious that such additional information is available in every scenario. Hence, we take the most generic and conservative approach available and consider the data as a single indivisible string. This also creates a "level of playing field" with the other algorithms, which also view the data as a single indivisible string. The heuristic is not perfect, but the results indicate that Q-learning is able to improve the accuracy of the model by incrementally learning the model.

So, given two PTA states s_1 and s_2 and a new method call (class label) $c^* \in C$, executed from state s_1, the reward of observing c^* is equal to the amount of change in the functions (methods) from the original state to the target state:

$$reward(s_1, c^*) = \max(\, diff_C(c^*, \sigma_{c^*}(s_1, s_2))) \qquad (10)$$

Where, $\sigma_{c^*}(s_1, s_2)$ determines the class labels (method calls), which are met by traversing from s_1 to s_2, except for c^*. For example, if three transitions exist from s_1 to s_2 which are labeled by $\{c^*, c', c''\}$, then $\sigma_{c^*}(s_1, s_2) = \{c', c''\}$.

This reward function is able to estimate the utility value of a specific observation but is not able to estimate the value of a path or a sequence of actions. To identify the paths governing transitions, we need to identify transitions that can potentially activate functions, which can subsequently lead to large state changes. Q-learning does this task by estimating Q-values. Q-values are computed using the reward values and according to the Q-value function, $Q^{\pi}(s, c^*)$, defined in Section 2.5.4:

$$Q^{\pi}(s, c^*) = (reward\ (s, c^*) + \gamma \max_{c \in C} Q(\varphi(s, c^*), c))/(1.9 \times N) \qquad (11)$$

$$\sum_{s \in S, c^* \in C} Q^{\pi}(s, c^*) = 1$$

Where N is the out-degree of state s or the number of outgoing edges emanating from s, and $\varphi(s, c^*)$ determines the state which is reached from state s by executing c^*.

As mentioned in Section 2.5.4, Definition 8, γ is a parameter called the discount factor in the range [0,1]. This parameter is used to balance the trade-off in importance between sooner versus later rewards. According to our experience with ReHMM along with other studies in RL [16],

[86], [87], [48], the best choice for this parameter is $\gamma = 0.9$. In addition, since the immediate utility value of a method call is calculated by a Q-value function, it is a number in the range [0,1].

In particular, a Q-value close to 1 represents higher volumes of changes that can be triggered by executing the corresponding method, and subsequently the higher probability of being executed compared to a Q-value close to 0. Therefore, we initially mark all transitions with the maximum Q-value = 1 to treat every undiscovered path as a potential path to be traversed. Hence, initially, all methods will have the same probability of being executed regardless of the current system state.

Also, because the probabilities of transitions emitting from a node in a PFSA (Definitions 4 and 9) must sum up to 1.0, a normalizing constant is required to guarantee that this assumption is being met during the inference process. In this study, the normalizing constant is equal to 1.9 because:

$$F(s, c^*) = \sum_{s \in S, c^* \in C} (reward\ (s, c^*) + \gamma \max_c Q(\varphi(s, c^*), c))/(N) \quad (12)$$

$$\begin{aligned} Max(F(s, c^*)) &= N \times (\max_{s \in S, c^* \in C} (reward\ (s, c^*)) + \max_{s \in S, c^* \in C} (\gamma \max_c Q(\varphi(s, c^*), c))/N \\ &= \max_{s \in S, c^* \in C} (reward\ (s, c^*)) + \max_{s \in S, c^* \in C} (\gamma\, max_c\, Q(\varphi(s, c^*), c)) \end{aligned} \quad (13)$$

Now, since $max_{s \in S, c^* \in C}\ (reward\ (s, c^*)) = 1$ and $max_{s \in S, c^* \in C}(\gamma\, max_c\, Q(\varphi(s, c^*), c)) = 0.9$, then the maximum amount of:

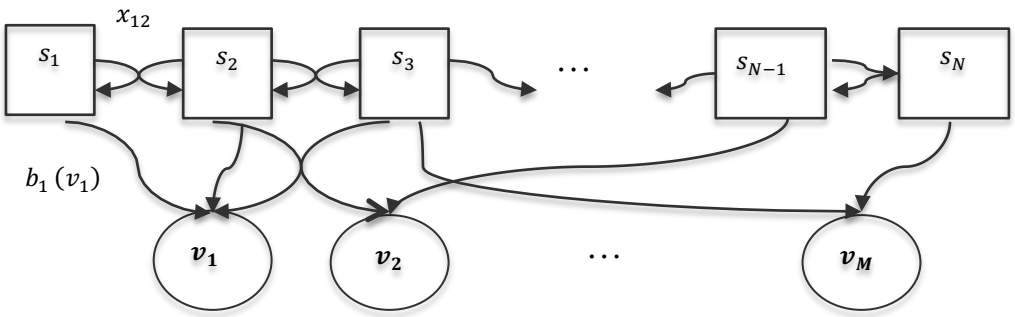
$$\max_{s \in S, c^* \in C} (reward\ (s, c^*) + \max_{s \in S, c^* \in C} (\gamma\, max_c\, Q(\varphi(s, c^*), c))) = 1.9 \quad (14)$$

Therefore, in order to assure that the summation of the transition probabilities of the nodes is equal to 1, $Q^{\pi}(s, c^{*})$ should be multiplied by the normalizing constant $1/_{1.9}$.

Training HMM Classifiers: ReHMM builds upon the data classifier approach [28] , [29] (Section 2.3), it uses an HMM classifier as a compatible stochastic approach with RL to infer a probabilistic, trainable model. Therefore, the TrainHMM function in Table 3 starts training an individual HMM, $\Lambda = (\mathrm{X}, \mathrm{Y}, \Pi)$, on the data obtained from each possible class label in the data trace. Therefore, having n class labels, n different HMMs will be trained. In each model:

1. The HMM can be used as a generator to create an observation sequence $O = (O_1, O_2, O_3, \dots, O_T)$ (where T is the number of observations in the sequence). The observation symbols ($O_i \in O$) in this study refer to the events and their corresponding parameter values ($v \in V$) in the data traces.
2. X is the state transition probability distribution $[X]_{ij} = \{x_{ij}\}$. Where: $x_{ij} = P\left(S_{t+1} = s_j \middle| S_t = s_i\right), 1 \leq i, \mathrm{j} \leq \mathrm{N}$. In this study, such probabilities are HMM components showing the probability of transiting from one hidden state to another one.
3. Y is the observation symbol (emission) probability distribution in hidden state j, $[Y]_{jk} = \{y_{j\,(vk)}\}$. Where $b_y\,(v_k) = P\left(O_t = v_k \middle| S_t = s_j\right), 1 \leq j \leq N,\ 1 \leq k \leq M$. In this study, the emission probabilities show the probability of observing a parameter value in a hidden state.
4. Π is the initial state distribution $\Pi = \{\pi_i\}$, where $\pi_i = P(S_1 = s_i), 1 \leq i \leq N$. Every hidden state that may be built during the classification process can be considered as an initial state in a Hidden Markov Model.

It is worth noting that the hidden state S is not a FSA state (not to be confused with the S notation). Also, the word *hidden* in HMM does not refer to the parameters of the model; even if these parameters are fully known, the model is still a *hidden* Markov model. **Error! Reference ource not found.** depicts a detailed HMM which is created per class label during the



classification process.

Figure 2. An overview of a HMM model, which is generated in the classification process

Executing TrainHMM function in this step of the inference process on the running example, leads to generating 8 HMM Classifiers, because 8 class labels can be extracted from the data trace. Each hidden Markov model depending on the number of its hidden and observable states is represented by transition, emission and initial distribution matrices. For instance, a model representing the class label "switch_pump_off" consists of 5 hidden states and 18 parameter values. Then:

- Transition probability matrix $X = [X]_{5\times5}$
- Emission probability matrix $Y = [Y]_{5\times18}$
- Initial distribution matrix $\Pi = [0.2, \dots, 0.2]_{1\times5}$

After creating a set of models and assigning them to separated class labels ($c_i \in C$), we can calculate the likelihood that a sequence of observations belongs to a specific class label.

Computing this likelihood is possible by executing the forward algorithm [73] (as defined in Section 2.5.5). After training the models, the likelihood of a sequence of observations belonging to a specific class label can be computed to predict the next method call in the FSA.

Merging Step: In the algorithm, Q, indicates the list of the corresponding Q-values. Then, both the HMM classifiers and PTA are used to carry out the same state-merging procedure examined in [28], [88]. The only difference is that in our proposed approach, the Q-values are also used to evaluate the similarity of the transitions. Thus, two states only share the same prefix if the inferred classifier predicts the exact same labels for every data point in the prefix of each state (including the Q-values). Since, Q-values are calculated incrementally, states with the same prefix but different Q-values reveal different reward values during the inference procedure, and subsequently different amounts of computations triggered by executing the corresponding events. Therefore, Q-values, along with the minimum merge score G, are used as extra data points to avoid merging these states. False merges may lead to missing some states of the model, representing a new behavior or a fault prone method.

Checking the Consistency: Once states and transitions have been merged, the consistency of the inferred model should also be checked to: (1) ensure the validity of the attached variables with the ReHMM classifiers; and, (2) prevent producing a non-deterministic EFSA. In this step, for each transition of the resulting model, the corresponding data variables are obtained and provided to the ReHMM classifier to predict the subsequent label (the name of the method to be executed). In this case, if the target state of the model does not have an outgoing transition with the predicted label, the CheckConsistency function the current merge is ignored. Otherwise, the algorithm looks for the next merge and the process continues until no more merges can be identified.

The result of running the ReHMM algorithm on a set of interaction traces show the value of paths, governing transitions, by assigning the corresponding Q-values to each transition. Consequently, ReHMM determines valuable (interesting) functions by identifying transitions that enable actions that trigger large-scale state changes. In this study, Q-values are also used as transition probabilities to infer an Extended PFSA. Therefore each node in this EPFSA represents a program state while every transition is attached with a probability. This transition probability indicate how likely the corresponding method call is to be invoked from the source code, with respect to the former executed method calls. This probabilistic FSA is able to reduce the effect of errors in training traces by pruning the transitions, which have a low likelihood of being traversed.

The PTA generation step, in the running example is implemented as below; the initial set of traces is used to produce an enhanced Prefix Tree Acceptor (PTA), carrying the data constraints and Q-values. Figure 3 shows the result of the first round of calculations. According to this figure, in order to move from state 0 to state 1 in the model, at least four direct paths exist. To save space, all of the transition labels are shown on one single edge. Otherwise there would be four edges from state 0 to state 1, labelled by: critical, switch_pump_on, highwater, not_critical and their corresponding data constraints. It is also worth noting that state 0 is the initial state in the model, and state 1 is the next state which is met after the execution of the corresponding transitions (critical, switch_pump_on, highwater and not_critical). The amount of Q-values is calculated per transition. For example, in order to reach state 1 from state 0 by traversing the edge, which is labeled as “critical”, first the reward function for this move should be calculated by considering the amount of changes triggered by this function-execution. Second, the maximum amount of Q-values of all possible future transitions from state 1 should be computed.

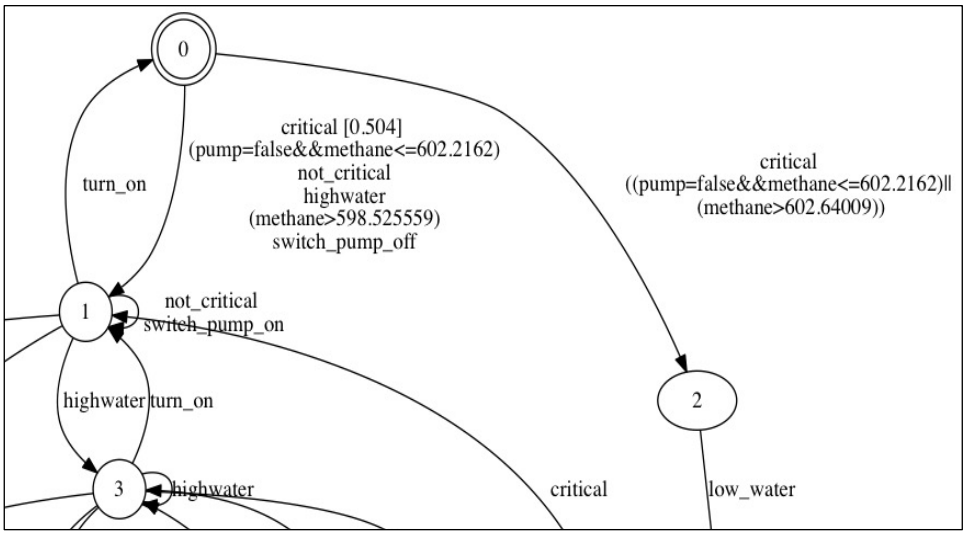


Figure 3. Excerpt of the model derived by ReHMM from the pump controller example

$$reward(0, critical) = \max\{diff_m(0,1)\}$$

$$max\ \{diff_m(0,1)\} = max\ \begin{Bmatrix} diff\ (critical\ , switch\_pump\_off), \\ diff\ (critical, \quad highwater, \dots) \end{Bmatrix} = 0.917$$

$$Q^{\pi}(0, critical) = \frac{0.917 + 0.9(1)}{1{,}9 \times 4} = 0.504 \simeq 0.5$$

Similarly, all transitions are incrementally annotated with Q-values as more events are discovered in the model. After the first iteration of calculating the Q-values, the model is reconsidered for the state-merging procedure. During the merging process, when transitions are merged, their corresponding data values are also merged (as mentioned in Section 2.3).

However, at this point, the HMM classifier is used to compare the attached data values (including Q-values) to the outgoing transitions from the merged state. If the produced results are similar enough; the Q-values have at least one identical number in their first decimal places (after being rounded to the nearest tenth); the transitions can be treated as equivalent and can be merged; otherwise the potential merge is ignored.

This procedure [29] also helps in increasing the accuracy of the model by detecting non-deterministic transitions. In this case, ReHMM infers an EFSA with 11 states while the proposed approach by [29] contains 16 states (See Appendix A for the EPFSA generated using ReHMM). Finally according to [29], the GK-tail algorithm inferred a model with 39 states using the same data traces.

It is also worth noting that similar to MINT framework, our approach is also able to implement Daikon's decorator class [4], [29] and use it to infer rules that link the variables together for each transition. As it is mentioned in sections 2.1 and 2.3 using Daikon to label transitions could be useful for the purposes of providing transition-specific information about the attached data. Also, the extracted rules (e.g. pump=false&methane<=602.2162), similar to other transitions' labels, are then utilized to calculate Q-values and determine whether or not a pair of states is compatible.

As it will be discussed in Section 2.9, ReHMM also is able to generate more accurate models when compared to MINT. The ability to generate small accurate models is a feature of ReHMM, which can be very helpful in addressing software engineering tasks such as test case generation.

2.7 Empirical Evaluation

To investigate the effectiveness of the proposed technique, we have performed an empirical evaluation. The evaluation compares the new technique with five data classifier inference approaches from MINT [28], [29] and one PFSA inference method (sk-Strings) [48]. It is argued in this chapter that these approaches represent the current state of the art. We have constructed an experimental framework, which addresses the following research questions:

- **(RQ1)** How does the ReHMM inference technique performs in terms of accuracy [29], [49] as compared to other state of the art inference approaches, specifically MINT (EFSA) and sk-strings (PFSA) [28]?
- **(RQ2)** Does the inferred EPFSA retain the probability distribution of the original specification [49]?

The first research question **(RQ1)** is answered using a new evaluation approach, proposed by Walkinshaw et al. [29]. Their proposed approach uses a well-known technique called k-folds cross validation to randomly partition a set of examples into k (non-overlapping) sets. Subsequently, a model is inferred using each set over k iterations and the remaining set is used to evaluate the model based upon the accuracy measure (defined in the next section). Since in each iteration a different set is used for the evaluation, the final accuracy score is equal to the mean of the k accuracy scores. This approach is successfully used to calculate the accuracy score for both EFSAs and PFSAs [29].

On the other hand, in order to evaluate the probablistic element of the infered Extended PFSAs[4] **(RQ2),** we implement the approach recommended by Lo et al. [16]; they suggest calculating a new metric called Probability Similarity (PS), measuring the similarity in terms of probabilities assigned to the common traces generated by both the simulator (Section 2.5.1) and the mined model. In other words, this metric determines if both the simulator and the mined models generate the same traces at similar frequencies. The PS metric also will be elaborated upon in the following section.

[4] In cases that the inference procedure leads to generating Extended Probabilistic FSAs, the quality of the produced EFSAs (without considering the transition values) can still be evaluated by comparing the BCR measure of the mined models but it is not sufficient in assessing the quality of state-action values (transition probabilities)

2.7.1 Comparison Criteria

In order to evaluate RQ1 and to assess the extent of agreement between a set of traces and a model, we follow the approach utilized by Walkinshaw et al. [29]; they recommend calculating the BCR (Binary Classification Rate) metric in order to assess the accuracy of inferred EFSAs. Essentially, each trace is compared to the model in order to consider— if the model correctly accepts positive traces and rejects negative ones. Therefore, we need a measure that not only considers the true positives, false positives and false negatives but also takes the true negatives into the account.

Basically, BCR is the mean of the Sensitivity and Specificity measures, where TP=True Positive, TN= True Negative, FP= False Positive, FN= False Negative and:

$$Sensitivity = \left(\frac{TP}{TP + FN}\right), \ \ Specificity = \left(\frac{TN}{TN + FP}\right) \qquad (15)$$

It is worth noting that Sensitivity is also known as "Recall", while Specificity is a recommended alternative for "Precision" in cases where True-Negatives also should be taken into account. In other words in this study, similar to [29], the ability of the automaton in correctly rejecting negative traces is as significant as correctly accepting the positive ones.

In addition, for the sake of completeness in analyzing the performance differences between diverse inference techniques, we also follow advice from [33], and apply the Wilcoxon Signed Ranks Test and effect size approaches.

Non-parametric Statistical Hypothesis Test: In this case, we established a null hypothesis and an alternative hypothesis to be evaluated. The null hypothesis (H_0) states that the two inference techniques provide the same accuracy, if the median of the BCR score for both techniques is the

same. On the other hand, the alternative hypothesis (H_1) states that if the difference between the medians of the BCR measures, which have been detected by each of the inference techniques, is not zero then they will be considered as different. Therefore, by considering a significance level $\alpha = 0.05$, we would be able to reject the null hypothesis if $p - value < 0.05$, for each independent situation.

Effect Size: In order to add a "size of difference" statement to our comparison criteria, we consider the strength or magnitude of a treatment effect, by calculating Cliff's Delta measure. Cliff's Delta statistic [86] is a nonparametric effect size measure that quantifies the difference between two groups of observations by testing the equivalence of probabilities of scores in one group being larger than the scores in the other. In this study, the magnitude of differences between inference techniques is assessed using the thresholds provided in [89] i.e. |d|<0.147 "negligible", |d|<0.33 "small", |d|<0.474 "medium", otherwise "large".

In order to evaluate RQ2, we estimate the Probability Similarity (PS): PS measures the similarity in terms of probabilities assigned to common traces, generated by a mined model Z

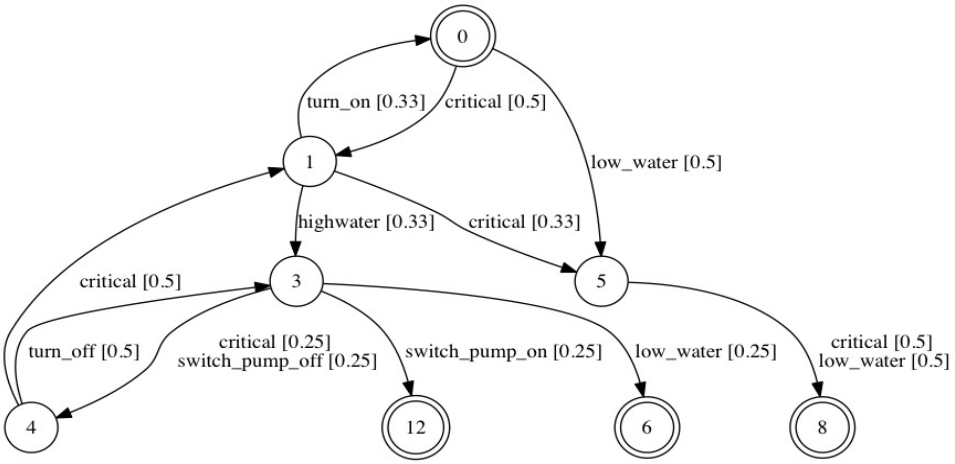


Figure 4. Excerpt of the simple simulator of the running example- the simulator is used as a basis for calculating the Probability Similarity of inferred PFSAs

and a simulator model Z'. Following the approach provided in [49], the simulator model in this study is also an initial version of the PTA (Section 2.5.1) generated using the same algorithm as applied in [29]. The only difference is that the produced PTA is also labeled with the transition probabilities, which (1) are distributed equally between transitions with the same source node and (2) are summed up to one. Figure 4 shows an excerpt of a simulator model.

Having both the simulator and the mined model, Z and Z', we are able to calculate the PS measure. First, we need to calculate the probability that a trace Tr is generated by both Z and Z' independently. It is the same procedure as the one which is normally used to measure the probability similarity between two Hidden Markov Models [76].

$$P_{CE}(Z, Z') = \sum_{Tr \in L(Z \cap Z')} (P_Z(Tr) P_{Z'}(Tr)) \qquad (16)$$

Where $P_Z(Tr)$ and $P_{Z'}(Tr)$ represent the probability of producing trace Tr by Z and by Z'. We can determine the probability of a trace by multiplying together the probability of its constituents. Now the Probability Similarity between Z and Z' can be calculated as follows:

$$PS(Z, Z') = \frac{2 \times P_{CE}(Z, Z')}{(P_{CE}(Z, Z) + P_{CE}(Z', Z'))} \qquad (17)$$

It is worth noting that this technique has also been successfully used in [49] and [3] in order to assess the quality of an inferred EPFSA.

2.8 Experimental Setup

Several techniques exist to empirically evaluate EFSA inference algorithms, but all of them rely on a reference model to be used as a basis for computing accuracy. In practice, generated models should be compared with the reference model for computing the accuracy of the model [14],

[36], [37]. Since these models are required to be generated manually for systems with complicated behavior, this can be considered as a significant limitation.

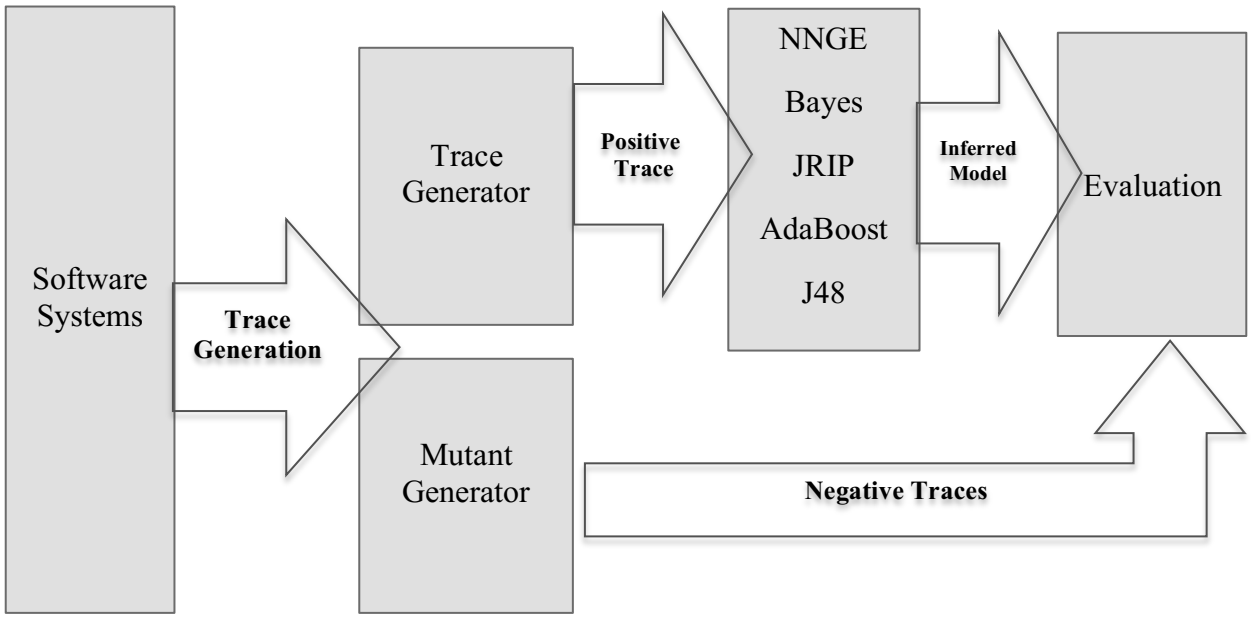


Figure 5. The toolset used in the empirical evaluation

To empirically evaluate ReHMM in terms of accuracy, seven different systems have been considered. All of these systems are sequential systems relying on an internal data-state, which makes them suitable for modeling with either EFSA or EPFSA inference tools. These choices also have been used in previous research projects [4], [10] and [39].

This selection also guarantees that the experimental objects are independent of the authors and specific programming languages.

- **SMTPTransport Class in OracleJavaMail[5]:** This module implements the Transport abstract class using SMTP for message submission and transport. The SMTPTransport

[5] http://www.oracle.com/technetwork/java/javamail/index.html

Junit tests (collected using a tracing-aspect in AspectJ[6]) along with the Apache Commons Mail test sets have been considered for the first experiment [28];

- *Number of traces:* 80 traces are used in this study;
- *Average trace length:* 20 (Input traces have different lengths; this metric calculates the mean of all traces' lengths); and
- *Average variables per event:* 1.5 (Min=1 and Max=5). This metric calculates the mean of all variables associated with each trace event.

- **Erlang Poolboy:** A module in the Basho Riak[7] distributed database, which implements a procedure for pooling connections
  - *Number of traces:* 80 traces have been generated using Poolboy Eunit tests (collected with an instrumentation system using the Wrangler refactoring API[8]) [28];
  - *Average trace length:* 30; and
  - *Average variables per event:* 3 (Min=1 and Max=7).
- **Mobile Frequency Server:** An Erlang server module, which assigns frequencies to mobile phones contacting each other.
  - *Number of traces:* 100 identical traces to [29] have been used in this case;
  - *Average trace length:* 20; and
  - *Average variables per event:* 3 (Min=1 and Max=6).
- **Resource Locker:** A distributed resource-locking algorithm written in Erlang, which is designed for a specific model of ATM switches.

[6] http://eclipse.org/aspectj/

[7] http://basho.com/riak/

[8] http://www.cs.kent.ac.uk/projects/wrangler/Wrangler/Home.html

- *Number of traces:* Again, 100 identical traces to [29] have been applied in this case;
- *Average trace length:* 20; and
- *Average variables per event:* 1.3 (Min=1 and Max=3).

- **Signature:** A Java class, which implements a digital signature algorithm.
  - *Number of traces:* 71 traces, which were collected through the execution of Columba[9] email clients in [29] are also used in this experiment;
  - *Average trace length:* 14.52; and
  - *Average variables per event:* 16.7 (Min=1 and Max=24).
- **Socket:** Another Java SDK class, which implements client sockets. A socket is an endpoint for the communication between two machines.
  - *Number of traces:* 100 traces are used as [29];
  - *Average trace length:* 44.18; and
  - *Average variables per event:* 13.17 (Min=1 and Max=18).
- StringTokenizer: Traces, which are collected from several execution of JEdit[10] in [29] on a Java utility class for breaking a string into tokens are applied in this case study.
  - *Number of traces:* 100 traces are used as [29];
  - *Average trace length:* 33.36; and
  - *Average variables per event:* 14.5 (Min=1 and Max=17).

Walkinshaw et al. [29] collected initial traces for Erlang modules using a random test generation framework. In this approach, random functions are invoked through random parameters and this

[9] http://sourceforge.net/projects/columba/

[10] http://www.jedit.org/

procedure is repeated 20 times to generate 100 traces for each system. Additionally, the traces for Java modules are collected through running real-world, publicly available applications that use the selected libraries [39].

In order to assure that the collected traces have been sufficient to infer reliable models mimicking actual systems behavior, we applied the measure provided by [90] and [91]. Using this measure, which is called the log's *confidence*, we consider the confidence of a log (set) of execution traces in order to estimate the expected faithfulness of the mining results. According to Cohen et al. [90], a low confidence (e.g. 0.2) indicates that the inference results might not be compatible with the actual behavior of the system under investigation. However, the high confidence (e.g. 0.95) hints that the inference results are probably very close to the actual behavior of the system. Therefore, the probability that a sequence of events Tr of length n does not appear in the traces ($Y^k[Tr] = 0$) but is possible to be generated by the inferred model ($f[Tr] = 1$) determines the log confidence of the considered traces, as below:

$$P[Y^k = f] \geq 1 - \sum_{\{Tr | p_{Tr} > 0\}} (1 - p_{Tr})^k \qquad (18)$$

Where, p_{Tr} denotes the probability that the Tr appears somewhere in the trace and k denotes the length of the log. (The interested reader should consult [90] for a derivation of equation (18)) The results indicate that all of traces used in this study have a confidence of at least 0.92.

In addition, computing the BCR measure and calculating the FNs, TNs, FPs and TPs also needs the definition of *negative traces*. As mentioned earlier, negative traces are traces covering the mutations. Walkinshaw et al. [28], [29] collected positive traces for considered case studies using a random test generation framework while, they collected the negative traces using both

automated and hand-picked mutation approaches. They proposed a customized mutation technique that can be replaced with random mutations of the program code by automatically applying the hand-picked changes to the positive traces. This method guarantees that newly generated traces are “negative”, since, they are generated by adding invalid suffixes to the valid prefix of positive traces. It is worth mentioning that all of the positive and negative traces used in this study are kindly provided by Walkinshaw et al. [28], [29].

Although the traces are available, the exact method used to extract them, are not. Even though, applying handpicked changes can address the problem caused by using quasi-random mutations; it makes replicating the trace production process very difficult (e.g. the set of rules applied to characterize impossible-to-happen sequences, are not available). Therefore, the process of creating further traces and using them to infer models comparable with the current case studies’ is almost impossible.

In order to evaluate our proposed approach in comparison with other techniques, we used Walkinshaw et al. ([15, 16]) studies as baselines, and updated their toolset as shown in Figure 5 by adding our proposed inference approach in the appropriate location. Their approach is a modular method, which means, that instead of relying on a single data model inference technique, it uses a set of arbitrary data classifier inference techniques. The classifiers used by [28] are: NNGE, Bayes, JRIP, AdaBoost and J48 (implementing the C4.5 algorithm [56]).

In addition, as mentioned earlier, the inferred EPFSAs by ReHMM are compared with a state of the art PFSA inference tool called sk-strings to investigate the accuracy of the Q-values in estimating the transition probabilities. In Accordance with several experiments conducted by Raman et al. [48], d%= 50% k=1 and the “AND variant” [33] are utilized as the default

parameters for the sk-strings algorithm. Therefore, as depicted in Figure 5, different models are inferred from positive software traces using ReHMM, MINT and sk-strings.

Then, in the evaluation phase, the k-folds cross validation technique, k=5, is used to address the problem of requiring reference models [36] and exercising the full range of behaviors in the systems under consideration. This approach randomly divides the input traces into 5 non-overlapping sets and infers a model using a set over 5 iterations. Another set is used to evaluate the model in terms of the BCR measure. Basically, the set of traces are classified by the model to find if the model accepts them as positive or rejects them as negative. As a result, positive traces can be used for k-folds cross validation, whereas the negative traces are added to the evaluation set in each iteration in order to consider the effect of false positives/negatives on the accuracy of the experiment (Figure 5). After inferring models based on the classifier algorithms [28], sk-strings and the proposed ReHMM technique, the BCR is measured to detect the most accurate approach, while PS is calculated to evaluate the probabilistic element of the inferred model.

This is also worth noting that depending on the software engineering problem we are trying to solve the impact of such issues may differ. For instance, in the situation that the EPFSA is inferred to mine software specifications, wrongly recognizing a negative trace as valid software execution leads to generate software specifications containing behaviors which are not possible in the software. Similarly, mistakenly detecting a valid trace as a negative one, cause the specification misses correct software behaviors. Additionally, in the case of applying the inferred model to generate software test cases, false positives and false negatives lead to subsequently producing invalid test cases and missing valid ones.

2.9 Experimental Results

According to [28] and [29], BCR values above 0.5 show that a model is able to determine correct traces better than a random model. In order to evaluate the accuracy of our suggested model, we calculate the BCR measure for all of the generated models using ReHMM and the state of the art approaches.

Table 5 shows the result of applying all five classifiers, the sk-strings algorithm and the ReHMM technique. We computed the average BCR measure on different inference iterations (k-fold) for two different minimum merging (G) scores 0 and 1 (See Section 2.2.3 for a definition of merging score). The accuracy of the inferred models is evaluated for different merging scores to investigate the impact of changes to the algorithm configuration, and consequently finding an ideal choice of G (if it exists).

In addition to this, Table 6 and Table 7 report the results of calculating the sensitivity and specificity measures for all of the considered case studies applying the inference approaches. These results indicate that ReHMM outperforms the other approaches in correctly determining the validity of the traces compared to the results provided in [29]. In addition, the result of the Wilcoxon signed rank test indicates that, regardless of the perceived closeness of the BCR scores, the ReHMM inference approach is significantly different from the other techniques ($p - value = 0.03125 < 0.05$ for G=0, and $p - value = 0.01562 < 0.05$ for G=1). The Cliff's Delta measure provides more detailed information to this picture by showing that a "large" effect size exists (in favour of ReHMM) for all of the experiments, when G=0, delta estimate=0.8688, and when G=1, delta estimate=0.9072, both with a 95% confidence interval. These results provide an answer to the first research question.

Table 5. The Results of Applying Inference Techniques on Seven Different Case Studies, for G=0,1

	Poolboy		**SMTPTransport**		**Resource Locker**		**Frequency Server**		**Signature**		**StringTokenizer**		**Socket**	
G	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***	***Inference Algorithm***	***BCR***
0	ReHMM	**0.70**	ReHMM	**0.91**	ReHMM	**0.75**	ReHMM	**0.84**	ReHMM	**0.75**	ReHMM	**0.79**	ReHMM	**0.92**
	Sk-strings	0.65	Sk-strings	0.63	Sk-strings	0.64	Sk-strings	0.64	Sk-strings	0.63	Sk-strings	0.45	Sk-strings	0.73
	NNGE	0.68	NNGE	0.59	NNGE	NA*	NNGE	NA	NNGE	NA	NNGE	NA	NNGE	NA
	Bayes	0.66	Bayes	0.71	Bayes	0.66	Bayes	0.75	Bayes	0.50	Bayes	0.50	Bayes	NA**
	JRIP	0.61	JRIP	0.72	JRIP	0.61	JRIP	0.61	JRIP	0.50	JRIP	0.51	JRIP	0.75
	AdaBoost	0.61	AdaBoost	0.67	AdaBoost	0.63	AdaBoost	0.76	AdaBoost	0.50	AdaBoost	0.50	AdaBoost	0.83
	J48	0.59	J48	0.67	J48	0.64	J48	0.67	J48	0.50	J48	0.50	J48	0.80
1	ReHMM	**0.76**	ReHMM	**0.99**	ReHMM	**0.80**	ReHMM	**0.85**	ReHMM	**0.79**	ReHMM	**0.90**	ReHMM	**0.90**
	Sk-strings	0.65	Sk-strings	0.79	Sk-strings	0.66	Sk-strings	0.66	Sk-strings	0.67	Sk-strings	0.71	Sk-strings	0.71
	NNGE	0.70	NNGE	0.98	NNGE	NA	NNGE	NA	NNGE	NA	NNGE	NA	NNGE	NA
	Bayes	0.71	Bayes	0.98	Bayes	0.71	Bayes	0.66	Bayes	0.73	Bayes	0.82	Bayes	NA
	JRIP	0.69	JRIP	0.98	JRIP	0.85	JRIP	0.61	JRIP	0.77	JRIP	0.75	JRIP	0.65
	AdaBoost	0.67	AdaBoost	0.96	AdaBoost	0.74	AdaBoost	0.77	AdaBoost	0.67	AdaBoost	0.78	AdaBoost	0.72
	J48	0.65	J48	0.96	J48	0.74	J48	0.77	J48	0.77	J48	0.80	J48	0.70

* NNGE has not be considered in Walkinshaw's new study [29] because of its poor performance in systems with big traces containing spurious events

** All Inference efforts for Naïve Bayes classifier in the Socket case study time out, because the inferred model using Naïve Bayes takes a significantly longer time to query compared to other cases and classifiers

They show that combining the RL and HMM approaches not only improves the accuracy of the inferred model but also adds valuable information (data function) to its data flow (all the transitions are labeled with the name of functions, Q-value and data-state variables), which provides an accurate EPFSA inference method, outperforming another PFSA inference algorithm (sk-strings). This result illustrates that the ReHMM algorithm produces more accurate solutions in all the systems considered compared with sk-strings.

The minimum, maximum and median of BCRs are also calculated from applying different inference approaches on considered case studies are provided in Appendix B (Table 46, Table 47). In this experiment both 5 and 10-folds for the cross validation are considered.

Table 6. The Result of calculating Sensitivity and Specificity Measures Using ReHMM for G=0

	Sensitivity	Specificity
Poolboy	0.81	0.89
SMPTransport	0.98	0.93
Resource Locker	0.96	0.54
Frequency Server	1.00	0.68
Signature	1.00	0.50
StringTockenizer	1.00	0.58
Socket	0.86	0.98

Table 7. The Result of calculating Sensitivity and Specificity Measures Using ReHMM for G=1

	Sensitivity	Specificity
Poolboy	0.79	0.97
SMPTransport	1.00	0.99
Resource Locker	1.00	0.60
Frequency Server	0.71	0.99
Signature	0.98	0.60
StringTockenizer	0.90	1.00
Socket	0.83	0.97

Table 8. The Result of Calculating Probability Similarity Measure in ReHMM and sk-strings for G=0,1

	Sk-strings (G=0)	ReHMM (G=0)	Sk-strings (G=1)	ReHMM (G=1)
Poolboy	0.69	0.76	0.74	0.87
SMPTransport	0.90	0.88	0.90	0.94
Resource Locker	0.53	0.88	0.62	0.94
Frequency Server	0.33	0.85	0.59	0.81
Signature	0.48	0.67	0.52	0.79
StringTockenizer	0.67	0.79	0.67	0.90
Socket	0.65	0.83	0.68	0.82

Additionally, the results of calculating the PS metric (Table 8) indicate that ReHMM improves upon sk-strings in terms of retaining the original probabilities. This results in generating the same traces at similar frequencies as the simulator. The result of the Wilcoxon signed rank test and Cliff's Delta measure also "prove" this claim, that the ReHMM inference approach is significantly different in terms of the accuracy of the generated PFSA with sk-strings: for G=0,1. Delta estimate=0.8979 for G=0 and delta estimate=0.8367 for G=1, both with a 95% confidence interval. Both of these estimates can be interpreted as a "large" effect size.

These results provide an answer to the second research question. In addition, this procedure helps in addressing the missing state-action value (transition probability) by identifying a new state-action value (Q-value) function, which is able to map a transition to its corresponding Q-value while the model is generated. Using the new proposed technique (Q-value function), the value of a transition can be calculated directly from the observations with no need to walk through the source code. In all of the considered experiments, ReHMM has the BCR value of 0.7 or more, which demonstrates an acceptable level of accuracy [29], while, there is no single data classifier algorithm that outperforms the others for all of the systems.

The results also indicate that changing the minimum merging score from 0 to 1 does not make much difference to the accuracy of some of the considered cases (e.g. Socket). This may have happened because of their lower level of learnability. This also would be due to the complex nested lists and tuples in their software-systems' structures, which makes it difficult to distinguish a pair of states by their suffixes compared with the other systems [28], [29]. Therefore, it can be concluded that there is no ideal choice of merging score (G), since it depends on the characteristics of the target system in terms of the design complexity and the manner in which the system is invoked [29]. However, ReHMM seems to provide improvements over the

merging score changes when compared with sk-strings. This strongly suggests an improved capability of ReHMM in identifying merge-able suffixes. We believe the reason of the improvement in the accuracy of the inferred model using ReHMM, is the ability of ReHMM to add transition probabilities to the model and using them as a parameter to be checked in the merging step. This procedure helps in reducing the number of inappropriate merges and consequently generating more accurate models.

Based upon this result, it could be concluded that ReHMM has obtained better accuracy in the systems containing a lower number of variables for each event. The reason lies in the fact that the inference process for the systems with high number of variables and events lead to generating very large models. The model becomes even larger when this feature is coupled with a larger value of G. It is also worth noting that the large-size models are slower to be inferred and subsequently are the subject of more false merges during the state-merging procedure (Table 9, illustrates the size of inferred models using ReHMM for all case studies).

It was also observed that the traces containing all systems' events with a low number of variables are sufficient to generate an accurate model representing a system behavior. Increasing the number of execution traces does not necessarily lead to an improvement in the model's quality by adding new states and transitions to the PTA. Simply, the newly added traces can only repeat the patterns of events that are already provided by the old traces.

Moreover, the results of implementing different edit distance measures in ReHMM as alternative heuristics in calculating the Q-values are provided in Table 10. These results illustrate the reason behind the decision of choosing Levenshtein distance over other approaches.

Table 9. Sizes of Inferred Models in Terms of State Numbers for All Case Studies, Applying ReHMM (G=0,1)

	Number of States (G=0)	Number of States (G=1)
Poolboy	18	50
SMPTransport	7	32
Resource Locker	406	601
Frequency Server	181	776
Signature	3	16
StringTockenizer	36	304
Socket	144	1153

Table 10. The Result of Calculating BCR Measure Using ReHMM Implemented by Different Edit Distance Heuristics

	Longest Common Subsequence Distance	Hamming Distance	Jaro Distance
Poolboy	0.68	0.6	0.66
SMPTransport	0.87	0.67	0.73
Resource Locker	0.75	0.63	0.7
Frequency Server	0.84	0.67	0.81
Signature	0.69	0.58	0.69
StringTockenizer	0.7	0.62	0.72
Socket	0.85	0.75	0.8

2.10 Time Complexity Analysis

In order to investigate the time complexity of the proposed method for situations with a long sequence of observations (i.e. large-scale software systems with a huge number of methods and parameter values), we have considered the computation order of the inference algorithm.

Based upon the Incremental Baum-Welch algorithms' time complexity [38], the computation order of the inference procedure is polynomial: $O(N^2\ logT)$, where N represents the number of hidden states and T indicates the number of observations (parameter values); and hence, it does not exponentially grow by increasing the system size.

The time complexity of the traditional approach (using an offline Baum-Welch algorithm) is $O(N^2T)$[39], while in the incremental estimation approach which has been used in this study, this has been reduced to $O(N^2\ logT)$. This improvement is possible by modifying the emission and transition probabilities incrementally using the RL-based forward algorithm and training an RL-based HMM with a Maximum Likelihood Estimation (MLE) using the Incremental Baum-Welch algorithm [38].

This procedure also helps in estimating more accurate ReHMM elements than regular non-incremental HMM estimation. The efficiency of this approach has been confirmed by the production of EFSAs with a minimum BCR value of 0.7. In addition, according to [94], a Q-learning algorithm with action-reward representation in a deterministic domain (similar to the algorithm, used in this study) reaches a goal state and terminates after at most $O(en)$ steps; where $e \leq n^2$ and n represents the number of states of the initial PTA. Therefore, the worst-case time complexity becomes $O(n^3\ )$.

We have also investigated the amount of time required to infer the models using the proposed inference approach by considering the performed experiments in terms of the time taken for each configuration. All experiments are run on a simple hardware and software platform consisting of a 2x2.4 GHz Quad-Core CPU, 32 GB RAM on a Mac Pro (manufactured in 2010), Eclipse Indigo. Clearly, massive performance gains can be made by moving the experiments onto a modern server; however, in these experiments we are not concerned with absolute performance. The times are listed in Table 11. The results indicate that in most cases, the inference time is less than one minute. However, in 3 cases of inference with merging score G=1 and one case study with merging score G=0, the inference time is more than a minute (between 2 and 9 minutes). Conducting the same experiment using MINT indicates that MINT's inference time is within the

same range as ReHMM. The time it takes for MINT to infer models in “Frequency server” and “Signature” are respectively equal to 0.509 and 0.028 minutes for G=1 is slightly lower than ReHMM. While, in Socket experiment, MINT hits the timeout limit for several classifiers and merging scores, ReHMM inference time is still less than the timeout limit (9.065 min). It is worth noting that both MINT and ReHMM are written in Java.

Table 11. Time Taken to Infer Models Across All Case Studies Using ReHMM for G=0,1

	Inference Time (min), G=0	**Inference Time (min), G=1**
Poolboy	0.014	0.112
SMPTransport	0.012	0.023
Resource Locker	0.928	2.520
Frequency Server	0.683	0.852
Signature	0.051	0.060
StringTockenizer	1.050	3.083
Socket	5.125	9.065

In the study conducted by Walkinshaw et.al [29] using the MINT inference algorithm, it is indicated that an increase in the value of G, can lead to the increase in the inference time. When G increases, the inference approach needs more time to evaluate more states and transitions to find suitable merge candidates. The effect of choosing a large merging score can be even more in cases with larger models. In the Socket case study, inference time reaches to more than 9 minutes for G=1. Walkinshaw et.al [29] believe that the traces are used to infer the model for Socket are “richer” than other examples. They consist of an alphabet of 56 events and an average of 32 variable values per event. These numbers of events and variable values (which are at least twice as many as other case studies) can easily lead to generating larger models and subsequently larger inference time.

It can be concluded that even if the time it takes for ReHMM to infer a model from software executions could be increased for large merging scores or traces including high number of variable values per event, it is still an accurate solution for addressing several software engineering related problems. For example, in a case that the inferred model is used to produce regression tests with high code coverage, the model only needs to be generated after each regression to cover newly developed portions. In other words, there is no need to re-produce the model on a regular basis, because the behavior of the model will not change unless a software component is updated or replaced. Therefore, spending time periodically, ideally offline, to generate an accurate model of the system, which could be used for software testing purposes or for detecting a system's architectural anomalies, is already recommended and applied in several software engineering studies [2], [95].

2.11 Threats to Validity

Some potential threats to the validity of our research and the method of addressing them are discussed in this section. In this study, we are principally concerned with three types of threats:

- Threats to the internal validity
- Threats to the external validity
- Threats to the power of the experiment

ReHMM similar to other dynamic inference techniques suffers from some limitations. It takes a set of software execution traces as input and infers models that generalize upon them. This procedure can easily lead to generating a partial view of the software behavioral model. The mentioned drawback has been addressed by combining the FSM with the values of the associated parameter or data-state variables. But with respect to the internal validity, there are qualitative

data constraints (such as class labels) in these models; the impact of these constraints on the method's accuracy cannot be directly evaluated. To address this issue, we added an additional quantitative value to the model, which provides insight into the model's accuracy by calculating the amount of forward probabilities and subsequent transition probabilities using Q-values (where Q-values reflect the degree of change between transition functions). The result proves that using a Q-value calculation procedure has been a significant aid in inferring an accurate model, since it increases the probability of discovering more diverse paths in each inference process. Therefore, generating a model with high accuracy may show the effectiveness of using Q-values to identify the functions governing transitions and generating the Probabilistic FSA of software systems. On the other hand, the threats to the external validity for our research are centered on the limitations of dynamic analysis. In dynamic specification mining, the system and its complete set of execution traces are not necessarily known. Therefore, the lack of complete traces describing the software behavior is a major limitation as identified in [10] and [37]. Walkinshaw et al. [28] reduces the risk of using incomplete trace samples by using k-folds cross validation. This approach is also replicated in this study. We also calculated the log-confidence measure [90], [91] to evaluate the sufficiency of the applied traces. In addition, choosing seven different case studies from two diverse programming languages shows this experiment is not dependant on any specific programming language. Therefore, the selected classes can be considered as representative of many others, but not all, possible software systems.

The third threat represents the *power* issue, which can lead to Type II errors in situations where a sufficient number of samples are not available. In order to address this issue, a k-fold cross validation approach and different configurations of learners are used, and several models are generated during every iteration of the inference algorithm (overall 700 models). In addition, in

order to assure that the inference approach is capable of handling traces of different lengths, diverse systems with different average trace lengths (from 14 to 44) are applied in this study.

2.12 Conclusion

Several techniques currently exist which are able to generate behavioral models from software execution traces. In this study, we introduce a new approach named ReHMM for inferring Extended PFSAs. This technique combines two different approaches: RL and HMM to generate an accurate behavioral model of the desired system. ReHMM is a new approach, training an RL-based HMM on data obtained from each class label of an execution trace.

Applying RL helps in exploring the transitions, which trigger more changes in the model; and hence, it helps to detect those functions that govern transitions. Then, we explore the ability of HMMs to estimate a model that maximizes the probabilities of identifying the correct traces.

This study makes a contribution to research in the area by offering a new algorithm to infer an accurate EPFSA from software execution traces. In addition, it suggests a solution to the problem of the missing state-action value by using Q-values in designing an Extended Probabilistic FSA.

In order to evaluate the proposed technique, we used traces extracted from seven modules in two different programming languages. Both systems have been used for inferring EFSAs [28] and [29]. The reason for choosing the same modules was the existence of a preliminary study, which we used to evaluate our approach. According to the results, our proposed technique outperforms other inference algorithms in terms of the BCR measure (used to evaluate the accuracy of the model). We also compared the generated models using the proposed approach with the models produced using a state of the art PFSA generation algorithm (sk-strings) in terms of accuracy. Again, the results confirmed the ReHMM's ability in generating the more accurate PFSAs too.

3 Inferring Reward Augmented Behavior Models from Log Files in Web Applications

3.1 Introduction

Many users with different needs and interests browse web applications by navigating through different pages. Navigational anomalies, deadlocks and unexpected inter-connections can easily lead to user dissatisfaction and, subsequently, lost audiences.

However, it is almost impossible to accurately predict and address all of the users' interaction expectations. Inferring a model by knowing and predicting users' behavioral patterns is required in order to understand users' interests and build applications addressing a wide range of requirements.

Several studies have been conducted to propose different inference approaches in order to model user behaviors by monitoring the usage of an application and consequently mining the collected data to extract user behavioral patterns. Applying data mining techniques on the data collected from user side or proxy servers in order to extract usage patterns is one of the proposed inference approaches [96]. The client side data can be collected using JavaScript or by modifying the source code of an existing browser. Some other existing solutions mine server-side log files as the historical user-interaction data and extract hidden behavioral patterns. For example, [2-7] generate probabilistic user behavioral models or user interest models from log files.

However, even if we have an accurate user-behavioral model available addressing many design questions remain problematic without being able to augment the generated model with

appropriate metrics and measures. For example, questions such as (1) Which pages[11] of the web application are poorly performing? (2) What are the landing pages? and (3) What are the common sessions?

In order to answer such questions, some extended inference techniques are suggested, which not only infer the user behavioral model but also **manually** augment models with a metric called reward values, .e.g [7],[8]. In general, rewards are non-negative values that are estimated and assigned to a model's states to provide insights on the benefits or losses associated with each state of the model. In our study, the benefits or losses are considered as the new and different content that a web page offers to the users in order to attract them, as compared to other web pages. **Critically our approach is fully automated!**

Later we show that the page reward value can be mapped to the popularity or accessibility of web pages. Therefore, high reward values can demonstrate the successful implementation of users' requirements, while low reward values for the less popular pages can indicate possible anomalies in the application design. Therefore, reward augmented behavioural models can represent a general abstraction of the model, which can be used to analyze user behaviour patterns.

Google Analytics[12] also provides a solution to answer the above types of questions. It tracks users' navigation actions by instrumenting web pages and uses a page tagging approach to gather website traffic data. In this case, a snippet of JavaScript code needs to be **manually** added to every page of the website. However, existing solutions including Google Analytics are either

[11] A web page in this study refers to the dynamic web page where its construction is either controlled on the server-side or the client-side

[12] https://www.google.ca/analytics/

limited to inferring predefined users' navigational profiles [97], [98] or they lack an automated inference process to generate reward augmented models.

In this study, we provide a comprehensive **automated** approach to generate reward augmented behavioural models from log files. In other words, this study contributes to current research in user-intensive behavioral models in the following distinct ways:

- This approach provides a new method to infer probabilistic user behavioural models using Markovian processes, and verifies the quantitative properties of users' behaviors using probabilistic model checking.
- This technique facilitates a model generation process by the use of a Reinforcement Learning (RL) based approach to automatically and incrementally learn the reward values from data, which represents the users' interactions with web applications.
- It uses the differences between the contents of different web pages to indicate the "attractive" pages from a user point of view.
- It is fully automated and requires no manual intervention neither in terms of code instrumentation nor reward estimation.
- In order to investigate the performance of the proposed approach, we have applied it on an enterprise scale, web and mobile application called "MyUAlberta". It is believed that this is the first time such processes (generating reward-augmented behavioural models from historical log files) have been applied to a real-world enterprise size application. Previous research [2], [99]–[102] has only considered small, artificially generated, data sets.
- In addition, the results of this investigation indicate that the calculated reward values are compatible with the values extracted from Google Analytics in determining a page's

importance. In other words, pages that are marked as interesting with a high number of page-views are also highly ranked by reward values.

The remainder of this chapter is organized as follows. Section 3.2 contains research motivations and the problem description. Section 3.3 overviews the steps of the proposed approach along with the background information relating to existing user behavioral models. Section 3.4 introduces a running example, used throughout the chapter to explain and evaluate the proposed approach. Section 3.5 provides a detailed description of the inference approach. Section 3.6 discusses the empirical evaluation of the proposed approach and presents the experimental results. Section 3.7 reviews related work and state of the work model inference approaches, while Section 3.8 summarizes the main contributions of the study and provides some thoughts on the ongoing research work.

3.2 Problem Statement and Research Motivation

User behavior models, which are generated from navigational patterns found in web applications, provide solutions to several software engineering problems. A navigational pattern is a record of where a user visits in an application. The pattern is extracted from the whole procedure of browsing the website, from the start to the end of a user session This information can be obtained from the server's log of a web application, which documents the history of user interactions and behaviors. Several inference studies have been performed to model users' behavioral flows using server log files [99], [100]. In this study, we try to provide a new user-behavioral inference approach, which has the following advantages compared to previous work:

Ideally, a user behavioral inference process should not require to instrument the application's web pages in order to generate user behavioral models. To instrument a web page, the source

code of the program is modified with additional commands. The purpose of instrumenting the web applications is data collection. An inference approach, which only uses data from log files, does not require access to the source code of the applications.

Providing a non-instrumented inference approach is preferably acceptable while achieving appropriate results. The results of the above mentioned inference approach, which only uses data from log files must be consistent with the results of an instrumentation-based approach. In this study, Google Analytics is used as an example of an instrumentation-based approach.

Web application evolution can be done by upgrading an application already in service or by releasing a new version or derivative. Any model generation approach should support the evolution of web applications. This can be helpful in sustaining web applications.

When users interact with a web application, the history of their requests and behaviors are stored in web server logs. In enterprise-scale web applications, log files will have millions of entries per day. It is essential that any model generation approach support the utilization of such large-scale server logs.

Behavioral models are normally generated by collecting data either from system instrumentation or from log files. To instrument a web page, it is necessary to insert additional code fragments into the source code. However, this can be difficult when the source code is inaccessible or too complex to instrument. Many software systems still in use were developed using technologies that are now obsolete. This becomes an issue when the original developers or the source code is no longer available, or source code only exists with outdated documentation, which is the only reliable source of information. These situations are frequently encountered, particularly in software systems that have been developed years ago. For instance, consider a web application,

which is developed and maintained over years by a single developer. If the developer leaves the company the company will encounter a situation called “maintaining the legacy code”. In this situation different developers need to familiarize themselves with the code they have never touched before. Another likely scenario is when an application is being licensed to a company without providing the source code. In such a case instrumenting the source code becomes very difficult and time consuming. Furthermore, legacy codes can be very difficult to read since they are written without exploiting object-oriented techniques. This old coding approach leads to lengthy and complex code that is difficult to understand and instrument. Although accessing source code of these systems is not always feasible, it is still possible to generate behavioral models for these systems from their log files.

A model generation approach needs to support the evolution of the web applications. In other words, a behavioral model should be generated incrementally during the evolution, and should play a role in the application’s evolution procedure. Consider an application in which, a specific link needs to be added to the main page. By adding the link, the new behavior model should be generated incrementally. This approach makes the model generation process quick, flexible and possible in the early phases of the application life cycle. As an example, consider a car dealership web application. By analyzing its users’ behavior model, pages with more visitors, will be determined and related advertisements will be added to the target pages. Then the new model will be created incrementally.

Analyzing the model also helps to detect design anomalies resulting in dissatisfaction among users. For instance, there might be some pages, in which users are being prevented from leaving the page without closing it. Such pages are called deadlocks [103]. Detecting deadlocks can significantly help in addressing design anomalies and providing solutions to retain users.

Therefore, generating behavior models in an incremental and analyzing them regularly play an essential role in the flawless evolution of the web application.

The model generation approach should also support the large-scale web applications. These applications have a large number of entries per day. Analyzing the behavior of numerous users in a large-scale application can provide information that may be helpful in understanding the users' requirements. From the business point of view, it's critical to understand how people interact with the company's web application, especially in cases where such applications are considered large or enterprise.

Understanding user behavior helps in improving the user experience; refine features and contents, and building a user-oriented product. Google, Microsoft and Facebook are examples of large-scale companies, which are using behavior model generation approaches to understand the behavior of their users.

Behavior models are also used to identify potential issues with the content or the design of enterprise applications. In the case of a commercial company's web application with hundreds of thousands of visits per day, analyzing the behavioral model can reveal difficulties users are experiencing while browsing the application or their preferences in the design of the user interface (UI). For instance, by tracking users' behavioral fellow, it can easily be detected if users prefer to see more information about a product on the same page that the product is placed on instead of clicking on the "more information" link, which shows the information on a separate page.

Developers can easily address such issues and increase users' satisfaction level. Satisfied customers are more likely to stay with the company and contribute to the company's success.

Generating and analyzing user behavioral models make detecting such issues possible in large-scale applications with huge log files recording user interaction data.

Google Analytics is a state of the art approach to infer user behavioral models. But it should be noted that generating such models using Google Analytics needs instrumenting the source code. Google Analytics is a web analytics service offered by Google that tracks and reports website traffic. Using Google Analytics users can create and review online campaigns by defining different conversions (goals). Goals are used to measure how well the web application is targeting the predefined objectives like sales or a specific location loads [104]. Google Analytics can also be used to identify poorly performing pages preventing the web application from reaching the defined targets (goals).

The behavioral models generated using non-instrumented inference approaches should be consistent with the results of Google Analytics or other equivalent systems. Therefore, one of our objectives in this study is to produce an algorithm, which has comparable performance to Google Analytics.

In order to evaluate the correctness of our proposed inference approach, we considered the compatibility of the approach with the results, which were extracted by Google Analytics from an instrumented website. Another objective in this study is to produce an automatic algorithm, which supports enterprise size web applications and its future evolutions. Section 3.6 discusses the evaluation of the proposed approach on a large-scale case study and presents the experimental results.

3.3 Augmenting Behavioral Models by Reward Values

In this section, we introduce our proposed inference approach with a short introduction about current user behavioral model generation procedures.

3.3.1 User Behavioral Model

Different users behave differently in their interactions with web applications. They browse web pages based upon their needs in a specific time frame. Therefore, providing a model representing possible user behavior and latent patterns with a graphical, and traceable, representation can be helpful. Several techniques are suggested to track users' navigation actions and generate models containing the paths users have taken through a web site. Some of these approaches instrument the web pages to collect users' interaction history, while others mine the server-side log files to extract interaction patterns. Moreover, the inferred models are also represented in various ways from tree-based data structures [105] to different types of probabilistic models [2], [97], [106]. To allow the universal application of this approach, we only assume the availability of server-side log files as the system input. This assumption also implies that the system requires no modification of the existing configuration, which is often a barrier to adoption. In terms of the output from the system, we are extracting probabilistic (state-oriented) models, analyzing them and augmenting their states with reward values to accurately represent the user behavioural patterns.

3.3.2 Proposed Model Inference Approach

Given our objectives, we commenced our research by considering pre-existing systems as partial solutions to address the raised concerns.

Our framework was designed and implemented to incrementally generate user-behavioral models for user-intensive web applications, and to overcome the limitations of former approaches.

The main steps of the proposed framework are shown in Figure 6 and briefly discussed in the following paragraphs. It is worth noting that steps 2 and 3 are intertwining and do not occur sequentially, but for the sake of clarity they are explained separately. Later in the chapter of the thesis, each step will be elaborated in a separate section.

1. **Identifying the initial parameters and processing the log-file:** at the first step, a set of Atomic Propositions (APs) is used to associate semantics to the URLs occurring in the log file. APs can be defined by the system expert or by automatically considering the URL of the page as a proposition. Also the system expert can define a set of user-classes to characterize different groups of users. For example, users' agents (internet browsers used to view the web pages) and locations could be considered as two user-classes. Classes categorize users based upon a set of common features. However in order to automatically infer a reward - augmented model, which is not limited to a specific scope, defining user-classes can be ignored. Also in this step, input logs are processed and classified. Each row of the log file is clustered into groups univocally identified by the sets of atomic propositions.
2. **Generating the behavioral model:** the model inference engine analyzes the processed log file and generates a Discrete Time Markov Chain (DTMC) for each "user-class", defined in the previous step.
3. **Calculating and assigning reward values:** Concurrently with generating each state of the DTMC, the corresponding reward value is also incrementally calculated and assigned

to the state. A Reinforcement Learning (RL) based approach is applied to automatically estimate the reward values for each state (i.e. web page).

4. **Analyzing the model:** when the DTMCs are generated and annotated with reward values, the analysis engine evaluates the properties of the interaction patterns against the inferred models using probabilistic model checking. The probabilistic model checker not only evaluates the correctness or incorrectness of a property, but also provides insights on the users' behaviors and on the impact of these behaviors on the reward values. Any probabilistic model checker can be applied; but in this study, we use PRISM [103].

3.4 Running Example

In order to define and demonstrate the proposed approach throughout this chapter, we introduce a real world, enterprise application called "MyUAlberta". This application includes several features for helping students and staff at the University of Alberta, enabling them to gain access to campus-related information through an easy-to-browse dashboard. The application has been active since September 2014 and more than 100000 app installs and more than 40000 monthly page views have been reported since then.

Users can view university news, events, and maps; and search for people who are registered as students or staff at the university. Students and academic staffs can view course details including seat availability and check their timetable. These features along with several other features are represented in the MyUAlberta mobile application and website.

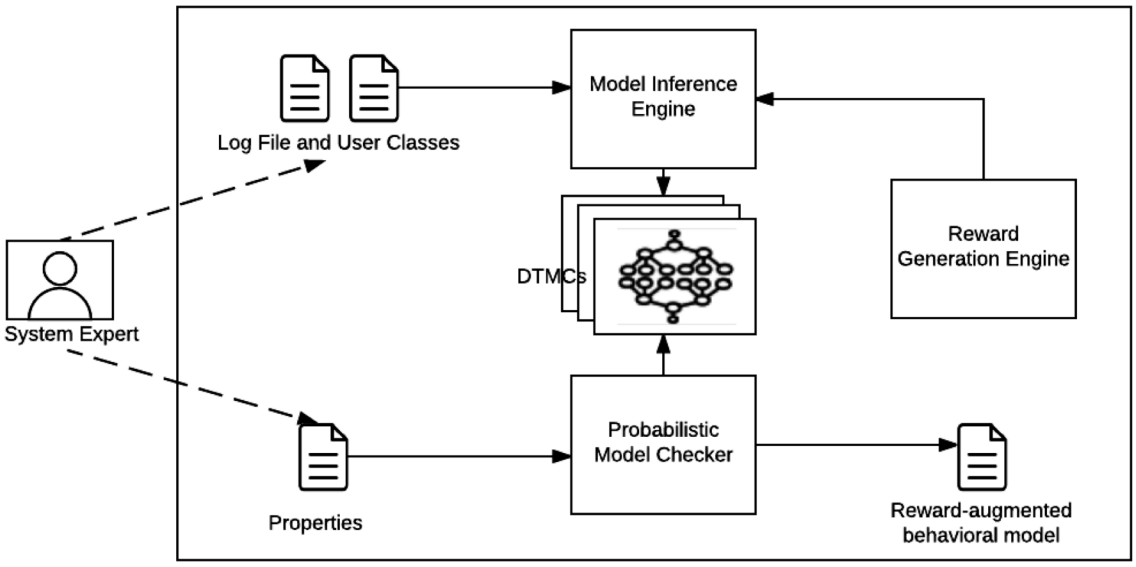


Figure 6. The Framework of the User Behavioral Model Inference Approach

This application has 58,498 iOS and 12,097 Android registered users with unique mobile devices. It is worth noting that these numbers do not include web users who only use the application through the web portal. In addition, the application contains 18 main modules; each provides a link to an external resource (e.g. link to the university website) or represents an in-app feature. As an example of an in-app implemented feature: students can use this application to login to the University authentication system and see their classes' schedule, marks, and course lists (Registrar module). They can also review course content and take quizzes online using the eClass module. Library, Events, Student Services, Find a Person, ONEcard, News, Athletics, Social Media, and Photos and Videos are other modules that provide different services to university students and staffs. Transit and Campus map modules are also two popular features in this application; they help users find out the departure time for several bus and LRT routes, as well as the campus-wide geographical map. Users are able to customize modules' order in the main menu, receive emergency push notifications, or send feedback about their experiences browsing the application. Therefore, according to the large scale of the application and its

numerous users, analyzing the behavior of the users would provide information that could lead to the fruitful future of the application in terms of improvements.

We extracted the server log files for the time period of one month composed of more than 120,000 lines. It can be easily anticipated that the generated behavioral model using such log files would be massive and complex, so manually extracting the behavioral patterns and assigning the reward values to them is neither possible nor accurate.

3.5 Inference Details

In this section we elaborate on the inference steps briefly discussed in the last section.

3.5.1 Identifying Initial Parameters and Processing Log File

In order to infer a model of users' behaviors, the inference approach needs a list of the interactions between the users and the web server of the application in the Common Log Format (CLF). This file consists of the rows and each row represents a request submitted by a user to the web server and contains the IP address, timestamp, requested URL and client's device information. Our proposed inference approach clusters each row of the log file into the groups identified by a set of Atomic Propositions (APs) [2]. It uses several code fragments called *filters* to indicate the set of atomic propositions, which can be associated to the relevant requested URLs in the log files. Filters are parameterized with a regular expression to only identify the URLs matching the expression. For example, the proposition *home* is going to be used as a label for a row in the log file, which contains the requested-URL that will lead the user to the home page of the application. This procedure helps in: (1) Identifying the requests corresponding to the same URLs and clustering them into the same group. (2) Detecting and filtering out the rows that

belong to CSS, JavaScript or any resources that are irrelevant to the users' interactions with the web application. The URLs and their corresponding atomic propositions for the MyUAlberta application are provided in Table 12.

Table 12. The URLS and Their Corresponding Atomic Propositions in MyUAlberta Application

Atomic Proposition	**URL**
home	…/home/
athletics	…/athletics/
social	…/social/
transit	…/trnst/
news	…/news/
video	…/video/
emergency	…/uaemergency/
calendar	…/calendar/
people	…/people/
login	…/login/
eclass	…/eclass.srv.ualberta.ca/portal/
map	…/campusmap.ualberta.ca/
onecard	…/myonecard.ualberta.ca/
caps	…/capsconnections.ualberta.ca/
studentservices	…/stustrv/
customize	…/customize/
feedback	…/MyUAlbertaFeedback/
search	…/search/
photos	…/photos/
fullweb	…/ualberta.ca
error	…/kurogoerror/
library	…/library/
registrar	…/registrar/

Our inference framework also contains two default classifiers to classify the users based upon (1) the *user-agents* (e.g. Mozilla, Firefox) and (2) the users' *location* extracted by geolocating the IP addresses [2]; more classifiers can be easily added. Classifiers help designers to extract domain specific information about the users by classifying users into several different customizable classes. For example, using this approach, we would be able to analyze specific users' behavioral patterns for clients who logged into the application using Chrome.

3.5.2 Generating the Behavioral Model

In the model inference step, the inference engine incrementally generates a set of Discrete Time Markov Chains (DTMCs) [2], [107].

DTMCs are probabilistic finite state automata, which follow a Markovian process and represent the users' behavioral patterns. Discrete time Markov chains are suitable options for representing the user behavioral models, because:

- The transition from one state to other states in the model only depends on the current state. Therefore, in a user behavioral model, the probability distribution of the next page (the user might visit), only depends on the links and content provided in the page that the user is currently browsing. This perfectly matches the user behavior pattern definition, which illustrates the users' movement flow from one state (page) to another.
- The system evolves through discrete time steps. In user behavioral modeling, we are interested in analyzing user behaviors at discrete time intervals to predict the next movement of the user solely based on the current state. Therefore, changes to the system cannot occur at any time along a continuous interval.

In our study, DTMCs are also annotated with a numerical value called a *reward.* Rewards indicate the quantitative value (benefit) of visiting a specific page in the web site or being in a specific state of the model. In the research conducted by Ghezzi et al. [2], rewards are manually determined and assigned by the system designer to the states of the models. Accordingly, a DTMC which is augmented with rewards is a tuple (S, P, L, ρ) where:

- S is a set of states, and $s_0 \in S$ indicates the initial state;
- $P: S \times S \rightarrow [0,1]$ is the probabilistic matrix indicating the probability of the occurrence of a transition between two connected states.
- L is a function which maps a state to a set of atomic propositions.
- ρ is a reward function with associates a non-negative number to each state.

In this study, we also infer a DTMC for each class defined by the classifiers.

In order to start the model inference process, an initial DTMC is generated. The initial DTMC consists of (1) two initial states: s_0 (initial state) and s_e (end state); (2) a zero transition matrix P indicating the probability transitions between states; (3) a set of state labels indicating the start and end labels $L = \{start, end\}$ and (4) a reward function ρ which assigns 0 to both start and end states as an initial reward value. Assigning 0 as a reward value to the initial states illustrates that the value of states is not calculated yet. Then, the initial DTMC will be incrementally developed by processing each row of the log file and adding more states and transitions to the model. The following paragraphs provide more details about this procedure:

- First, when a raw r is processed, the algorithm assumes that the IP address in the raw corresponds to a new user unless the IP address has been previously detected within a predefined *time-window*. A time window is the **minimum** time span between timestamps

that is defined by the system expert to identify the requests that are issued by two different users but the same IP addresses. In another words, when the time-window for a certain IP address expires, the algorithm assumes that the user associated with that IP address left the system [2]. In this study, we assumed that the time-window is equal to 1 minute, which is equal to the minimum session timeout in Google Analytics. This should not be confused with the default session duration in Google Analytics, which is equal to 30 minutes. If the previous step considers r as a request issued by a new user, the algorithm adds a new state to the model, and labels it by the set of propositions associated with r. At this point, it considers the start state s_0 as its parent state. But if the request belongs to a known user, the algorithm still builds the new state with the same labels, but considers the latest state associated with the previous request as its parent state [2].

- Then the transition probability p_{ij} is assigned to the transition between s_i and s_j. p_{ij} is equal to the ratio between the number of transitions between s_i and s_j the total number of transitions with source state s_i [27].
- During the inference procedure, if the times-window expires for a certain IP address, the algorithm generates a transition from the current state to the end state s_e and updates the transition probability.
- The previous steps will be repeated until no new request is found in the log file.

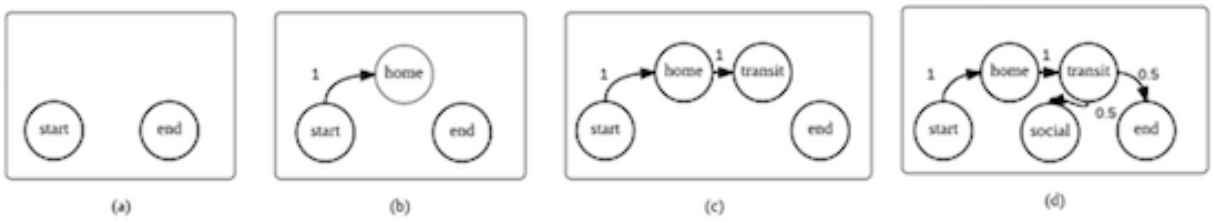


Figure 7. An excerpt of the model inference procedure for MyUAlberta application

Table 13 shows an excerpt of the MyUAlberta log file containing the user' IP address, the requests' timestamps and the requested URLs. This log file represents the interaction of a user with the application.

Table 13. The URLS and Their Corresponding Atomic Propositions in MyUAlberta Application

1.1.1.1 - - [24/Jan/2016:06:50:56 +0700] "GET /home
1.1.1.1 - - [24/Jan/2016:06:51:00 +0700] "GET /trnst
1.1.1.1 - - [24/Jan/2016:06:51:04 +0700] "GET /social

Figure 7 depicts implementing the above step on MyUAlberta case study. In this case, as the log file is processed, the initial DTMC is generated by building *start* and *end* states (See Figure 7a). When the first line of the log file is read, since this is the first time a request to the home page is getting processed, a new state labeled *home* is generated. In other words, since the IP address of this request has not been already encountered, the algorithm connects it to the *start* state (Figure 7b). At the same time, the inference engine assigns a probability of 1 to this transition. This is due to the fact that, there is no other state with the source of *start* state yet. As the algorithm processes the second row of the log file that contains the same IP address, it adds a new state labeled *transit* to the DTMC and considers the *home* state as the source state for it. The transition probability of the new connection is also 1 (see Figure 7c).

The third row of the log file again belongs to the same IP address but containing a request to load the social media page. Therefore, the algorithm adds the new state called *social* to the model in a same way as previous states. The only difference is that since this state is not the first state getting yielded from the *home* state, the number of the outgoing states from home should be

divided by the transition probability and all of the corresponding transitions should get updated. Thus, the probabilities of transitions derived from *home* state are equal to 0.5. This means that the probability of the transition between the home and transit states will be also updated to 0.5 (see Figure 7d).

When the time-window for a specific IP address expires, the algorithm assumes that the user left the application. Therefore, it generates a new transition that connects the latest discovered state (which is associated with this user) to the *end* state. Figure 7d depicts this step.

During the inference process the reward values are also calculated and assigned to the states of the model, but for the sake of clarity we present the remaining steps in the next section.

3.5.3 Calculating and Assigning Reward Values

As mentioned previously, the necessity for manually producing the reward signal is a major problem with the previous inference approaches, such as [2].

In order to illustrate the use of rewards in user behavioral models clearly, we first provide an example related to a sell and buy web site. Assuming that the goal of the web site is to increase the number of advertisements, the designer can assign reward values to states by considering the number of advertisements in each page. For example, if there are 10 advertisements in the homepage, the system expert associates the reward value 10 with the proposition homepage. Accordingly, other states of the model also get annotated by the number of advertisements their corresponding proposition contains. Depending on the web site's goals and missions, designers only define one technical or non-technical metric of interest and assign rewards based upon this metric only once during the setup phase of the inference procedure. Therefore, in order to recalculate reward values based on the new metrics: (1) system experts should recalculate reward

values, and (2) models should be regenerated. In such situations, defining an approach, which is able to represent the rewards values of the states from a general perspective, would be very helpful. However, automating the calculation process makes the approach more effective specifically for large-scale software systems.

The following paragraphs outline our proposed technique to solve this issue. We utilize a reinforcement learning strategy to automate the estimation of the rewards signal. Therefore, we first provide a quick background about the applied techniques and definitions.

3.5.3.1 Reinforcement Learning

This section provides an overview and definition of Reinforcement Learning.

RL is located between supervised and unsupervised learning to learn what to do to maximize a numerical *reward signal* [66]. The learner does not know what actions to take to reach the goal of maximizing the reward signal and only can pick and try actions to detect those increasing the accumulative reward [67].

Reinforcement learning algorithms are defined in an iterative way not by characterizing learning methods, but by characterizing a learning problem. The agent and the environment are interacting continually: the agent selects actions and the environment responds to the actions, presents new states to the agent and which give rise to rewards. This cycle is repeated as part of a Markov Decision Process (MDP) [66], [68]. MDPs are used as stochastic extensions of finite automata or Markovian process to model the decision making process and solve optimizing problems. They are augmented by actions and rewards so that they consist of actions, transitions, labels, and states. In the following paragraphs, some definitions are introduced that are helpful in demonstrating our proposed approach in the next session.

Definition 10. **Markov decision process (MDP).** MDPs provide a mathematical framework to model the decision making process in situations where outcomes are partly random and partly under the control of a decision maker. They are useful in addressing a wide range of optimization problems solved via dynamic programming and reinforcement learning [66].

More precisely, an MDP is a discrete time stochastic control process. In other words, an MDP contains:

- A set of possible states S.
- A set of possible actions A.
- (Transition) probability distribution, $X: S \times A \times S \rightarrow [0,1]$ giving for each state and action. It computes the probability of reaching state s' by performing action a in state s and is denoted as $X(s, a, s'),\ s, s' \in S, a \in A$.
- The immediate reward received after transitioning from state s to state s' due to action is denoted as $R(s, a, s')$.

Therefore, an MDP has a set of states. These states will play the role of outcomes in the decision making approach as well as providing information, which is necessary for choosing actions. For example, in the case of a robot navigating through a building, the state might be the room it is in, or the x and y coordinates. For a factory controller, it might be the temperature and pressure in the boiler.

A MDP also has a set of actions. When the process is in state s, the decision maker may choose any action a that is available in state s. The process responds at the next time-step by randomly moving into a new state s'. So, $s_t \in S$ denotes the state at time t [68]; according to this definition of a Markovian process, we would have:

$$P(s_{t+1}|s_t, s_{t-1}, s_{t-2}, \dots) = P(s_{t+1}|s_t) = X(s_t, a_t, s_{t+1}) \quad (19)$$

This process is called Markovian, because the probability of reaching state s' depends only on the current state s and the action a. In other words, the next state is independent of all the previous states and actions and only the current state predicts what the next state will be [72].

Definition 11. **Reward Function.** R specifies the reward the agent receives by performing an action. So, $R: S \times A \times S \rightarrow \mathbb{R}$ presents the reward function that computes the immediate utility of an action, indicating the intrinsic desirability of that state. So an MDP can be denoted by the tuple $\langle S, A, T, R \rangle$ depicting it as a state transition graph [68].

In this study we are interested in calculating the value of performing an action in a specific state to compute reward values. In order to achieve this goal a new function called state-action value function needs to be defined. Since the accurate definition of this function requires a background on two other definitions: policy function and state-value function, in the following paragraphs we discuss both concepts respectively.

Definition 12. **Policy.** The goal in a MDP is to find a function called policy, which determines which action to take in each state, so as to maximize the reward function. Policy map π gives the probability of taking action a when in state s:

$$\pi: S \times A \rightarrow [0,1]$$

$$\pi(s, a) = P(a_t = a | s_t = s) \quad (20)$$

Definition 13. **State-Value Function.** The state-value Function $V^{\pi}(s)$ specifies the value of a state is equal to the total amount of reward a learner can accumulate, starting from that state. We can define the value of a state, under a policy π, formally $V^{\pi}(s)$, as [67]:

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\} \qquad (21)$$

Where:

E_{π} is the expected return earned by following policy π and discount factor γ, $0 \leq \gamma < 1$. This models the fact that future rewards are worth less than an immediate reward. Similarly, in order to calculate the value of performing an action a in state s, a **state-action value function**: Q: S×A → [0,1], can be defined as:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right\} \qquad (22)$$

All RL-based algorithms are based upon providing an approach for appropriately estimating **state-action value functions**. This has led to the exploration and production of several different estimating methods and techniques. One of the most popular of these is ***Q-Learning*** [69], which is an off-policy Temporal Difference control algorithm. In other words, Q-Learning is able to estimate Q-value functions (Q-learning based estimations of the state-action value function) without requiring an initial model of the environment.

Additionally, *Q*-learning can handle problems with stochastic transitions and rewards without requiring any adaptations. It has been proven that for any finite MDP, *Q*-learning eventually finds an optimal policy. This means that the expected value of the total reward that has been returned over all successive steps is the maximum achievable.

In this situation, because of the lack of known transition and reward models, the algorithm handle problems with stochastic transitions and rewards and uses exploration and sampling approaches to learn the required model. Therefore, Q-learning finds an optimal policy for any finite MDP and estimates the agent's Q-value function based on the Q-value estimation of an

action. In other words, using the above definitions equation 23 is inferred. This process is incrementally evaluated as follows [68]:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q_k(s_t, a) - Q_k(s_t, a_t) \right) \quad (23)$$

Where, α $(0 < \alpha \leq 1)$ is the *learning rate,* which determines the extent to which new information can override old information and how fast we modify our estimates [70]. In the next section, we illustrate how we use Q-value functions to incrementally calculate reward values.

3.5.3.2 Automated Reward Calculation Algorithm

Because of the proven ability of Q-value function to converge to an optimal policy [71] and estimate the state-action value-function in free model problems [69], we have used Q-learning to estimate the reward values in this study. In other words, the problem of estimating the reward values for the user behavioral model needs to be addressed using an approach, which: (1) is able to incrementally learn the values from the current state of the model; and (2) can easily get configured to generate meaningful reward values for web applications. Therefore, Q-value function is an appropriate option to address this issue.

In order to present a Q-value function, which fits the behavioral model generation process, we have modified the Q-value function (5) as below:

Equation 24 illustrates how the Q-value function has been used in this approach to calculate the value of state s_i, which is called $\rho(s_i)$:

$$\rho(s_i) = 1 - similarity\ (CrawlResultsA, CrawlResultsB) + \gamma \max \rho(s_{i+1}) \quad (24)$$

Since the reward calculation algorithms has been synchronized with the model-inference engine in terms of being executed every time a new state is generated, the rewards are also calculated and updated incrementally during the model generation process. Therefore:

When a pair of states is created, a function (StateMatcher) searches in the regular expression library to find the request URLs matching these states. For example, if the state label is *library*, the method detects the corresponding URL that has been requested by the users; in this case, that is *"www.myualberta.ualberta.ca/library"*.

When both URLs are retrieved, another method (Crawl) is called. This function crawls (spiders) the detected URLs and collects all words and links on the pages and stores them in two Strings (CrawlResultA, CrawlResultB). Crawlers have been previously used in mining behavioral models from web applications [108], [109], but to the best of our knowledge, this is the first time they are used to estimate the reward values of behavioral models. It is worth noting that using crawlers as the model inference approach limits its application to small-scale software systems and therefore, it cannot be applied to large scale applications with complicated DOM trees and huge log files. In such cases, crawling each page browsed by a user and creating its corresponding DOM tree can easily leads to non-determinism in the model and huge data-space usage. However, in the reward calculation procedure, the agent only crawls the two states of the model at the time, which leads to saving time and space in large-scale applications. Additionally, in this study we used CRAWLJAX [110] as the crawler implemented in the reward calculation step. It is able to discover various navigational paths and user interface states within AJAX applications.

After storing the crawl results, the difference between the two URLs' content is calculated using a similarity method. This function works by calculating the Levenshtein distance between two states' content. (See Table 14 andTable 15 for the reward calculation and similarity algorithms).

Table 14. Reward Calculation Algorithm

Input: Model states $(s_i, s_{i+1}) \in AP$; $i = 0$; $\rho = 0$
Output: Reward value ($Reward \in \mathbb{R} \geq 0$)

begin
For each state s_0 **To** s_{N-1} **do**
urlA⟵**StateMatcher**(s_i)
urlB⟵**StateMatcher**(s_{i+1})
CrawlResultsA⟵**Crawl**(urlA)
CrawlResultsB⟵**Crawl**(urlB)
$\rho(s_i) \leftarrow 1 - \boldsymbol{Similarity}\ (CrawlResultsA, CrawlResultsB)$
$max \leftarrow \max \rho(\delta(s_{i+1}))$
$\rho(s_i) \leftarrow (\rho(s_i) + 0.9 \max)/100$
For each (s_i, s_j) ; $s_j \in S_e$ *
merge (s_i, s_j) **if**
$(s_i = s_j)$ **AND** (Reward (s_i)=Reward (s_j)) **AND** (Adjacent (s_i) = Adjacent (s_j)) **AND** (Reward(Adjacent (s_i))= Reward(Adjacent (s_j)))
Repeat Until no new $s_j \in S_e$ is found
Repeat Until no new $s_i \in S$ is found
end

* S_e is the set of states which are already labeled with the reward values

Table 15. Similarity Calculation Algorithm

Input: Method calls in String format (c_1, c_2)
Output: Similarity Score

$Similarity\ (c_1, c_2)$**begin**
if ($Length.c_1 < Length.c_2$)
then $Swap\ (c_1, c_2)$
BigLength ⟵ $Length.c_1$
Return bigLength – *ComputeEditDistance*[*](c_1, c_2) / bigLength

* *We have implemented the "Levenshtein distance" algorithm to compute the Edit distance in this study*

The reason for applying the Levenshtein distance is it's proven capability in measuring the similarity between two strings. In this study the outputs of Crawl methods are strings too. The similarity function plays the role of the Q-value function to initiate the Q-value ($\rho(s_i)$) for state

s_i, so when the user sends another request URL from the current state (the web page the user is currently viewing), the algorithm calculates the Q-values of upcoming states (web pages) and chooses the maximum amount to learn the current states' reward.

In order to eliminate the redundancy in the model, the merging step of gkTail inference algorithm is applied to merge the equivalent states [64]. According to this state-merging procedure, two states are considered equivalent if they have the same future of length k (in our study k=1). Therefore, here two states can be merged if they share the same label, rewards and immediate future, which means their adjacent states also have the same labels and reward values. This procedure prunes the model from redundant states with the same values.

It is also worth noting that we initialize the process to give all states the same reward value (zero) and the same chance of being observed (requested by users).

Using this procedure, all reward values are calculated incrementally during the model generation process. Eventually, the inference engine assigns the rewards of states as the sum of the reward values of the propositions associated with the states. Our proposed reward calculation process not only automatically computes the reward values in an incremental way, but also uses the server side logs as the only source of the input. Moreover, the empirical evaluation (provided in the next section) shows that the reward values calculated by our proposed approach correctly represent the benefits or losses associated with the states.

Figure 8 shows the initial steps of the reward calculation process for the MyUAlberta application. As depicted, the model generation process initially goes through the same approach discussed in Section 3.5; and when a state is generated, the reward value is also calculated in an incremental manner. Therefore, first the content of the current state are crawled and compared

with the new state's using $similarity\ (start, home)$. Then, the maximum amount of previously initialized or calculated values for the next state is considered. Therefore, in this case, $\gamma \max \rho\big(\delta(s_{i+1})\big) = 0$, because there is no already traversed state after the home state. So, for instance, in a case that the user browses only one page, the model contains at least 3 states: start, browsed page and end; and the reward value of the considered page would be 1.

Moreover, in the situation that the pages are dynamically created using Ajax requests, it would be possible that the states of pages do not get registered in the browser's history engine. Therefore, only navigations between pages can be tracked. This issue is not imposing any drawbacks in our study, since our proposed approach only cares about the changes between pages in log files and does not specifically deal with the state changes in Ajax-enabled pages. However, there is always a way to consider these changes using the workaround, implemented by Ajax techniques to change the URL fragment identifier when an Ajax-enabled page is loaded. Also, as it is already mentioned, in this study we used CRAWLJAX to crawl the URLs, discover various navigational patterns and calculate the reward values within the AJAX applications.

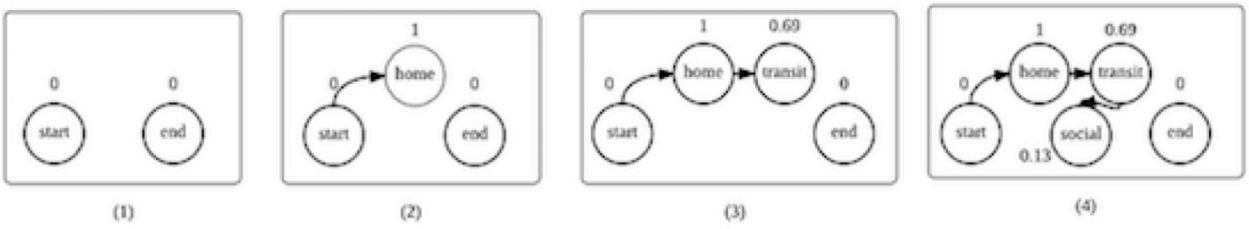


Figure 8. An excerpt of the reward calculation procedure for MyUAlberta application

3.5.4 Analyzing the Model

In order to analyze the behavioral model that has been inferred, it is necessary to identify one or more *properties* of the model, which can be evaluated by a probabilistic model checker (PRISM). In this step, the system expert defines the properties of interest using the reward-

augmented Probabilistic Computation Tree Logic (PCTL). This approach helps to identify the set of DTMCs, which are more relevant to the specified property [2], [111], [112].

In this study, we are more interested in properties that are specifying the reward values of different states in the final model. Therefore, this approach analyzes properties, which are related to the *expected values* of the rewards. This is achieved using the $\boldsymbol{\mathcal{R}}$ operator, which can be used either in a Boolean-valued query: $\boldsymbol{\mathcal{R}}\, bound\, [rewardprop]$ or a real-valued query: $\boldsymbol{\mathcal{R}}\, query\, [\, rewardprop\,]$.

Where *bound* takes the form $< r,\ <= r,\ > r$ or $>= r$ for an expression r and *query* is $=?,\ min =?$ or $max =?$.

Additionally, the *rewardprop* represents the reward property, which can be considered in following types:

- Reachability reward $\boldsymbol{\mathcal{F}}$: Reward accumulated along a path until a certain point is reached;
- Cumulative reward $\boldsymbol{C <= k}$: Expected state reward cumulated after k steps;
- Instantaneous reward $\boldsymbol{I = k}$: Expected state reward to be gained in the state entered at step k;
- Steady-state reward $\boldsymbol{S}$.

For example, in order to consider the reward value of all the states up to the state labeled as "news", following property can be used:

$$\{\}\, \mathcal{R} =_{?}\, [\mathcal{F}\ news]$$

Inside the bracket {}, the system expert can also specify the scope of the property for a defined a user class (e.g. a user agent) or leave it empty to not be limited to any specific scope.

Given a property and a set of inferred DTMCs, the algorithm identifies DTMCs, which are relevant to the scope of the property. For instance, if the scope of the property is limited to the users who browsed the application with Chrome, the inference engine only selects the DTMC, which are associated to this specific user-class. In case the algorithm selects more than one DTMC for the specified scope, the extracted DTMCs need to be merged together to build a single DTMC. Ghezzi et al. [2] suggested following approach to merge DTMCs:

- The set of states in the new DTMC is consisted of the union of the states of the input DTMCs.
- The transition probabilities in the new DTMC is calculated using the law of total probability:

$$P_T(s_i, s_j) = \sum_{1 \le k \le n} P_k(s_i, s_j) \times P_i(u_k) \qquad (25)$$

  where, $P_i(u_k)$: the probability for a user that exited state s_i to belong to the user-class u_k.
- Labels of the states in the new DTMC are the same as labels in their corresponding input DTMC.
- Reward values of the states in the new DTMC are the same as reward values in their corresponding input DTMC.

It is worth noting that the similar approach is used in our study to build a single DTMC from selected DTMCs (if there are multiple DTMCs) in the property analysis step.

As it is previously mentioned, our approach evaluates the specified property for the final DTMC using PRISM. PRISM is not only able to evaluate the truth or falsity of a property, but also can

compute the reward functions using considered reward properties. Therefore, our framework passes the property and the DTMC to PRISM, and receives the results of the evaluation through the API. In the following section we empirically evaluate the performance of this approach in a real-life case study.

3.6 Empirical Evaluation

3.6.1 Industrial Case Study

In order to evaluate the performance of the proposed approach, we applied it on MyUAlberta application as a large-scale web and mobile application.

3.6.2 Experimental Results

Since our proposed approach labels the most interesting pages (in terms of the amount of differentiation they provide) with the highest reward values, we are able to compare its performance with the results of users' behavioral-flow from the Google Analytics. It is worth noting that, we only use Google Analytics to show that our approach is assigning meaningful values to the model. So, neither our method nor Google Analytics can be replaced by one another due to their different applications in software engineering. But in cases that the Google Analytics data is not available, reward values may be able to provide meaningful information as well. In the following paragraphs, we provide the experimental results and corresponding explanations.

Table 16 shows the results of calculating reward values using the automated reward calculation algorithm for the main states (modules). These results illustrate that some pages (URLs) have higher reward values compared to others, which, indicates that these pages have provided more varied and interesting content than the others. For instance, the homepage has a reward value

equal to 0.8263, which is the maximum amount among all other pages. The transit module is ranked second in terms of the reward value, at 0.4989. In order to evaluate the correctness of the calculated reward values, we considered their compatibility with the results we extracted from Google Analytics. The MyUAlberta project has been continuously attached to a Google Analytics' account from the day the application was launched for the first time in 2014. Thus, historical data from actual interacting users is stored accurately. This data is helpful in indicating interesting pages from users' points of view, along with users' behavioral flows. Table 17 provides the number of page-views for each considered page (URL). According to this table, the homepage has had the maximum number of viewers at 35,385 page-views in a month, while the /trnst/ page has been viewed 10,563 times. These examples along with reward values of other pages and views indicate that, the pages with higher views also have higher corresponding reward values.

These results can also indicate that pages with higher reward values are offering more varied content compared to the previously viewed page. So, the users are more interested and, consequently, apt to explore the pages containing different links and text. In addition to this, many users are brought to the homepage directly (without referrals from other pages).

Therefore, in many cases, there is no actual URL exists to be compared to the homepage. As a result, the homepage receives more accumulated rewards (Equation 24). This can easily explain the reason for the high reward value for the home state. This shows that our proposed approach for calculating the reward values not only automates the calculation process, but also produces the results, which are technically explainable and compatible with the data extracted from Google Analytics (as the proof of the concept). Moreover, as provided in Table 16 (third

column), the time needed for calculating the reward value is variable. Pages with higher reward values require more time for the calculation process than those with lower rewards.

Table 16. Results of Running Reward Calculation Algorithm on MyUAlberta Case Study

URL	Reward Value	Time for Reward Calculation (seconds)
…/home/	0.8263	0.043
…/athletics/	0.0096	0.002
…/social/	0.0284	0.006
…/trnst/	0.4989	0.031
…/news/	0.0024	0.003
…/video/	0.0048	0.007
…/uaemergency/	0.0001	0.006
…/calendar/	0.0048	0.008
…/people/	0.0024	0.003
…/login/	0.1721	0.013
…/eclass.srv.ualberta.ca/portal/	0.4012	0.028
…/campusmap.ualberta.ca/	0.1274	0.013
…/myonecard.ualberta.ca/	0.0192	0.003
…/capsconnections.ualberta.ca/	0.0024	0.002
…/stustrv/	0.1522	0.016
…/customize/	0.0024	0.003
…/MyUAlbertaFeedback/	0	0.001
…/search/	0.0096	0.002
…/photos/	0.0024	0.003
…/ualberta.ca	0.0074	0.003
…/kurogoerror/	0.0072	0.002
…/library/	0.0216	0.008
…/registrar/	0.1298	0.015

Obviously, pages containing more links and text need more time for crawling the page and, subsequently, more time for calculating the reward values. Since the number of strings and links are finite in a web page, and the designed crawler only discovers the first layer of links, the time to calculate the reward values never grows exponentially [113].

Table 17. Number of Page-views for Each Considered Page (URL) on MyUAlberta Case Study

URL	Page-views
…/home/	35385
…/athletics/	419
…/social/	290
…/trnst/	10563
…/news/	507
…/video/	241
…/uaemergency/	109
…/calendar/	640
…/people/	271
…/login/	4881
…/eclass.srv.ualberta.ca/portal/	7871
…/campusmap.ualberta.ca/	2001
…/myonecard.ualberta.ca/	1356
…/capsconnections.ualberta.ca/	123
…/stustrv/	1747
…/customize/	140
…/MyUAlbertaFeedback/	15
…/search/	937
…/photos/	267
…/ualberta.ca	1159
…/kurogoerror/	282
…/library/	1100
…/registrar	4113

Figure 9 shows the users' behavioral flow extracted from the Google Analytics account, representing the most engaging content in the web application. The nodes in the behavioural flow indicate the total number of sessions for different URLs, while the connections between nodes represent the total number of sessions that passed through that connection. Figure 9 shows the home page in the MyUAberta application is the point of entry or the starting page, where users mainly start browsing the application. Then, during the first engagement (interaction), eclass and transit pages receive the highest portion of the traffic going out from the home page. Following the connections and nodes through different engagement levels demonstrates the overall traffic flow in the web application.

Figure 9. One-Month User Flow Extracted From Google Analytics

This figure also confirms the compatibility of our results with the information extracted from Google Analytics, in cases in which the number of sessions (instead of the number of page-views) is considered. In our proposed approach, we also expect sessions contain more URLs with high reward values and less URLs with low reward values. Therefore sessions which contains URLs leading to pages like home, eclass and transit are more likely to be seen on top of each interaction, indicating higher traffic in such pages in compared to the sessions containing URLs with the low reward values.

3.6.3 Correlation coefficients

The Pearson correlation coefficient of two random variables is a measure of the linear dependence or correlation between them. The value of the Pearson correlation coefficient varies between -1 and 1. When the random variables are directly (positively) correlated to each other, the value of the measure would be 1. When there is no linear correlation or there is a total

negative linear correlation between the variables, the values of the correlation coefficient would be respectively 0 and -1. In this study in order to indicate the correlations between the reward values calculated using our approach, and the page view metric, extracted from the Google analytics account, we also calculate the Pearson correlation between these two variables. The correlation coefficient matrix of two random variables (A and B) is the matrix of correlation coefficients for each pairwise variable combination:

$$R = \begin{pmatrix} \rho(A,A) & \rho(A,B) \\ \rho(B,A) & \rho(B,B) \end{pmatrix} \qquad (26)$$

Where:

$$\rho(A,B) = \frac{1}{N-1}\sum_{i=1}^{N}\left(\frac{A_i-\mu_A}{\sigma_A}\right)\left(\frac{B_i-\mu_B}{\sigma_B}\right) \qquad (27)$$

if each variable has *N* scalar observations.

Therefore, in this study the correlation coefficient matrix of reward values and page views is calculated as below:

$$R = \begin{pmatrix} 1 & 0.9396 \\ 0.9396 & 1 \end{pmatrix}$$

Which indicates the positive correlation between the two variables is statistically significant. This result again shows the importance of calculating reward values in the user behavioural models especially in the cases that Google analytics data does not exist or is not available.

Figure 10 depicts the coefficients of two polynomials that fit into two sets of data (reward values calculated using our proposed approach, and the page views, extracted from the Google analytics account). This figure can be used to demonstrate the correlations among pairs of variables. The

slopes and peaks in the fitting polynomials can be visually compared. For example, the peak in both plots belongs to the home page, which has the highest reward value and page views.

It is worth noting that we also calculated the Spearman's rank correlation coefficients [114] for the same data as a nonparametric measure of rank correlation. Intuitively, the Spearman correlation between two variables will be high when observations have a similar rank between the two variables. In this study, the value of rho (Spearman's rank correlation) is 0.92202, which indicates that the association between the considered variables (page views and reward values) is statistically significant.

According to the experimental results and correlation coefficients analysis, the outcome of our model generation approach is consistent with Google Analytics, while it is also able to correctly detect design anomalies and deadlocks. Moreover, our approach: (1) does not need any code instrumentation; (2) is applicable for large-scale web applications (MyUAlberta is an enterprise example); (3) supports web evolution processes (as it generates models in an incremental way); and (4) works (perfectly) for legacy applications (the only input it needs is the server logs).

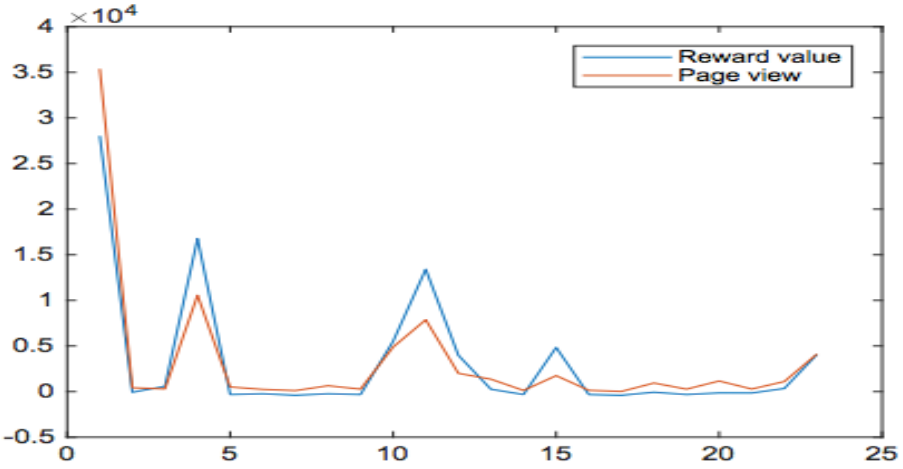


Figure 10. The correlations among pairs of variables in MyUAlberta case study

3.7 Related Work

User models represent a description of the users' behavior based upon observations. These observations arrive in two forms—explicit feedbacks and settings provided directly by the users themselves, or implicit conclusions about users derived from their actions, traits, or past behaviors.

Explicit feedback can be used to generate a model based on preferences, expressed interests, or similar attributes. For example, explicit ratings are used in systems such as Netflix and Amazon in order to produce future recommendations for a user. To draw implicit conclusions and adapt models, systems often leverage machine learning and data mining techniques.

Antwarg et al. [115] conducted research on predicting user search intentions using an attribute-driven hidden Markov model obtained from a web application used at Ben-Gurion University (BGU). Understanding user behavior and discovering the valuable information within such huge databases involves several phases that are addressed differently in various research studies: (1) data cleaning and preprocessing, where, typically, noise is removed, log files are broken into sessions, and users are identified; (2) data transformation, where useful features are selected to represent the data, and/or dimension reduction techniques are used to reduce the size of the data; (3) applying data mining techniques to identify interesting patterns, statistical or predictive models, or correlations among parts of the data; (4) interpretation of the results, which includes visualization of the discovered knowledge and transforming it into user-friendly formats.

The process of generating user models using data mining/machine learning techniques can be seen as a standard process of extracting knowledge from data where a user model is used as a wrapper for the entire process. Indeed, this is a very popular approach that has been employed in

a variety of settings. For instance, Englebrech et al. used hidden Markov models to model users' behavior based upon satisfaction in spoken dialogue systems [116], [117]. Ruvini [118] studied user interactions with the Google search engine using a SVM algorithm to infer the user's goals. While, Mobasher et al. captured web usage interests by using K-means clustering [119]. In this study, the input data consisted of user logs from the University of Minnesota Computing Science department's server. In addition, Virvou et al. [120] used K-means clustering to model students in tutoring systems, while Pennacchiotti and Popescu [121] applied machine-learning techniques to classify Twitter users. Beck et al. [122] performed the construction of a user behavioral model for an adaptive tutor using J4.8 and Naïve Bayes Classifiers on the input data collected from the interaction of students with the tutor. As a result, Naïve Bayes Classifier outperforms J4.8 in terms of accuracy. Therefore, it can be concluded that machine learning techniques in particular offer the advantage of taking diverse data and developing a classification based on observations. Another technique involves the use of basic statistical relationships [123].

In addition to this, probabilistic graphical models are used to discover and represent dependencies among different variables, such as the evaluation of the effect of a shopper's gender on their shopping behavior [124]. Dependency [122], [125] and Bayesian networks [126] are examples of such techniques, while sequential pattern analysis algorithms use time-ordered sessions or episodes and discover patterns from users' interactions with the system.

Another successful set of techniques aimed at providing formal guarantees (usually expressed in some form of temporal logic) for models that can be specified as transition systems in Model Checking [124], [127] (MC). There has been a lot of interest in the MC community for extensions of the classical algorithms to probabilistic settings, which are more expressive but significantly harder to analyze. These extensions study the Probabilistic Model Checking (PMC)

problem, where the goal is to find the probability that a property holds in some stochastic model. There is extensive work on how the PMC problem can be solved through exact techniques [101], [103], [124], [127], [128], which compute correct probability bounds. In PMC, the models are augmented with quantitative information regarding the likelihood that transitions occur and the times at which they do so. In practice, these models are typically Markov chains or Markov decision processes [124].

Another important and related research area is log analysis. Interaction log analysis has provided researchers with insight into users' behavior. These investigations typically involve examining user behavior through query log analysis [101], [129], [130]. White et al. [101] describe a systematic, log-based approach for modeling user interests during web interactions. The goal of their modeling system is predicting future behavior, and evaluating the effectiveness of different sources of contextual evidence. Using log data, Terai et al. [130], also analyze the influence of task characteristics on information-seeking behavior in the Web applications.

Examining the applicability of implicit feedback for recommender systems has also been studied [131], [132]. Applications of implicit feedback to web page recommender systems are also considered in [133], [134]. These systems typically establish historical click trails of a user or a community of users, and they assess the accuracy of statistical machine learning models, which predict future page visits [101].

3.8 Conclusion

In this chapter, we present a novel stochastic approach to (1) generate a user behavioral model from the log files, (2) automatically calculate the states' rewards, (3) annotate and analyze the models to verify the quantitative properties, and (4) address some limitations found in existing

approaches. Our proposed approach not only builds a fully automated inference framework, but also provides the following advantages as compared to other behavioral model generation methods:

- Our proposed approach is applicable on any web application of any size – new or legacy, since it is not dependent on the specific data input. In other words, a server log file would be sufficient to start the modeling procedure.
- This approach provides the capability to evaluate and verify the property of the inferred models.
- Reward values can easily add semantics to inferred behavioral models. They help in interpreting model behaviours and detecting anomalies. Calculating reward values and assigning them to the states of the model during the inference procedure would be a more accurate and time-saving approach as compared to manually assigning them by systems' experts. The proposed technique uses Reinforcement Learning to incrementally calculate the value of the reward measure using the information extracted from browsed web pages in different states. In other words, the proposed approach adds meaningful reward values to the model, explaining the real user's interest or willingness in browsing web pages.
- It is easy to apply this procedure to calculate domain-specific reward values and identify different measures.
- It makes the deadlock or anomaly detection procedure faster and more meaningful by limiting the search space to the states with low reward values. In this study, a deadlock was identified in the registrar page, which might have negative affect on the reward value in this page.

- In addition, the experimental results approved that the proposed inference approach is applicable on large-scale applications with many web pages and huge log files, and is able to generate meaningful and compatible reward values in a considered case study.

4 Test Case Prioritization Using Extended Digraphs

4.1 Introduction

Stochastic models are used in different software testing tasks. Many Model-based Testing (MBT) techniques are devolved to either generate new test cases or execute and evaluate the exiting ones [15].

The focus of this study is only on the second phase to evaluate the generated test cases and prioritize them based upon suggested factors. Researchers have proposed various methods to reduce the second phase costs. These techniques can be divided into three significant categories: *Test Suite Minimization Techniques* (which select a minimal subset of test suites with respect to maintaining the original coverage) [135]–[137], *Test Case Selection Techniques* (which identify the test cases that are relevant to the set of recent modifications) [138] and *Test Case Prioritization Techniques* (which sort the test cases by determining their execution priority based upon different criteria) [139]–[141].

Despite the existence of safe test selection and minimization techniques [142], [143], empirical evidence shows that some of them can severely decrease the fault detection capabilities of test suites or disturb the conditions under which safety can be achieved [142]. For instance, [144] evaluated several similarity-based selection techniques (STCS) and then compared the effectiveness of the best similarity-based selection technique with other common selection techniques in the literature. The results show that the best STCS is never worse than non-STCS techniques regardless of the failure rate. On the other hand, Test Case Prioritization Techniques schedule test cases in order to increase their capabilities to address significant issues [139] specifically:

- Increasing the rate of fault detection.
- Increasing the code coverage.
- Increasing the confidence of the testers in the reliability of the system under test (SUT).
- Increasing the rate of detecting high-risk defects. And,
- Increasing the early detection of faults, which are correlated with specific code changes.

Running all of the test cases and then fixing the faults may lead to the execution of unnecessary test cases containing redundant functionality and delay in the regression testing process.

Within this domain of application, the following reasons motivate us to propose a novel prioritization technique:

- Reducing the time required to execute test cases and increasing the likelihood of spending testing time more beneficially in the case of an unexpected termination of regression-testing activities.
- During the test case prioritization process, no test case is discarded. Hence, it can be concluded that the prioritization techniques do not suffer the drawbacks that can occur when the test case selection and test suite minimization mechanism discards test cases.
- In the situation that decreasing the number of test cases is considered as an objective, prioritization techniques can be applied in conjunction with minimization and selection methods. So test case prioritization can be applied as a complementary step in regression testing.
- It is also worth noting that in regression testing, we may be interested in prioritizing test cases in a way that can be effective for a particular version. [140] call this approach

"version-specific prioritization". In this study, we also focus on version-specific test case prioritization.

Therefore, in this chapter, after a comprehensive explanation about the proposed techniques and their computation and estimation processes, we walk through applying them on a successful model-based technique in Graphical User Interface (GUI) testing [16], while later we show that it can be used with any type of MBT technique. In addition, the domain of application is also not restricted to GUI testing; it is believed that the approach is appropriate to most software testing scenarios. Finally, we compare the approach with traditional (Random, Additional statement coverage, Worst and Optimal) prioritization schemes, and a recent contribution in the literature, to demonstrate its effectiveness.

Specifically, the principle contributions of this study are:

- The introduction of a new behavioral model – the extended (multi-) digraph. This model provides a richer set of information than traditional behavioral models, such as a regular digraph. It is argued that this extended model offers superior performance.
- A concrete mechanism to (numerically) populate this extended model. Specifically, it shows how to use reinforcement learning and Hidden Markov models to realize this model.
- A demonstration of how this model can be utilized to automatically prioritize test cases for GUI applications.
- An empirical demonstration that the technique(s) derived from this extended model outperform other common dynamic test case prioritization techniques.

In the next section, we present the motivations behind this research. In Section 4.3, we provide some required background information about Reinforcement Learning and Hidden Markov models. Section 4.4 represents several test case prioritization techniques that have been used in this research. Sections 4.5 and 4.6 include the design of the proposed techniques by describing the RL-based HMMs' parameters estimation and prioritization methods and a short motivating example. Section 4.7 describes the evaluation phase and experimental setup. It also discusses the results and analysis of our empirical studies. Section 4.8 provides a discussion on the achieved results. Section 4.9 discusses some related work to the contribution of this research. While, Section 4.10 considers the threats to internal and external validity of the study; and finally, Section 4.11 presents overall conclusions and some thoughts on potential future work.

4.2 Motivation

The concept of test case prioritization is a well-established task in software verification and validation. Several different test case prioritization techniques with various aims have been considered through multiple research experiences. The most common aim is increasing the fault detection rate of a test suite; this is also the main motivation of this chapter. The prioritization technique can be *dynamic,* building upon dynamic data flow information or *static,* based upon static analysis of the System Under Test (SUT). This section presents the motivations behind proposing a new dynamic prioritization technique.

4.2.1 Static or Dynamic Prioritization

Most prioritization techniques are based upon structural code coverage information gathered through the dynamic execution of the SUT, while others focus on static analysis of the test cases. Static prioritization techniques mainly focus on ordering the test cases based on analyzing the

source code or test cases' static call graph. For example, [145] propose a static prioritization approach to estimate the ability of each test case to achieve code coverage and to use this information for the re-ordering of the test cases. Prioritizing test cases based on static approaches suffers from the following drawbacks [146]:

Since this technique walks through the source code, there is a need to have access to the SUT or the test cases' source code.

Choosing an appropriate method to statically analyze the source code depends on the programming language used to write the code. [147] present a static technique ranking test cases by extracting and analyzing topic models. Their approach is only applicable in Java-based and medium-sized SUTs, which also indicates that generalizability of this approach is not certain.

Prioritizing test cases using their static call graph also affects both the prioritization precision and cost. Since, a static call graph intends to represent every possible execution of the program; the problem of generating an exact static call graph is undecidable. So generating a comprehensive and correct static call graph causes over-approximations or unrealistic transitions and relationships.

These reasons, along with the proven capability of dynamic approaches in effectively ranking test cases, imply that developing new dynamic test case prioritization techniques is a viable alternative.

4.2.2 Overview of the utilized testing models and domain of application

Our dynamic test case prioritization technique uses an MBT-based approach. It assumes that an MBT-based "front end" (automatic test case generator) exists and in essence it "extends" this

"front end" model and system to complete the testing process. In MBT, the test modeler creates an abstract model (state machine) of the SUT and then generates a set of test cases by walking (traversing) through the model [148]. So, MBT is about the Automatic generation of efficient tests using models of the SUT [20]. This model is a depiction of SUT behavior including input sequences; actions, conditions, output logic and data flow through modules and routines [15]. The most significant feature of MBT is automating both the test generation and execution processes along with the capability of generating test cases. For instance, [19] combines MBT with Evolutionary Functional Testing (EFT) to achieve a fully automatic test case design and evaluation framework. While, [149] represent a new approach to apply model-based testing in service-oriented applications' testing process.

The behavioral model is often instantiated as labeled multi-digraphs where nodes are connected to each other via multiple directed edges. The model is built and updated according to an agent's activities [16]. This approach has been successfully used in generating GUI behavioral models using many different techniques to automatically generate test suites that simulate user behavior in interacting with GUIs. In particular, model-based GUI testing techniques generate test cases based upon a state transition diagram. Several techniques, with diverse modeling methods have been proposed [150]–[152]. Most of them depict a graph-based model, where nodes represent events relating to GUI widgets, and edges (or arcs) represent the relationship between events. Hence, a GUI test case can be generated from such a model by selecting any possible path. In this study, our proposed technique is able to prioritize generated test cases using any such MBT approach. Our approach is based upon building an Extended multi-digraph as a SUT behavioral model. In the following, the definition of both regular and Extended digraphs are considered. Extended digraphs are the basis for our research contribution.

Definition 14. **Regular Digraph.** We define a regular digraph as $G = (S, E, V)$ consisting of a set of nodes (S) representing GUI states of the SUT, and edges $(e_k \in E = (s_i, s_j), where\ s_i, s_j \in S)$ indicating the transition between states. The transitions are labeled with the name of the actions triggering transitions, and in some cases, an estimated utility value of the action. Figure 11 shows an overview of a regular digraph. In this figure, a_{ij} denotes the utility of the executed action between states s_i and s_j. (Obviously, transitions between s_j and s_i; and s_i and s_k, where $k \in 1, 2 \ldots N$ and $k \neq i$ are also feasible.) The method of calculating this value is explained in Section 4.5.

Figure 11. An overview of the behavioral model as a directed graph

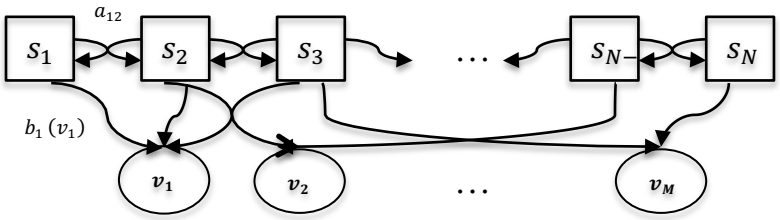


Figure 12. An overview of the behavioral model as an *Extended* directed graph

Definition 15. **Extended Digraph.** An Extended digraph $G' = (S, E, V, A, B)$ is an extended version of a regular digraph G which contains additional information. In the Extended digraph, V indicates a set of actions triggering transitions, where $v_k \in V$ ($k \in 1, 2 \ldots M$ and M is the number of possible actions); A is the probability transition matrix, in which each element $a_{ij} \in A$ represents the probability of moving from state s_i to state s_j; and, B shows the emission matrix in which each element $b_i(v_k) \in B$ indicates the probability of observing action v_k in state s_i (Figure 12). The methods of estimating and calculating these elements are also considered in

Section 4.5. The extra information helps in evaluating the effect of executing actions on GUI states enabling superior prioritization of test cases.

Generating the Extended model only requires simple interfacing glue code to be added to pre-existing MBT approaches. So the Extended model can be generated using information gathered during the test case execution procedure, hence, there is no need to re-implement test cases to generate the model. Once generated, this model can be utilized to prioritize the test cases (to maximize their (test cases) effectiveness during regression testing and other activities). To elaborate the motivation behind this approach we consider the following example.

The main reason for proposing a stochastic approach is the proven ability of such techniques in estimating these probabilistic models, specifically in calculating the forward probability of a sequence of events; in this study, the sequence of events is interpreted as a test case. It should be noticed that the proposed techniques in this research are not limited to GUI based applications and can be applied in a wide range of circumstances. However, to demonstrate the utility of the approach, a domain of application needs to be selected. We have chosen to evaluate our techniques by prioritizing test cases of GUI based applications because:

- GUIs are very common, and the need for GUI testing tools becomes ever greater.
- Number of generated test cases in GUI testing is often large. Hence, it is time-consuming to re-execute them during regression testing [153].
- No GUI test case generation tool prioritizes test cases for regression testing objectives.
- Because event-flow graphs [150], action-based behavioral models [16] or other types of SUT models contain information about user-observations (based on interacting with the

SUT). Hence, it is feasible to determine the observable and latent states required to estimate the extended behavioral model.

Thus, based upon a well-known GUI testing definition [150], the focus of this research would be on prioritizing test cases consisting of a sequence of events (an episode), which is performed during user-GUI interactions. (An episode is a path from the initial to a terminal state, or a sequence of produced executions by an agent.) In other words, we only cover testing interactions that are performed between a user and the GUI; other types of test cases are outside our scope. The definition is further restricted as actual users would introduce variation in experimental results, hence in our experiments, we utilize an automated test case generation tool which acts as a proxy for actual users [16]. This allows for the exploration of a fully automated testing process from user action generation to test case prioritization.

4.3 Theoretical Background

The proposed prioritization technique is established by combining Reinforcement Learning (RL) and Hidden Markov Model (HMM) concepts to efficiently and rapidly prioritize test cases. The main reasons for choosing Reinforcement Learning are its strong statistical background, its proven ability in handling a wide range of data, and its ability to re-estimate the Markov model efficiently. Using RL, we are able to estimate an appropriate HMM and then use it to compute each test case's forward probability- the likelihood of executing a specific test case based upon the SUT's inferred HMM.

Briefly, this method proposes a probabilistic-based prioritization technique using the following steps:

- Estimating initial RL-based HMM parameters according to the generated test cases using MBT techniques.
- Training an RL-based HMM with a maximum likelihood, using the Baum-Welch algorithm [93].
- Prioritizing test cases based on estimating the forward probability of each test case in the execution phase by applying a forward algorithm [73].

The RL-based HMM is estimated using the generated model through MBT techniques. This approach uses a System Under Test (SUT) behavioral model (currently an extended digraph graph; however, changing the model is straightforward) to generate a Markov chain containing hidden states and transition probabilities estimated using RL algorithms. Thus, this technique is able to prioritize test cases based upon the amount of computations (changes), a test case may cause in GUI states (using Q-learning, a type of RL) **and** the probability of each action happening in each specific state (using the HMM).

Moreover, we investigate another prioritization approach (called Accumulated Q-value) by computing the accumulated amounts of Q-values for each specific test case. This can be interpreted as a prioritization technique where the technique is derived from only a test case's contribution in changing an application's state. It is worth noting that the Q-values can be calculated during the generation of the extended behavioral model based on definition 15. The method of this computation will be considered in Section 4.4. Before investigating the proposed techniques, some background on RL and HMM is provided.

4.3.1 Reinforcement Learning

Reinforcement Learning (RL) contains one of the best known classes of machine learning algorithms which "teach" an agent how to interact with an environment [68]. It is located between supervised and unsupervised learning since only a limited feedback, named a *reward signal*, is received by an agent about the agent's predictions [66]. The long-term objective of this agent is performing an action (i.e. mapping situations to actions), which maximizes the overall reward signal [67]. RL's practical applications not only address learning paradigms in operations research and control engineering [73], [154]–[156], but are also one of the most active research areas in artificial intelligence [157]. RL algorithms' progress is typically iterative. They learn during iterations by observing the current environment, inferring the environment's state and executing an action, which guides the agent to the next state. In other words, the agent receives the system's state and the reward score associated with the last transition. Then, it evaluates potential actions according to the expected reward to be realized and selects an action, which is sent back to the system. In response, the system makes a transition to a new state and this cycle will be repeated as part of a Markov Decision Process (MDP) [66], [68]. MDPs can be categorized as stochastic extensions of finite automata or Markovian Processes, which are augmented, by actions and rewards, so they consist of actions and transitions as well as states.

Definition 16. **Markovian System.** The system under consideration is called Markovian if executing an action does not depend on previous actions and visited states (i.e.it only depends on the current state and status). So, an MDP contains:

- A finite set of environmental states $S = \{s_1, s_2, \dots, s_N\}$ where N is the number of states;
- A finite set of actions $A = \{a_1, a_2, \dots, a_k\}$, where k is the size of the action space;

- The transition function $T: S \times A \times S \rightarrow [0,1]$ which computes the probability of reaching the state s' by performing action a in state s and is denoted as $(T(s, a, s'))$; and

To compare different states and actions, during agent and environment interaction, they must be ordered according to their occurrence. So s_t denotes the state at time t [68].

Thus, according to the definition of a Markovian process, we have:

$$P(s_{t+1}|s_t, s_{t-1}, s_{t-2}, \dots) = P(s_{t+1}|s_t) = T(s_t, a_t, s_{t+1}) \quad (28)$$

Definition 17. **Reward Function.** The *Reward Function R* specifies the reward, or penalty, the agent receives by performing an action. So, $R: S \times A \times S \rightarrow \mathbb{R}$ presents the reward function that computes the immediate utility of an action to define the model of the MDP. So an MDP can be denoted by the tuple $\langle S, A, T, R \rangle$ depicting it as a state transition graph [68].

Definition 18. **Value Function.** The Value Function $V^{\pi}(s)$, specifies *"how good"* it is for the agent to be in a given state. The *How good* notation here is expressed in terms of future rewards that can be expected. We can define the value of state *s* under policy π, formally $V^{\pi}(s)$, as [67]:

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\} \quad (29)$$

Where,

The stochastic policy $\pi: S \times A \rightarrow [0,1]$ is a mapping from each state s and action a, to the probability $\pi(s, a)$ by performing an action a when in state s.

E_{π} is the expected value earned by following policy π and *discount factor* γ, with $0 \leq \gamma < 1$. This models the fact that future rewards are worth less than an immediate reward, i.e. if $\gamma = 0$

the agent only would be concerned about the immediate reward while, a value close to 1 gives a large weighting to future actions.

Definition 19. **State-action Value Function.** Similarly, the value of performing an action a in state s (the state-action value function or Q-value function Q: $S \times A \rightarrow \mathbb{R}$) can be defined as:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right\} \quad (30)$$

Almost all RL based paradigms are based on estimating the value functions appropriately, which has led to the exploration and production of several different estimating methods and techniques. One of the most popular, which also is applied in this research, is ***Q-Learning*** [69].

Q-Leaning is a method to estimate Q-value functions, when there is no available model of the MDPs (*model-free* fashion). In this situation, because of the lack of priori transition and reward models, there is a need for sampling and exploration to learn the required model or step *directly* into estimating values for actions (Q-values). Therefore, Q-learning estimates the agent's Q-value function based upon an action's Q-value estimation; this process is incrementally evaluated as follows [68]:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha\left(r_t + \gamma \max_a Q_k(s_t, a) - Q_k(s_t, a_t)\right) \quad (31)$$

Where, α $(0 < \alpha \leq 1)$ is the *learning rate*, which determines the extent that new information can override old information [70].

Because of its proof of convergence to an optimal policy [71], and its proven ability in value-function estimation in free-model problems [69], we apply Q-learning for estimating the Q-values in this research.

4.3.2 Hidden Markov Model

Hidden Markov Models (HMMs) are popular statistical tools for modeling data in various areas such as bioinformatics, speech recognition, partial discharges and many other temporal pattern situations [158]–[160]. In the broadest sense of the word, an HMM is a Markovian process that is split into two components: the observable states and the hidden (latent) states. Compared to a HMM, regular Markov models only consist of observable states, which are directly visible to the observer. Therefore, only the state transition probabilities require estimation. However, in a hidden Markov model, only the outputs (observations), depending on the latent states (GUI states), are visible. Therefore, as the GUI states are invisible from an observer view and only the outputs in this model are completely observable, regular Markov models and Partially Observable Markov Decision Processes (POMDPs) are not applicable in this study [161]. An HMM is usually characterized by the following elements [72]:

- N, the number of hidden states in the model, $S = \{s_1, s_2, \dots, s_N\}$.
- M, the number of distinct observation symbols per hidden state, $V = \{v_1, v_2, \dots, v_M\}$.
- The state transition probability distribution $[A]_{ij} = \{a_{ij}\}$, where: $a_{ij} = P\left(Q_{t+1} = s_j \middle| Q_t = s_j\right), 1 \leq \text{i,j} \leq \text{N}$.
- The observation symbol probability distribution in hidden state j, $[B]_{jk} = \{b_{j\,(vk)}\}$, where $b_j\,(v_k) = P\left(O_t = v_k \middle| Q_t = s_j\right), 1 \leq \text{j} \leq \text{N}, 1 \leq \text{k} \leq \text{M}$. And
- The initial state distribution $\Pi = \{\pi_i\}$, where $\pi_i = P(Q_1 = s_i), 1 \leq \text{i} \leq \text{N}$.

Using the values of N, M, A, B and Π, the HMM can be used as a generator to create an observation sequence (where T is the number of observations in the sequence): $O = \{O_1, O_2, O_3, \dots, O_T\}$. We use the notation Λ = (A, B, Π) to simply indicate the complete

parameter set of the HMM. There are three key issues with HMMs, which are commonly considered when applying them to a problem domain:

Problem 1:

- Given the observation sequence $O = \{O_1, O_2, O_3, \dots, O_T\}$ and an HMM, how to efficiently compute the probability of the observation sequence?

Problem 2:

- Given the observation sequence $O = \{O_1, O_2, O_3, \dots, O_T\}$ and an HMM, how do we find the state sequence that best explains the observations?

Problem 3:

- Given the observation sequence$O = \{O_1, O_2, O_3, \dots, O_T\}$, how to choose the model parameters in an HMM?

In this research, we present a novel application of HMMs in software testing, i.e. we show how they can be utilized in test case prioritization processes by addressing the first and third issues. The third one can be solved in software testing by estimating the SUT's Hidden Markov Model (based on the software's behavioral model) and extracting the known sequence of test cases (observations). Having the model and observation sequence, we will be able to accurately compute the forward probability of each test case, using a forward dynamic programming procedure. This probability is what we need to evaluate and prioritize test cases; this evaluation also resolves the first issue of applying HMMs to this new problem domain. Addressing the second problem is beyond the scope of this initial research; however, a suggested approach is briefly presented in Section 4.11.

4.3.2.1 Forward Probability

A forward algorithm computes the probability of encountering a sequence of observations, supposing that the sequence has been generated by a given HMM. In another words, a forward algorithm is used to calculate a belief state or the probability of a state at a certain time (forward probability). A forward probability can be calculated at each time step by considering the most likely state, given the previous history. Therefore, the forward probability $\alpha_k(t)$ is the sum of the probabilities of all constrained paths of length i (where i is the length of a sequence of observations) that end in state k.

Since, test suites are considered as a set of observed sequences, this algorithm estimates the forward probability of each test case, given an HMM learned from an Extended digraph.

According to the RL approach (Section 4.5) that has been used in this study, the forward probability of a test case is related to the amount of computation (changes) in GUI states; test cases with higher amounts of changes are more likely to encounter unexpected behavior than a test case causing fewer changes. In other words, test cases that activate more actions during their execution process are more likely to detect faults. Thus, a forward algorithm is an appropriate choice to compute this likelihood as a set of test cases can be modelled as a sequence of actions. This procedure has the following steps [73]:

Defining the forward probability,$\alpha_k(t)$, as the joint probability of observing the first *t* vectors v_t, $\mathrm{T} = 1, \dots, t$ while in state k at time *t*. Another way to state this would be that $\alpha_k(t)$ is the probability of observing $v_1, v_2, \dots, v_t$, in addition, at time *t* the state is *k*.

$$\alpha_k(t) = P(v_1, v_2, \dots, v_t, s_t = k|\Lambda) \qquad (32)$$

This probability can be evaluated by the following recursive formula.

$$\alpha_k(1) = \pi_k b_k(v_1)\,, \qquad 1 \le k \le N \tag{33}$$

$$\alpha_k(t) = \pi_k b_k(v_t) \sum_{l=1}^{N} \alpha_l(t-1) a_{l,k}\ , \quad 1 \le k \le N, 1 \le t \le T$$

$$\alpha_k(t) = \sum_{l=1}^{N} \alpha_l(t-1) b_k(v_t) a_{l,k}$$

However, when the sequences of observations (the length of the episodes) become larger, the probabilistic values in the forward algorithm get increasingly small, and after multiple iterations the values tend to zero. For that reason, $\alpha_k(t)$ are scaled during the iterations of the algorithm to avoid underflow problems. The scaling coefficients are used to keep the probability values in the dynamic range of the machine. So, the coefficient $c_t$is defined as follow [162]:

$$c_t = \frac{1}{\sum_{k=1}^{N} \alpha_k(t)} \tag{34}$$

Using c_t, the scaled value of $\alpha_k(t)$ would be:

$$\alpha_k^*(t) = c_t \times \alpha_k(t) = \frac{\alpha_k(t)}{\sum_{k=1}^{N} \alpha_k(t)} \tag{35}$$

To conclude, after estimating an appropriate HMM, we would be able to prioritize test cases by computing their corresponding forward algorithm.

4.4 Test Case Prioritization

The Test Case Prioritization (TCP) approach sorts test cases within a test suite in order to maximize some pre-defined criteria such as additional code coverage or fault detection rate [163]. In other words, test cases with the highest score, with respect to the prioritization criteria, have the highest priority to be executed. [139] provide a formal definition for the test case prioritization problem.

Definition 20. **Given:** T (a test suite), PT (a set of permutations of T), and f (a function that maps PT onto a real number). **Problem:** Find $T' \in PT$ such that:

$$(\forall T^{"})(T^{"} \in PT)(T^{"} \neq T')\,[f(T') \geq f(T^{"})] \qquad (36)$$

In this definition, PT is the set of all possible orderings of test suite T. In addition, function f yields an award value for each specific prioritization showing its value (Prioritization with highest award values are preferable). We customize this definition based upon the research's objectives in the following section.

Several deferent test case prioritization techniques have been proposed through multiple research experiences. [140] considered 18 different approaches and compared them in terms of effectiveness with respect to cost and performance. Moreover, they classify them into three separate groups. The first group is named *comparator* containing: Random and Optimal ordering. The second one is the *statement level* group consisting of four fine granularity techniques and the third group is the *function level* group, which contains 12 coarse granularity techniques. In this research, we will consider Random, Optimal and Worst ordering techniques to define upper and lower boundaries in achieving the test case prioritization goal, here, improving

the fault detection rate. In addition, we prioritize test cases based upon the Additional statement coverage technique from the second category and two novel RL-based prioritization approaches, which will be illustrated in following sections. Since the number of statements is greater than the number of functions, and statement level group of techniques is a "superset" of the function-level techniques, hence, we don't consider the function-level group and just apply the statement coverage techniques.

4.4.1 Random, Optimal and Worst Prioritization Techniques

One of three prioritization techniques which, we consider in this research is Random test case prioritization. In this approach, we randomly pick a test case among the list of test cases and remove it from the list, then we repeat the picking process until no test case exists [141]. We consider it as an experimental control to evaluate the effectiveness of other heuristics in comparison with the Random Ordering. We also select and consider the Optimal and Worst prioritization approaches. Both of these techniques are not practical, since information about which test cases reveal faults and which do not expose them is not available [140]. But, in this research the SUTs' faults are known and we can determine their corresponding test sets. Thus, we consider Optimal and Worst prioritization techniques to theoretically obtain the upper and lower boundaries in the fault detection rate. In the Optimal prioritization, we prioritize test cases in an order, which maximizes this rate, while in the worst prioritization approach, we are trying to prioritize test cases that can detect the fewest new faults, thus minimizing the fault detection rate [164].

4.4.2 Additional Statement Coverage Prioritization

Determining which statements in the program were explored (covered) by a specific test case can be considered through a program instrumentation phase. Then, test cases can be prioritized according to the total number of statements they cover *(total coverage prioritization)* [139]. It is possible to achieve additional coverage in subsequent testing processes by considering statements that have not been covered in earlier processes. So, here we use Additional statement coverage prioritization, which is based upon feedback about the coverage gained so far. We iteratively select a test case that covers the maximum number of statements then, we adjust the coverage information of the remaining test cases to calculate the coverage of statements that have not been covered [140]. This process is repeated until all of the statements are covered by at least one test case. It is worth noting that if the multiple test cases cover the same number of not previously covered statements, we need to choose one of them in a random way. [165] demonstrate that the Additional coverage approach as the best coverage-based prioritization technique in terms of fault detection capability. They evaluated the effectiveness of different coverage-based prioritization techniques on an industrial case study and concluded that prioritization methods based on Additional coverage using finer grained coverage criteria outperformed all other coverage-based techniques.

Cobertura[13], an open source Java tool, is able to determine which parts of a Java program are covered by a test case; Cobertura can be embedded into a test case generator (AutoBlackTest in this study). AutoBlackTest collects coverage data incrementally during the execution of test cases on the current version of the application. Gained information is utilized in order to prioritize test cases in an Additional statement coverage manner.

[13] http://cobertura.github.io/cobertura

4.5 Test Case Prioritization Using RL-Based HMM

Combining Hidden Markov Model and Reinforcement Learning approaches as a part of a stochastic estimation process has been considered in several artificial intelligent problems. [74] suggest using RL to re-estimate the HMM parameters when the recognition model does not work during motion prediction scenarios. [166] also integrated HMMs and RL to recognize communication channels and translate human actions into instruction symbols. This process was done by estimating HMMs and then applying RL to decide the next fixation point. On the contrary, [167] propose an RL-based approach to train HMMs in order to cope with the local optima problem that happens when using a local search method.

In this research, we propose a novel approach to prioritize test cases using an RL-based HMM. In other words, we want to learn a model which is not only able to generate test cases by traversing model paths but also can easily rank test cases based upon their forward probabilities. Achieving this goal is possible by integrating the HMM and RL approaches producing a graph-based model as an MDP based learning technique. The RL-based HMM approach provides a three-step framework: the first step contains a Reinforcement Learning algorithm, here Q-learning, to learn to interact with the software under test, to stimulate its functionalities, and to automatically generate test cases. The second step includes an HMM estimation process to prioritize generated test cases. The final step computes the test cases' forward probabilities, using the *Forward Algorithm.*

Before going through the prioritization steps, we state two assumptions, which are used to realize our work in the GUI-testing domain:

- In this domain, actions represent GUI actions such as click, select, deselect and etc. According to the [16] research, Q-values indicate the values of the actions based upon the computations activated by them. They believe that the reward function should encourage the system to perform actions triggering "large volumes of computations" (critical events [164]) rather than those that only activate "small volumes of computations". For instance, clicking on a button or typing a word in the text area causes smaller changes to the GUI state than filing out a form and submitting it into a database. So the actions that contribute more GUI changes are more likely to gain higher rewards and consequently higher Q-values than others. Also, based upon this research, a GUI state consists of a set of widgets $\{w_1, .., w_n\}$ and each widget (w_i) can be interpreted as a pair $(type_i, P_i)$, where $type_i$ indicates the type of a widget such as textarea, label … and P_i is a collection of widget properties and their values. We accept and build upon [16] work in this respect. They also define a function that generates traits from widgets, where a trait is subset of the widget properties. So, given w_1, w_2; $w_1 =_t w_2$ iff $trait\ (w_1) = trait(w_2)$. For example, the trait of a button includes its type, position and label shown on it. A similar approach is also offered by IBM functional Tester[14] to compare widgets.
- A GUI visualizes and processes data through menus, buttons, labels etc. Extracting such information embedded into widgets can be helpful to automatically produce interesting executions in the application. [168] suggest an algorithm to extract the descriptions of widgets. [16] use this technique in designing their GUI testing framework. Again, we accept and build upon this work [16], [168].

[14] http://www-03.ibm.com/software/products/us/en/functional

4.5.1 Step 1: Q-Learning Estimation Method

As mentioned earlier Q-learning is one of the most popular Q-value function estimation techniques in the situation where there is no access to a pre-defined MDP model. Table 18 shows the Q-value estimation process has been customized for GUI testing [16]. It computes the reward function based upon the proportion of properties that change value; and then the fraction of widgets that are different when observed in different states. According to the computed reward function and initial inputs, we are able to estimate the Q-values for each specific episode.

4.5.2 Step 2: HMMs' Parameters Estimation

As mentioned before, a Hidden Markov Model has five significant elements; each element requires an initial estimation. For the HMM Λ = (A, B, Π) with a (discrete) observation distribution, we consider the following parameters:

- N: The number of hidden states, which is equivalent to the total number of states (possible widgets such as save or open) in the SUT.
- M: The number of distinct observation symbols which is equivalent to the number of possible events or actions (such as click, double click or type something in a text box).
- The state transition probability (N*N) matrix is the probability of traversing from one state to an adjacent state. In RL, and in particular Q-learning, we are able to assign an RL-Score or Q-value to each edge of the generated model. Because this value is assigned after the execution of the corresponding action to the edge, it represents the likelihood of the corresponding transition occurring [16]. It can be computed (Table 18) for each s during action $a's$ execution: $Q(s,a) = reward(s,a) + \gamma\, max_{a'} Q(\delta(s,a), a')$. If more than one possible action between two adjacent states exists, the action with the largest Q-

value would be selected as the value of the transition probability between the two connected (hidden) states.

- The observation probability matrix is an N*M matrix (emission matrix) calculated as below, where f_{jk} is the frequency of observing a specific action (v_k) in a specific state j.

$$b_j(v_k) = \begin{cases} \frac{f_{jk}}{\sum_{k \epsilon v} f_{jk}}, & f_{jk} > 0 \\ 0, & f_{jk} = 0 \end{cases} \quad (37)$$

- The initial state probability distribution, which describes the starting state of the model. To generate the model in this study, every state is considered as a potential starting state.

Table 18. Customized Q-Learning for GUI-based Applications

Input n, number of executions that the agent must produce for each episode (length of episode)
Initialize $Q(s,a) \leftarrow^{*} 0, \forall s \in S, \forall a \in A$ [16]
$\gamma \leftarrow 0.9$
$i \leftarrow 0$
$\varepsilon \leftarrow 0.8$

For each episode **do**

s is a random state initialized as the starting state

Repeat

- Choose a possible action a ∈ A (s)
- Perform action a based on $\varepsilon - \text{greedy}^{**}$ policy [86]
- i ← i + 1
- Observe the new state s′
- Compute the reward(s, a):
- Compute diff_w: the degree of change to a GUI widget while moving from s to s′:

  $\mathbf{diff_w(w_1, w_2)} \leftarrow \frac{|\mathbf{P_1 \backslash P_2}| + |\mathbf{P_2 \backslash P_1}|}{|\mathbf{P_1}| + |\mathbf{P_2}|}$, where $w_1 =_t w_2$, are the same widgets monitored in s and s′
- Compute $\mathbf{diff_S(s, s')} \leftarrow \frac{|\mathbf{s} \backslash_t \mathbf{s'}| + \sum_{\mathbf{w_1} \in s, \mathbf{w_2} \in s', \mathbf{w_1} =_t \mathbf{w_2}} \mathbf{diff_w(w_1, w_2)}}{|\mathbf{s'}|}$,where |s| denotes the fraction of widgets in state s.
- reward(s, a) ← $\text{diff}_s(s, \delta(s, a))$, δ denotes the state reached by executing action a.
- Compute $\boldsymbol{Q(s, a)} \leftarrow \mathbf{reward(s, a)} + \boldsymbol{\gamma}\, \mathbf{max_{a'}} \boldsymbol{Q(\delta(s, a), a')}$
- s ← s′

Until n is equal to (i-1).

* ← shows the assignment

** $\boldsymbol{\varepsilon - \text{greedy}}$ policy selects a random action with probability $\boldsymbol{\varepsilon}$ and the one with highest Q-value with probability $\boldsymbol{1 - \varepsilon}$

Now, we have an initial estimation for each required element of the HMM with respectively N and M latent and observable states. This model illustrates two types of relationships:

- An edge label between latent states, i and j, represents the (a_{ij}) transition probability between these two states.
- An edge label between latent state j and visible state k ($b_j\,(v_k)$) is calculated using the above formula.

In the next step, this initial model is used as an input to an Expectation–maximization (EM) algorithm; specifically we utilize the Baum-Welch algorithm [93]. This algorithm estimates the best model with the highest likelihood of the estimated parameters solving the first problem (see Section 4.3.2). If we consider each test case as a sequence of observations, we are able to predict the forward probability of test cases by utilizing the estimated HMM and implementing the forward algorithm.

4.5.3 Step 3: Computing Forward Probabilities and Considering their Application in Test Case prioritization

Before applying the forward probability in prioritizing test cases, the reasons of choosing it as a prioritization metric should be reviewed and considered. According to the forward probability calculation procedure (See formulas 32, 33 and 34 in Section 4.3.2.1) both transition and emission probabilities are essential parameters in computing the forward probability $\alpha_k(t)$. In addition, as mentioned earlier (See steps 1, 2 and Table 18) the transition probabilities are directly calculated using the amount of the Q-values. (Q-values indicate the likelihood of the corresponding transition occurring by computing the amount of changes, triggered during the transition.) Therefore, it can be concluded that the RL-based HMM prioritizes test cases based on

the amount of changes that can be triggered in the GUI state (Q-values) by executing each specific sequence of observations (actions). This technique not only considers the state changes, but also evaluates the effect of a specific action's performance on each state (HMM emission matrix). So both the states and the edges of the application's behavioral model are involved in the prioritization technique. According to [164], test cases containing more critical events (large volumes of computations), are more likely to uncover faults during testing. Therefore, test cases with higher forward probabilities are likely to contain such events more than those with lower forward probability.

Now, in order to complete the procedure of applying forward probabilities in test case prioritization, we modify [139] definition to involve the forward algorithm in calculating an award value (Definition 20). In the new definition, F_p would be a new function, which yields forward probabilities for each "test case". It is worthwhile to mention that in the original definition, we compute the award value for each different ordering, but in the new definition, we calculate it for every test case in an offline mode with no need to re-execute the test cases. Therefore, we are able to sort test cases by assigning higher ranks to the test cases with higher amounts of forward probabilities.

Definition 21. **Given:** T (a test case), TC (a set of generated test cases), and F_p(a function that maps from TC to a real number). **Problem:** Find $T' \in TC$ such that

$$\left(\forall T^{"}\right)\left(T^{"} \in TC\right)\left(T^{"} \neq T'\right)\left[F_p\left(T'\right) \geq F_p\left(T^{"}\right)\right] \qquad (38)$$

According to the new definition, we prioritize test cases based upon the estimated HMM. The following algorithm (Table 19) describes the proposed method. It accepts a set of test cases, $TC = \{T_1, T_2, \ldots, T_c\}$, where $T = \{v_1, v_2, \ldots, v_t\}$ is a sequence of events or actions. Then, it

estimates a corresponding HMM and prioritizes the test cases according to their F_p value. It is worth noting that if the multiple test cases have the same amount of F_ps (rarely happens), we need to choose one of them in a random way.

Table 19. RL-based HMM Test case Prioritization Algorithm

Input: $TC = \{T_1, T_2, \dots, T_c\}$; **Output:** TC': Prioritized TC

1: While $\exists\ T_i \in TC$ **do**:

For each T_i **do**

- $TC' \longleftarrow T_i$ and Extract all corresponding hidden states (S_i): $N \longleftarrow N + 1$ (per visit to each unique state)
- Extract all corresponding actions (v_i): $M \longleftarrow M + 1$ (per visit to each unique event)
- If a_{ij} does not exist then, compute Q-value among S_i and S_j and assign it as a_{ij} into the transition matrix $[A]_{N\times N}$
- If b_{ij} does not exist then, compute the corresponding observation symbol probabilities and assign it as b_{ij} into matrix $[B]_{N\times M}$

2: Train HMM (Λ) by implementing Baum-Welch (A, B, Π)

3: While $\exists\ T_i \in TC'$ **do**:

- $F_{Pi} = \text{forward}(\Lambda, T_i) = \alpha_k(t) = P(v_1, v_2, \dots, v_t, s_t = k|\Lambda)$

4. $TC' \longleftarrow \text{DescendSort}(TC')$ based upon F_{Pi} values

4.6 Accumulated Test Cases' Q-values in Descending Order

In order to investigate the effect of test case dissimilarities in the fault detection rate, we also decided to prioritize test cases based upon the amount of computations activated by the corresponding action in each test case. As mentioned earlier, GUI state changes can easily be calculated using the Q-learning technique. Given each test case's accumulated Q-value (summation of each test case's Q value), we only need to rank every test case in a descending order, then label the one with the highest Accumulated Q-value (SQ) as the test case with the highest execution priority. Table 20 illustrates the overview of this technique.

Table 20. Accumulated Test Cases' Q-values Ordering

Input: $TC = \{T_1, T_2, \dots, T_c\}$
Output: TC': Prioritized TC

1: While $\exists\ T_i \in TC$ **do**:
2: For each T_i **do**
$TC' \leftarrow T_i$
a. Extract all corresponding Q-values ($Qvalue_j$)
b. $SQ_i = \sum(Qvalue_j)$
3: $TC' \leftarrow DescendSort(TC')$ based upon $\mathbf{SQ_i}$values

This technique can be considered as an attempt to optimally use the information in a regular digraph; whereas the RL-HMM technique explicitly utilizes additional information only available in the extended diagraph. In the following section, we discuss how these techniques improve the test case prioritization problem with regard to failure detection objective.

4.6.1 Motivating Example

In this study we cover two well-known, MBT based GUI testing tools: GUITAR[15] and AutoBlackTest[16] (ABT). GUITAR uses a test automation framework based on state machines, named the event flow model [150]; while AutoBlackTest generates a behavioral model based upon agent activity [151].

To motivate this work, we consider a small example to investigate the difference between state of the art behavioral models (such as a regular digraph) and the Extended multi-digraph. Figure 13 shows a small excerpt of a behavioral model derived by a GUI-based MBT test case generation tool (AutoBlackTest) for the UPM application (an application for building a personal database of accounts and one of the case studies presented in this chapter) [16]. In this figure, the

[15] http://sourceforge.net/apps/mediawiki/guitar/index.php?title=GUITARHome_Page
[16] http://www.lta.disco.unimib.it/tools/AutoBlackTest

nodes of the graph represent the GUI states of the SUT and the edges represent the transition between the states. They are labeled with the name of the actions triggering the transitions, and in some cases, the estimated utility value of the action. This is an example of a regular digraph. Applying our new technique, we would be able to generate an extended form of directed graphs utilizing stochastic information. The additional information will be used in the test case prioritization procedure to calculate the forward probability of each test case. Figure 14 depicts an excerpt of an Extended behavioral model applying our new approach on the UPM application.

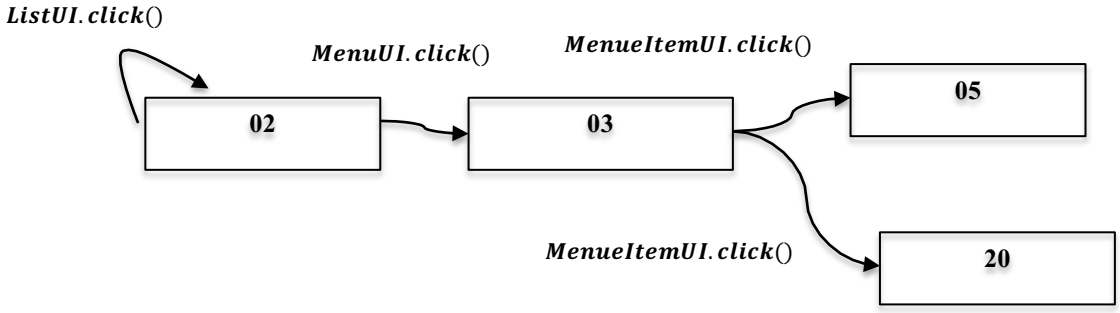


Figure 13. Expert of a behavioral model as a directed graph-motivating example

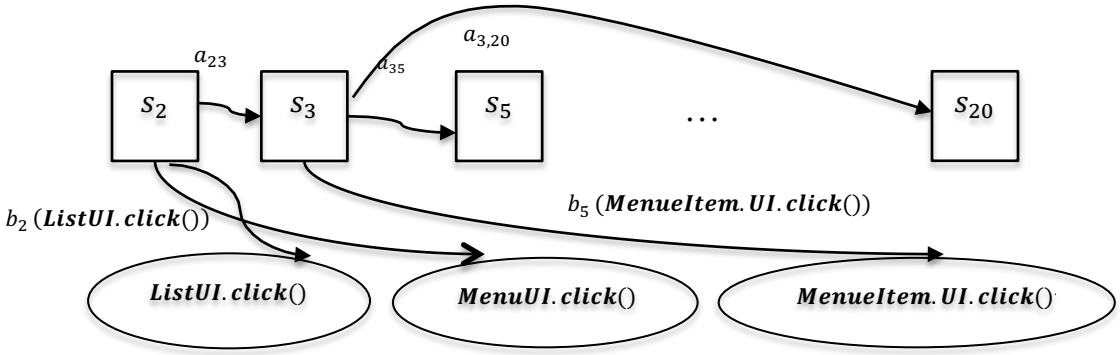


Figure 14. Expert of a behavioral model as an *extended* directed graph- motivating example

Based on definition 15, this figure is an extended behavioral model consisting of a set of GUI states (rectangular-shapes), actions (elliptical-shapes) and additional information (labels). In this model a_{ij} indicates the transition probability from state i to j and $b_i\ (v_k)$ indicates the probability

of observing action v_k in state i. This is the only information needed to be stored for test case prioritization; specifically if it is stored in the form of two probability matrices covering a_{ij} and $b_i\ (v_k)$ for all possible transitions.

Thus, considering following sequences as possible test cases extracted from above directed graph (Figure 13), we would be able to calculate required data to estimate an RL-based HMM and prioritize test case based upon the forward probabilities. This obviously shows that the extended directed graph contains more useful information than the regular one, which helps in estimating the probabilistic model and calculating the forward probability of a sequence of events (i.e. test cases). As mentioned earlier, the forward probability of the sequence is used to effectively prioritize test cases by assigning the higher fault detection probability to test cases which are able to trigger more actions and computations in each GUI states.

$T_1 = \{02\ (ListUI.click()), 02\ \}$

$T_2 = \{02\ (MenuUI.click\ ()), 03\ \}$

$T_3 = \{02\ (MenuUI.click()), 03\ (MenueItemUI.click()), 05\}$

$T_4 = \{02\ (MenuUI.click()), 03\ (MenueItemUI.click()), 20\}$

For example, for test case 2 (T_2), we should calculate the Q-value and the immediate utility value of an action using the following reward function, based on the differences between widgets in each state (Table 18). So:

$$reward(s,a) = diff_s\big(s, \delta(s,a)\big) = diff_s\big(02, \delta(02, MenuUI.click())\big)$$

$$= \frac{|s \backslash_t s'| + \sum_{w_1 \in s, w_2 \in s', w_1 =_t w_2} diff_w(w_1, w_2)}{|s'|} = 0.55$$

In addition, because this test case terminates in state 3 and no more actions are executable from this state, $\gamma\ max_{a'}Q(\delta(s,a),a') = 0$ and:

$$a_{23} = Q(s,a) = reward(s,a) + \gamma\ max_{a'}Q(\delta(s,a),a') = 0.55$$

$$b_2\ ((MenuUI.click\ ())) = \frac{f_{jk}}{\sum_{k\epsilon v} f_{jk}} = \frac{1}{2} = 0.5$$

Calculating the same parameters for all test cases gives us following transition probability and emission matrices.

$$[A] = \begin{bmatrix} 0 & 0.55 & 0 & 0 \\ 0 & 0 & 0.71 & 0.7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, [B] = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Having all the required parameters to estimate an accurate HMM prepares the model for RL-HMM based prioritization.

Estimating the RL-based HMM using the Baum-Welch algorithm, we would be able to compute the forward probability of each test case using the forward algorithm. Therefore, applying Formula 33. We would have:

$$(Forward\ (RL-HMM, T1)) = 0.137$$

$$(Forward\ (RL-HMM, T2)) = 0.275$$

$$(Forward\ (RL-HMM, T3)) = 0.5$$

$$(Forward\ (RL-HMM, T4)) = 0.5$$

And, we can sort test cases in following order:

1: T_4, T_3, T_2, T_1 OR

2: T_3, T_4, T_2, T_1

As we expected, the test cases with larger quantities of forward probabilities should be prioritized due to the higher amounts of computations and actions, which have been triggered in the GUI states during their executions. We expect test cases with such characteristics are able to reveal more faults in GUI-based applications because of their capabilities in triggering more actions in GUI states. T_4 matches with episode 2 in the complete example (Section 4.7.3.1), which is able to reveal a fault in the Application Under Test (AUT). So, choosing either the first or second ordered test suite we would be able to detect a fault by running only one or two test cases.

Also, we can go further in this example and replace the initial model, which is generated by [16] technique with the event-flow graph, which is generated using GUITAR [150], another GUI test case generator tool (Figure 15). This procedure shows that the proposed test case prioritization technique is applicable to any test cases that have been generated by any GUI-based MBT test case generation tool.

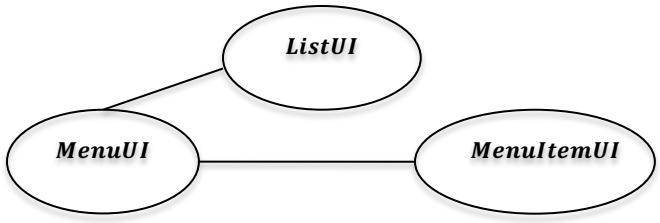


Figure 15. Event flow graph, extracted from GUITAR- Motivating example

Unlike the graph, which was extracted from AutoBlackTest, GUITAR's event-flow graph is not a state-based model and only contains the relations between actions. Based upon the changes between GUI states during test case generation process [168], we are able to create a GUITAR

based extended digraph (Figure 16). In this situation, we assume that GUI state has been changed at least one time in each single event execution.

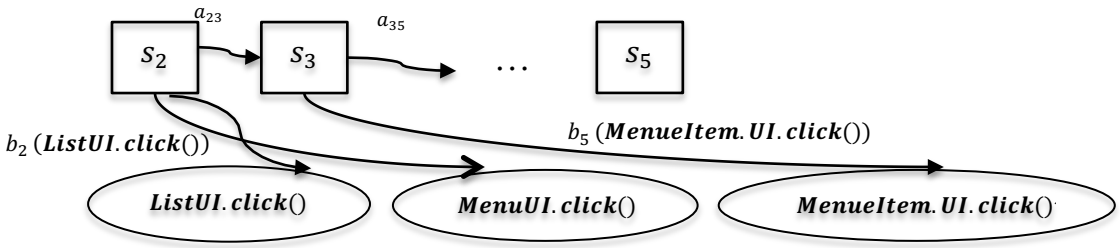


Figure 16. Extended directed graph- generated using event flow graph

Based upon the generated event-flow graph, GUITAR is able to generate following test cases:

$T_1 = \{(ListUI.click())\}$

$T_2 = \{(ListUI.click()), (MenuUI.click\ ())\}$

$T_3 = \{\{(ListUI.click()), (MenuUI.click\ ())\}, (MenueItemUI.click()),\}$

Similar to the previous steps, both emission and transition probability matrices should be calculated before estimating the Hidden Markov Model. Similarly the amount of Q-values and immediate utility values should be computed based upon the differences between the widgets in corresponding GUI states. Matrix A indicates the transition probabilities and B shows the emission probabilities in the initial HMM.

$$[A] = \begin{bmatrix} 0 & 0.55 & 0 \\ 0 & 0 & 0.71 \\ 0 & 0 & 0 \end{bmatrix}, [B] = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Calculating the forward probabilities we would have the following results

$(Forward\ (RL - HMM, T1)) = 0.25$

$(Forward\ (RL-HMM, T2)) = 0.068$

$(Forward\ (RL-HMM, T3)) = 0.5$

And, subsequently we can sort the test cases in following orders: T_3, T_1, T_2

The test suite, which is generated by GUITAR missed the fault-revealing test case T_4 during the test case generation phase; thus in this case, the final test suite is not able to detect any fault and prioritization doesn't affect the speed of defect detection. This demonstrates the higher ability of AutoBlackTest in generating test cases with better coverage. In addition, it highlights the capability of our proposed approach in prioritizing test cases that have been generated using different GUI test case generators. Our approach is clearly independent of their architectures and strategies.

4.7 Empirical Evaluation

To investigate the effectiveness of the proposed techniques (RL-based HMM and Accumulated Q-value) relative to existing prioritization techniques, we have constructed an experimental framework and addressed the following research questions:

- ***(RQ1):*** How does the RL-based HMM technique compare in terms of effectiveness with other prioritization techniques?
- ***(RQ2):*** How does the Accumulated Q-value technique compare in terms of effectiveness with other prioritization techniques?
- ***(RQ3):*** Does RL-Based HMM prioritize test cases more effective than the Accumulated Q-value approach?

- ***(RQ4):*** Does RL-Based HMM prioritize test cases more effective than the weight-based methods presented by [164]?

4.7.1 Comparison Criteria

4.7.1.1 Average Percentage Faults Detected (APFD)

Based upon the test case prioritization goal, there are many possible comparison criteria that can be utilized in order to evaluate the effectiveness of the applied approach [139]. In this research, we have focused on the goal of increasing the "likelihood of revealing faults earlier in the execution of the test run", or in the other words, "the fault detection rate" during regression testing.

To measure how rapidly a prioritized test suite is able to detect faults, we have used the APDF measure. APFD is defined as [139]:

$$APFD\ (T') = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \qquad (39)$$

Where T' is a prioritized test suite, and T is a test suite containing n test cases, and F is a set of m faults revealed by T. In addition, TF_i is the first test case in the ordering T' of T that detects fault i. The range of APFD is between 0 and 1, and a higher APFD means superior fault detection.

4.7.2 Statistical Testing[17]

[169] provide guidance on the statistical analysis of automated processes in software engineering contexts. We essentially follow their advice and present effect size estimations [170] as the principle descriptor of performance differences between algorithms. The estimation of effect size

[17] We have implemented the statistical results and plotted the boxplots using OriginLab

is commonly preceded by traditional statistical testing, and hence, we provide both for the sake of completeness.

The central limit theorem states [171] that the distribution of the mean of a sufficiently large number of independent, identically distributed variables will be approximately normal, regardless of the underlying distribution. In addition, [172] and many other texts provide a rule of thumb saying that ~30 data points are sufficient to get reliable results. Hence, a solid theoretical basis exists for applying a paired t-test on the mean of these types of variables. Therefore, we "normalized" our statistical analysis by executing the prioritization techniques 1000 times for each of the generated test suites and construct the mean value of the APFD (robust estimator). Then, for each prioritization approach, we select a prioritized test suite (R) with the minimal discrepancy to the mean APFD. Thus:

In this study, the random variable is the mean of the percentage of faults detected over all orderings by a given prioritization technique.

X_i, Y_i indicate two paired measurements from the n measured values; the measured values are the mean of the percentage of detected faults by a given prioritization technique (i) on a test suite of size (n).

Therefore, after computing this (ordered) element, we would be able to apply statistical testing by calculating the differences between *the mean of the percentages of detected faults in* all braces of paired sets X_i *and* Y_i (paired test). To apply the paired t-test, the t-statistic needs be calculated as below:

$$\widehat{X}_\iota = (X_i - \bar{X}) \qquad (40)$$

$$\widehat{Y}_\iota = (Y_i - \bar{Y}),$$

$$t = (\bar{X} - \bar{Y}) \sqrt{\frac{n(n-1)}{\sum_{i=1}^{n} (\widehat{X}_\iota - \widehat{Y}_\iota)^2}}$$

This brace-wise approach is standard practice in such situations in many domains [173] when the variable meets the paired t-test assumptions[18].

In this case, we established a null hypothesis and an alternative hypothesis to be evaluated. The null hypothesis (H_0) states the two prioritization techniques provide the same effectiveness of fault detection, if the mean of the percentage of the detected faults (over all orderings) for both techniques is the same. On the other hand, alternative hypothesis (H_1) states that if the difference between the mean of the percentages of faults, which have been detected by each of prioritization technique, is not zero then they will be considered as different. We independently evaluate this hypothesis for a number of random situations; the randomization is proved by changing the SUT under consideration without pattern or rationale. Therefore, by considering a significance level $\alpha = 0.05$, we would be able to reject null hypothesis if $p - value < 0.05$ for each independent situation.

4.7.2.1 Effect-Size

In order to add a "size of difference" statement to our comparison criteria and normalized the statistical evaluation of multiple (with independent subjects (SUT)) tests, we principally consider the strength or magnitude of a treatment effect, by calculating Cohen's d measure. Cohen's d is

[18] As a cross-check, the samples for this test were subjected to a Shapiro-Wilk test for normality. In all situations, the samples passed the test

defined as the difference between the means, $M_1 - M_2$, divided by standard deviation (SD), s_{pooled}. So,

$$d = \frac{M_1 - M_2}{s_{pooled}} \tag{41}$$

where,

$$s_{pooled} = \sqrt{SD^2_{group1} + SD^2_{group2} \Big/ 2}$$

Since, Cohen's d computes the standardized difference between two groups, or a brace, of samples; it can help us to interpret the strength of our proposed techniques in comparison with others. [174] suggests that d=0.2 can be considered as a "small" effect size, while 0.5 can be interpreted as a "medium" effect size, and 0.8 "large" effect size. In this research, we consider this measure on the percentage of detected faults, which have been detected by a brace of prioritization techniques with mean APFD, in order to investigate the magnitude of each approach's effect. So, for each brace:

$(R_i, R_j), i, j \in \{All\ considered\ prioritization\ approaches\}$, both $\overline{P_i}$ and $\overline{P_j}$ (the average of percentage of detected faults) should be calculated to compute:

$$d = \frac{\overline{P_i} - \overline{P_j}}{s_{pooled}}, s_{pooled} = \sqrt{SD^2_{R(i)} + SD^2_{R(j)} \Big/ 2} \tag{42}$$

4.7.3 Experimental Setup

According to a recent empirical comparison, AutoBlackTest outperforms GUITAR in both code-coverage and the number of detected faults [16]. Therefore, in this study, we principally use AutoBlackTest to generate the required test cases.

4.7.3.1 AutoBlackTest and Modified Framework

AutoBlackTest is an automatic testing tool that builds a model of a SUT and produces test cases by walking through the model. The most significant feature of AutoBlackTest is its ability to interact with an unknown environment. To address this issue, AutoBlackTest uses a Q-learning approach [16].

Therefore, AutoBlackTest is a combination of a Q-Learning Agent and a Test Case Selector. The Q-learning agent is responsible for executing a sequence of actions (episodes) extracted from the model. On the other hand, the test case selector eliminates redundant test cases, which cover the same statements within the code.

Table 21. Investigated applications

Application	**Version**	**Statement Cov.(%)**	**KLOC**
UPM[19] (a personal password manager)	V 1.6	86	2.515
Buddi[20] (a finance tool)	V 3. 4. 0. 8	64	10.580
PDFSAM[21] (a merging and splitting PDF documents tool)	V 0.7 Stable release 1	68	3.138
TimeSlotTracker[22] (tasks and activities management tool)	V 0.4	59	3.499

[19] http://upm.sourceforge.net/

[20] http://buddi.digitalcave.ca/

[21] http://sourceforge.net/projects/pdfsam/

[22] http://sourceforge.net/projects/timeslottracker/

The derived model by AutoBlackTest is a regular digraph [16]. Therefore, according to the Definition 14, AutoBlackTest generates a graph $G = (S, E, V)$, where the set of nodes (S) represent the abstract GUI states and the edges $(e_k \in E = (s_i, s_j), where\ s_i, s_j \in S)$ indicate the transition between states. The transitions are labeled with the name of actions and corresponding Q-values (V).

It is believed that AutoBlackTest is a state of the art mechanism to automatically infer a regular digraph, an essential building block for the construction of the extended digraph. While AutoBlackTest is a state of the art technique, it is far from perfect and only constructs a portion of the total regular digraph for the "average" GUI-based application. Figure 17 depicts the proposed framework and the procedure to determine the effectiveness of the proposed approaches both in terms of fault detection, and in comparison with other existing techniques (Random, Optimal, Worst, Coverage-based and Accumulated Q-value). The experimental environment consists of two steps:

- **Step1:** Examine the generated test cases by parsing episodes' produced by AutoBlackTest. Each episode, or test case, is a sub-graph of the SUT's behavioral model, which is generated as a Dot file[23]. Then, extract the initial HMM parameters and coverage information.
- **Step2:** Train the most appropriate HMM[24] with a maximum likelihood estimate of the parameters. Prioritize test cases based on RL-based HMM and Accumulated Q-value ordering techniques, as well as ranking them through Random, Optimal, Worst and

[23] DOcument Template file format

[24] RHMM package [89] of the **R** programming environment [250] have been used

Additional statement coverage approaches. Finally, analyze the effectiveness of the considered prioritization techniques based upon the APFD measure.

To empirically evaluate AutoBlackTest, widgets from four different-domain applications were extracted [168] and tested. These applications are listed in Table 21 and are reused in this study, as it allows us to use [16] as an initial reference point. [16] provide results after their test case selection algorithm runs, and hence it can be considered as a control case (base value for improvement).

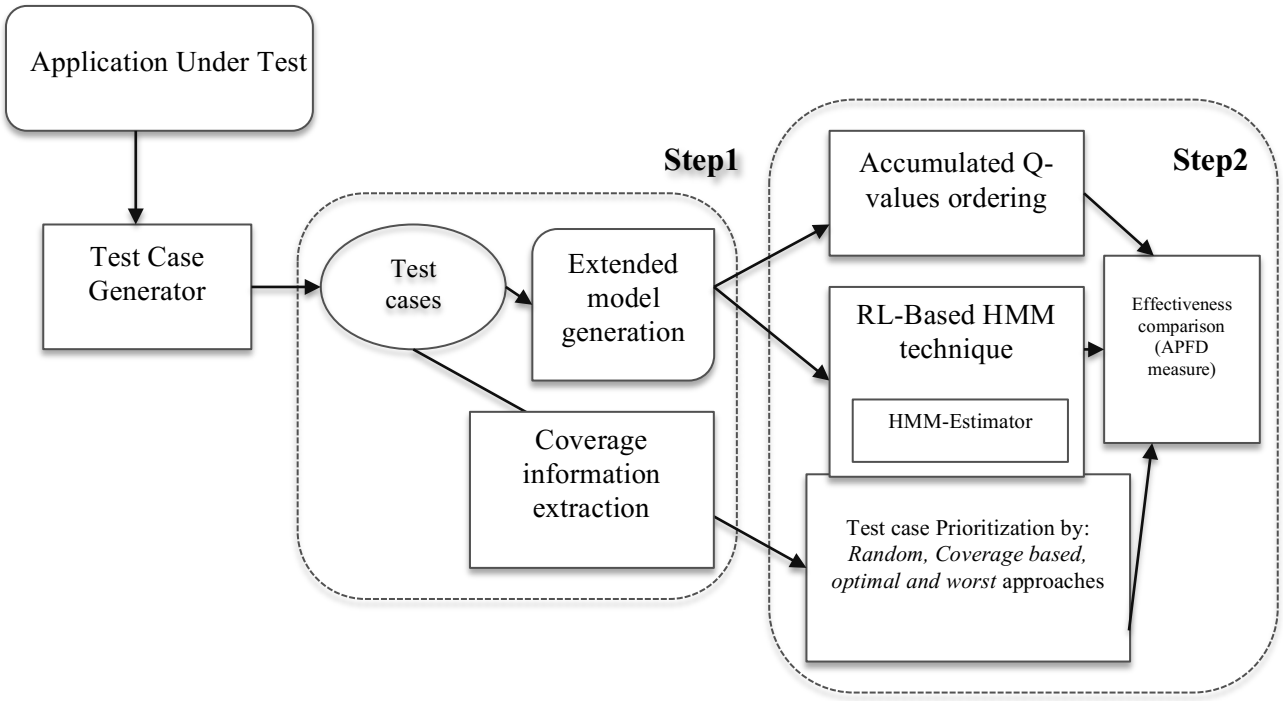


Figure 17. Experimental Setup; Combination of a test case generator (ABT) and prioritization techniques

In subsequent experiments, we also consider an extended version of PDFSAM with a larger size test suite and three additional GUI applications from [164].

4.7.3.2 Fault Matrix

In a fault matrix, rows represent the test cases and columns represents the faults in the System Under Test. If a test case i detects fault j then the entry i,j in the matrix would be equal to 1.

So using a fault matrix, we would be able to indicate the capability of each test case in detecting faults by counting the number of 1s in the corresponding row [147]. In this research, we have applied several test case prioritization approaches on 8 different GUI applications. Hence, providing the details of all fault matrices in this study is not possible, Table 22 summarizes the various characteristics of each fault matrix.

Table 22. Fault matrix summary

Application	#Test cases	#Faults	Avg. faults per test	% tests detecting X% faults		
				X=0	X=20	X=50
UPM	15	8	0.5	46.6	0.0	0.0
Buddi	15	2	0.5	86.6	0.0	0.0
PDFSAM	26	3	0.5	88.4	0.0	0.0
TimeSlotTracker	19	1	0.5	94.7	0.0	0.0
Extended PDFSAM	60	6	0.5	90.0	0.0	0.0
WordProcessor	18	12	2.7	72.2	16.6	0
TerpPaint	23	12	3	82.6	8.6	4.3
Calculator	15	12	1.5	40.0	0.0	0.0

4.7.4 Experimental Results

4.7.4.1 Study 1: UPM

In this experiment, we investigate the effectiveness of utilizing the considered prioritization techniques on the UPM application. To estimate a suitable RL-based HMM for UPM, the following parameters are needed:

- N: The number of hidden states which is equivalent to the total number of states in the UPM behavioral model, 52 for this package.
- M: The number of observation symbols which is equivalent to the number of distinct actions in UPM, 36.
- The state transition probability matrix is calculated using the estimate of Q-values between the distinct states (if there is more than one possible action between states, the action with highest Q-value would be selected); In this case, we have a 52*52 transition probability matrix.
- Calculating the frequency of observing a specific action in a specific state; this produces a 52*36 matrix. This matrix is the observation symbol probability matrix.

At the next step, we use the forward algorithm on UPM's Hidden Markov Model to find the probability of an observed sequence given an estimated HMM. It exploits recursion in the calculations to avoid the necessity for exhaustive calculation of all the paths through the execution process. According to Definition 21, we prioritized test cases based upon their corresponding F_p.

Therefore, in this case, we calculate 15 F_p, for 15 UPM episodes (test cases). Obviously, the accumulated forward probabilities tend to get small (converge to 0) since they are affected by previous probabilities in the forward and recursive process (Numerical stability is therefore guaranteed by the scaling introduced in Section 4.3.2.1).

In addition, it is worthwhile to mention that the summation of the probabilities will not be one because:

The generated graph is not a complete graph as there will be distinct vertices that cannot be connected by a pair of unique edges. So some paths cannot be considered.

Some paths are impossible to walk through, because of the application design. AutoBlackTest only explores the paths (episodes), which are reachable and logical. In this case, 15 episodes are the minimum number of paths that can detect failures.

Therefore, only when we were able to compute the forward probabilities for all possible episodes (paths) of a graph (preferably a complete graph), can we expect the summation of the forward probabilities to converge to one. According to the F_p value for each episode, the probabilities can be sorted in the following descending order ($\boldsymbol{E_{10}}$ denotes the Episode (test case) number 10):

$$\{\boldsymbol{F_p(E_{10}) > F_p(E_6) > F_p(E_3) > F_p(E_{14}) > F_p(E_1) > F_p(E_{13}) > F_p(E_9) > F_p(E_2) > F_p(E_4) > F_p(E_7) > F_p(E_8) > F_p(E_{12}) > F_p(E_0) > F_p(E_{11}) > F_p(E_{15})}\}$$

AutoBlackTest reports that episodes 2, 3, 6, 7, 8, 10, 13 and 14 are able to detect faults in UPM [16].

However, according to its test case selection policy, it only selects episodes 4, 8, 11, and 13 and eliminates the others. Eliminating episodes, which are able to detect faults, may cause unexpected costs and problems in regression testing processes.

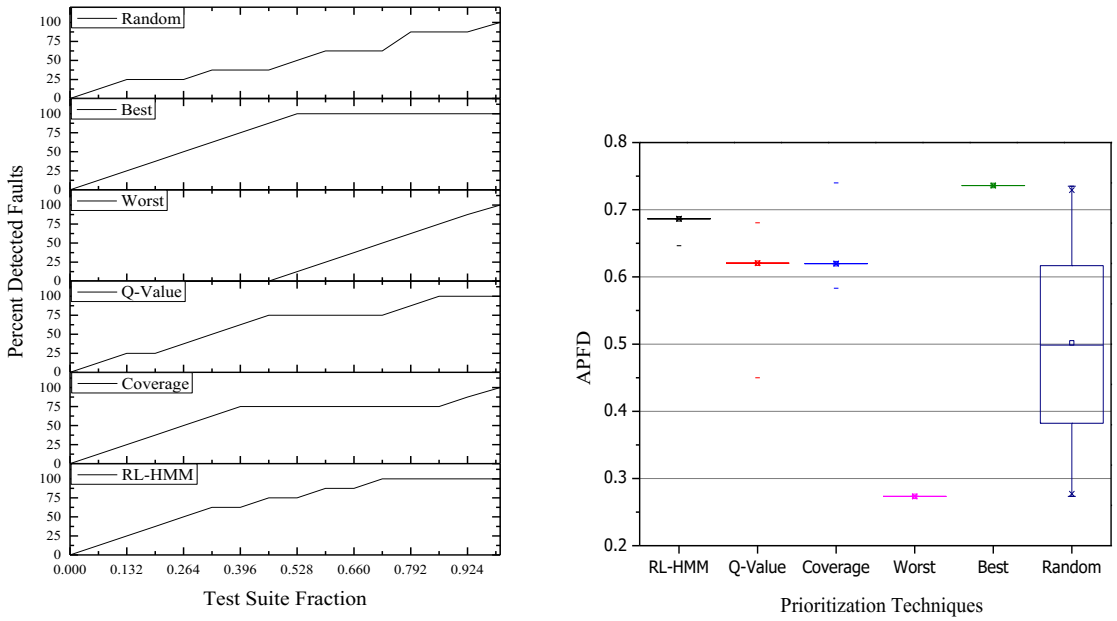


Figure 18. (a) Percent of faults detected versus test suite fraction. **(b)** Box plot of APFD for UPM

On the other hand, the proposed techniques (RL-base HMM and Accumulated Q-value) are able to prioritize test cases (episodes) along with increasing the original fault detection rate.

To get more reliable results, we execute each of the techniques 1000 times. Then we calculated the means and standard deviations of the APFDs to evaluate each prioritization technique with respect to the prioritization objective (improving fault detection rate). Then we select an ordering of test cases with the closest APFD to the mean APFD.

As can be seen from Table 23, both proposed techniques have larger mean APFDs (and smaller standard deviations), implying that their performance is universally "good". (The reason of getting zero SD for both the Worst and Best approaches are because there are only one test suite which satisfies the best and worst prioritizations.)

Figure 18(a) shows the percentage of detected faults versus the execution sequence of test cases for the prioritized test suites, with mean APFD (average performance). This figure depicts that both the RL-based HMM and Accumulated Q-value techniques are able to detect a considerable portion of faults by executing a smaller number of test cases. For example, by executing 85% of test cases both techniques detect 100% of the faults (8 out of 8 faults). This amount is 75% faults for Additional statement coverage, and 87% for the Random prioritization technique.

In addition, Figure 18(b) shows the distribution of the 1000 APFD values using a boxplot. This figure indicates that the APFDs' distribution for the RL-based HMM and Accumulated Q-value approaches are not variant in comparison with the Random technique.

Table 23. APFD of the applied prioritization techniques for UPM

Prioritization Techniques	**Means of APFDs**	**Standard Deviation (SD)**
RL-based HMM	0.6865	0.00012
Accumulated Q-value	0.6205	0.00472
Additional Statement Coverage	0.6198	0.0046
Random	0.5018	0.5018
Worst	0.2733	0
Optimal (Best)	0.736	0

Table 24. The statistical analysis for RL-based HMM technique vs. other techniques in UPM

RL-based HMM Technique Vs. Others	**t-statistic**	**DF**	**p-value (one-tail)**	**Cohen's d**
RL-based HMM vs. Accumulated Q-Value	3.16228	14	0.00644	1.6329
RL-based HMM vs. Statement Coverage	2.23607	14	0.04047	1.1547
RL-based HMM vs. Random	5.36745	14	7.8360E-5	2.7717
RL-based HMM vs. Worst	6.48385	14	1.03159E-5	3.3482
RL-based HMM vs. Optimal (Best)	-2.42272	14	0.02853	-1.2511

Table 24[25] shows the results of applying the statistical tests and the magnitude of Cohen's d (effect size), which are used to determine the effectiveness of RL-based HMM in comparison with the other approaches. In this table, we can see that the RL-based HMM technique is better than the Additional statement coverage and Random prioritization techniques. In addition, the RL-based HMM method is prioritizing test cases significantly better than Accumulated Q-value approach. Large Cohen's ds also prove that there is a significant difference, on average between the fault detection capability of the proposed techniques and other prioritization approaches. This measure is more than 0.8 (large) for all of the considered braces in this case.

We repeat this process for the other three applications to conclude the empirical comparison about the effectiveness of the proposed approaches in improving the fault detection rate.

4.7.4.2 Study 2: Buddi

To estimate an HMM for the Buddi application, we walk through its behavioral model to compute and extract the following parameters:

- N: The number of hidden states is 76.
- M: The number of observation symbols is 45.
- A 76*76 transition probability matrix estimated using Q-values. And,
- A 76*45 observation symbol probability matrix.

After achieving the best HMM using the Baum-welch algorithm, the same process as for the UPM investigation process was followed, we executed the forward algorithm to compute the

[25] DF denotes the Degree of Freedom

forward probability of episodes. 15 episodes (F_p) were produced by AutoBlackTest from the Buddi application. Again, we can sort the forward probabilities:

$$\{\boldsymbol{F_p(E_7) > F_p(E_{14}) > F_p(E_0) > F_p(E_3) > F_p(E_1) > F_p(E_2) > F_p(E_6) > F_p(E_{10}) > F_p(E_5) > F_p(E_9) > F_p(E_8) > F_p(E_{11}) > F_p(E_{12}) > F_p(E_{13}) > F_p(E_4)}\}$$

According to AutoBlackTest [16] episodes 7 and 14 are considered as fault-prone test cases (test cases that reveal faults). The RL-based HMM technique correctly prioritizes these two episodes, in its final test suite. This test suite is able to find 100% of Buddi's faults in regression testing.

Both proposed techniques are able to detect all the faults by executing only 20% of the test cases, while only one fault can be revealed by executing more than 50% of the test cases using Additional statement coverage and Random prioritization approaches. This fact is also visible in Figure 19(a) showing the percent of detected faults using each technique. Also in Figure 19(b), boxplots summarize the distribution of the performance of the prioritization techniques. Again, this distribution shows lower variance for both of the proposed techniques when compared against Random and Additional statement coverage approaches.

Similarly to UPM, mean and standard deviation of the APFDs for each technique are calculated (Table 25). According to this table, and formally verified in Table 23, we can conclude that RL-based HMM is not statistically significantly different to the Optimal approach (upper bound). Using Table 26, we provide a statistical statement of RL-based HMM performance when compared with the other techniques. The RL-based HMM technique is significantly different from Additional statement coverage and Random prioritization as well as Worst prioritization.

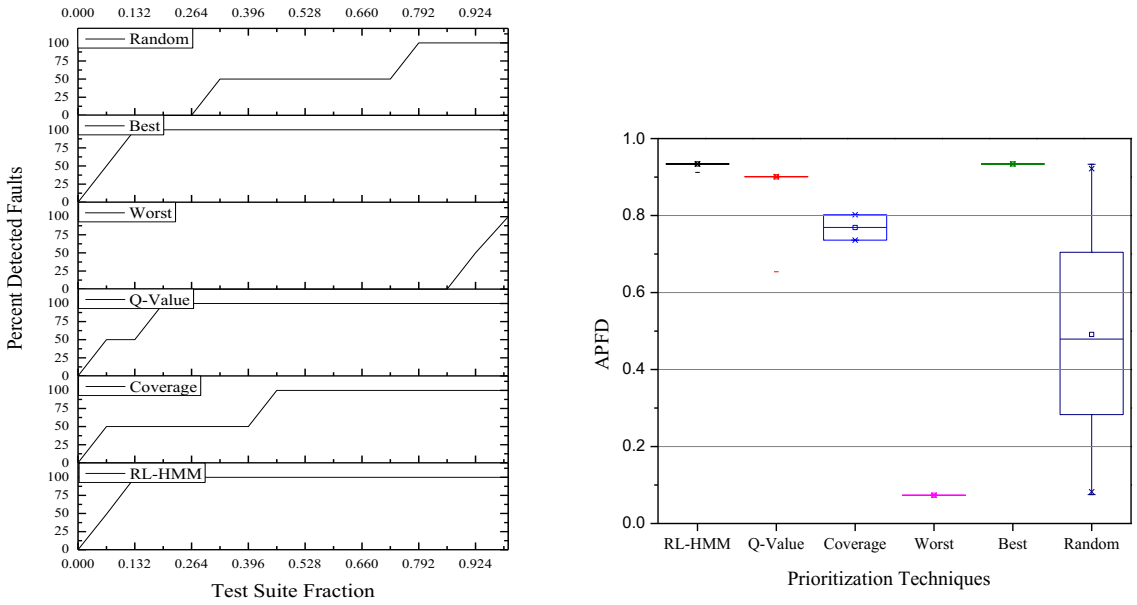


Figure 19. (a) Percent of faults detected versus test suite fraction. (b) Box plot of APFD for Buddi

The Cohen's d measure also indicates that the difference between RL-based HMM and other techniques is large, while it is medium for RL-based HMM vs. Accumulated Q-Value, although the difference is not considered statistically significant. There is a same story for the Accumulated Q-value prioritization technique. It prioritizes test cases significantly better than other techniques, except obviously for RL-based HMM.

Table 25. APFD of the applied prioritization techniques for Buddi

Prioritization Techniques	**Means of APFDs**	**Standard Deviation**
RL-based HMM	0.9339	7.0698E-4
Accumulated Q-value	0.901	0.0089
Additional Statement Coverage	0.769	0.02688
Random	0.491	0.24868
Worst	0.073	0
Optimal (Best)	0.934	0

Table 26. The statistical analysis for RL-based HMM technique vs. other techniques in Buddi

RL-based HMM Technique Vs. Others	**t-statistic**	**DF**	**p-value** (one-tail)	**Cohen' d**
RL-based HMM vs. Accumulated Q-Value	1	14	0.33317	0.5163
RL-based HMM vs. Statement Coverage	2.61116	14	0.01966	1.3483
RL-based HMM vs. Random	4.86926	14	2.04238E-4	2.5144
RL-based HMM vs. Worst	7.45575	14	2.02796E-6	3.8501
RL-based HMM vs. Optimal (Best)	-1	14	0.34463	-0.5163

4.7.4.3 Study 3: PDFSAM

PDFSAM is a Java-based tool for merging and splitting PDF files. Again, the package was used to evaluate the proposed techniques [16].The Initial HMM parameters for PDFAM are:

- N: The number of hidden states is 79.
- M: The number of observation symbols is 49.
- Transition probability matrix has 79 rows and 79 columns.
- The observation symbol probability matrix has 79 rows and 43 columns.

In this case, we considered 26 episodes, which according to AutoBlackTest [16] reports that episodes 0, 2 and 25 are able to detect faults. Ordering the forward probabilities, demonstrates that the RL-based HMM prioritization technique prioritizes episodes according to their contribution in fault detection and no test cases (episodes) are withdrawn.

$$\{F_p(E_{10}) > F_p(E_0) > F_p(E_2) > F_p(E_7) > F_p(E_{11}) > F_p(E_{25}) > F_p(E_{14}) > F_p(E_{15}) > F_p(E_{18}) > F_p(E_6) > F_p(E_3) > F_p(E_{24}) > F_p(E_{20}) > F_p(E_4) > F_p(E_9) > F_p(E_1) > F_p(E_{19}) > F_p(E_5) > F_p(E_{22}) > F_p(E_8) > F_p(E_{17}) > F_p(E_{13}) > F_p(E_{21}) > F_p(E_{12}) > F_p(E_{16}) > F_p(E_{23})\}$$

This prioritization is able to detect 100% of the faults in PDFSAM's regression testing process by only executing 23% of test cases. The Accumulated Q-value approach is able to only detect

67% of faults by executing 78% of prioritized test cases. Figure 20(a) shows the number of detected faults versus test case execution. The percent of faults detected by the Additional statement coverage technique is 33% after executing 42% of the test cases. Also, the APFD distribution (Figure 20(b)) for both proposed techniques is relatively invariant, when compared to the Random and Additional statement coverage prioritizations.

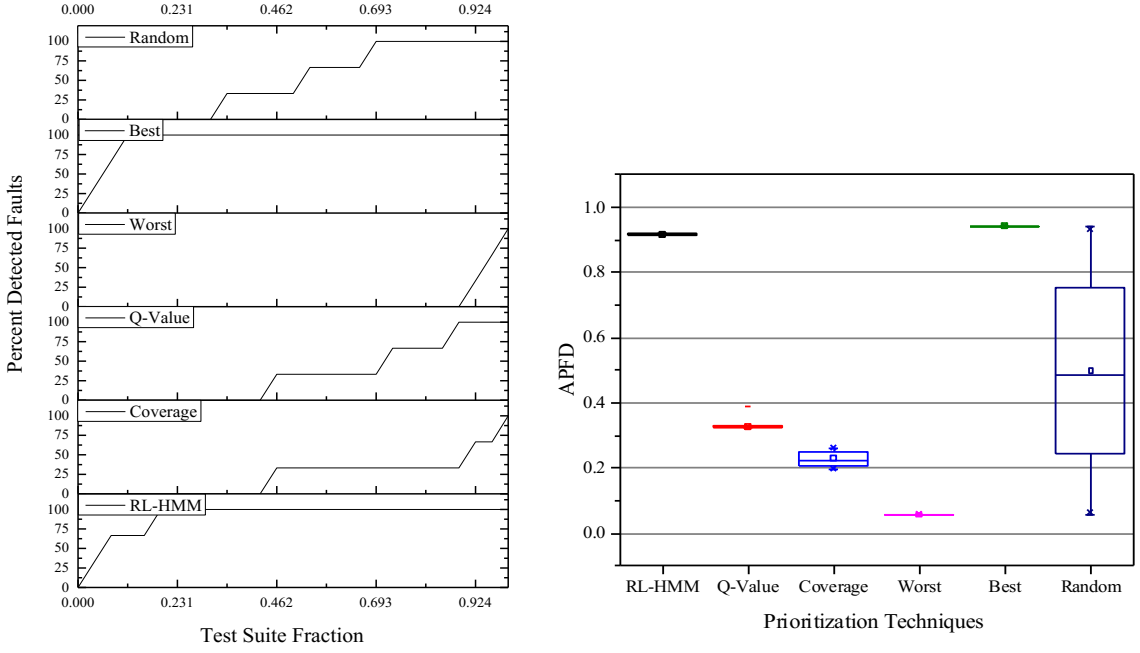


Figure 20. (a) Percent of faults detected versus the test suite fraction. (b) Box plot of APFD for PDFSAM

Table 27. APFD of the applied prioritization techniques for PDFSAM

Prioritization Techniques	**Means of APFDs**	**Standard Deviation**
RL-based HMM	0.9164	3.1868E-4
Accumulated Q-value	0.3260	0.00206
Additional Statement Coverage	0.2285	0.0217
Random	0.4985	0.25259
Worst	0.0568	0
Optimal (Best)	0.9422	0

Table 28. The statistical analysis for RL-based HMM technique vs. other techniques in PDFSAM

RL-based HMM Technique Vs. Others	**t-statistic**	**DF**	**p-value** (one-tail)	**Cohen' d**
RL-based HMM vs. Accumulated Q-Value	8.2955	25	8.9005E-9	3.2537
RL-based HMM vs. Statement Coverage	12.49415	25	1.70486E-12	4.9006
RL-based HMM vs. Random	5.6673	25	5.83228E-6	2.2228
RL-based HMM vs. Worst	13.78585	25	1.82592E-13	5.4072
RL-based HMM vs. Optimal (Best)	-1.44222	25	0.16118	-0.5656

Table 27 lists the mean and standard deviations of the APFDs for each test case prioritization technique. Table 28 shows that the RL-based HMM technique is significantly better than the other techniques. The Accumulated Q-value technique is significantly better than the Additional statement coverage technique. Moreover, again, all the differences between the RL-based HMM and the other techniques are large according to Cohen's d measure.

4.7.4.4 Study 4: TimeSlotTracker

TimeSlotTracker is the fourth and last application that has been investigated in the evaluation phase. Similar to previous applications, we need to initialize the HMM and finally estimate it with the best parameters to meet the maximum likelihood condition.

- N: The number of hidden states is 246.
- M: The number of observation symbols is 46.
- A 246* 246 matrix representing the transition probability matrix
- The observation symbol probability matrix has 246 rows and 46 columns.

In this case, AutoBlackTest generates 19 episodes throughout the testing process and episode 18 is considered as the only fault revealing test case. Estimating an appropriate RL- based HMM, we compute the forward probabilities as before.

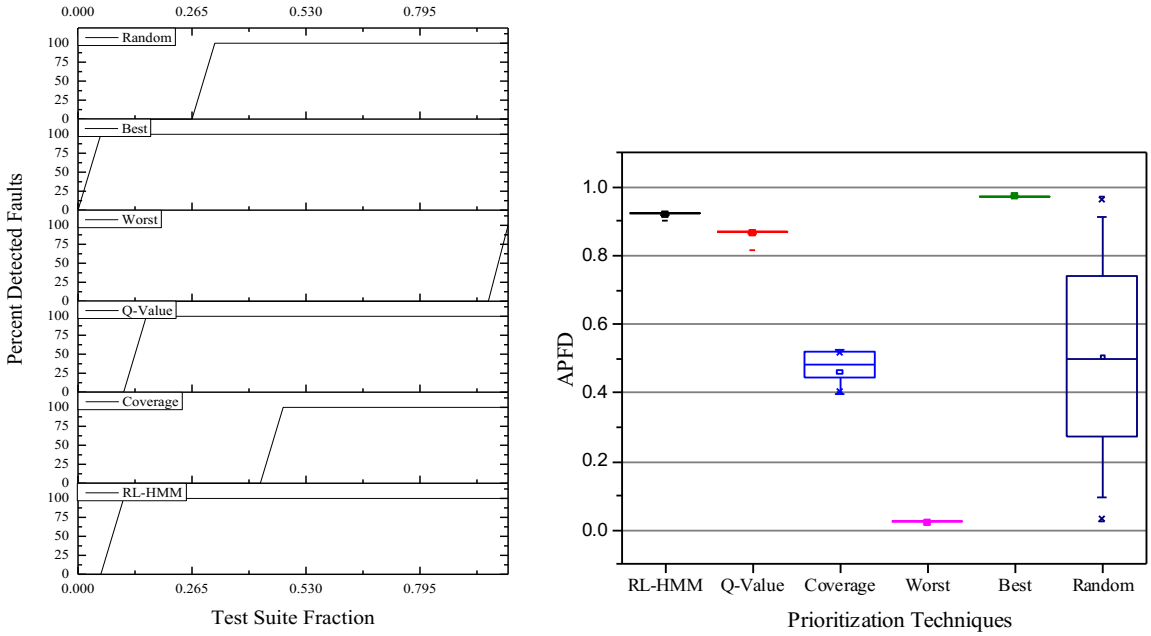


Figure 21. (a) Percent of faults detected versus test suite fraction. (b) Box plot of APFD for TimeSlotTracker

Table 29. APFD of the applied prioritization techniques for TimeSlotTracker

Prioritization Techniques	**Means of APFDs**	**Standard Deviation (SD)**
RL-based HMM	0.9209	7.2276E-4
Accumulated Q-value	0.8675	0.0017
Additional Statement Coverage	0.462	0.04305
Random	0.5028	0.27145
Worst	0.023	0
Optimal (Best)	0.9735	0

Table 30. The statistical analysis for RL-based HMM technique vs. other techniques in TimeSlotTracker

RL-based HMM Technique Vs. Others	**t-statistic**	**DF**	**p-value** (one-tail)	**Cohen' d**
RL-based HMM vs. Accumulated Q-Value	**1**	18	**0.32988**	**0.4588**
RL-based HMM vs. Statement Coverage	**2.51661**	18	**0.02099**	**1.1546**
RL-based HMM vs. Random	**2.17946**	18	**0.04209**	**1.000004**
RL-based HMM vs. Worst	10.37642	18	2.89352E-9	4.7610
RL-based HMM vs. Optimal (Best)	-1	18	0.32988	-0.4588

After sorting the probabilities, the RL-based HMM prioritization technique reports the following test suite.

$$\{F_p(E_{18}) > F_p(E_5) > F_p(E_7) > F_p(E_0) > F_p(E_{12}) > F_p(E_1) > F_p(E_9) > F_p(E_6) > F_p(E_{11}) > F_p(E_4) > F_p(E_{10}) > F_p(E_{15}) > F_p(E_{14}) > F_p(E_{13}) > F_p(E_3) > F_p(E_8) > F_p(E_2) > F_p(E_{16})\}$$

It shows that the fault detection rate using prioritized test cases would be 100% for TimeSlotTracker; it means that by executing only one test case the fault would be detected. On the other hand, this fault would be found after the execution of 21% and 37% of the test cases, respectively, using Accumulated Q-value and Additional statement coverage approaches. Figure 21(a) also depicts the percentage of detected faults versus test case executions for each applied prioritization technique. In addition, Additional statement coverage and Random approaches still have more variant distributions than the other techniques, while the RL-based HMM distribution is "close" to the upper bound (Figure 21(b)). The mean and standard deviation APFD of the prioritization processes are shown in Table 29. Table 30 indicates that RL-based HMM is significantly better than Random and Additional statement coverage, but it is not significantly different from the Accumulated Q-value prioritization method. Cohen's d measure also confirms this result. The Accumulated Q-value approach again provides better a better performance than the Additional statement coverage technique.

4.7.4.5 Study 5: Extended PDFSAM

To demonstrate that the proposed techniques are also applicable of prioritizing test cases with somewhat larger size test suites, we applied the approaches to order 60 test cases.

For this study, we run the testing process (AutoBlackTest) of PDFSAM again, but now for two hours. The default testing time had been set to 1 hour for the other case studies in order to allow an unbiased comparison across application programs. In the other case studies, except PDFSAM, the test case generation procedure completed before the testing time finished. In this experiment, we generated 60 test cases of which test cases numbered 0, 2,25,29,40 and 52 were able to detect faults. In order to initialize the HMM and finally estimate it with the best parameters to meet the maximum likelihood condition, we consider the following parameters:

- N: The number of hidden states is 207.
- M: The number of observation symbols would be 79.
- A 207* 207 matrix representing the transition probability matrix
- The observation symbol probability matrix also has 207 rows and 79 columns.

After estimating the model and calculating the accumulated Q-values, we sort the test cases based upon each considered technique. The results show that the RL-based HMM technique again outperforms the other techniques when considering improving the fault detection rate. For instance, a prioritized test suite using RL-based HMM is able to detect all 6 faults by executing only 16 test cases (only 26% of the test suite) while statement coverage technique needs to run 48 test cases (80% of test suite) to find all of the faults. Figure 22(a) shows the percentage of detected faults versus the fraction of test suite executed with the mean APFD. Figure 22(b) also shows that both proposed techniques are relatively invariant in terms of APFD.

In addition, Table 31 illustrates the results of computing the mean and standard deviation of the APFD measure for 1000 trials per application. Table 32 reports the result of applying a paired t-test and calculating Cohen's d measure on the percentage of detected faults by executing a

prioritized test suite with mean amount of APFD. According to this investigation, the RL-based HMM technique prioritizes test cases significantly better than the other techniques except Optimal. Also, the Accumulated Q-value technique has better APFD measure than coverage-based prioritization and sorts test cases significantly better than it.

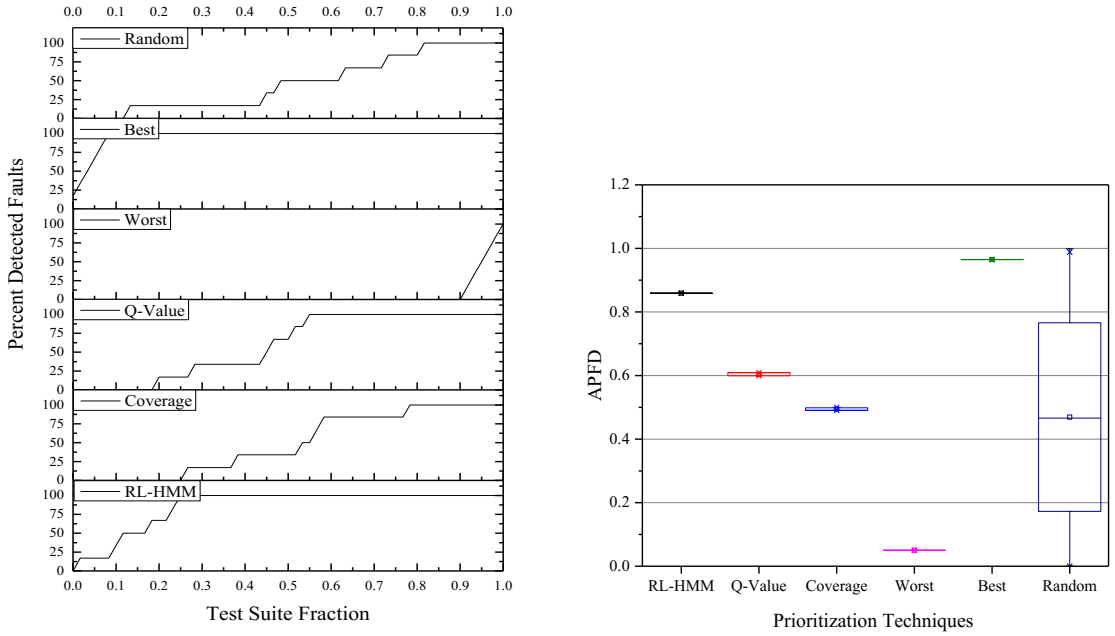


Figure 22. (a) Percent of faults detected versus test suite fraction. (b) Box plot of APFD for Extended PDFSAM

Table 31. APFD of the applied prioritization techniques for Extended PDFSAM

Prioritization Techniques	Means of APFDs	Standard Deviation (SD)
RL-based HMM	0.895	0
Accumulated Q-value	0.604	0.005
Additional Statement Coverage	0.494	0.0042
Random	0.469	0.2967
Worst	0.0503	0
Optimal (Best)	0.965	0

Table 32. The statistical analysis for RL-based HMM technique vs. other techniques in Extended PDFSAM

RL-based HMM Technique Vs. Others	t-statistic	DF	p-value (one-tail)	Cohen' d
RL-based HMM vs. Accumulated Q-Value	6.89838	59	3.72746E-9	1.7961
RL-based HMM vs. Statement Coverage	8.91787	59	1.35721E-12	2.3220
RL-based HMM vs. Random	10.16044	59	1.16392E-14	2.6455
RL-based HMM vs. Worst	19.28205	59	2.13086E-27	5.0206
RL-based HMM vs. Optimal (Best)	-3.94484	59	2.1174E-4	-1.0271

4.7.4.6 Study 6: WordProcessor, TerpPaint and Calculator

As our last experiment, we have applied our proposed techniques on three GUI applications: WordPrcessor, TerpPaint and Calculator[26]. TerpPaint is a paint program written using Java and AWT/Swing. It is similar to Microsoft Paint and is developed at the University of Maryland. WordProcessor is a small application, which was developed by [175] as part of their efforts in a "fast-paced guide" for teaching Swing. Finally, Calculator is an open-source program, which implements a four-function calculator; Calculator is developed using Java Swing[27].

The purpose of choosing these applications is comparing the results of our test case prioritization approaches against those presented in [164]. Unfortunately, the framework used in their experiment is not available. Further, the paper's explanations are insufficient to re-implement their experimental environment.

Hence, we can only use the paper's results and run our methods on identical applications. This implies that the results are not directly comparable as the test data will be different in the two papers. Since, AutoBlackTest generates meaningful test cases with optimum lengths, combining

[26] http://sourceforge.net/projects/terppaint

[27] http://beginner-java-tutorial.com/java-swing-calculator.html

test cases to generate different-length test suites can lead to generating false positive fault-revealing test cases. Therefore, we only consider the prioritization results for L1 test cases from Huang et al.'s research [164].

In order to increase the similarities between the two experimental frameworks, we also seeded 12 faults from the same types into a random location in each application and ran (100 times) RL-based HMM, Accumulated Q-value and Coverage techniques on each set of test cases generated using AutoBlackTest. In addition, the untreated (un-prioritized) and best orderings of test cases has been considered to estimate the test effectiveness boundaries and hence will be used as the point of comparison.

It is worth noting that the reason of considering the untreated ordering in this research is that Huang et al.'s research only covers two techniques (untreated and coverage) as non-weight based prioritization methods and applies t-tests and statistical analysis on them. Also, they didn't compare weight-based techniques with the Best and Worst orderings. Therefore, unlike the previous studies, we have compared our proposed technique with untreated method in order to follow the same approach as theirs.

Because the exact test cases, which have been used in Huang et al.'s research, were not available for doing an accurate comparison, it cannot be claimed that the test cases have been generated and used in this research have the exact same coverage or fault detection capability as Huang et al's test cases. Therefore we have decided to investigate the effectiveness of each considered technique by comparing its fault detection capability with untreated, coverage-based and best orderings. In order to clarify the effectiveness of each test case prioritization technique, we also compared the mean of APFD of each method with the Best prioritization (the optimal ordering of

test cases). As the set of test cases are different in the two papers the best (or optimal) results are different. The following formula shows the method we have used to calculate the relative ratio of each method versus their own Best (optimal) approach.

$$Relative\ Ratio\ (RR) = \frac{Best_{APFD-Treatment_{APFD}}}{Best_{APFD}} \qquad (16)$$

Obviously, prioritization techniques with a low RR value approach optimality. Table 33-Table 37 show the mean APFDs and Relative Ratios for each applied technique. In addition, the results of the statistical analysis for each considered application are provided.

Table 33. APFD of the applied prioritization techniques for TerpPaint

Prioritization Techniques	**Means of APFDs**	**RR**
RL-based HMM	0.924	**0.0149**
Accumulated Q-value	0.891	0.0501
Additional Statement Coverage	0.755	0.1950
Untreated	0.690	0.2643
Best	0.938	0
High-to-low equal weight (L1)	0.934	0.603
High-to-low fault-prone weight (L1)	0.844	0.1590
High-to-low adjusted weight (L1)	0.966	**0.0281**
Best (L1)	0.994	0

Table 34. The statistical analysis for RL-based HMM technique vs. other techniques in TerpPaint

RL-based HMM Technique Vs. Others	**t-statistic**	**p-value** (one-tail)
RL-based HMM vs. Statement Coverage	**3.81329**	**8.93385E-4**
RL-based HMM vs. Untreated	**4.12466**	**4.12614E-4**
RL-based HMM vs. Best	1.2916	0.20932
Untreated vs. fault-prone weight (L1)	**2.6730**	**0.0043**
Untreated vs. adjusted weight (L1)	-3.1059	0.0012
Coverage vs. fault-prone weight (L1)	**3.6082**	**0.0002**
Coverage vs. fault-prone weight (L1)	-1.9221	0.0286

Table 35. APFD of the applied prioritization techniques for WordProcessor

Prioritization Techniques	**Means of APFDs**	**RR**
RL-based HMM	0.899	**0.0088**
Accumulated Q-value	0.887	0.0220
Additional Statement Coverage	0.716	0.2105
Untreated	0.707	0.2205
Best	0.907	0
High-to-low equal weight (L1)	0.946	0.0463
High-to-low fault-prone weight (L1)	0.969	0.0231
High-to-low adjusted weight (L1)	0.972	**0.0201**
Best (L1)	0.992	0

Table 36. The statistical analysis for RL-based HMM technique vs. other techniques in WordProcessor

RL-based HMM Technique Vs. Others	**t-statistic**	**p-value** (one-tail)
RL-based HMM vs. Statement Coverage	**4.265**	**4.65314E-4**
RL-based HMM vs. Untreated	**3.67374**	**0.00174**
RL-based HMM vs. Best	-1.0063	0.327
Untreated vs. fault-prone weight (L1)	-3.7138	0.0001
Untreated vs. adjusted weight (L1)	-3.6246	0.0002
Coverage vs. fault-prone weight (L1)	-0.6824	0.2485
Coverage vs. fault-prone weight (L1)	-0.7929	0.2151

Table 37. APFD of the applied prioritization techniques for Calculator

Prioritization Techniques	**Means of APFDs**	**RR**
RL-based HMM	0.708	**0.0432**
Accumulated Q-value	0.622	0.1594
Additional Statement Coverage	0.577	0.2202
Untreated	0.408	0.4480
Best	0.740	0
High-to-low equal weight (L1)	0.838	0.1440
High-to-low fault-prone weight (L1)	0.898	0.0827
High-to-low adjusted weight (L1)	0.931	**0.0490**
Best (L1)	0.979	0

Table 38. The statistical analysis for RL-based HMM technique vs. other techniques in Calculator

RL-based HMM Technique Vs. Others	**t-statistic**	**p-value** (one-tail)
RL-based HMM vs. Statement Coverage	**3.81329**	**8.93385E-4**
RL-based HMM vs. Untreated	**4.12466**	**4.12614E-4**
RL-based HMM vs. Best	-2.99924	0.00899
Untreated vs. fault-prone weight (L1)	-3.3235	0.0013
Untreated vs. adjusted weight (L1)	-3.8280	0.0003
Coverage vs. fault-prone weight (L1)	-2.4221	0.0113
Coverage vs. fault-prone weight (L1)	-3.2269	0.0016

Table 38 indicates that the RL-based HMM technique has the closest behavior to the Best ordering due to the low RR (=0.0149). In addition, according to the Table 34, it also prioritizes test cases significantly better than untreated and coverage-based prioritizations in TerpPaint. The same prioritization efficiency can be interpreted based uponTable 35 to Table 38 for both the WordProcessor and the Calculator application. For example (Table 35), the RL-based HMM approach prioritizes test cases with the lowest difference compared to the Best technique (RR=0.0088) while, for the High-to-low adjusted weight (L1), the most accurate proposed method by [164] the RR value is equal to 0.0201.

4.8 Discussion

We have investigated four different applications by inferring four different RL-based Hidden Markov Models. The results show that the RL-based HMM prioritization technique is successful in all four cases, plus a second trial on PDFSAM plus three extra GUI applications proposed by [164], specifically in prioritizing test cases that contribute to discovering faults.

According to the empirical results represented in Section 4.7, since the RL-based HMM technique worked significantly better than other applied approaches according to multiple statistical tests, effect size measures and more informal investigations, the first research question

(***RQ1***) defined in the beginning of this section has been addressed. In addition, according to Table 22, we can conclude that even though in some cases (like UPM) other approaches (such as Additional Statement Converge and Random) have an acceptable performance, RL-based HMM is more successful in detecting difficult-to-find faults. For example in TimeSlotTracker, where the chance of finding the fault is 0.05, RL-based HMM is able to correctly prioritize the fault-revealing test case.

To address the second research question (***RQ2***), we replicated all the experiments and analysis for another proposed technique (Accumulated Q-value). This technique achieves results that demonstrate that this technique has better mean APFD than the other techniques (except RL-based HMM and Optimal), and is significantly better than Random and Additional coverage when applied to the PDFSAM and Buddi applications.

The third question (***RQ3***) investigates the effectiveness of both proposed techniques in comparison to each other. As mentioned earlier, statistical analysis illustrates that the RL-based HMM is significantly more effective than the Accumulated Q-value technique in 3 of the 4 applications, and in the 4th (TimeSlotTracker) application they are not significantly different. The results from the TimeSlotTracker experiment demonstrates a (worst case) situation of when the effect of executing actions (causing further computations) in a specific (GUI) state is similar to the effect of triggering different actions in each state. That is, they contribute the "same amount" of information for making prioritization decisions. The TimeSlotTracker application is a good example of an application, which only has actions, which trigger small amounts of computations. Under these circumstances, the performance difference between the RL-based HMM technique and the Accumulated Q-value approach will be minimalized. However, it is believed that the RL-based HMM technique will always out-perform the Accumulated Q-value approach.

In addition, in order to investigate the effectiveness of the proposed techniques in cases with larger test suite sizes, we ran AutoBlackTest for an extended period and re-applied all the techniques on a test suite now with 60 test cases. The results shows that both of our approaches are still applicable to larger-sized test suites, and that they also prioritize them better than the other state of the art methods.

To answer the fourth question (***RQ4***), we also have considered three GUI-based applications from [164] in order to show the effectiveness of our proposed technique in comparison with other GUI test case prioritization approaches. The results demonstrate that in all the applications, that the RL-based HMM technique is significantly better than untreated and coverage-based prioritizations; while only the fault-prone weight method outperforms coverage based and untreated orderings in terms of fault detection rate for the TrepPaint application. In addition, the RL-based HMM is not significantly different to the Best ordering in all considered applications according to the Relative Ratio measure.

It is believed that these results illustrate that the combination of Reinforcement Learning and Hidden Markov models can be highly successful in prioritizing test cases, this is because this enhanced model considers the test cases from two different and novel perspectives: the tendency of the system to prioritize test cases that, (a) trigger different actions in each state, and (b) executing actions causing more computations in GUI states.

4.8.1 Run-Time Analysis

To investigate the computational complexity of the RL-based HMM prioritization approach, we ran a single instance of this technique on a modest hardware and software platform consisting of a 3.4 GHz CPU, 8 GB RAM on an HP Compaq machine (6200 Pro SFF PC) using the RHMM

package which has been developed using the R language (R is a statistical scripting language[28]) for estimating HMM parameters.

In order to investigate the systems with the largest number of test cases generated by AutoBlackTest and GUITAR, we have considered both the Extended PDFSAM and TerpPaint applications in terms of prioritization run-time. The RL-based HMM technique ran in 12.3 seconds in Extended PDFSAM (8.1s to extract the Q-values and initial values of the HMM parameters, 3.26 for estimating an HMM using the Baum Welch algorithm and less than a second to calculate the forward probabilities). Also, based upon Baum Welch and Forward algorithms' time complexities, the computation order is polynomial $O(N^2T)$, where N represents the number of hidden states and T indicates the number of observations in a sequence; and hence, will not exponentially grow by increasing the number of test cases. The corresponding time for the TerpPaint application is 17.1 seconds. It is worth noting that the parameter extraction techniques used in this study can be optimized in terms of implementation (e.g. anecdotal comparisons often state that algorithms written in R run 1000 times slower than equivalent algorithms in C[29]) to decrease the overall run-time in future studies. However, as the result, it can be concluded that time is needed to prioritize test cases using our approach is trivial compared to the improvements it is providing in early detection of faults.

4.9 Related Work

Applying Markov chain models in software testing processes dates back to 1994, when [10] described a method for statistical software testing. They used Markov chains as a finite state, discrete parameter, and time homogeneous modeling approach to develop and analyze a model-

[28] http://www.r-project.org

[29] http://lists.nongnu.org/archive/html/igraph-help/2011-02/msg00045.html

based testing technique for automatic test case generation. They concluded that Markov chain usage models can be utilized in a diverse set of application domains and are useful during a statistical testing process. Correspondingly, Markov chain usage models have been utilized several times in software testing and reliability research [176]–[178]. Also Bayesian Network models have been applied in software failure detection problems to design software reliability models [179] or in probabilistic test case prioritization to incorporate source code changes and test coverage data into a unified model [180], [181]. Moreover, web application testing is another interesting area for applying Markov chain models. [182] considered the effectiveness of Unified Markov Models (UMM) as a suitable testing mechanism through two empirical studies. They confirmed that UMMs are appropriate methods to test web applications accurately. [183] present a controlled Markov chains (CMC) approach to software testing. They considered software testing as a control problem, where the software under test serves as a controlled object. [18] also presented an MBT method to generate test cases to perform load testing for any software system that is "model-able" by a Markov chain. Their algorithm was successful in preserving faults that would have been likely to be missed in traditional load testing methods. The purpose of the authors in designing this technique was not "test case prioritization". However, their thoughts are statistically close to what is presented in this research.

One of the most critical parameters in using Markov chain models in the MBT area is in estimating transition probabilities. Many researches provided diverse techniques to estimate Markov chain parameters, however, [184] proposed a novel technique to target coverage criteria rather than using a classical uniform probability generation approach. It would be an applicable technique in creating a Markov chain with appropriate initial parameters leading to a global optimized model.

As mentioned earlier, the application of HMM models to pattern recognition and bioinformatics problems, has been well-established. On the other hand, limited numbers of papers have been published on Hidden Markov Models (HMM) applied to software testing; however, it has been used for modeling computer security problems and intrusion detection in web applications. HMMpayl is a combination of HMM usage and multiple classifier systems that has been reported as an effective tool against the most frequent attacks to web applications [185].

In addition, combining HMM and Reinforcement learning concepts to re-estimate and improve the model has been considered in several artificial intelligent problems such as robots' motion prediction [74], speech recognition [75] and natural language generation [76]. Also [77] suggests a method to handle RL algorithms in partially observable Markov Decision Problems. [78] provides a comprehensive study on RL with hidden states. He also represents four new RL algorithms for environments with hidden crucial states.

One of the reasons, which initially motivated the ideas beyond the design of an RL-based HMM test case prioritization technique, was the existence of prioritization approaches, which are only applicable for non-GUI testing. [139] describe several techniques for using information gathered from the test case generation phase to prioritize test cases in regression testing. They measured the fault detection rate of each method and found their proposed techniques are able to improve this rate. Furthermore, [186] present a novel test suite prioritization technique, which concentrates on test cases' fault detecting ability. They used a fault model, which generates test cases to guide specification-based testing. Results confirm the effectiveness of this approach in logical expression testing, but there is no evidence of it being applicable to the testing of state-based models, which are very applicable in testing interactive applications. In addition, [187] consider the application of test case prioritization in regression testing procedure from a time-

constraint perspective. The authors believe that the time constraints, which can be imposed on regression testing, can strongly affect the behavior of the prioritization techniques. They also provide helpful suggestions about determining the appropriate situation for prioritizing test case based upon the system's constraints.

Finally, [164] have proposed a method for adding weights to event flow graphs (a very primitive version of a regular digraph). They use the GUITAR tool to generate test cases and a behavioral model to which they add weights to the transitions of this graph. This results in a weighted graph similar to the graph produced by AutoBlackTest. While AutoBlackTest dynamically analyses the SUT to produce system-specific weights, [164] simply use static weights derived from heuristics, which have very limited theoretical underpinnings. This implies that the digraph produce by AutoBlackTest is clearly superior to the digraph produced by [164]. And hence, we use the output of AutoBlackTest, rather than that of [164] as the baseline for analyzing test case prioritization techniques in interaction-driven applications.

4.10 Threats to Validity

In this section, some potential threats to validity of our research are discussed. We consider two types of threats:

- Threats to internal validity, which are cause-effect relationships between independent and dependent variables.
- Threats to external validity, which are limitations about generalizations from our study, or the threats to transform the research from the specific case study to general subjects.

With respect to internal validity, the most significant threats are (a) the four different models (one per application) generating diverse levels of forward probabilities; and (b) the size of the test suites derived by AutoBlackTest from these applications. This first threat can be classified as an instrumentation effect. This can add bias into the results of the APFD measures. To reduce the likelihood of this effect, we have normalized the Q-values and APFD measure to be within the range [0, 1], thus minimizing any impact in the values of the transition matrices and the APFD measure. The second threat represents the most significant issue. Given the choice of the domain of application – extending the automatic generation of (black-box) test cases for GUI-based applications – no obvious approach exists to minimize this threat. AutoBlackTest produces superior results when compared to other systems (such as GUITAR) in this domain of application. However, the volume of test cases generated by it can be viewed as "rather small" leading to prioritization problems of "limited extent". In fact, AutoBlackTest explicitly discards tests which it considers to be of no value which compounds this "size of test suite" issue further. Hence, further research is required in producing more complete regular digraphs and extended digraphs models to advance the topic further.

On the other hand, the threats to external validity for our research are centered on the selected applications as representative of any possible application. To address this issue, we evaluate our method on four different GUI-based applications. As mentioned before, any application with a weighted or a non-weighted behavioral model can be utilized, but because results show that AutoBlackTest is a successful model–based GUI testing tool, we have concentrated on working in this area. Also by evaluating the proposed method on known and pre-evaluated applications, we can compare our results and illustrate the effectiveness of the RL-based HMM technique.

4.11 Conclusion

Many studies have investigated automatic test case generation, test case prioritization and GUI testing, but few of them specifically focused on the fault based test case prioritization issue in GUIs.

In this research, we propose a novel fault-based technique, the extended digraph. It is argued that the extended digraph provides a richer explanation of program behavior than a regular digraph. This digraph is generated by using an RL-based HMM approach to prioritize test cases. We present an approach to initialize an appropriate HMM based on a Q-learning algorithm that leads to a final HMM with the maximum likelihood estimate of parameters after applying an EM algorithm. Then we use the estimated model to compute the likelihood (forward probability) of the generated test cases. Finally, we presented a new definition for test case prioritization based upon these forward probabilities.

In addition, we propose another technique, which uses the summation of each test case's Q value in order to sort them in a descending order. Thus, test cases with the higher amount of accumulated Q-value than others get higher priority in the sorted list.

To evaluate the proposed methods, we designed an experimental setup using AutoBlackTest as a test case generator tool. Random, Additional statement coverage, Worst and Optimal prioritization are also included in order to provide a comprehensive comparison. We applied all of the prioritization techniques on four GUI applications to evaluate the effectiveness of our proposed approach and address the first three research questions. In addition, as an extra experiment, we have compared the RL-based HMM technique with the weight-based prioritization approach presented by [164].

According to the APFD measure, Relative Ratio, boxplots, statistical tests and effect size estimates, RL-based HMM outperforms the other approaches in terms of fault detection effectiveness. It illustrates that considering GUI states and actions are playing an important role in improving the fault detection rate. Because the RL-based HMM approach combines RL and HMM concepts, it is able to sort a test suite by prioritizing test cases with special focus in two important aspects:

- The amount of computations (changes), a test case may cause in GUI states (using Q-leaning) and,
- The probability of triggering each action in each specific state, plus the amount of computations (changes), a test case may cause in GUI states (using HMM).

5 Automated Testing of Motion-based Events in Mobile Application

5.1 Introduction

In recent years, mobile devices have been produced in various types and shapes, offering a wide range of services and features. It is a very difficult task to develop mobile applications that are able to work appropriately on different mobile devices and operating systems (OSs) [188], [189]. On the other hand, releasing applications that are not fully functional, usable, and consistent can risk the developer's reputation in such a competitive environment. Testing the application's functionality and verifying its robustness are key factors in improving the application's quality.

Embedding new hardware devices, such as movement sensors (accelerometers and gyroscopes), in smartphones and tablets further complicates the testing procedure. Users are able to interact with the application by touching, tilting, shaking, and rotating the mobile devices. When a device is in motion or it's screen is continuously touched, the probability of unintentional inputs increases; in such circumstances, automatically generated test suites are needed to produce accurate test cases and accelerate the mobile application testing procedure.

Some tools and techniques have been developed to test the quality of the source code for mobile applications[30] [190]–[192], but the number of approaches that focus on automated testing is still very limited. The majority of these automated testing tools offer capture-and-replay functionality to test the application's User Interface (UI) [14], [193], [194]. For instance, Choudhary et al. [192] have conducted a study on existing testing tools for Android applications. Although the study dealt with techniques for testing the mobile applications, it doesn't provide any insight or

[30] Android: http://developer.android.com/tools/help/lint.html ; iPhone: http://clang-analyzer.llvm.org/

mechanism for generating test cases for motion-based mobile applications. In addition to this, the case studies considered in [192] do not have any motion-based facilities, and hence are not suitable to be utilized in this study.

Writing and continually improving motion-based test cases is a difficult task when testing mobile applications that use movement-sensor data. Therefore, considering existing mobile testing tools and approaches, two problems can be noticed: 1) no automated approach is provided (this problem is considered as a technical challenge in this study); and 2) generating test cases for motion-based mobile applications remains unconsidered (this problem is considered as a scientific challenge in this study). Thus, in this chapter of the thesis, we propose a new approach to address these limitations. It is argued that mimicking users' behaviours is one of the key factors in generating gesture-based test cases. It helps in executing realistic test scenarios and standard gestures [195], [196].

In order to automatically generate test cases, mimicking human generated gestures in motion-based mobile applications, we propose a novel approach, which synthesizes the motions, and subsequently, simulates the test cases based upon the formalized gestures. Motion data is represented by the data captured, using the movement sensors and the objects' positions (2D coordinates) on the screen. An application can then use the sequences of motions to simulate the gestures and test the UI. To increase the chance of generating realistic movements, a set of training data is generated by human users and is used to train the hidden Markov model (HMM) classifiers. The models are iteratively used to generate new motion sequences for the application testing procedure. Gestures and animations are commonly considered to be the key components in modern mobile user interface design; hence this work directly targets the heart of the matter in this new and evolving application domain.

Our experiments on sample Android applications that support motion-based gestures reveal the effectiveness of the proposed approach as an automated testing process. Although, the focus of the empirical evaluation of the proposed approach is on Android applications, it is worth noting that the algorithm is also applicable to motion-based iOS applications and applications found on other mobile platforms.

In summary, the generated motions are used to automatically produce test cases, mimicking human-generated gestures with the technical goal of increasing code coverage. Therefore, the process is highly beneficial during regression testing, since the generated test cases can later be executed on newer versions of the application to uncover issues in the system.

This study contributes to the research in this area by:

1. Proposing a new approach to synthesize motion data, and make it executable as a test input to the application being tested.
2. Applying a HMM classifier on the training data to create a set of HMMs, and subsequently using them to generate motion sequences.
3. Evaluating the effectiveness of the proposed approach in terms of, (1) mimicking the user's behaviour, and (2) increasing the code coverage of the software under test (SUT).

This chapter is organized as follows. Section 5.2 provides related work and background information and definitions relating to mobile applications, particularly motion-based gesture testing. Section 5.3 describes an overview of the proposed approach, the gesture synthesis and simulation procedures, while Section 5.4 provides the design and implementation details of the proposed technique. Section 5.5 provides a running example of the proposed test case generation approach using real data. Section 5.6 discusses the evaluation phase, experimental setup, and

results. Section 5.7 explains the experiment's run-time analysis. Section 5.8 examines the study's limitations and the threats to its validity. Finally, Section 5.9 presents the overall conclusions and some thoughts on potential future research.

5.2 Related Work

5.2.1 Mobile Application Testing

Testing is a crucial activity in a software development procedure. Producing a defect-free application, addressing all of the requirements of the users, along with providing fully functional, consistent, and highly usable services are vital in highly competitive environments. Over the past few years, phenomenal progress in the mobile device market has led to an outstanding growth in the mobile application development industry. However, the growth in developing mobile testing procedures and techniques has been insufficient. Although many testing methods and tools exist for desktop and server/host software, most of them are not applicable for testing "mobile software" [197]. Moreover, most existing test generation techniques rely on a crawler to explore the dynamic states of the application under test. Such approaches are automated and systematic but lack the domain knowledge of system experts. Far et al. [198] propose a new technique to combine the human knowledge present in the form of input values with the inferred knowledge of automated crawling. Similar approaches have not been applied in testing mobile applications. Hence, in this study, we propose an approach, which relies on both human and automated exploration data. Ermuth et al. [199] also presents a UI-level test case generation technique that applies human-produced execution traces in order to automatically create complex sequences of events that are able to cover more pages, scenarios and code lines compared to a purely random test generator. This approach relies on inferring sequences of low-level UI events (macro events)

using data mining techniques and the inference of finite state machines (FSMs). However, Ermuth's technique [199] also has never used to test mobile applications.

Although many traditional testing tasks are common between mobile applications and the desktop/web-based applications, several key factors cause challenges in the mobile testing procedure. For example, the variety of mobile devices and diversity in OSs cause difficulties in testing device-specific factors [195]. Mobile devices are different in terms of screen sizes, platforms, input methods, and the quality of the sensor data. Such differences can easily multiply testing efforts. For testing an application, it needs to be exposed to a sufficient number of devices from different models, screen sizes, and OS versions. Covering an adequate number of factors leads to generating a large number of test cases that are required to be executed in an environment where short release cycles are common. This can easily affect the quality of the application, along with the time of the marketplace and the costs of construction. Integrating automation approaches with test case generation procedures is a key factor in addressing these issues in the "mobile testing era", where many test cases need to be executed on a large selection of mobile devices and configurations to reproduce defects.

In this regard, [200] presents a framework to test the functionality of mobile applications when a device is moved to a new network. The framework uses an application-level emulator as a mobile agent to carry the application across networks to ease the testing process under different network technologies. Additionally, [201] suggests a quality assurance framework to define key patterns and metrics in mobile application testing. Although these research studies provide insights into the testing of the mobile applications, they still do not cover the test case generation phase. Several studies with a special focus on automated testing for mobile applications have also been conducted; [202]–[206] suggest different, automated, graphical user interface (GUI)

testing approaches for Android applications. For example, [202] produces *AndroidRipper*; this tool seeks to explore the application's GUI and evaluate its effectiveness in terms of fault detection ability when compared to random approaches. Android Monkey[31] generates purely random tests for Android mobile applications using a brute-force mechanism. Android Monkey usually achieves shallow code coverage compared to other state of the art approaches used for testing GUI in Android applications [207]–[209]. Mao et al. [210] also proposes another framework which combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation in order to automatically explore and optimise test sequences in Android applications. Moreover, [211] presents a new approach to automatically generate test oracles for testing user-interaction features found in mobile applications. Given a model of the mobile application's UI, this framework uses a library of oracles and generates a test suite to test the user-interaction features in the application. There also has been some work [212] that uses contextual information to randomly generate inputs to test mobile applications and automatically find crashes. Such approaches are more focused on discovering; reporting and reproducing crashes are not practically used to generate functional test cases covering the source code.

Although, some automated test case generation techniques are suggested for testing the UI of mobile applications, but their functionality and applicability in testing the new features of today's mobile phones are far from perfect. To test the UI, the mobile application needs to be executed with user interaction events. With technological advancement in smartphones and tablets, natural user interfaces (NUIs), which no longer use keyboards and keypads as human-machine interfaces, have become popular. Touch-sensitive screens, speech recognizers, and

[31] https://developer.android.com/studio/test/index.html

gesture detectors are the primary interaction channels in the new generation of mobile applications. This era of application testing is relatively new, and only a limited number of studies have been performed to address these testing challenges [195], [196].

5.2.2 Testing Motion-based Gestures

Mobile applications, which allow users to control the applications' functionality through NUIs, normally recognize gestures by using the data provided by the embedded sensors in the mobile device [213]. Several smartphones and tablets contain accelerometers to control motion inputs. One of the most common applications of accelerometers is presenting the landscape and portrait views of the screen based on the way the device is being held [214]. The 3-axis model of the accelerometer is able to measure the magnitude and direction of the acceleration (gravitational force) as a vector $[ax_k, ay_k, az_k]$ for a motion k in a 3D space. Each acceleration parameter measures changes in velocity over time along a linear path. Combining all three accelerations, lets the application detect the device's movement in any direction and obtain the device's current orientation. Depending on the graphical capabilities of mobile applications, 2D or 3D versions of the acceleration vector are considered. Obviously, 2D applications do not use $az_k$to indicate a motion k. From the tester's perspective, testing applications that support motion-based events introduce a new complexity to the testing procedure; motion-based gestures should be accurately specified and reliably reproduced [195]. The lack of formal motion-gesture specification prevents testers from developing an automated test generation approach. To simulate the motions, atomic gestures should be formalized. The next section presents the simulation and synthesis procedures of motion-based events (gestures).

5.3 Gesture Simulation

In the simplest test-case generation process, the test data-points can be provided to the application by using a random test generation approach, which randomly creates data frames within a defined range to move the object on the screen. It can be expected that the number of reasonable gestures, which are created randomly, are very limited. Therefore, even if these test cases are able to cover an acceptable number of branches in the source code, they may not be able to reveal faults a human user can discover simply because they cannot replicate standard gestures [196].

This study considers an automated test case generation procedure for mobile applications interacting with users using motion-based events. However, it is not limited to the applications only supporting motion-based events and can be applied on the application covering both types of inputs (motion-based and non-motion based). In such applications, users normally interact with these types of applications by performing a sequence of gestures, e.g. by moving a flying or bouncing object on the screen or drawing geometrical shapes by touching the screen. In other words, user-generated gestures are transferred to the object or touched location to move the object toward the desired direction or to draw a geometrical shape (e.g. circle) around the touched point on the screen. It is noteworthy that motion-based events are not only used to move an object on the screen; sometimes, shaking a mobile phone in a specific direction or touching and dragging the screen leads to executing a function or opening another application [215]. This study focuses on the procedure to automatically generate test motions on both types of applications (1) applications only supporting the data generated using accelerometer sensors; and (2) applications supporting both the data generated using the accelerometer sensors and the data generated using other types of events such as those produced by touching the sensitive screen. In

such cases, several parameters can affect a single event (such as the object size, the size of the screen, an object's location, etc.).

Since users are free to touch, move and shake their mobile phones in any desirable direction and speed, a testing approach must be able to generate sets of standard gestures, which are not only executable on the application but also resemble the human-generated motions. Therefore, to automatically generate more reasonable gestures – mimicking human users – this research proposes a novel approach. It is hypothesized that this mimicking may also result in an increased level of code coverage of the SUT. The correctness of this assumption is examined in the empirical evaluation section (Section 5.6).

The proposed technique contains several steps and details, which are depicted in the framework provided in Figure 23. This figure shows the schematic overview of our proposed approach for a complex application containing acceleration parameters moving an object (bouncing ball) on the screen in different directions (as an example). This framework can easily be adjusted for any applications supporting motion-based events. The proposed approach consists of the following sequential steps:

- Gathering training data: A human user is asked to interact with the application and generate motions to be used as a training set. (It is worth noting that the person is not trained or instructed to generate any specific types of motions from the applications and the generated motions are the result of a volunteer interacting with the application for the first time. This prevents the data-gathering phase from collecting biased data.)
- Clustering motions: k-means clustering algorithm as a classic clustering technique is used to identify the relationship between data points (motions) generated by human users, and

to determine the cluster of behaviours that they belong to. It is well known that data clustering is a successful approach in recognizing and categorizing human expressions, gestures and actions [196], [216], [217]. More specifically, in this study, the motion parameters are partitioned into k clusters, such that each motion is allocated to the cluster with the nearest mean. The clustered data later will be used to train an initial model of the gestures.

- Training Initial HMM: In order to produce the first standard test gesture, an initial HMM is trained using the human-generated motions and their corresponding clusters. Hidden Markov models are well known for their application in pattern recognition such as speech, handwriting and gesture recognition. As we utilize time-varying motion sequences, HMMs can be used to model and learn human skills such as reasonable interactions with mobile applications [218], [219]. Basically, the initial HMM trains a model, where its hidden states indicate motions' clusters, generated in the first step. Training the model using the expectation-maximization (EM) algorithm, the probability of a gesture belonging to a specific cluster (state) is estimated and used to calculate the first motion acceleration parameters. The first motion's acceleration is calculated by computing the mean of the accelerations in each HMM state and by selecting one pair randomly. Using this approach, we can assure that the whole test generation procedure - including the initial motion - is produced through the models trained from the user-generated data, so they potentially mimic human generated gestures.
- Generating the test data using HMM classifiers: In this step, we apply HMM classifiers on clustered data to generate test motions using the previously produced gestures. HMM classifiers are successfully used in several studies, considering the prediction of human

activities and gestures [220]–[224]. For each cluster, the dynamics of each motion class is learned with one HMM. Thus, having m motion-clusters, m HMM classifiers need to be applied. HMM classifiers classify each motion as a function of a future time frame [225]. Thus, the probability of a test case belonging to each cluster is calculated using the well-known Forward algorithm [19]. The motion-cluster with highest Forward probability is selected and the mean of the acceleration of the motions belong to this cluster is considered as the next motion's acceleration.

- Adding generated motions to the training set: In order to avoid over-fitting the model, the generated motions should be added to the training set. This helps the model to learn from the data rather than memorizing the trend.
- Storing and Executing test cases: Once, for example, the ball hits the vertical wall (or a terminal condition happens), sets of test motions generated since the last hit are stored as test cases and will be used to generate real motions in the mobile applications. Terminal conditions can be defined generally or per application. For example, a general terminal condition can happen once a specific number of test motions are generated, while a customized terminal condition happens when a flying object (if applicable) hit the edges. In this study, we considered the customized option for the stop criteria.

It is worth noting that this framework provides an overview of the proposed approach. The implementation details of this framework are discussed in Section 5.4.

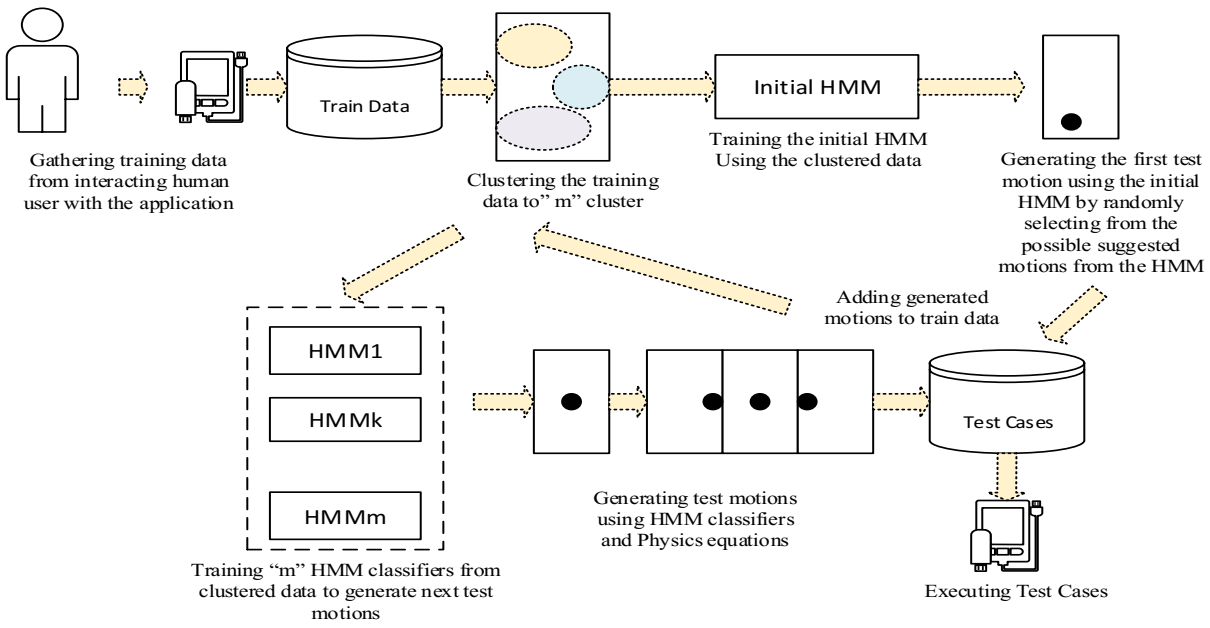


Figure 23. An overview of applying the proposed approach on the application with flying object. It consists of both training the initial HMM (top) and test generation process using HMM classifiers (bottom)

5.3.1 Synthesizing Motion Sequences

This section describes the method of instantiating the motion sequences for complicated motion-based applications, which transfer the users' gestures to a bouncing object. However, the application of this approach is not limited to events using sensor-generated data; it can be easily used to generate automated test cases for any type of motion-based events. Following the previous section, two sets of data (motion sequences) are considered in this study:

- The training data, which is captured during a real user's interaction with the application and is used to train the initial HMMs.
- The second set is the test data, which is generated by using the test generation algorithm and is presented to the application being tested to evaluate its functionality. To create meaningful test data, which is recognizable by the trained HMM and its corresponding

classifier, we describe a single motion k by a 6-tuple $(lx_k, ly_k, vx_k, vy_k, ax_k, ay_k)$, where lx_k, ly_k indicates the object's location, vx_k, vy_k determine the velocity, and ax_k, ay_k describe the acceleration of the motion in 2D space at a specific time interval. Figure 24(a) shows the 3D acceleration axes on a smartphone, which also contains a z-axis. In order to simplify the explanation of the algorithm and cover more common applications, only 2D applications have been considered in this study. However, it is worth noting that it is possible to apply the same algorithm in 3D versions as well.

An example of a single motion in a bouncing ball application is provided below:

05-07 17:36:15.828: Vx(32065): -2.7148619

05-07 17:36:15.828: Vy(32065): -2.7148619

05-07 17:36:15.828: lBallX(32065): 549.0

05-07 17:36:15.828: lBallY(32065): 20.0

05-07 17:36:15.828: Ax(32065): 0.090979666

05-07 17:36:15.828: Ay(32065): -0.12330139

This can be presented in a 6-tuple format (the data is rounded for the sake of clarity): (549,20, -2.714, -2.714, 0.09, -0.123).

This study also considers two time intervals during the test generation procedure:

- The first time interval constantly happens every φ milliseconds [193] to capture the information regarding the current motion and position of the object on the screen and to calculate the next motion using the well-known SUVAT equations [215], [226].

- The second time interval happens every θ milliseconds, which is estimated by selecting the minimum possible time between two gestures, generated by human users. (This time can vary with the complexity of the gestures in different applications). Hence, the estimation of θ assists the algorithm to generate more realistic (complex) gestures as it accounts for the limitations of kinematics.

  It is worth noting these time intervals can overlap in the sense that in the time window between two θ intervals, φ interval may happen when $\theta > \varphi$.

Figure 24(b) shows an atomic gesture consisting of a sequence of motions happening within these two intervals. Each sequence of motions is terminated by the occurrence of a specific condition in the application being tested, depending on the application's objectives and functionalities. For example, a simple terminal condition can happen when the flying object hits another object (such as the edges of the screen or another flying object) on the screen.

Additionally, in the following paragraph, some of the SUVAT equations (equation of motions), which are useful in calculating the coordinates of the motions are provided:

- $v = at + v_0$
- $l = l_0 + v_0 t + \frac{1}{2} a t^2$
- $l = l_0 + \frac{1}{2}(v_0 + v)t$
- $l = l_0 + vt - \frac{1}{2} a t^2$
- $v^2 = v_0^2 + 2a(l - l_0)$

where: l_0 is the object's initial position

l is the object's final position

v_0 is the object's initial velocity

v is the object's final velocity

a is the object's acceleration

t is the time

Definition 22. A test case (TC) consists of a set of motions $(M = \{m_1, \dots, m_n\})$,where $m_{k \leq n}$ is a 6-tuple $(lx_k, ly_k, vx_k, vy_k, ax_k, ay_k)$. The number of tuples (motions) in each TC depends on the number of detectable motions before the termination situation happens.

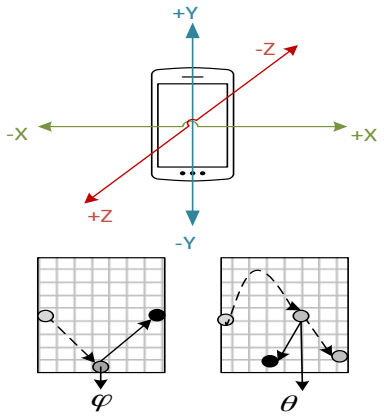


Figure 24. (a) 3D acceleration axes on smartphones; and (b) an atomic gesture containing a sequence of motions happening within two intervals: (left) a bouncing object keeps moving in the screen after hitting the edge in first time-interval φ; (right) the proposed approach calculates the next movement after the second time-interval θ happens

5.4 HMM-Based Test Case Generation

Human activity recognition and classification has been studied using several different machine learning approaches such as multi-class support vector machines (SVM) [227], k-Nearest Neighbor (k-NN) [228], Neural Networks (NN) and HMM-based approaches, but in cases that the activity sequences are time-varying, HMM based approaches have produced better performance and results [218], [229], [230]. In addition, the Markovian process had been used in several motion detection-related studies to create statistical models from clustered data [196]. Following these studies, we also cluster our training data by using the *k*-means algorithm [231] to identify the data points (motions) containing related gestures and to assign them to the same clusters (the number of clusters (*k*) is selected by using the silhouette score [232]).

In other words, the clustering algorithm is applied to groups of motions with similar behaviour and allocates them into a single cluster. These clusters will be used as the class labels for the HMM classifiers. This means that each class indicates a set of similar motions in the corresponding cluster. Therefore, a motion, which belongs to a class during the classification process, also shares similar characteristics with the motions in their corresponding cluster. It is also worth noting that since the motions' clusters, detected by the clustering algorithm, play the role of class labels in the proposed HMM classification procedure; we use the term of class label instead of cluster to avoid unwanted ambiguities.

Consequently, the clustered data will be used to train an initial Hidden Markov Model. Since an HMM is a Markovian process that contains two sets of states (the observable and the hidden [latent] states), only the motion sequences, depending on the latent states (motions' clusters or classes), are visible in such a model. Therefore, as the classes are invisible from an *observer's*

view, only the motions in this model are completely observable, an HMM can create a more powerful model compared to regular Markov models or partially observable Markov decision processes (POMDP) [26].

The HMM in this study is characterized by the following elements [19]:

- a set of latent states $S = \{s_1, s_2, \dots, s_L\}$, which are hidden from the external observer and indicates the class of motion sequences;
- a set of observable states $V = \{v_1, v_2, \dots, v_N\}$, where each is mapped to a corresponding motion sequence (m_k);
- a transition probability $[A]_{ij} = \{a_{ij}\}$, $a_{ij} = P\left(Q_{t+1} = s_j \middle| Q_t = s_j\right), 1 \leq \text{i,j} \leq \text{L}$, which determines the transition probability between different classes. For the initial modelling process, because human users generate the motions, the initial transition probabilities between different classes of motions can be extracted directly from the training data;
- an emission probability $[B]_{jk} = \{b_{j\,(vk)}\}$, $b_j\,(v_k) = P\left(M_t = v_k \middle| Q_t = s_j\right), 1 \leq \text{j} \leq \text{L}, 1 \leq \text{k} \leq \text{N}$, which indicates the probability of a motion sequence belonging to a specific class (estimated by frequency counting on the clustered training corpus); and
- initial state distribution, $\Pi = \{\pi_i\}$, $\pi_i = P(Q_1 = s_i), 1 \leq \text{i} \leq \text{L}$. Each and every state can be an initial state in this study.

Using the values of A, B, and Π, an HMM can be used as a generator to create an observation sequence (where T is the number of motions in the test case): $M = \{M_1, M_2, M_3, \dots, M_T\}$. We use the notation Λ = (A, B, Π) to simply indicate the complete parameter set of the HMM with respect to the Markovian process, which illustrates that the probability of a motion's occurrence only depends on the previous motion:

$$P(s_{t+1}|s_t, s_{t-1}, s_{t-2}, \dots) = P(s_{t+1}|s_t) \qquad (43)$$

This initial HMM model is used as an input to an expectation-maximization (EM) algorithm; specifically, we utilize the Baum-Welch algorithm in this study [27]. This algorithm estimates the optimal model with the highest likelihood of the estimated parameters. In Table 39, this procedure is done by running the HMM function in the second line. Then, the initialAccel function initializes, the acceleration parameters of the first test motion by calculating the mean of the acceleration pairs (i.e. (ax, ay) in 2D space) in each HMM state and by selecting one pair randomly. Then, in lines three and four of this algorithm, the CreateMotion function is generating a motion sequence using the SUVAT equations and the Update function is storing the newly created motion sequence as the current motion. After generating the initial motion, the CreateMotion and Update functions are called again but this time within the time interval φ, until a termination condition happens (line 5-10). This procedure generates a simple gesture based upon the previous motion, using appropriate physics equations. In order to generate more realistic and complicated gestures, we propose using the HMM classifier to detect the sequence class label at each interval θ [18], [28], [29].

The HMMClassifier function in line 12 of the algorithm classifies the current motion sequence into an appropriate class of gestures. This function combines a set of sequences of motions and a list of class labels to train one HMM per class label (where L is the number of class labels). Subsequently, the trained models are used to calculate the forward probability of a motion sequence M per model $\Lambda_{i\leq L}$ ($P(M|\Lambda_i)$). $P(M|\Lambda_i)$ is calculated using the Forward algorithm, which is internally called during the execution of the HMMClassifier function. The forward algorithm computes the forward probability,$\alpha_k(t)$, as the joint probability of observing the first *t*

vectors $m_t, T = 1, \dots, t$ while in state k at time t. Another way to state this would be that $\alpha_k(t) = P(m_1, m_2, \dots, m_t, s_t = k|\Lambda)$ which is the probability of observing $(m_1, m_2, \dots, m_t)$, assuming that the system is in state k at time t. Given a list of forward probabilities for a motion sequence M, we are able to easily detect a model with the maximum probability and assign its corresponding class label as the motion's class label [19]. Determining the class label of a motion sequence allows us to easily detect the motion sequences belonging to the same class from the training data set, and estimate the next motions values by calculating the mean of the accelerations of the motions (the Accel function in line 12). Moreover, the generated motion is added to the training set to avoid over-fitting. This helps the model to learn from the data rather than memorizing the trend (lines 10 and 16). It is worth noting that in this study, we also use the term of "occurrence likelihood" to refer to the forward probability.

Putting it all together, lines five to seventeen of Table 39 create a set of motion sequences within two different intervals. Simple gestures are generated based on physics equations once the first time-interval happens. But, the more complicated motions (e.g. gestures with variable accelerations) that may require a longer time period to be created by a human user are generated within the second time interval. This process provides sufficient duration to allow the method to generate more complex gestures. An example of a simple motion is the one calculated by the SUVAT equations after the bouncing ball hitting the horizontal wall. While the complex one is a motion calculated by HMM classifiers for a ball slowly bouncing in the middle of the screen. In reality, when the ball is slowly moving in the screen, the human user can change the direction of movement by shaking the phone in several different directions. Therefore in an automated test generation process, a trained model is needed to predict the most probable acceleration of the

gesture from the last motion's parameters. In this study, the motion generation is stopped and a test case is generated once a terminal condition (e.g. hitting the vertical wall) happens.

When the application under test is motion-based applications with no acceleration parameter involved, lines 18-30 of this algorithm will be applicable. In such situations, the first motion can be created by randomly selecting a touched-point in the screen. In a 2D space, the motion sequence only contains the coordinates of the touched-point(x, y). Similar to the algorithm provided in Table 39, within different intervals φ and θ, random touch points are generated, or the HMMClassifier function predicts the next motion class-label and the Position function returns the position of a touched point by calculating the mean of the position pairs in the predicted class. Depending on the application design's objectives, the position of a touched-point can be used to draw a shape or render functionality such as vibrating the phone or opening a dialogue box. In this algorithm, in order to focus more on the second case study, we consider creating a geometrical shape (e.g. circle) with the touched-point coordinates as its center (CreateShape function). Additionally, when an application covers non-gyroscopic events such as clicking a button or choosing an item from the menu, RandomEvent function randomly generate an event, executable within the current state of the application.

5.5 Running Example

In order to clarify the proposed test case generation procedure, we considered a very small portion of the training data generated by a human user in the bouncing ball application. An example of a single motion is provided below:

05-07 17:36:15.828: Vx(32065): -2.7148619

05-07 17:36:15.828: Vy(32065): -2.7148619

05-07 17:36:15.828: lBallX(32065): 549.0

05-07 17:36:15.828: lBallY(32065): 20.0

05-07 17:36:15.828: Ax(32065): 0.090979666

05-07 17:36:15.828: Ay(32065): -0.12330139

Table 39. Test case generation procedure for cases with acceleration involved

Input: Initial position of the bouncing object (x,y), training data set (S), set of class labels (C); i = 2;
Output: Test case (TC)

1. ***if*** $(Accel\_MotionEvent)\{$
2. (ax,ay)⟵ initialAccel(*HMM(S,C)*)
3. $m_1 \leftarrow$CreateMotion(ax,ay,x,y)
4. Update(ax,ay,x,y)
5. **While** *(!terminalCondition)*
6. ***if*** $(curTime - lastUpdate1) \geq \varphi)$
7. $i \leftarrow i + 1$
8. $m_i \leftarrow$CreateMotion(ax,ay,x,y)
9. Update(ax,ay,x,y)
10. $S \leftarrow S \cup \{m_i\}$
11. ***if*** $(curTime - lastUpdate2) \geq \theta)$
12. (ax,ay)⟵Accel(*HMMClassifier(m_i,S,C)*)
13. $i \leftarrow i + 1$
14. $m_i \leftarrow$ CreateMotion(ax,ay,x,y)
15. Update(ax,ay,x,y)
16. $S \leftarrow S \cup \{m_i\}$
17. **End while**
18. ***if*** $(NonAccel\_MotionEvent)\{$
19. **While** *(!terminalCondition)*
20. ***If*** $(curTime - lastUpdate1) \geq \varphi)$
21. $i \leftarrow i + 1$
22. $m_i \leftarrow$RandomPosition (x,y)
23. CreateShape(x,y)
24. ***If*** $(curTime - lastUpdate2) \geq \theta)$
25. $i \leftarrow i + 1$
26. $m_i \leftarrow$Position (*HMMClassifier(m_{i-1},S,C)*)
27. $S \leftarrow S \cup \{m_i\}$
28. CreateShape(x,y)
29. **End while**
30. }
31. ***else***{
32. $t_j \leftarrow$RandomEvent();
33. $j \leftarrow j + 1$
34. }
35. **Return** $TC \leftarrow \{m_1, \dots, m_i\} + \{t_1, \dots, t_j\}$

*lastUpdate1 indicates the last update that happened at interval φ while lastUpdate2 indicates the last update that happened at interval θ

Table 40. Training motions clustered in two distinct clusters

Cluster 1	**Cluster 2**
(211.362, 502, 9.787, 9.787, -0.306, 7.948)	(20.0, 344.450, 24.511, 24.511, -4.563, -3.260)
(220.376, 502, 9.259, 9.259, 0.550, 7.753)	(20.0, 45.517, 8.898, 8.898, -6.545, 6.408)
(229.642, 502, 8.681, 8.681, -0.550, 7.753)	(26.236, 20.0, 3.899, 3.899, 11.504, 5.465)
(245.160, 502, 7.443, 7.443, -0.835, 7.907)	(20.0, 182.771, 23.407, 23.407, 8.195, -7.834)
(252.285, 502, 6.566, 6.566, -0.835, 7.907)	(20.0, 235.211, 19.966, 19.966, -8.742, 0.182)
(270.067, 502, 2.694, 2.694, -1.039, 7.953)	(300.0, 118.057, -28.868, -28.868, -8.030, -4.404)
(272.012, 502, 1.572, 1.572, -1.067, 7.899)	(300.0, 367.611, -36.233, -36.233, 8.330, 1.120)
(271.131, 502, -1.667, -1.667, -1.170, 7.818)	(20.0, 378.444, 28.222, 28.222, -2.802, -0.751)
…	…

In this running example, we follow the test generation framework (Figure 23) provided in Section 5.3 step by step to generate test cases:

- Gathering training data: 30 motions in the format of 6-tuple $(lx_k, ly_k, vx_k, vy_k, ax_k, ay_k)$ are gathered as the result of user interaction with the application.
- Clustering motions: the training data is clustered into 2 distinct clusters (classes) using the k-means algorithm. Due the space limitations a partial view of the clusters are provided in Table 40.
- Initial HMM training: the clustered data is then used to train the initial HMM using Baum Welch algorithm. In this case, the HMM model contains 30 observable states and 2 hidden states (since there are only two clusters). Then, the acceleration parameter of the first test data motion is generated by calculating the mean of the acceleration pairs of the motions belonging to each hidden state of the initial HMM and subsequently selecting

one pair randomly. After determining the initial acceleration parameter, the first motion is created using this parameter and the SUVAT equations. In this case, given:

the (1) initial acceleration parameter:

$$(ax, ay) = (0.59855044, -0.91578215)$$

(2) the initial location of the ball in the screen:

$$(lx_0, ly_0) = (309,253)$$

and (3) knowing that the initial velocity is equal to zero (ball is not moving at the beginning):

$$(vx_0, vy_0) = (0,0)$$

motion

$m_1(309.080798,252.876369,0.1755132, -0.2747346,0.059855044, -0.091578215)$

is generated using physics equations: $v = at + v_0$ and $l = l_0 + v_0 t + \frac{1}{2} at^2$.

Then within the time interval $\varphi = 300ms$ other motions are also generated through the same process with the difference that the acceleration of the current motion is used as the initial acceleration for the next ones. These motions will be added to the training set to avoid over-fitting. (Figure 25 depicts a schema of the trained initial HMM).

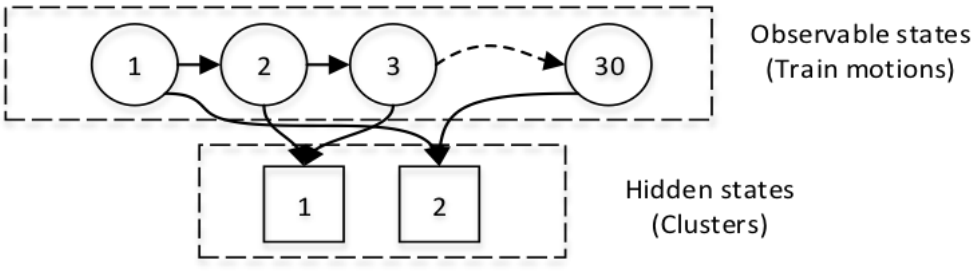


Figure 25. An overview of trained HMM in running example

- Test data generation using HMM classifiers: Now, in order to generate more complex motions (within time interval $\theta = 500\,ms$), covering unexpected human-generated gestures, two (number of classes) HMM classifiers are trained and the forward

probability of the current motion is calculated to reveal the class of motions it belongs to. Then, the mean of the accelerations of the motions belonging to this class are calculated; and again, are used as input of the motion equations to calculate the velocity and location parameters. For example, if the occurrence likelihood (forward probability) of the current motion $m_i(20, 492.07,2.1625056,2.1625056, -0.00778115, 0.24600422)$ in class c_2 reaches the maximum amount compared to the other class (c_1), the mean of the acceleration of the motions in class c_2 is calculated and will be used as the new current motion's acceleration. In this case, the mean of the accelerations in c_2 is equal to (0.3471,1.1162). Therefor, using physics equations, the next motion would be:

$m_{i+1}(21.1246403,493.2907778, 2.3360556,2.7206056,0.3471,1.1162)$,

This motion also will be added to the training set.

Once, the ball hits the vertical wall, the motions generated since the last hit, are saved in the form of a test case and will be executed to move the ball toward different directions on the screen.

5.6 Empirical Evaluation

5.6.1 Experimental Setup

To study the proposed approach, we performed an experiment on four case studies from three different mobile applications that could detect and execute motion-based gestures. Unfortunately, finding case studies in this area is far from straightforward. Firstly, the volume of open-source games is limited and many of them are ports of existing games from traditional platforms. For example, Wikipedia[32], AOpenSource.com[33], F=Droid[34], and Prism-break[35], provide lists of notable

[32] https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications

open-source applications for the Android platform. However, upon review, the reader soon discovers that nearly all of the applications are ports of desktop or laptop applications. Hence, none of the applications feature user-interaction via gyroscopic input devices. This obviously renders these applications useless as case studies.

Additionally, it is worth noting that even though our approach is able to generate test cases for mobile applications covering both gyroscopic and non-gyroscopic events, the focus of the study is on providing a practical approach for generating motion-based events. This means that the portion of our algorithm, which is producing the test cases for non-gyroscopic events, can be easily replaced with other effective GUI-based test case generation techniques such as [206], [207].

In addition, even if inputs for these types of inputs could be generated in an unbiased form, it is far from clear a coverage statement could be realized for solely the gyroscopic portion of the product. And hence, we have decided, regrettably, to limit this study to applications where the inputs are of a gyroscopic-nature only. This allows us to comprehensively examine these applications and produce a set of unbiased results from experimenting with these applications.

In the evaluation section, we attempt to answer the following research questions:

- Can the test-generated motions mimic actual user behaviour?
- Does the proposed method improve the code coverage of the SUT when compared to existing automated techniques (random testing)?

---

[33] http://www.aopensource.com/

[34] https://f-droid.org/

[35] https://prism-break.org/en/categories/android/

- (a) How does the proposed approach compare with random algorithms in terms of the computational complexity? (b) Can random algorithms produce better test cases (in terms of the code coverage) than our proposed approach, if the same volume of computational resources, as given to the HMM-based approach, is assigned to them? The answers to these questions are provided in the separate section (Section 5.7: Run-time analysis)

5.6.1.1 Case Study 1: Bouncing ball

The first case study is an Android application, a bouncing ball application, which is designed to record a data set of coordinates from shake and tilt gestures performed by human users *(LOC=716).* This application only contains one flying object (round ball), which bounces on the screen; the ball moves by processing the information it captures from a phone's accelerometer.

The dynamics of a bouncing ball follows a set of well-studied physics laws and equations [30], which are used in this study. Since covering the details of such equations is beyond the scope of this research, we only discuss some of the case-specific motions and equations:

- When the application starts, the ball position is stable in a corner of the screen, waiting for a motivation. Depending on the power of the first motion, the ball starts moving toward the motion's direction. In this study, the time interval φ is fixed at 300 milliseconds to capture the information regarding the current motion and position of the ball on the screen and to calculate its next position. The time interval is set to 300 milliseconds to follow the approaches proposed in [195], [196]. In other words: $300 < the\ time\ periods\ between\ user-created\ motions.$

- The second time interval θ is equal to 500 milliseconds in this study because the time windows between gestures created by users vary from 500 milliseconds to one second, we select the lower bound to create more standard motions.
- While the ball is moving on the screen, the motion data is re-ordered in the 6-tuple format, used to express test motions (Section 5.3.1). Each sequence is terminated whenever the ball hits the vertical edges of the screen.

Table 41. Simplest Supported Actions and Gestures in Both Types of Application

Bouncing Ball / Extended Bouncing Ball		**Bubbles**		**Diamond**	
Action	**Gesture**	**Action**	**Gesture**	**Action**	**Gesture**
Tilt the device toward left	The ball bounces to the left side of the screen	Touch/Push the screen.	The circle is drawn around the touched-point	Tilt the device toward left	The object bounces to the left side of the screen
Tilt the device toward right	The ball bounces to the right side of the screen			Tilt the device toward right	The object bounces to the right side of the screen
Tilt the device to the front	The ball bounces down.			Tilt the device to the front	The object bounces down
Tilt the device to the back	The ball bounces up			Touch/Push the buttons	The action recorded in the button will be triggered

Table 41 (First two columns) indicates the simplest possible actions that can be performed through this application, along with their corresponding gestures. It is noteworthy that in designing this table, it is assumed that the ball has enough space to move toward each direction. Obviously, it cannot for example move to the left when it has already hit the right-side edge. Any combinations of these actions (e.g. curving), which may be produced by rotating, tilting the

device, and so on is also considered in this case study. For example, when the user rotates or tilts the mobile phone toward the right, the ball can moves in a curve instead of moving in a straight line to the right.

5.6.1.2 Case Study 2: Bubbles

The second case study is another android application called Bubbles, which is able to draw circles around the touched points on the screen *(LOC=423)*. In order to generate circles (bubbles), the user touches or pushes the screen resulting in a circle being gradually grown from the touched point. The maximum length of the circle's radius is predefined and fixed, so the circle keeps growing until it's radius is equal to the maximum number or the user touches another point in the screen. Table 41 (Second two columns) shows the action (motion event) and its corresponding gesture. In this case, the motions containing the coordinates of the touched points are captured within the same time intervals as the first case study to generate a set of motions. The sequences of motions are continuously generated until a border is touched. Then, the generated set is considered as a test case.

5.6.1.3 Case Study 3: Extended Bouncing ball

We also modified the Bouncing ball application by adding one more flying object in the screen. The second ball behaves the same as the first one (Table 41– First two columns), except for the difference that its initial location in the screen is in the bottom right-hand corner (the original ball is located in the left side), thus depending to the amount of acceleration received from the sensors, they can move in diverse directions. The same test generation algorithm is used to produce test cases for the extended Bouncing ball application (*LOC= 1054*) as the simple version and test motions are stored in two separate sets of test suites for each ball.

5.6.1.4 Case Study 3: Diamond

In order to evaluate the performance of our proposed test case generation approach in a more complex framework we also applied our technique to generate test cases for another real world mobile game. This game, which is called Diamond (*LOC= 4311*), is a classic game implemented in a modern way using accelerometer sensor. In this application, the user controls an object in the screen and tries to collect as many diamonds as possible by moving the mobile phone toward the correct direction. The player also has to avoid hitting enemy objects and reach to the end point safely. The moving object follows the common behaviour of a bouncing object (Table II – third two columns) and the terminal condition happens when the game is over (the game is over, when the player hits an enemy object or reaches the end point). Moreover, in order to enter the game, change the settings or quit the game, the player needs to select items from the menu by pushing some buttons. Therefore, the application is able to handle more than one input type (both gyroscopic and non-gyroscopic data).

5.6.1.5 Comparison Criteria

In order to address the first research question and analyze the performance differences between the proposed approach and other test case generation methods, Non-parametric Statistical Hypothesis Tests and Effect Size (cliff's delta) Measures are applied:

Non-parametric Statistical Hypothesis Test: In this case, we established a null hypothesis and an alternative hypothesis to be evaluated. The null hypothesis (H_0) states the two test case generation techniques provide the same performance in covering the source code. On the other hand, the alternative hypothesis (H_1) states that if the difference between the medians of the coverage percentages, is not zero then they will be considered as different. Therefore, by

considering a significance level $\alpha = 0.05$, we would be able to reject null hypothesis if $p - value < 0.05$ for each independent situation.

Effect Size: In order to add a "magnitude of a treatment effect" to our comparison criteria, Cliff's Delta measure is calculated. Cliff's Delta statistic [86] is a nonparametric effect size measure that quantifies the difference between two groups of observations by testing the equivalence of probabilities of scores. In this study, the magnitude of differences between test generation techniques is assessed using the following thresholds: $|d| < 0.147$ "negligible", $|d| < 0.33$ "small", $|d| < 0.474$ "medium", otherwise "large" [236]. In addition, Cliff's Delta is a bounded measure [-1, +1] where the limiting values indicate that the two populations have no overlap.

5.6.2 Experimental Results

To answer the research questions and evaluate the efficiency of the proposed test generation approach, a volunteer interacted with all the applications and produced motion sequences which are then used as training sets. For instance, in the Bouncing ball application, a set of training data was obtained by recording the motion coordinates for three minutes from a total of 317 gestures performed on two different Android devices[36]. Applying the silhouette score, we grouped the motions into 95 clusters. For the extended version of this application, 600 motions and 105 clusters were considered. This data is recorded in 6 minutes. For the Bubble application, these numbers were 481 and 95 respectively (motions are stored for 2 minutes). Training motions are also stored in the 6-tuple format used to express test motions (Section 5.3.1). The amount of time allocated to each training process is estimated based upon the time a new user

[36] Samsung Galaxy S5 (Android version 4.4.2), Samsung Galaxy S4 (Android version 4.3)

needs to become visually familiar with the application and to generate a set of motions. In this study, this time is estimated by calculating the mean of the time that new users require to generate a reasonable set of motions for the considered applications.

To evaluate the quality of the generated test cases in all case studies, 20 sets of 200 motion sequences were generated using the proposed technique. In addition, for the Bouncing ball application, the same number of motion sequences (20 sets of 200 motions) was created by two random test generator procedures:

- Physics-based: takes a human-user motion to initialize the acceleration or position parameters then creates the next motions based on the current one by randomly selecting a physics equation (Table 42).
- Simple Random Algorithm: Creates test cases by simply generating random motion sequences within the data ranges supported by the hardware. The well-known Mersenne Twister (MT) approach, a pseudo random number generator (PRNG) is used in this study, which generates random numbers based on Mersenne prime $2^{19937} - 1$, using a 32-bit word length [237]. In this study, a human user also generates the initial motion. Since, the HMM-based technique is using human-data to train the initial model and generate the first motion, the simple random test case generation process also get initialized by human-generated data.
- Hybrid approach: In order to conduct a fair comparison between approaches, some experiments have been designed to execute combinations of human and randomly generated test cases (e.g. “Human + Simple random” and “Human + Physics-based”). This means that using human data is not limited to the initialization phase and user-

generated data forms half of the test cases. Therefore, a hybrid test case consists of a combination of human generated motions and random motions.

- Random-Human: we also created another random-based approach by randomly selecting motion events from the training data. In this approach we generated test cases by picking random motions from human generated training set.
- Monkey [199]: In order to compare the performance of proposed approach with other well-known existing tools. We also generated test uses for the considered case studies using the Monkey tool. This tool is able to send a pseudo-random stream of user events (such as clicks, touches, or gestures, as well as a number of system-level events) to the system. Such streams act as a set of test cases for the application under test.
- Sapienz [210]: Another well-known testing approach for Android mobile applications is called Sapienz. This technique uses multi-objective search-based testing to automatically generate test cases. In another word, Sapienz combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation together to generate automated tests cases for Android application. In this study we applied the white-box manner, which uses fine-grained instrumentation at the statement level.

Table 42. Random Test case generation procedure for cases with acceleration involved (Physics-based)

- **Input:** Initial position of the bouncing object (x,y); $i = 2$;
- **Output:** Random Test case (TC)

1. (ax,ay)⟵ getHumanMotion()
2. **While** (!terminalCondition*)
3. e ⟵Select RandomEquation()
4. m_i ⟵CreateMotion(ax,ay,e)
5. $i \leftarrow i + 1$
6. **End while**
7. **Return** $TC \leftarrow \{m_1, \dots, m_i\}$

*In this case we terminated the process after generating 200 test cases

It is worth noting that in cases (e.g. the second case study), where the acceleration parameter is playing a significant role in defining a motion, the generated accelerations in random test cases are limited to the acceleration range supported by the hardware. In addition, since the acceleration parameter and its corresponding physics equations are not considered in the second case study, only the simple random algorithm is implemented to generate the random touched-points.

To answer the first research question, we classified two sets of test cases (derived from Bouncing ball and Bubbles applications) by using the HMM classifier into 95 classes which are defined based upon the data generated by the human users. The same procedure is applied on the test data generated for the extended Bouncing ball and Diamond applications and they are classified into 105 clusters. Then the occurrence likelihood (LC) of each sequence of motions for each class label are calculated where $\{LC = P(M|\Lambda_i), \Lambda_{i\leq L} and\ M \in TC\}$, where L is the number of classes. In this case, when $\max_L P(M|\Lambda_i)$ is a small quantity, it can be concluded that the test case TC is not behaving similar to the test cases that were used to create the classes. Additionally, since these classes are created using human-generated motions, it can be implied that the probability of the test case TC being generated by a human user is low.

The results show that the motions generated using the HMM-related technique have a higher forward probability (occurrence likelihood) compared to both Simple Random and Physics-based approaches. Accordingly, it can be concluded that the test cases generated using the proposed technique are more likely to be generated by a human user. The reason is that each class label describes a set of human-generated motions; therefore once a motion has high occurrence likelihood in one of these classes, it can be concluded that the probability of being generated by a human user for this motion is high.

Figure 26 (a), (b), (c) and (d) depict boxplots showing the distribution of the occurrence likelihoods of the motions produced by the HMM classifier model and the simple random approach for all considered applications. According to these figures, it also can be concluded that the generated random tests in Bubble application are "behaving better" than the random tests in the rest of applications. The reason is that the gestures in the Diamond and Bouncing ball applications are more complex than the gestures in the Bubble application in terms of motion sequences. This makes it more difficult to generate gestures resembling human behaviour using the random approach in the Bouncing ball and Diamond applications when compared to the Bubble.

To address the second research question, the JaCoCo[37] code coverage library was used. Using this toolkit, bytecode instrumentation is applied, and the branch coverage value is measured. Since we generated 20 sets of 200 test cases using each approach, the means of the coverage percentages on all sets, are calculated to achieve more accurate results (In total, 64000 motion sequences are generated during the experiments). Table 43 and Table 44 report the means of the branch-coverage percentages calculated by running each of the test case generation approaches in all applications.

Figure 27 (a), (b) and (c) also show the distribution of the code coverage using box plots. According to these results, HMM-based test cases achieve better coverage compared to the random and human-generated test cases. In addition the results of applying the Wilcoxon signed rank test indicates the HMM-based approach is significantly different from the other techniques in terms of code coverage.

[37] http://www.eclemma.org

Table 43. Results of Calculating Effect Size Measure and the Mean of Code Coverage For Test Case Generation Methods in Bouncing ball and Extended Bouncing ball Application

	Approach	Mean of Code Coverage (%)	Approach	Delta Estimate	p-value
Bouncing ball	HMM-based	79.26	HMM-based Vs. Physics-based	-0.965	4E-05
	Physics-based	55.95	HMM-based Vs. Simple Random	-1	4E-05
	Simple Random	33.05	HMM-based Vs. Human + Physics-based	-0.7357	0.00019
	Human + Physics-based	63.63	HMM-based Vs. Human + Simple Random	-0.6761	0.00034
	Human + Simple Random	62.73	HMM-based Vs. Human	-0.7225	0.00017
	Human	60.2	HMM-based Vs. Random-Human	-0.8575	7.6E-06
	Random-Human	59.05	HMM-based Vs. Monkey	-0.95	1.9E-06
	Monkey	37	HMM-based Vs. Sapienz	-0.95	1.9E-06
	Sapienz	41.15			
Extended Bouncing ball	HMM-based	81.3	HMM-based Vs. Physics-based	-0.95	1.9E-06
	Physics-based	52.75	HMM-based Vs. Simple Random	-0.95	1.9E-06
	Simple Random	31.77	HMM-based Vs. Human + Physics-based	-0.71	3.4E-05
	Human + Physics-based	62.78	HMM-based Vs. Human + Simple Random	-0.575	0.0028
	Human + Simple Random	62.98	HMM-based Vs. Human	-0.62	0.0002
	Human	62.05	HMM-based Vs. Random-Human	-0.87	1.3E-05
	Random-Human	58.7	HMM-based Vs. Monkey	-0.95	1.9E-06
	Monkey	37.75	HMM-based Vs. Sapienz	-0.95	1.9E-06
	Sapienz	42.3			

Table 43 and Table 44 also report the p-values and delta estimates at the 95% confidence interval. The Cliff's Delta measure provides more detailed information to this picture by showing that a "large" effect size exists (in favour of HMM-based approach) for all of the comparisons.

The achieved results confirm that the HMM-based test case generation approach not only automates the test generation and execution procedure for motion-based events, but also (1) creates better test cases in terms of mimicking actual user gestures; and (2) improves the

(branch) code coverage for the SUT. In the next section, we compare our proposed approach with random generation in terms of the time complexity.

Table 44. Results of Calculating Effect Size Measure and the Mean of Code Coverage For Test Case Generation Methods in Bubble and Diamond Applications

	Approach	Mean of Code Coverage (%)	Approach	Delta Estimate	p-value
Bubbles	HMM-based	92.06	HMM-based Vs. Human + Simple Random	-0.9325	5E-05
	Simple Random	74.28	HMM-based Vs. Human	-0.9325	4E-05
	Human + Simple Random	79.97	HMM-based Vs. Simple Random	-1	4E-05
	Human	78.94	HMM-based Vs. Random-Human	-0.9425	1.9E-06
	Random-Human	76.8	HMM-based Vs. Monkey	-0.95	1.9E-06
	Monkey	74.85	HMM-based Vs. Sapienz	-0.985	1.9E-06
	Sapienz	75.65			
Diamond	Human + Physics-based	63.02	HMM-based Vs. Human + Simple Random	-0.7025	0.0003
	Human + Simple Random	61.39	HMM-based Vs. Human	-0.5525	0.0022
	Human	63.85	HMM-based Vs. Random-Human	-0.575	0.0016
	Random-Human	62.99	HMM-based Vs. Monkey	-0.95	1.9E-06
	Monkey	42.25	HMM-based Vs. Sapienz	-0.95	1.9E-06
	Sapienz	50.91			

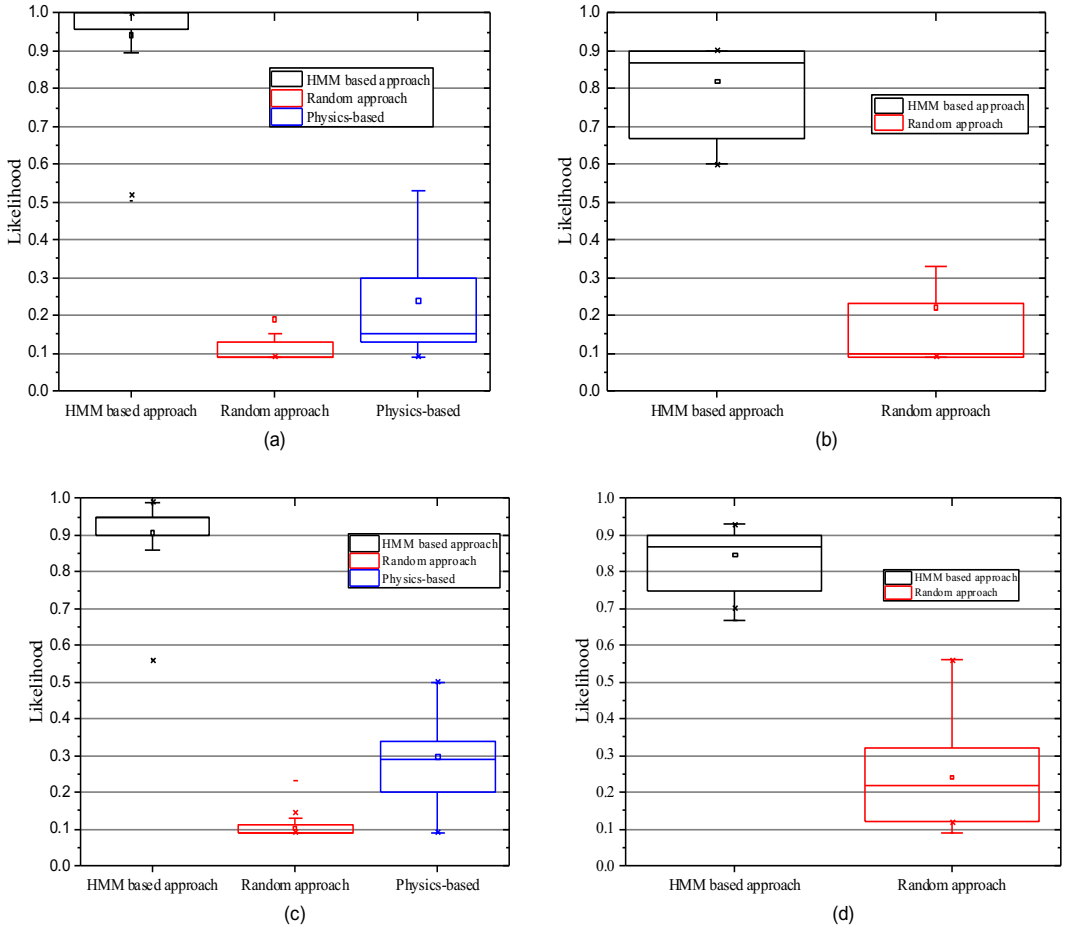
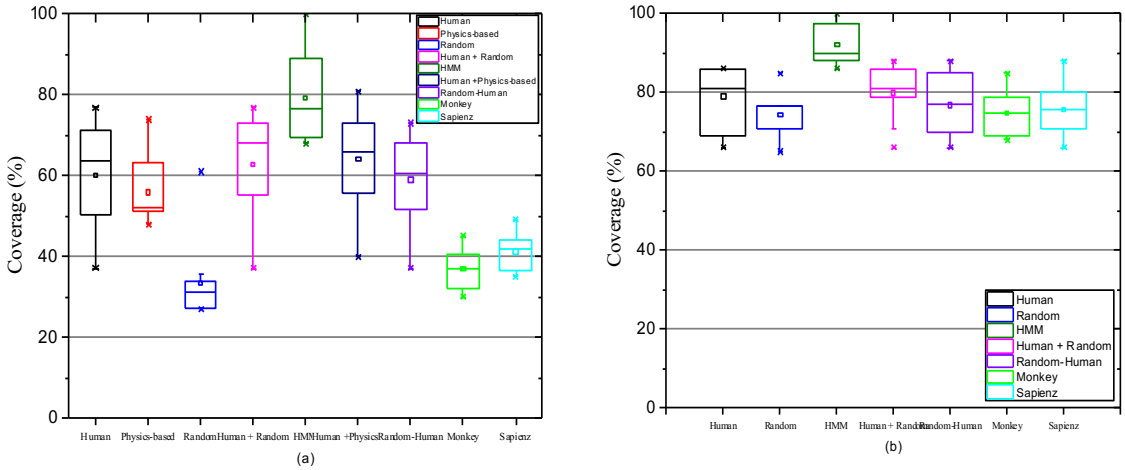


Figure 26. (a) Boxplot summarizing the achieved likelihoods for each approach in the Bouncing ball application. (b) Boxplot summarizing the achieved likelihoods for each approach in Bubbles application (c) Boxplot summarizing the achieved likelihoods for each approach in Extended Bouncing ball application (d) Boxplot summarizing the achieved likelihoods for each considered approach in Diamond application



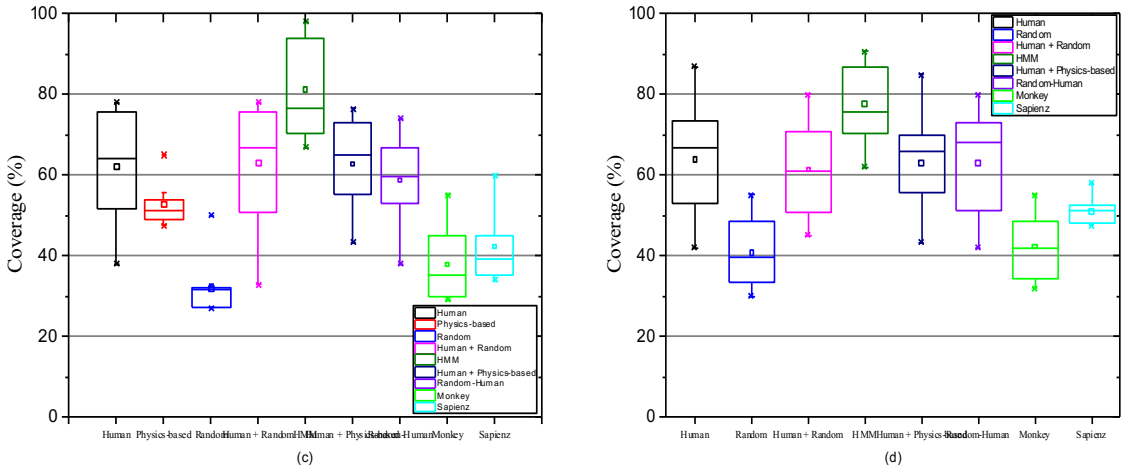


Figure 27. (a) Boxplot summarizing the results of calculating the code coverage for each approach in the Bouncing ball application. (b). Boxplot summarizing the achieved results of calculating the code coverage for each approach in the Bubbles application. (c) Boxplot summarizing the results of calculating the code coverage for each approach in the Extended Bouncing ball application (d) Boxplot summarizing the results of calculating the code coverage for each approach in the Diamond application

5.7 Run-Time Analysis

In order to answer the part (a) of the third research question, we considered the computational complexity of the proposed test generation approach by running a single instance of this technique on a hardware and software platform consisting of a 2x2.4 GHz Quad-Core CPU, 32 GB RAM on a Mac Pro, Eclipse Indigo[38] and a Samsung Galaxy S5.

To investigate the time complexity of HMM-based approach, we analyzed the complexity of the involved algorithms. Based on the Baum-Welch and forward algorithms' time complexities, the computation order of our approach is polynomial $O(T^2 n)$, where T represents the number of hidden states, and n indicates the number of observations. Hence, each algorithm's time complexity will not grow exponentially by increasing the number of motions.

[38] https://eclipse.org/indigo

Moreover, recent literature has considered the question of should algorithms be compared against their speed of performance – if an algorithm is twice as slow as the other algorithm should the quick algorithm get twice as many tries at getting it correct? The answers to these questions should also address part (b) of the third research question.

To provide an accurate answer to this question it should be noted that, while it is easy to have sympathy for this viewpoint, it is very difficult to construct an unbiased examination of algorithms from this perspective. Consider, testing and test case generation, the topic of this article, the first problem encountered is that test generation is only a sub-process. Following [238] an automated testing system has three components; test generation, test execution, and examination of the test results. So, the total time (t_t) is combination of all; $t_t = t_g + t_e + t_o$ where t_g, t_e, and $t_o$stand for generation time, execution time, and result examination time, respectively. Test generation and execution can be automated easier than test result examination – the production of meaningful test oracles is still at a very early stage in research. With respect to examination of the test results, two options are normally used:

- A test oracle is constructed to automate the test examination. The test oracle usually has a simplified definition of a defect. Does the system crash or not, is an example of such a description. Here each crash is considered a "defect".
- The test results are investigated manually by the tester.

When $\mathrm{t_e}$ is small (very small programs) and the test result examination is fully automated (small t_o), one would be better off running more test cases instead of generating more efficient test cases [239]. In such a situation, methods that have high runtime compared to random generation are not cost effective. However, industrial software's execution runtime is usually large enough

to have adequate time for test generation. Yoo et al. [240] have considered using parallelised search based optimisation algorithms to find optimal sets of test cases or to prioritize test cases for regression testing, since executing all test cases for large-scale applications is a very time consuming task. The total test execution time in their study is equal to the times for initializing test cases, evaluating the fitness values of different generations and the remaining parts of the execution time.

Further, test result examination is not typically fully automated, unless a simplistic test oracle (e.g. finding system crashes) is utilized. Hence, test result examination normally requires manual work by the tester. Hence, generating more effective test cases, which normally have higher runtime than random test cases is believed to improve failure detection in most cases. Hence, in many situations running test case generation algorithms for equal amounts of time may actually be a rather poor objective.

Additionally, fast algorithms producing large numbers of poor test cases have a significantly detrimental effect on the effectiveness of the entire testing process. Previous research shows that individual aspects such as testers' skills have as strong an effect on the results of testing, as do the test case generation techniques. Other components, including test case execution and especially manual test oracle processes are far from straightforward. Several empirically-based findings [241]–[245] have emphasized the role of experience and skills in these software testing activities. Hence, in general, making the execution and manual test oracles components more complex by running test case generation algorithms that produce large volumes of poor test cases is normally a bad idea. Other studies explicitly warn against producing large numbers of unproductive test cases. For example, in Williams et al. [246] based upon interviews with actual practitioners at Microsoft, state "Unit testing coverage needs to be measured. The quantity of test

cases is a bad measurement. More reliable is some form of code coverage (class, function, block etc.). Those measurements need to be communicated to the team and be very visible."

This implies that the practitioners are looking for techniques which assist in producing test sets which have good characteristics (such as coverage) while avoiding bad characteristics (such as large volume).

In addition, many new testing initiatives such as continuous integration (CI) become impossible to implement as the size of the test set increases. CI is highly dependent on cycle time – time to compile and automatically execute the test set – unless the cycle time remains short, developers quickly become disenfranchised by the process. This leads to abandon of the execution of the test set and results in increased defect rates. In conclusion, much research exists which suggests that automatic test case generation algorithms should be principally concerned with producing high-quality test cases rather than worrying about execution times except in extreme situations.

Even if we ignore this, comparing execution times are still a highly problematic undertaking. Often the algorithms will be produced by different authors, be at different stages of development, and utilize different technologies. For example:

1. The current algorithm is produced by a student programmer, whereas a random library Mersenne Twister has been actively produced and evolved over a substantive period by a large pool of professions, who actively ensure that the code is efficiency whereas the current algorithm is simply a first-cut prototype with no real interest in efficiency.
2. Mersenne Twister has seen decades on development with many proposals on producing more and more efficient versions whereas the current algorithm has seen none.

3. Most random libraries are written in C; whereas the current algorithm is written in R. Anecdotal comparisons often state that algorithms written in R run 1000 times slower than equivalent algorithms in C[39]. Hence, any attempt to compare two such algorithms via execution time would be highly biased rendering any such results next to useless.

Perhaps, a better viewpoint is to consider the algorithms via their algorithmic complexity statements. However, even here the volume of work on developing an algorithm creates a significant bias. [169] noted that adaptive testing algorithms (ART) such as [247] were not effective because of their $O(n^2)$ time complexity, where n is the number of test cases. However, the field had previously made no real attempt at producing more efficient algorithms. Recently Shahbazi et al. [238] has produced a new ART algorithm, which produces more effective test cases than previous ART algorithms. In addition, the paper also looked at time complexity and produces test cases with a time complexity of $O(n)$. Following up from this work, Singh et al. [248] have recently produced a concurrent version of this algorithm with time complexity of $O(n/p)$, where p is the number of processors available to the algorithm. Hence, even the algorithmic complex of an algorithmic tends to reduce over time as more effort is spent upon a topic. Implying that for any algorithm with a known polynomial-time solution, that even algorithmic complexity is a non-stable indicator of performance.

Having said all of this, we still provide some guidance on the effectiveness of the algorithms with regard to computational complexity. Therefore, in this study, which the computational complexity of the proposed approach is $O(T^2n)$, assuming that the method generates 200 motions using train data clustered into 13 different classes, the asymptotic complexity of the test

[39] http://lists.nongnu.org/archive/html/igraph-help/2011-02/msg00045.html

generation process would be $13\times13\times200 = 33800$. Thus, if we allocate the same asymptotic complexity to random test generation, with computational complexity $O(n)$, random test generation approach will be able to generate 33800 motions in the provided time. Obviously, it would be more expensive to run 33800 random motions compared to 200 motions generated by HMM-based approach.

As illustrated in Table 45, it has been noticed that running all of these motions (33800) in the Bouncing ball application improves the average code coverage up to 42% for the random approach, which is sill lower than coverage, reached by HMM-based technique (75%), running 200 motions. In addition, Running the 33800 test motions in Bubbles increases the coverage to 78% for random, while the percentage of code coverage is 92% for the HMM-based test case generation technique using 200 test motions. Therefore, it can be concluded that providing the same resources as HMM-based approach to random does not necessarily lead to significant improvement in the code coverage. Additionally, the time it takes to generate 200 motions using the HMM-based technique ($\boldsymbol{t_g}$) is less than a minute, for the bouncing ball application, while it takes 3 minutes to execute them; therefore $\boldsymbol{t_g < t_e}$. While, the time is required to execute the 33800 test cases generated by random approach is 23 minutes. This result confirms the statement provided at the beginning of this section, illustrating that generating too many test cases using random techniques is not always a good option for improving code coverage. Specifically, high test-execution time in industrial case studies with large test suites provides sufficient time to generate more efficient test cases, using well-designed test case generation approaches.

Table 45. Results of Providing Same Resources as HMM-based to Random

	Approach	Code Coverage (%)	t_g(min)	t_e(min)
Bouncing ball	HMM-based (200 motions)	75%	0.5	3
	Random (33800 motions)	42%	0.17	23
Extended Bouncing ball	HMM-based (200 motions)	75%	0.5	3.2
	Random (33800 motions)	40%	0.17	24
Bubbles	HMM-based (200 motions)	92%	0.2	1.2
	Random (33800 motions)	78%	0.08	15.6
Diamond	HMM-based (200 motions)	71%	0.7	2.8
	Random (33800 motions)	46%	0.25	20

5.8 Threat to Validity

In this section, we consider the potential threats to the validity of our research and discuss the methods used to address them. In this study, we are principally concerned with three types of threats: internal validity, external validity, and the power of the experiment

Threats to the internal validity might come from the method of assigning the time intervals in the empirical study. If the time intervals are estimated to be too short (long), then more (less) motions will be generated compared to when a human user is interacting with the application. To address this issue, we estimated the minimum and maximum numbers of generated movements via several users' experiences and considered their average as a type-one interval (φ).

On the other hand, the threats to the external validity of our research are centred on the generalization of the results to other SUT motion-based applications. In this study, we consider

four applications and two types of motion-based applications (both with and without gyroscopic inputs). However, we also point out that the proposed technique should be applied to more and different case studies (e.g. 3D applications) in future work.

The third threat represents the *power* issue. This can lead to type-two errors in studies with insufficient numbers of samples. To address this issue, we recorded three sets of 317, 481 and 600 motion sequences to design the training sets. The data was also grouped into 95 and 105 classes, which led to training two sets of 95 HMMs and a set of 105 HMMs.

Finally, at the meta-level an obvious risk exists: Are the three research questions good proxies for defect finding capabilities? Ideally, any paper would wish to consider this research question directly. However, given the relative infancy of these types of systems, insufficient data (with regard to defects) is believed to exist to allow such an experiment to be adequately constructed. Hence, the adoption of the proxies for the exploration is required.

5.9 Conclusion

Testing mobile applications that use motion-based gestures to interact with users poses a new challenge. Test inputs should be realistic motion sequences, which are able to simulate the user's behaviour in interacting with the application. This helps in revealing defects, which remain unknown in applications because they do not conform to expected human-generated motions. Since, Markovian models have been successfully used in software testing studies to generate models representing common user behaviour in UI testing [10], [27], [196].

In this study, we have proposed a new HMM-based approach, which presents a solution for automating the testing process for applications supporting motion-based events. Using this

method, gestures can be formally specified as sequences of motions, which are easy to re-execute in the application. Therefore, an HMM classification approach is used to classify the current movement into a class of motions providing the best description of the gesture's characteristics. Then, according to the results provided by the classification approach and using standard movement equations, a realistic proxy for the likely next movement coordinates can be estimated.

We evaluated our approach by generating a set of test inputs for four Android applications with a gaming theme. The empirical results show that the generated test cases using HMM-based approach not only cover a higher number of branches in the source code compared to randomly generated test cases, but the occurrence likelihood of the corresponding motion sequences in model trained by user generated data is also higher in HMM-based approach. This indicates that the new approach outperformed the random methods (including Monkey, Sapienz and Random-Human) in generating test cases that mimic human-user behaviour.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we introduce and develop the new idea of inferring behavioral models from both software executions and user-interaction log files. In both cases, a hybrid approach is used to apply RL and HMM concepts to dynamically generate Extended PFSAs and a set of probabilistic Markovian models.

In chapter 2 of this thesis, ReHMM (our proposed inference approach) is applied on the execution traces extracted from seven modules in two different programming languages. According to experimental results, ReHMM outperforms other EFSA inference algorithms in terms of the BCR (the measure used to evaluate the accuracy of the model). Moreover, ReHMM is compared with sk-strings algorithm (a well-known PFSA generation algorithm) and outperformed it in terms of accuracy. Therefore it can be concluded that ReHMM is able to generate more accurate models than considered EFSA and PFSA inference approaches.

It should be noted that this study makes a contribution to research in the area by proposing a new EPFSA inference approach from software execution traces. The proposed technique uses RL and HMM to explore the transitions, which trigger more changes in the model; and is able to detect functions governing transitions. This procedure also provides a solution to the problem of the missing state-action value by assigning Q-valued to the transitions.

In chapter 3 of this study, we propose another inference approach to automatically generate a reward-augmented user behavioral model from the user-interaction log files. This study makes a contribution to research in the area by (1) automatically calculating and assigning the states' rewards which add semantics to the models and ease interpreting the models and detecting

design anomalies; (2) covering enterprise-size web applications with no need of instrumenting the source code; and (3) generating comparable results with the data extracted from Google Analytics. In order to evaluate our technique, we apply it on user-interaction log files extracted from the enterprise mobile and web application, called MyUAlberta. Then we compared the generated model with the user-behavioral workflow extracted from the associated Google Analytic account. The result indicated that the calculated reward values are compatible with the values extracted from Google Analytics in determining a page's importance. Since, unlike Google Analytics, our approach does not need instrumenting the source code and these results are only achieved by running the inference algorithm on the log files, it could be concluded that the proposed approach is useful in legacy applications and in those, where the source code or the system expert is not available.

In chapter 4, we present a new fault-based test case prioritization approach using an extended digraph. Again, the digraph is generated by using an RL-based HMM approach. We initialize an appropriate HMM based on a Q-learning algorithm to infer an HMM with the maximum likelihood estimate of the parameters. Then we use the estimated model to compute the forward probabilities of the test cases and prioritized them based upon their corresponding forward probabilities.

In order to evaluate the proposed method, we used AutoBlackTest, a GUI-based test case generation tool as a baseline to generate and prioritize test cases. We also included Random, Additional statement coverage, Worst, Optimal and Weight-based [164] prioritizations to provide a comprehensive comparison. All considered prioritization techniques are applied on four different GUI applications. According to the results of different considered measures including the APFD, relative ratio, boxplots, statistical tests and effect size estimates, RL-based

HMM outperforms the other approaches in terms of fault detection effectiveness. It indicates that the amount of change, a test case may cause in GUI states and, the probability of triggering each action in each specific state play essential roles in determining the capability of a test case in detecting faults.

Finally in chapter 5, we propose another HMM-based approach to generate test cases for mobile applications, supporting motion-based events. Our proposed approach applies an HMM classification technique to determine the class of the current motion and uses it to predict the next movement coordinates. In addition, an empirical study is performed to compare the proposed approach too randomly, and human-generated, test cases in terms of (1): the code coverage, and (2): the capability of generating motions mimicking human-generated test cases. It is worth noting that in order to cover generating both simple and complex gestures; we consider producing test cases within two different time intervals. Within the first interval motions are generated using SUVAT equations, but within the second time interval, more complicated test cases are generated using the HMM classifiers and the forward probabilities.

In order to evaluate the HMM-based test case generation approach, we apply this technique to three Android applications supporting motion-based events. The empirical results indicate that HMM-based technique covers more branches in the code compared to randomly generated test cases while, the motions generated using this approach has also a higher occurrence likelihood compared to other considered techniques. This indicates that the new approach outperforms the random method in (1) generating motions that mimic actual human-user behaviour; and (2) reaching the high code coverage.

It is worth noting that we also investigated all of our proposed stochastic approaches in terms of their time complexity. We performed a run-time analysis on all involved algorithms and

concluded that the run-time of none of our proposed algorithms grows exponentially when the system size increases, which means they all can be applied in large-scale software systems.

6.2 Recommendations for Future Research

Although the results of this research demonstrate improvements in the accuracy and effectiveness of the model-based testing and behavioral model inference procedures, there is still room for more improvements. This research can be extended for further investigation as follows:

- The ReHMM algorithm generates a more optimized model with higher quality [36] in comparison with other considered inference techniques, making it possible to avoid many false merges during the merging procedure. We believe in order to avoid all inappropriate merges in such algorithm a more accurate merging protocol should be defined and applied. Further studies to prove this claim should be considered in the future.
- We also believe that our proposed modelling approach in chapter 3 could be extended to be applicable on any probabilistic timed automata [107] to capture other user behaviors and their corresponding reward values.
- Additionally, even though the results of this research improve the effectiveness of test case prioritization, the RL-based HMM approach still has room for improvement. First, additional studies can be performed on more applications such as web-based applications. Second, in this study, we only consider GUI applications. This method can be evaluated further in order to present a generic approach to generate an RL-based weighted model for every type of application. [147] represented a static approach to prioritizing Junit test cases by defining the distance between pairs of test cases based upon using topic modeling. Such techniques can be utilized to compute the reward function and Q-values

in non-GUI based applications. Third, detecting the best sequence of GUI states contributing to the most appropriate prioritized test suite would be helpful in addressing the *second* HMM problem (mentioned in section 5.4.3.2) using Viterbi Algorithm and finding the most suitable ordering which maximizes the HMM's likelihood of the estimated parameters.

- With regard to our motion-based test case generation technique, although there are promising results, we believe that our experiments only cover an initial exploration of this area, and several issues remain to be addressed in further studies which could be covered in future studies: (1) **Different types of motion-based applications.** This study only considers two types of mobile applications that use motion events to interact with users. Future studies should be performed on the proposed technique in more complicated applications with 3D graphical design. (2) **Different time intervals.** Using further empirical experience with different time intervals should also be considered in future work. (3) **Influence of training data on efficiency of generated test cases.** Our evidence is based on estimating HMMs on a single set of training data. There is the potential to use different methods of sampling the training data, and evaluating their impact on the efficiency of test cases. For example, different time intervals and terminating conditions can be used during the training data capturing process. (4) **Fault detection capability.** The ability of generated test cases to detect faults should also be investigated. Unfortunately, we are currently unaware of any suitable application with a published list of motion-based real-life defects.

  Moreover, with the recent developments in the Virtual Reality (VR) technologies; testing motion-based applications in devices running VR programs is a new challenge. Providing

a testing approach, which is able to detect the natural human gestures from users interactions with devices like Oculus Rifts[40] or similar touch controllers introduces a new and interesting area of research which can be considered in futures studies.

[40] https://www.oculus.com/rift/

References

[1] J. E. Cook and A. L. Wolf, “Discovering Models of Software Processes from Event-Based Data,” *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, 1996.

[2] C. Ghezzi, P. Milano, G. Tamburrelli, and M. Sama, “Mining Behavior Models from User-Intensive Web Applications,” *Proceeding 36th Int. Conf. Softw. Eng.*, pp. 227–287, 2014.

[3] D. Lorenzoli, L. Mariani, and M. Pezzè, “Automatic generation of software behavioral models,” *Proc. 13th Int. Conf. Softw. Eng.*, 2008.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically Discovering Likely Program Invariants to Support Program Evolution,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, 2001.

[5] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, A. Zeller, G. Fraser, S. Hack, and A. Zeller, “Generating Test Cases for Specification Mining,” in *International Symposium on Software Testing and Analysis (ACM)*, 2010, vol. 38, no. Section 4, pp. 85–95.

[6] S. Hangal and M. S. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 291–301.

[7] A. . Biermann and J. . Feldman, “On the Synthesis of Finnite State Machines from Samples of Their Behavior,” *IEEE Trans. Comput.*, no. June, pp. 592–597, 1972.

[8] I. Beschastnikh, Y. Brun, J. Abrahamson, M. Ernst, and A. Krishnamurthy, “Using

declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 408–428, 2015.

[9] E. Manavoglu, D. Pavlov, and C. L. Giles, "Probabilistic User Behavior Models," in *Third IEEE Conference o Data Mining*, 2003, pp. 203–210.

[10] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, 1994.

[11] K. Dejaeger, T. Verbraken, and B. Baesens, "Prediction Models Using Bayesian Network Classifiers," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 237–257, 2013.

[12] A. Avritzer, E. de Souza e Silva, R. M. . Lea˜o, and E. J. Weyuker, "Automated generation of test cases using a performability model," *IET Softw.*, no. March 2010, pp. 113–119, 2011.

[13] I. K. El-Far and J. A. Whittaker, "Model- based software testing," *Encycl. Softw. Eng.*, pp. 1–22, 2002.

[14] M. Utting, B. Legeard, and M. Utting, *Practical model-based testing: A tools approach*. Morgan- Kaufmann, 2006.

[15] H. Hemmati, "Similarity- based test case selection toward scalable and practical model-based testing," University of Oslo, 2011.

[16] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest : Automatic Black-Box Testing of interactive applications," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 81–90.

[17] X. Li, M. Rezvanizaniani, Z. Ge, M. Abuali, and J. Lee, “Bayesian optimal design of step stress accelerated degradation testing,” *J. Syst. Eng. Electron.*, vol. 26, no. 2, pp. 502–513, 2015.

[18] a. Avritzer and E. R. Weyuker, “The automatic generation of load test suites and the assessment of the resulting software,” *IEEE Trans. Softw. Eng. Softw. Eng.*, vol. 21, no. 9, pp. 705–716, 1995.

[19] F. Lindlar, A. Windisch, and J. Wegener, “Integrating Model-Based Testing with Evolutionary Functional Testing,” in *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 163–172.

[20] S. Arlt, S. Pahl, C. Berolini, and M. Schaf, “Trends in model-based GUI testing,” *Adv. Comput.*, vol. 86, pp. 183–222, 2012.

[21] M. Leonardo, F. Pastore, M. Pezzè, and M. Santoro, “Mining Finite State Automata with Annotations,” in *Mining Software Specifications*, no. February, C. Liu, Ed. CRC Press, 2011, pp. 29–57.

[22] L. Mariani and F. Pastore, “Automated Identification of Failure Causes in System Logs,” in *19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008, pp. 117–126.

[23] E. M. Gold, “Language Identification in the Limit,” *Inf. Control*, vol. 10, no. 5, pp. 447–474, 1967.

[24] G. Ammons, R. Bodik, and J. R. Larus, “Mining Specifications,” in *26th Annual ACM Symposium on Theory of Computing*, 2002, pp. 273–282.

[25] K. Mukherjee and A. Ray, “State splitting and merging in probabilistic finite state automata for signal representation and analysis,” *Signal Processing*, vol. 104, pp. 105–119, 2014.

[26] D. Freitag, D. Freitag, A. K. McCallum, and a. McCallum, “Information extraction with HMM structures learned by stochastic optimization,” *Proc. Natl. Conf. Artif. Intell.*, pp. 584–589, 2000.

[27] S. S. Emam and J. Miller, “Test Case Prioritization Using Extended Digraphs,” *ACM Trans. Softw. Eng. Methodol.*, 2015.

[28] N. Walkinshaw, R. Taylor, and J. Derrick, “Inferring Extended Finite State Machine models from software executions,” in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 301–310.

[29] N. Walkinshaw, R. Taylor, and J. Derrick, “Inferring extended finite state machine models from software executions,” *Emperical Softw. Eng.*, 2015.

[30] S. Kanjilal, S. T. Chakradhar, and V. D. Agrawal, “Test function embedding algorithms with application to interconnected finite state machines,” *Comput. Des. Integr. Circuits Syst. IEEE Trans.*, vol. 14, no. 9, pp. 1115–1127, 1995.

[31] K. N. Oikonomou, “Abstractions of Finite-State Machines Optimal with Respect to Single Undetectable Output Faults,” *IEEE Trans. Comput.*, vol. C-36, no. 2, pp. 185–200, 1987.

[32] A. W. Biermann and J. A. Feldman, “On the Synthesis of Finite State Acceptors,” *Stanford Artif. Intell. Proj.*, 1970.

[33] C. Luo, F. He, and C. Ghezzi, “Inferring Software Behavioral Models with MapReduce,”

Dependable Softw. Eng. Theor. Tools, Appl., vol. 9409, pp. 135–149, 2015.

[34] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014*, no. Section 6, pp. 468–479, 2014.

[35] W. Bartussek and D. L. Parnas, "Using assertions about traces to write abstract specifications for software modules," in *Information Systems Methodology: Proceedings, 2nd Conference of the European Cooperation in Informatics, Venice, October 10--12, 1978*, G. Bracchi and P. C. Lockemann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 211–236.

[36] R. Janicki, "Foundations of the Trace Assertion Method of Module Interface Specification," *IEEE Trans. Softw. Eng.*, vol. 27, no. 7, pp. 577–598, 2001.

[37] J. A. Brzozowski, "Representation of a class of nondeterministic semiautomata by canonical words ," *Theor. Comput. Sci.*, vol. 356, pp. 46–57, 2006.

[38] J. Brzozowski and J. Helmut, "Representation of Semiautomata by Canonical Words and Equivalences," *Int. J. Found. Comput. Sci.*, vol. 16, no. 831, 2005.

[39] I. Krka, "Automatic Mining of Specifications from Invocation Traces and Method Invariants," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 178–189.

[40] C. Ghezzi, P. Milano, P. L. Vinci, A. Mocci, and P. Milano, "Synthesizing Intensional Behavior Models by Graph Transformation," in *International Conference on Software Engineering*, 2009, pp. 430–440.

[41] M. Gabel, “Javert : Fully Automatic Mining of General Temporal Properties from Dynamic Traces,” *Current*, pp. 339–349, 2008.

[42] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, “Mining Object Behavior with ADABU,” in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, 2006, pp. 17–24.

[43] M. Santoro, “Inference of Behavioral Models that Support Program Analysis,” Università degli Studi di Milano-Bicocca, 2011.

[44] M. O. Rabin, “Probabilistic automata,” *Inf. Control*, vol. 6, no. 3, pp. 230–245, 1963.

[45] E. Vidal, I. C. Society, F. Thollard, C. De Higuera, F. Casacuberta, I. C. Society, and R. C. Carrasco, “Probabilistic Finite-State Machines — Part I,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 7, pp. 1013–1025, 2005.

[46] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, “Probabilistic Finite-State Machines--Part II,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 7, pp. 1026–1039, 2005.

[47] D. Lo and S.-C. Khoo, “SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner,” *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, pp. 265–275, 2006.

[48] J. Patrick, “The sk-strings method for inferring PFSA,” 1978.

[49] D. Lo and S. Khoo, “QUARK: Empirical Assessment of Automaton-based Specification Miners,” in *13th Working Conference on Reverse Engineering*, 2006, pp. 51–60.

[50] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[51] K. Bogdanov and N. Walkinshaw, "Computing the structural difference between state-based models," *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 177–186, 2009.

[52] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 651–666, 2008.

[53] R. B. Lyngsø, C. N. Pedersen, and H. Nielsen, "Metrics and similarity measures for hidden Markov models.," *Proc. Int. Conf. Intell. Syst. Mol. Biol.*, pp. 178–86, 1999.

[54] N. Walkinshaw and K. Bogdanov, "Automated Comparison of State-Based Software Models in Terms of Their Language and Structure," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 2, 2013.

[55] T. B. Le, X. D. Le, D. Lo, and I. Beschastnikh, "Synergizing Specification Miners through Model Fissions and Fusions," in *30th IEEE/ACM International Conference on: Automated Software Engineering (ASE)*, 2015, pp. 115–125.

[56] M. Hall, H. National, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software : An Update," *ACM SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.

[57] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, and J. Dick, "Using Formal Specifications to Support Testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 1–78, 2009.

[58] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, “Automated discovery of state transitions and their functions in source code,” *Softw. Test. Verif. Reliab. Wiley Intersci.*, vol. 18, pp. 99–121, 2008.

[59] G. Fraser and N. Walkinshaw, “Assessing and generating test sets in terms of behavioural adequacy,” *Softw. Testing, Verif. Reliab.*, vol. 25, no. 8, pp. 749–780, 2015.

[60] S. Selvakumar, M. R. C. Dinesh, C. Dhineshkumar, N. Ramaraj, and C. Paper, “Extended Finite State Machine Model-Based Regression Test Suite Reduction Using Dynamic Interaction Patterns,” in *Information Processing and Management: International Conference on Recent Trends in Business Administration and Information Processing, BAIP 2010, Trivandrum, Kerala, India, March 26-27, 2010. Proceedings*, no. 2016, V. V Das, R. Vijayakumar, N. C. Debnath, J. Stephen, N. Meghanathan, S. Sankaranarayanan, P. M. Thankachan, F. L. Gaol, and N. Thankachan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 475–481.

[61] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, “Using Behaviour Inference to Optimise Regression Test Sets,” *Test. Softw. Syst.*, pp. 184–199, 2012.

[62] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, “STAMINA: a competition to encourage the development and assessment of software model inference techniques,” *Empir. Softw. Eng.*, vol. 18, no. 4, pp. 791–824, May 2012.

[63] G. Rozenberg and D. Volker, *The Book of Traces*. World Scientific, 1995.

[64] D. Lo, L. Mariani, and M. Santoro, “Learning extended FSA from software: An empirical assessment,” *J. Syst. Softw.*, vol. 85, no. 9, pp. 2063–2076, Sep. 2012.

[65] G. Lu and H. Miao, “An Approach to Generating Test Data for EFSM Paths Considering Condition Coverage,” *Electron. Notes Theor. Comput. Sci.*, vol. 309, no. 61073050, pp. 13–29, Dec. 2014.

[66] C. Szepesvári, *Algorithms for Reinforcement Learning*, vol. 4, no. 1. Morgan & Claypool Publishers, 2010.

[67] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, vol. 9, no. 5. Cambridge, Massachusetts: The MIT Press, 1998.

[68] M. Wiering and M. van Otterlo, *Reinforcement Learning (State of the Art)*. Springer, 2012.

[69] C. J. Watkin, “Learning from Delayed Rewards,” University of Cambridge, 1989.

[70] P. Dayan and C. J. Watkin, “Technical Note Q -Learning,” *Kluwer Acad. Publ.*, vol. 292, pp. 279–292, 1992.

[71] L. J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” *Mach. Learn.*, vol. 8, no. 3–4, pp. 293–321, May 1992.

[72] W. Ching, *Markov chains: model, algorithms and applications*. Springer Science, 2006.

[73] L. R. Rabiner and B. H. Juang, “An introduction to hidden Markov models.,” *IEEE ASSP Mag.*, no. January, pp. 4–15, Jun. 1986.

[74] K. Hamamoto, K. Morooka, and H. Nagahashi, “Motion Recognition By Combining HMM and Reinforcement Learning,” in *IEEE International Conference on Systems, Man and Cybernetics*, 2004, pp. 5259–5264.

[75] M. A. Walker, “An Application of Reinforcement Learning to Dialogue Strategy Selection in a Spoken Dialogue System for Email,” *J. Artif. Intell. Res.*, vol. 12, pp. 387–416, 2000.

[76] H. Cuay and N. Dethlefs, “Hierarchical Reinforcement Learning and Hidden Markov Models for Task-Oriented Natural Language Generation,” in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics:shortpapers*, 2011, pp. 654–659.

[77] T. Jaakkola, S. S. P., and M. I. Jordan, “Reinforcement Learning Algorithm for Partially Observable Markov Decision Problems,” in *MIT Press*, Cambridge, Massachusetts: MIT Press, 1995, pp. 345–352.

[78] A. K. McCallum, “Reinforcement Learning with Selective Perception and Hidden State,” University of Rochester, 1995.

[79] W. W. Cohen and S. E. Fienberg, “A Comparison of String Metrics for Matching Names and Records,” in *KDD Workshop on Data Cleaning and Object Consolidation*, 2003.

[80] W. Heeringa, “Measuring Dialect Pronunciation Differences using Levenshtein Distance,” University of Groningen, 2004.

[81] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, vol. 10, no. 8. pp. 707–710, 1966.

[82] J. Quante and R. Koschke, “Dynamic protocol recovery,” *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 219–228, 2007.

[83] V. G. Timkovskii, “Complexity of common subsequence and supersequence problems and related problems,” *Cybernetics*, vol. 25, no. 5, pp. 565–580, Sep. 1989.

[84] R. Hamming, “Error detecting and error correcting codes,” *Bell Syst. Tech. J.*, vol. 2, no. 29, 1950.

[85] M. A. Jaro, “Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida,” *J. Am. Stat. Assoc.*, vol. 84, no. 406, pp. 414–420, 1989.

[86] L. J. Lin, “Reinforcement Learning for Robots Using Neural Networks,” Carnegie Mellon University, 1993.

[87] O. Abul, F. Polat, and R. Alhajj, “Multiagent Reinforcement Learning Using Function,” *IEEE Trans. Syst. Man, Cybern. Part C Appl. Rev.*, vol. 30, no. 4, pp. 485–497, 2000.

[88] K. J. Lang, B. A. Pearlmutter, and R. A. Price, “Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm,” in *Proceedings of the 4th International Colloquium on Grammatical Inference*, 1998, pp. 1–12.

[89] O. Tramasco and S. Bauer, “Package ‘RHmm.’” 2013.

[90] H. Cohen and S. Maoz, “Have We Seen Enough Traces ?,” in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 93–103.

[91] N. Busany and S. Maoz, “Behavioral Log Analysis with Statistical Guarantees,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 898–901.

[92] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora, “A Hidden Markov Model to Detect Coded Information Islands in Free Text,” in *13th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2013, pp. 157–166.

[93] L. Baum, T. Petrie, G. Soules, and N. Weiss, “A maximization technique occuring in the statistical analysis of probabilistic functions of Markov chains,” *Annu. Math. Stat.*, vol. 41, no. 1, pp. 164–171, 1970.

[94] S. Koenig and R. G. Simmons, “Complexity Analysis of Real-Time Reinforcement Learning,” *Proc. AAAI Conf. Artif. Intell.*, pp. 99–105, 1993.

[95] P. Zech, M. Felderer, P. Kalb, and R. Breu, “A Generic Platform for Model-Based Regression Testing,” in *Technologies for Mastering Change*, 2012, pp. 112–126.

[96] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, “Web Usage Mining : Discovery and Applications of Usage Patterns from Web Data,” *ACM SIGKDD Explor. Newsl.*, vol. 1, no. 2, pp. 12–23, 2000.

[97] R. R. Sarukkai, “Link prediction and path analysis using Markov chains,” *Comput. Networks*, vol. 33, pp. 377–386, 2000.

[98] Q. Yang and H. H. Zhang, “Web-Log Mining for Predictive Web Caching,” *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1050–1053, 2003.

[99] G. Greco, A. Guzzo, L. Pontieri, and D. Sacca, “by Clustering Log Traces,” *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 8, pp. 1010–1027, 2006.

[100] T. Zhu, R. Greiner, and H. Gerald, “Learning a Model of a Web User ’ s Interests,” *Int. Conf. User Model.*, pp. 65–75, 2003.

[101] R. W. White, P. Bailey, and L. Chen, “Predicting User Interests from Contextual Information,” in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009, pp. 363–370.

[102] F. M. Facca and P. L. Lanzi, “Mining interesting knowledge from weblogs : a survey,” *Data Knowl. Eng.*, vol. 53, pp. 225–241, 2005.

[103] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM : Probabilistic Model Checking for Performance and Reliability Analysis,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 4, pp. 40–45, 2009.

[104] B. Clifton, *Advanced Web Metrics with Google Analytics*. Alameda,CA , USA: SYBEX Inc., 2008.

[105] S. Schechter, K. Murali, and M. D. Smith, “Using path profiles to predict HTTP requests,” *Comput. Networks ISDN Syst.*, vol. 30, pp. 457–467, 1998.

[106] F. Chierichetti and R. Kumar, “Are Web Users Really Markovian ?,” in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 609–618.

[107] C. Baier and J. Katoen, *Principles of Model Checking*. London: The MIT Press, 2008.

[108] M. Schur, A. Roth, and A. Zeller, “Mining Workflow Models from Web Applications,” *IEEE Trans. Softw. Eng.*, vol. 5589, no. MAY, pp. 1–1, 2015.

[109] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, “Automatically Generating Test Cases for Specification Mining,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 243–257, 2012.

[110] A. Mesbah, A. van Deursen, and S. Lenselink, “Crawling Ajax-based Web applications through dynamic analysis of user interface state changes,” *ACM Trans. Web*, vol. 6, no. 1, pp. 1–30, 2012.

[111] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability,” *Form. Asp. Comput.*, vol. 6, no. 5, pp. 512–535, 1994.

[112] M. Kwiatkowska, G. Norman, and D. Parker, “Stochastic model checking,” *Form. methods Perform. ...*, pp. 220–270, 2007.

[113] J. Jiang, X. Song, N. Yu, and C. Lin, “FoCUS : Learning to Crawl Web Forums,” *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 6, pp. 1293–1306, 2013.

[114] C. Spearman, “The Proof and Measurement of Association between Two Things,” *Am. J. Psychol.*, vol. 15, no. 1, pp. 72–101, 1904.

[115] L. Antwarg, L. Rokach, and B. Shapira, “Attribute-driven Hidden Markov Model Trees for Intention Prediction,” *IEEE Trans. Syst. Man, Cybern. Part C (Applications Rev.*, vol. 42, no. 6, pp. 1103–1119, 2012.

[116] K. Engelbrecht, F. Gödde, F. Hartard, H. Ketabdar, K. Engelbrecht, F. Goedde, H. Ketabdar, and S. M. De, “Modeling User Satisfaction with Hidden Markov Models,” in *Proceedings of the SIGDIAL 2009 Conference: The 10th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2009, no. September, pp. 170–177.

[117] L. C. Stuart, “User Modeling via Machine Learning and Rule- Based Reasoning to Understand and Predict Errors in Survey Systems,” 2013.

[118] J. Ruvini, “Adapting to the User’s Internet Search Strategy on Small Devices,” pp. 284–286, 2003.

[119] B. Mobasher, R. Cooley, and J. Srivastava, “Automatic Personalization Based on Web Usage Mining Architecture for Usage-based Web Personalization Mining Usage Data for

Web Personalization," *Commun. ACM*, vol. 43, no. 8, pp. 142–151, 2000.

[120] M. Virvou, C. Troussas, and E. Alepis, "Machine learning for user modeling in a multilingual learning system," in *International Conference on Information Society (i-Society)*, 2012, pp. 292–297.

[121] M. Pennacchiotti and A. Popescu, "A Machine Learning Approach to Twitter User Classification," in *Proceedings of the Fifth International AAAI Conference on Weblogs and Social Media*, 2010, pp. 281–288.

[122] J. E. Beck, P. Jia, J. Sison, and J. Mostow, "Predicting Student Help-Request Behavior in an Intelligent Tutor for Reading," in *User Modeling 2003: 9th International Conference, UM 2003 Johnstown, PA, USA, June 22--26, 2003 Proceedings*, P. Brusilovsky, A. Corbett, and F. de Rosis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 303–312.

[123] A. Jameson, "Adaptive Interfaces and Agents," in *The human-computer interaction handbook*, 2002, pp. 305–330.

[124] D. Henriques, P. Zuliani, and E. M. Clarke, "Statistical Model Checking for Markov Decision Processes," *Int. Conf. Quant. Eval. Syst.*, pp. 17–20, 2012.

[125] P. Shitole and M. A. Potey, "Survey of User Modeling Techniques with Specific Emphasis on Considering Demographic Attributes," vol. 3, no. 12. pp. 1366–1370, 2014.

[126] F. Lin and L. Wenyin, "User Modeling for Efficient Use of Multimedia Files."

[127] C. Baier, E. M. Clarke, V. Hartonas-garmhausen, M. Kwiatkowska, and M. Ryan, "Symbolic Model Checking for Probabilistic Processes," in *International Colloquium on*

Automata, Languages, and Programming, 1997, pp. 430–440.

[128] F. Ciesinski and M. Gr, "On Probabilistic Computation Tree Logic," *Valid. Stoch. Syst.*, pp. 147–188, 2004.

[129] R. W. White, S. T. Dumais, and J. Teevan, "Characterizing the Influence of Domain Expertise on Web Search Behavior," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, 2009, pp. 132–141.

[130] H. Terai, "Differences between Informational and Transactional Tasks in Information Seeking on the Web," in *Proceedings of the second international symposium on Information interaction in context*, 2008, pp. 152–159.

[131] M. Claypool, D. Brown, P. Le, and M. Waseda, "Inferring User Interest," *IEEE Internet Comput.*, vol. 5, no. 6, pp. 32–39, 2001.

[132] D. W. Oard and J. Kim, "Implicit Feedback for Recommender Systems," in *Proceeding of 5th DELOS Workshop on Filtering and Collaborative Filtering*, 1998, pp. 31–36.

[133] S. Gündüz and M. T. Ozsu, "Recommendation Models for User Accesses to Web Pages (Invited Paper)," in *Artificial Neural Networks and Neural Information Processing*, 2003, pp. 1003–1010.

[134] F. Khalil, J. Li, and H. Wang, "Integrating Recommendation Models for Improved Web Page Prediction Accuracy," in *Proceedings of the thirty-first Australasian conference on Computer science*, 2008, vol. 74, pp. 91–100.

[135] U. Farooq, "Model based test suite minimization using metaheuristics," Cowan University, 2011.

[136] S. Parsa and A. Khalilian, “On the optimization approach towards test suite minimization approach,” *Int. J. Softw. Eng. Its Appl.*, vol. 4, no. 1, pp. 15–28, 2010.

[137] D. Jeffrey and N. Gupta, “Improving fault detection capability by selectively retaining test cases during test suite reduction,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 2, pp. 108–123, 2007.

[138] P. G. Frankl, G. Rothermel, K. Sayre, and F. I. Vokolos, “An empirical comparison of two safe regression test selection techniques,” in *International Symposium on Empirical Software Engineering (ISESE)*, 2003, vol. 195–204, pp. 195–204.

[139] G. Rothermel, R. Untch, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, 2001.

[140] S. Elbaum, G. Rothermel, A. G. Malishevsky, and S. Member, “Test case prioritization : A family of empirical studies test case prioritization,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, 2002.

[141] Z. Zhang, W. K. Chan, and T. H. Tse, “Adaptive Random Test Case Prioritization,” in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009, no. Ase, pp. 1–3.

[142] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An empirical Study of the effects of minimization on the fault detection capabilities of test suites,” in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 34–43.

[143] G. Rothermel and D. Hall, “A safe , effcient regression test selection technique,” *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, 1997.

[144] H. Hemmati, A. Arcuri, and L. Briand, “Achieving scalable model-based testing through test case diversity,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 1–42, Feb. 2013.

[145] H. Mei, S. Member, D. Hao, L. Zhang, S. Member, L. Zhang, J. Zhou, G. Rothermel, and I. C. Society, “A Static Approach to Prioritizing JUnit Test Cases,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1258–1275, 2012.

[146] S. Yoo and M. Harman, “Regression Testing Minimisation , Selection and Prioritisation : A Survey,” *Softw. TESTING, Verif. Reliab.*, vol. 7, pp. 1–60, 2007.

[147] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, *Static test case prioritization using topic models*. Springer Science, 2012.

[148] R. Dev, A. Jaaskelainen, and M. Katara, “Model-based GUI testing : Case smartphone camera and messaging development,” *Adv. Comput.*, vol. 85, pp. 65–122, 2012.

[149] A. T. Endo and A. Simao, “Model-Based Testing of Service-Oriented Applications via State Models,” in *IEEE International Conference on Services Computing*, 2011, pp. 432–439.

[150] A. M. Memon, “An event-flow model of GUI-based applications for testing,” *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, 2007.

[151] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, “AutoBlackTest : A tool for automatic black-box testing,” in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1013–1015.

[152] H. Reza, S. Endapally, and E. Grant, “A model-based approach for testing GUI using

hierarchical predicate transition nets," in *Fourth International Conference on Information Technology*, 2007, pp. 336–370.

[153] A. M. Memon, D. R. Hackner, and G. U. I. T. Field, "Test case generator for GUITAR," in *International Journal of Software Engineering*, 2008.

[154] H. S. Chang, "Reinforcement learning with supervision by combining multiple learnings and expert advices," in *American Control Conference*, 2006, pp. 159–161.

[155] R. Davoodi and B. J. Andrews, "Computer simulation of FES standing up in paraplegia: a self-adaptive fuzzy controller with reinforcement learning.," *IEEE Trans. Rehabil. Eng.*, vol. 6, no. 2, pp. 151–61, Jun. 1998.

[156] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intell. Transp. Syst.*, vol. 4, no. 2, pp. 128–135, 2010.

[157] M. McPartland and M. Gallagher, "Reinforcement Learning in First Person Shooter Games," *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 1, pp. 43–56, Mar. 2011.

[158] M. M. Yin and J. T. L. Wang, "Effective hidden Markov models for detecting splicing junction sites in DNA sequences," *Inf. Sci. (Ny).*, vol. 139, no. 1–2, pp. 139–163, 2001.

[159] K. Lee, H. Hon, M. Hwang, and X. Huang, "Speech recognition using hidden Markov model: A CMU prespective," *Speech Comun.*, vol. 9, no. 5–6, pp. 497–508, 1990.

[160] K. Aas and L. Eikvil, "Text page recognition using Grey-level features and hidden Markov models," *Pattern Recognit.*, vol. 29, no. 6, pp. 977–985, Jun. 1996.

[161] J. D. Williams and S. Young, “Partially observable Markov decision processes for spoken dialog systems,” *Comput. Speech Lang.*, vol. 21, no. 2, pp. 393–422, Apr. 2007.

[162] P. Blunsom, “Hidden Markov Models.” pp. 1–7, 2004.

[163] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, “A Study of Effective Regression Testing in Practice,” in *Proceedings of the 8th IEEE In ternational Symposium on Software Reliability Engineering*, 1997, pp. 264–274.

[164] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, “Design and analysis of GUI test-case prioritization using weight-based methods,” *J. Syst. Softw.*, vol. 83, no. 4, pp. 646–659, Apr. 2010.

[165] D. Di Nardo, N. Alshahwan, and L. Briand, “Coverage-Based Test Case Prioritisation : An Industrial Case Study,” in *Proceeding of Sixth International Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 302–311.

[166] M. a. T. Ho, Y. Yamada, and Y. Umetani, “An HMM-based temporal difference learning with model-updating capability for visual tracking of human communicational behaviors,” in *Proceedings of Fifth IEEE International Conference on Automatic Face Gesture Recognition*, 2002, pp. 170–175.

[167] J. Kabudian, M. R. Meybodi, and M. M. Homayounpour, “Applying continuous action reinforcement learning automata(CARLA) to global training of hidden Markov models,” in *International Conference on Information Technology: Coding and Computing.*, 2004, p. 638–642 Vol.2.

[168] G. Becce, L. Mariani, O. Riganelli, and M. Santoro, “Extracting widget descriptions from

GUIs," in *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2012, pp. 347–361.

[169] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *33rd International Conference onSoftware Engineering (ICSE)*, 2011, pp. 1–10.

[170] P. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge: Cambridge University Press, 2010.

[171] F. Galton, *Natural Inheritance*. Macmillan, 1889.

[172] J. A. Rice, *Mathematical Statistics and Data Analysis*. Ducbury, 1994.

[173] H. Hsu and P. A. Lachenbruch, "Paired t Test," *Wiley Encycl. Clin. Trials*, pp. 1–3, 2008.

[174] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associate Publishers, 1988.

[175] M. Robinson and P. Vorobiev, *Swing: A Fast Pased Guide with Production-Quality Code Examples*. Manning Publications, 1999.

[176] G. S. Semmel and D. G. Linton, "Determining optimal testing times for Markov chain usage models," in *IEEE Proceedings Southeastcon*, 1998, pp. 1–4.

[177] A. L. White and Sjorgen, "Markov chains for testing redundant software," in *Proceedings of Reliability and Maintainability Symposium*, 1988, pp. 426–433.

[178] F. Zhen and P. Chenglian, "A system test methodology based on the Markov chain usage model," in *Proceedings of the 8th International Conference on Computer Supported*

Cooperative Work in Design, 2004, vol. 96, pp. 160–165.

[179] C. G. Bai, Q. P. Hu, M. Xie, and S. H. Ng, "Software failure prediction based on a Markov Bayesian network model," *J. Syst. Softw.*, vol. 74, no. 3, pp. 275–282, Feb. 2005.

[180] S. Mirarab and L. Tahvildari, "A Prioritization Approach for Software," in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, 2007, pp. 276–290.

[181] S. Mirarab and L. Tahvildari, "An Empirical Study on Bayesian Network-based Approach for Test Case Prioritization," in *International Conference on Software Testing, Verification, and Validation*, 2008, pp. 278–287.

[182] J. Hao and E. Mendes, "Usage-based statistical testing of web applications," in *Proceedings of the 6th international conference on Web engineering - ICWE '06*, 2006, pp. 17–24.

[183] K.-Y. Cai, Y.-C. Li, and W.-Y. Ning, "Optimal software testing in the setting of controlled Markov chains," *Eur. J. Oper. Res.*, vol. 162, no. 2, pp. 552–579, Apr. 2005.

[184] A. Feliachi and H. Le Guen, "Generating transition probabilities for automatic model-based test generation," in *3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010, no. X, pp. 99–102.

[185] D. Ariu, R. Tronci, and G. Giacinto, "HMMPayl: An intrusion detection system based on Hidden Markov Models," *Comput. Secur.*, vol. 30, no. 4, pp. 221–241, Jun. 2011.

[186] Y. T. Yu and M. F. Lau, "Fault-based test suite prioritization for specification-based testing," *Inf. Softw. Technol.*, vol. 54, no. 2, pp. 179–202, Feb. 2012.

[187] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, “The Effects of Time Constraints on Test Case Prioritization : A Series of Controlled Experiments,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 593–617, 2010.

[188] A. Valdi, E. Lever, S. Benefico, D. Quarta, S. Zanero, and F. Maggi, “Scalable Testing of Mobile Antivirus Applications,” *Computer (Long. Beach. Calif).*, vol. 48, no. 11, pp. 60–68, 2015.

[189] J. Gao, X. Bai, W. T. Tsai, and T. Uehara, “Mobile Application Testing: A Tutorial,” *Computer (Long. Beach. Calif).*, vol. 47, no. 2, pp. 46–55, 2014.

[190] B. Jiang, X. Long, and X. Gao, “MobileTest: A tool supporting automatic black box test for software on smart mobile devices,” in *29th International Conference on Software Engineering, ICSE’07*, 2007, pp. 8–14.

[191] A. I. Wasserman, “Software Engineering Issues for Mobile Application Development,” *ACM Trans. Inf. Syst.*, pp. 1–4, 2010.

[192] S. R. Choudhary, A. Gorla, and A. Orso, “Automated Test Input Generation for Android: Are We There Yet?,” in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[193] D. Amalfitano, A. R. Fasolino, and P. Tramontana, “A GUI crawling-based technique for android mobile application testing,” in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, 2011, pp. 252–261.

[194] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, “RERAN: Timing- and touch-sensitive

record and replay for Android," *Proc. - Int. Conf. Softw. Eng.*, pp. 72–81, 2013.

[195] M. Hesenius, T. Griebe, S. Gries, and V. Gruhn, "Automating UI Tests for Mobile Applications with Formal Gesture Descriptions," in *Proceedings of MobileHCI'14*, 2014, pp. 213–222.

[196] C. J. Hunt, G. Brown, and G. Fraser, "Automatic testing of natural user interfaces," *IEEE 7th Int. Conf. Softw. Testing, Verif. Valid.*, pp. 123–132, 2014.

[197] B. Kirubakaran and V. Karthikeyani, "Mobile application testing — Challenges and solution approach through automation," in *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, 2013, pp. 79–84.

[198] A. M. Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging Existing Tests in Automated Test Generation for Web Applications," in *29th ACM/IEEE international conference on Automated software engineering (ASE)*, 2014.

[199] M. Ermuth and M. Pradel, "Monkey See , Monkey Do : Effective Generation of GUI Tests with Inferred Macro Events," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 82–93.

[200] I. Satoh, "A testing framework for mobile computing software," *IEEE Trans. Softw. Eng.*, vol. 29, no. 12, pp. 1112–1121, 2003.

[201] D. Franke and C. Weise, "Providing a software quality framework for testing of mobile applications," in *4th IEEE International Conference on Software Testing, Verification, and Validation*, 2011, pp. 431–434.

[202] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De, U. Federico, and I. I. Napoli,

"Using GUI Ripping for Automated Testing of Android Applications," in *Proceedings of the 27th IEEE international conference on Automated Software Engineering*, 2012, pp. 258–261.

[203] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceeding of the 6th international workshop on Automation of software test - AST '11*, 2011, no. Section 4, p. 77.

[204] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012, p. 100.

[205] C. M. Prathibhan, A. Maliani, N. Venkatesh, and K. Sundarakantham, "An automated testing framework for testing android mobile applications in the cloud," in *IEEE International Conference on Advanced Communication Control and Computing Teclmologies (ICACCCT)*, 2014, no. 978, pp. 1216–1219.

[206] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013.

[207] S. Malek, "EvoDroid : Segmented Evolutionary Testing of Android Apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 599–609.

[208] M. Linares-v, M. White, C. Bernal-c, K. Moran, and D. Poshyvanyk, "Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios," in *roceedings of*

the 12th Working Conference on Mining Software Repositories (MSR), 2015.

[209] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android : Are We Really There Yet in an Industrial Case ?," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 3–8.

[210] K. Mao, M. Harman, and Y. Jia, "Sapienz : Multi-objective Automated Testing for Android Applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 94–105.

[211] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," *IEEE 7th Int. Conf. Softw. Testing, Verif. Valid.*, pp. 183–192, 2014.

[212] K. Moran, M. Linares-v, C. Bernal-c, C. Vendome, D. Poshyvanyk, and C. William, "Automatically Discovering , Reporting and Reproducing Android Application Crashes," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[213] J. An and K.-S. Hong, "Finger gesture-based mobile user interface using a rear-facing camera," in *2011 IEEE International Conference on Consumer Electronics (ICCE)*, 2011, pp. 303–304.

[214] E. S. Choi, W. C. Bang, S. J. Cho, J. Yang, D. Y. Kim, and S. R. Kim, "Beatbox music phone: Gesture-based interactive mobile phone using a tri-axis accelerometer," in *Proceedings of the IEEE International Conference on Industrial Technology*, 2005, vol.

2005, pp. 97–102.

[215] C. B. Park and S. W. Lee, “Real-time 3D pointing gesture recognition for mobile robots with cascade HMM and particle filter,” *Image Vis. Comput.*, vol. 29, no. 1, pp. 51–63, 2011.

[216] S. O. Hara, Y. M. Lui, and B. A. Draper, “Unsupervised Learning of Human Expressions , Gestures , and Actions,” in *IEEE International Conference on Automatic Face & Gesture Recognition and Workshops*, 2011, pp. 1–8.

[217] S. Gibet, N. Country, and J.-F. Kamp, *Lecture Notes in Artificial Intelligence*. 2005.

[218] D. Trabelsi, S. Mohammed, F. Chamroukhi, L. Oukhellou, and Y. Amirat, “Supervised and unsupervised classification approaches for human activity recognition using body-mounted sensors,” in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2012, no. April, pp. 25–27.

[219] W. J. Li, “A hybrid HMM / SVM classifier for motion recognition using μ IMU data A Hybrid HMM / SVM Classifier for Motion Recognition Using 3IMU Data *,” in *IEEE International Conference on Robotics and Bioimetics*, 2008, no. January, pp. 115–120.

[220] D. Trabelsi, S. Mohammed, F. Chamroukhi, L. Oukhellou, and Y. Amirat, “An Unsupervised Approach for Automatic Activity Recognition based on Hidden Markov Model Regression,” *IEEE Trans. Autom. Sci. Eng.*, vol. 10, no. 3, pp. 1–7, 2013.

[221] M. S. K. Gaikwad, “HMM Classifier for Human Activity Recognition,” *Comput. Sci. Eng. AN Int. J.*, vol. 2, no. 4, pp. 27–36, 2012.

[222] A. Mannini and A. M. Sabatini, “Accelerometry-Based Classification of Human Activities

Using Markov Modeling," *Comput. Intell. Neurosci.*, 2011.

[223] T. Yang and Y. Xu, "Hidden Markov Model for Gesture Recognition," Carnegie Mellon Universioty, 1994.

[224] R. Cilla, M. A. Patricio, A. Berlanga, and J. M. Molina, "Recognizing Human Activities from Sensors Using Hidden Markov Models Constructed by Feature Selection Techniques," *J. Algorithms*, vol. 2, pp. 282–300, 2009.

[225] O. Perez, M. Piccardi, G. Jesus, and J. M. Molina, "Comparison of Classifiers for Human Activity," in *Lecture Notes in Computer Science*, 2007, pp. 192–201.

[226] D. Kleppner and R. Kolenjow, *An Introduction to Mechanics*. Cambridge University Press, 2013.

[227] C. Nello and Shawe-Taylor John, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*, vol. 22, no. 2. 2001.

[228] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *Am. Stat.*, vol. 46, no. 3, pp. 175–185, 2007.

[229] C. i. Wang and S. Dubnov, "The Variable Markov Oracle: Algorithms for Human Gesture Applications," *IEEE Multimed.*, vol. 22, no. 4, pp. 52–67, 2015.

[230] Z. Moghaddam and M. Piccardi, "Training Initialization of Hidden Markov Models in Human Action Recognition," *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 2, pp. 394–408, 2014.

[231] J. B. MacQueen, "Some Methods for classification and Analysis of Multivariate

Observations," *5th Berkeley Symp. Math. Stat. Probab. 1967*, vol. 1, no. 233, pp. 281–297, 1967.

[232] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, 1987.

[233] T. Tunys, "Gesture detection and NFC for Android OS," Czech Technical University in Prague, 2012.

[234] J. Ravikiran, K. Mahesh, S. Mahishi, R. Dheeraj, S. Sudheender, and N. V Pujari, "Finger detection for sign language recognition," in *International MultiConference of Engineers and Computer Scientists*, 2009, vol. I, pp. 0–4.

[235] R. Cross, "Enhancing the Bounce of a Ball," *Am. Assoc. Phys. Teach.*, vol. 48, no. 7, p. 450, 2010.

[236] M. Torchiano, "R Package: 'effsize,'" 2015.

[237] A. Jagannatam, "Mersenne Twister – A Pseudo Random Number Generator and and its Variants," 2008.

[238] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal voronoi tessellations-a new approach to random testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 163–183, 2013.

[239] A. Arcuri and L. Briand, "Adaptive Random Testing : An Illusion of Effectiveness ?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 265–275.

[240] S. Yoo, M. Harman, and S. Ur, "GPGPU Test Suite Minimisation : Search Based

Software Engineering Performance Improvement Using Graphics Cards," *Empir. Softw. Eng.*, vol. 18, no. 3, pp. 550–593, 2013.

[241] J. Bach, "Exploratory Testing Explained," *Den Bosch UTN Publ.*, pp. 253–265, 2003.

[242] M. Dirk, L. Begona, van der P. K. Rob, and W. Alan, *Software Quality and Software Testing in Internet Times (High-tech software quality management)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[243] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[244] A. Beer and R. Ramler, "The Role of Experience in Software Testing Practice," in *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications,* 2008, pp. 258–265.

[245] J. Itkonen, M. V Mäntylä, and C. Lassenius, "Defect Detection Efficiency : Test Case Based vs . Exploratory Testing," in *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 61–70.

[246] L. Williams, G. Kudrjavets, and N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft 1," in *20th International Symposium on Software Reliability Engineering*, 2009, pp. 81–89.

[247] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive Random Testing," in *Annual Asian Computing Science Conference*, 2004, pp. 320–329.

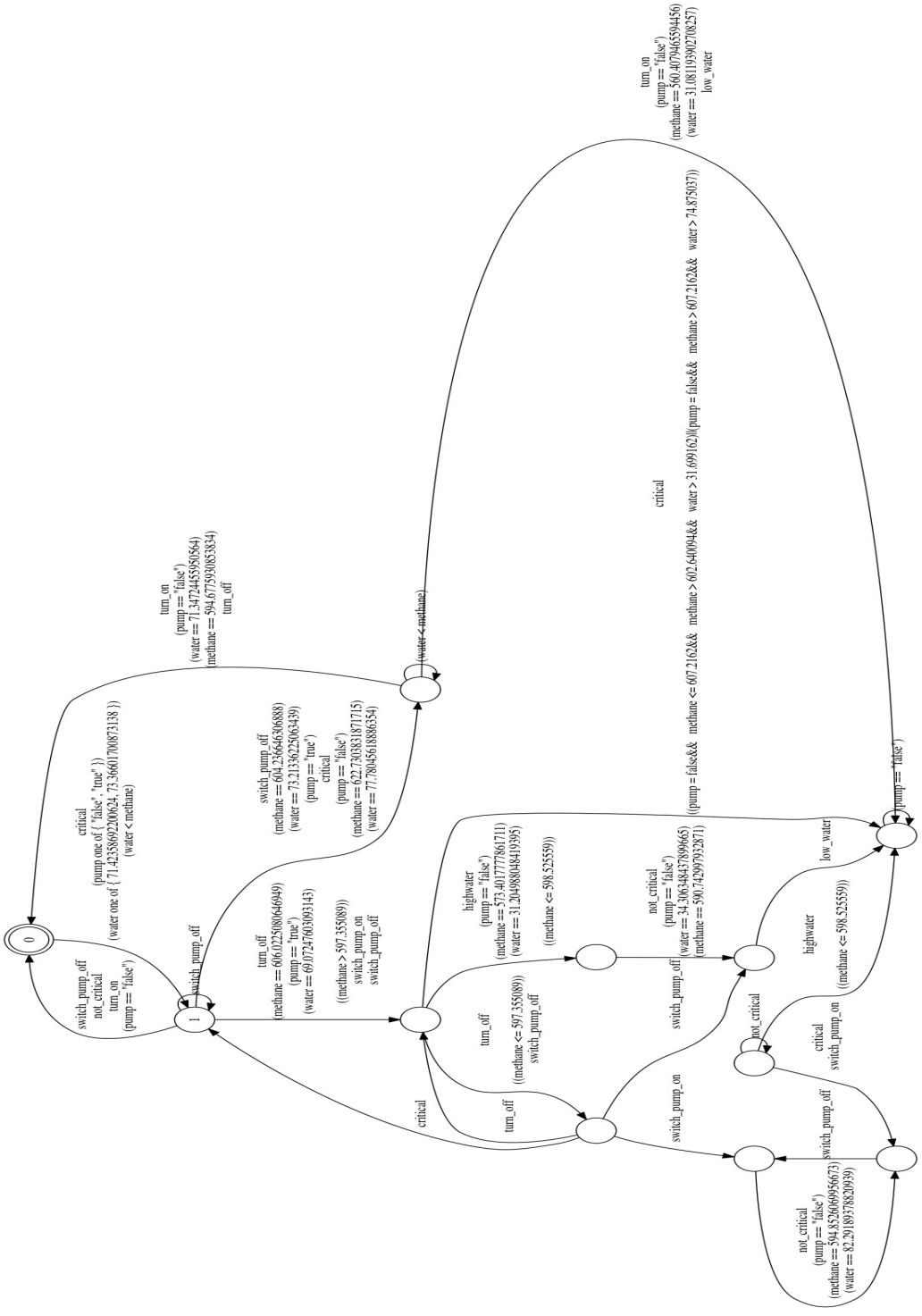
[248] R. Singh, J. Miller, and M. Smith, "Random Border Centroidal Voronoi Tessellations (RBCVT) Improved by Parallel Selection using Regular Sampling," *IEEE Trans. Reliab.*

[249] S. H. Houmb, “Method, device and computer program for monitoring an industrial control system,” 2014.

[250] R. Ihaka and R. Gentleman, “R: A Language for Data Analysis and Graphics,” *J. Comput. Graph. Stat.*, vol. 5, no. 3, pp. 299–314, 1996.

Appendices

Appendix A - The overview of the EFSA generated by ReHMM



Appendix B - The Results of Applying Inference Techniques on 7 Different Case Studies for k=5 and k=10

Table 46. The Results of Applying Inference Techniques on Poolboy, SMTPTransport, Resource Locker and Frequency Server, for k=5,10 (in terms of BCR)

	G=0	Poolboy			SMTPTransport			Resource Locker			Frequency Server		
k	***Inference Algorithm***	***Min BCR***	***Max BCR***	***Median BCR***	***Min BCR***	***Max BCR***	***Median BCR***	***Min BCR***	***Max BCR***	***Median BCR***	***Min BCR***	***Max BCR***	***Median BCR***
5	ReHMM	0.66	0.72	0.705	0.89	0.92	0.91	0.71	0.83	0.76	0.73	0.9	0.835
	Sk-strings	0.63	0.66	0.66	0.58	0.66	0.645	0.6	0.69	0.645	0.58	0.68	0.655
	Bayes	0.64	0.7	0.64	0.68	0.72	0.7	0.55	0.68	0.67	0.55	0.75	0.68
	JRIP	0.5	0.7	0.6	0.68	0.8	0.71	0.55	0.73	0.62	0.55	0.65	0.615
	AdaBoost	0.58	0.72	0.6	0.62	0.81	0.67	0.6	0.74	0.625	0.65	0.8	0.79
	J48	0.53	0.61	0.6	0.6	0.7	0.685	0.62	0.66	0.63	0.55	0.72	0.675
10	ReHMM	0.66	0.8	0.71	0.9	0.95	0.91	0.75	0.87	0.77	0.77	0.96	0.845
	Sk-strings	0.65	0.69	0.66	0.55	0.66	0.645	0.65	0.69	0.66	0.58	0.68	0.66
	Bayes	0.64	0.7	0.64	0.68	0.75	0.71	NA	NA	NA	NA	NA	NA
	JRIP	0.5	0.69	0.6	0.68	0.83	0.715	0.55	0.8	0.63	0.6	0.65	0.635
	AdaBoost	0.59	0.75	0.62	0.7	0.81	0.72	0.62	0.85	0.64	0.75	0.88	0.79
	J48	0.59	0.63	0.605	0.6	0.72	0.7	0.61	0.69	0.65	0.6	0.72	0.67

Table 47. The Results of Applying Inference Techniques on Signature, StringTokenizer and Socket, for k=5,10 (in terms of BCR)

	G=0	**Signature**			**StringTokenizer**			**Socket**		
k	***Inference Algorithm***	***Min BCR***	***Max BCR***	***Median BCR***	***Min BCR***	***Max BCR***	***Median BCR***	***Min BCR***	***Max BCR***	***Median BCR***
5	ReHMM	0.7	0.95	0.74	0.7	0.95	0.74	0.7	0.95	0.74
	Sk-strings	0.55	0.73	0.635	0.55	0.73	0.635	0.55	0.73	0.635
	Bayes	NA	NA	NA	NA	NA	NA	NA	NA	NA
	JRIP	0.44	0.77	0.55	0.44	0.77	0.55	0.44	0.77	0.55
	AdaBoost	0.45	0.86	0.51	0.45	0.86	0.51	0.45	0.86	0.51
	J48	0.44	0.8	0.49	0.44	0.8	0.49	0.44	0.8	0.49
10	ReHMM	0.73	0.95	0.77	0.73	0.95	0.77	0.73	0.95	0.77
	Sk-strings	0.55	0.73	0.675	0.55	0.73	0.675	0.55	0.73	0.675
	Bayes	NA	NA	NA	NA	NA	NA	NA	NA	NA
	JRIP	0.5	0.79	0.545	0.5	0.79	0.545	0.5	0.79	0.545
	AdaBoost	0.6	0.9	0.64	0.6	0.9	0.64	0.6	0.9	0.64
	J48	0.48	0.8	0.525	0.48	0.8	0.525	0.48	0.8	0.525