AN AUTOMATED BLACK-BOX MODEL DISCOVERY WITH SYSTEMATIC SAMPLING ON ANDROID MOBILE APPLICATIONS

by
ÖMER KORKMAZ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
August 2020

AN AUTOMATED BLACK-BOX MODEL DISCOVERY WITH SYSTEMATIC SAMPLING ON ANDROID MOBILE APPLICATIONS

APPROVED BY:

Assoc. Prof. Dr. Cemal YILMAZ.. [illegible]
(Thesis Supervisor)

Assoc. Prof. Dr. Hüsnü YENIGUN.. [illegible]

Assoc. Prof. Dr. Mert OZKAYA.. [illegible]

DATE OF APPROVAL: 18/08/2020

ABSTRACT

AN AUTOMATED BLACK-BOX MODEL DISCOVERY WITH SYSTEMATIC SAMPLING ON ANDROID MOBILE APPLICATIONS

ÖMER KORKMAZ

COMPUTER SCIENCE AND ENGINEERING M.Sc. THESIS, AUGUST 2020

Assoc. Prof. Cemal Yılmaz

Keywords: automated model discovery, systematic sampling, covering arrays, combinatorial testing

Clients progressively depend on mobile applications for computational needs. With the popularity of Google Android and the rise of interest in Android devices, Android applications have been valuable and millions of mobile applications have increased the importance and demand of test processes in the complex systems. Since the applications had well-developed strong conditions that need to be tested, automation in the testing has played a significant role. Many types of researches have primarily focused on different model discovery strategies to be used for different purposes (e.g., test generation, bug detection). However, they were not used systematically for testing of mobile applications. We present a tool that provides an automated black-box model discovery by applying systematic sampling to build a model of an application dynamically for different uses. The approach includes two purposes: (1) discovering the model of an application by providing systematic sampling, and (2) predicting guard conditions of the discovered model. The results of our experiments have confirmed the ability of the approach to acquire higher code coverage and the accuracy of predicted guard conditions than existing approaches.

ÖZET

ANDROID UYGULAMALARDA SISTEMATIK ÖRNEKLEME ILE OTOMATIKLEŞTIRILMIŞ MODEL KEŞIF YAKLAŞIMI

ÖMER KORKMAZ

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ, YÜKSEK LİSANS TEZİ, AĞUSTOS 2020

Tez Danışmanı: Doç. Dr. Cemal Yılmaz

Anahtar Kelimeler: model keşif, sistematik örnekleme, kapsayan diziler, kombinatoryal test

İstemciler, hesaplama ihtiyaçları için mobil uygulamalara giderek daha fazla güveniyor. Google Android'in popülaritesi ve Android cihazlara olan ilginin artması ile Android uygulamaları değerli hale geldi ve milyonlarca mobil uygulama, karmaşık sistemlerde test süreçlerinin önemini ve talebini artırdı. Uygulamalar test edilmesi gereken iyi geliştirilmiş güçlü koşullara sahip olduğundan, testteki otomasyon önemli bir rol oynamıştır. Birçok araştırma türü, öncelikle farklı amaçlar için kullanılacak farklı model keşif stratejilerine odaklanmıştır (örneğin, test oluşturma, hata algılama). Ancak, mobil uygulamaların test edilmesinde veya farklı amaçlar için kullanılabilecek olan uygulama modeli sistematik örnekleme ile oluşturulmadı. Farklı kullanımlar için dinamik olarak bir uygulama modeli oluşturmak üzere sistematik örnekleme uygulayarak otomatik bir kara kutu modeli keşfi sağlayan bir araç sunuyoruz. Yaklaşım iki amaç içerir: (1) sistematik örnekleme sağlayarak bir uygulamanın modelini keşfetmek ve (2) keşfedilen modelin koruma koşullarını tahmin etmek. Deneylerimizin sonuçları, yaklaşımın mevcut yaklaşımlardan daha yüksek kod kapsamı ve koruma koşullarının doğruluğunu elde etme yeteneğini doğruladı.

ACKNOWLEDGEMENTS

First, I wish to express my sincere appreciation to my thesis supervisor Cemal Yılmaz for continuous support throughout my master's degree. I am grateful for his precious guidance and having the chance to work with him.

I am also grateful to the jury members Hüsnü Yenigün and Mert Özkaya for their valuable time.

I wish to extend my special thanks to my family for their endless support, love, and effort. They always trusted and encouraged me.

I would like to thank my all friends for their supports and the wonderful moments that we shared. Thanks to their helpfulness and sincerity, my adaption to the new environment became easier.

to my family

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Mobile devices have been increasingly becoming smarter and more powerful. Therefore, mobile applications in many areas such as education, health, economy, or management are used by millions of people on a daily basis. As the failures in the field may have some severe consequences, these applications need to be tested thoroughly.

One frequently used approach for this purpose is *model-based testing* (D. Amalfitano & Memon., 2015,1; Nariman Mirzaei & Malek, 2016; S. Hao & Govindan., 2014; W. Yang & Xie., 2013). In model-based testing, given a model representing the behavior of the system under test (SUT), test cases are automatically generated typically by employing a structural coverage criterion, such as those based on state and transition coverage (Pradhan, 2019; Shafique, 2010) Many empirical studies strongly suggest that model-based testing is an efficient and effective approach for testing mobile applications (D. Amalfitano & Memon., 2015,1; Nariman Mirzaei & Malek, 2016; S. Hao & Govindan., 2014; W. Yang & Xie., 2013).

One down side of model-based testing, however, is that it takes as input the model of the SUT. As these models often need to be created manually and updated as the underlying codebase is modified, this greatly affects the practicality of the model-based approaches.

Many approaches have been proposed in the past to automatically discover the models of software systems, especially the mobile applications, so that these models can be used with various model-based testing approaches to automate testing from end to end (A. Machiry & Naik., 2013; AndroidMonkey, 2018; Claessen & Hughes, 2000; H. van der Merwe & Visser, 2014; R. Mahmood & Malek., 2014; S. Anand & Yang., 2012).

The existing model discovery approaches can be categorized into two main groups; random testing-based approaches (A. Machiry & Naik., 2013; AndroidMonkey, 2018; Claessen & Hughes, 2000) and somewhat-systematic testing-based approaches (H. van der Merwe & Visser, 2014; R. Mahmood & Malek., 2014; S. Anand & Yang., 2012). While the former approaches randomly generate user events, such tapping a button or swiping the current screen from top to bottom, the latter approaches aim to verify the requirements of the SUT in a somewhat-systematic manner.

One observation we make, however, is that even the somewhat-systematic testing approaches do not systematically take the interactions between various entities in the SUT into account, such as the interactions between the inputs fields.

In this thesis, we conjecture that systematically sampling the input space of applications by taking the interactions between various factors into account can greatly improve the effectiveness of model discovery.

To this end, we present an automated model discovery approach in this thesis. More specifically, we discover finite state machine-based models, where states represent distinct screens discovered during crawling and the transitions between states depict the transitions between screens. The transitions are further annotated with guard conditions (if any), which represent the conditions that must be satisfied in order to take the transitions.

To systematically, sample the input space of the SUT, we use a well-known combinatorial object for testing, called t-way covering arrays (D. M. Cohen & Patton, 1997). A t-way covering array, where t is often referred to as coverage strength, takes as input an *input space model*. The model includes a set of parameters, each of which takes its value from a discrete domain, together with inter-parameter constraints (if any), which invalidate certain combinations of parameter values. Given a model, a t-way covering array is a set of test cases (where each test case is comprised of values for all the parameters in the model), in which each possible combination of parameter values for every combination of t parameters appears at least once (C. Yilmaz & Koc, 2014; Nie & Leung, 2011).

The basic justification for using t-way covering arrays is that they (under certain assumptions) can efficiently and effectively exercise all program behaviors caused by the interaction of t or fewer parameters (D. M. Cohen & Patton, 1997). Therefore, they have been extensively used for software testing (R. Mahmood & Malek., 2014; S. Anand & Yang., 2012). In this work, however, we used them (and, to the best of our knowledge, for the first time) to systematically sample the input spaces for automated model discovery.

At a very high level, the proposed approach operates as follows: we start with an initially empty model. For each screen encountered during the discovery process, we first check to see if we have seen the screen or not. If the screen has not been seen before, we add a new state to the current model together with a transition from the previous state to the newly discovered state. Otherwise, i.e., if the screen has already been seen, we map the screen to a state in the model and add an appropriate transition (if not already included in the model) from the previous state to the current state.

We then determine the input fields for the current screen (i.e., user interface objects with which the end-users can interact). To systematically test the interactions between these input fields, we compute a t-way covering array by discretizing parameter domains using equivalence class partitioning (Bhat & Quadri, 2015; Fang & Li, 2015).

Note that t (i.e., the coverage strength) is an input parameter of the proposed approach and we compute a covering array for every distinct screen discovered. The covering array is computed when the screen is discovered for the first time. Then, every time the screen is encountered, we randomly pick a previously untested test case from the respective covering array, which, indeed, is comprised of the values to be fed to the input fields on the screen, and execute the test case. The crawling process terminates when the test cases in the covering arrays computed for all the discovered test cases are executed.

Once the crawling process terminates, the likely guard conditions are discovered. To this end, for each state, the test results obtained from the covering array computed for the state, are fed to a classification tree algorithm by using the destination states as classes. For every transition originating from the state, the output is a condition, which is comprised of parameters defined in the source state together with their values, representing the likely condition that needs to be satisfied before the transition can be taken.

To evaluate the proposed approach, we have conducted a number of empirical studies. In the first set of experiments, we used simulations to measure the sensitivity of various parameters on the performance of the proposed approach. We used simulations for this purpose as it was not possible for us to control these parameters in real subject applications. And, in the second set of experiments, we used a number of real applications as subject applications, which were, indeed, frequently used in related works (D. Amalfitano & Memon., 2015,1; S. Hao & Govindan., 2014; W. Yang & Xie., 2013).

The results of our experiments strongly suggest that our hypothesis holds true in practice that systematically testing the interactions between various factors can improve the performance of model discovery approaches. We have arrived at this conclusion by noting that compared to existing approaches the proposed approach increased state and transition coverage, structural code coverage, and the accuracy of the guard conditions.

The contributions of this thesis can be summarized as follows:

- an approach for automated model discovery by systematically sampling the interactions between factors that can affect program executions;
- a framework implementing the proposed approach;
- a series of experiments evaluating the proposed approach in a multi-faceted manner.

The remainder of the paper is organized as follows: Section **2** provides background information on the technologies and concepts used in the study, including covering arrays and equivalence classes; Section **3** discusses related work; Section **4** introduces the proposed approach; Section **5** presents the experiments carried out to evaluate the proposed approach; Section **6** provides more discussion (and experiments) on the practicality of the proposed approach; Section **7** discusses threats to validity; and Section **8** presents concluding remarks and discusses possible future works.

2. BACKGROUND

In this section, we give background information on Android platforms, Combinatorial Interaction Testing (CIT) including covering arrays, and the equivalence classes.

2.1 Android Platforms

Android platform developed by Google includes a full ARM processor-based Linux operating system, libraries related to the system, middleware, and a suite of pre-installed apps. It is optimized on running programs written in Java on the *Dalvik Virtual Machine (DVM)* (Dalvik, 2018). Android also provides one the most important feature called *Application Development Framework (ADF)* which is an API for the development of the apps and includes services that needed to build component types and GUI-based applications (AndroidDeveloper, 2018). Android framework is improved for the integrity and reusability of the components.

Android-based applications are generated by using XML manifest file which has significant information on Android platforms to manage the life cycle of the application. The information of the manifest file is mostly about the description of the components on the app based on configuration and architectural properties. Components are four different types (*Activities, Services, Broadcast Receivers, and Content Providers*). This file is produced for each activity of an application. An Activity typically corresponds to a screen the application that consists of components and layouts. The layout includes GUI elements (e.g., Button for triggering defined operations text and EditText for text inputs). Developers can control the behavior of each activity with the callbacks. Since the activities provide a user interface, services do not have any view that users can interact. However, they are used to run the operations in the background as an application component. Broadcast Receivers and Intents offers inter-process communication in the running time. They can be defined in the manifest file or the application code so that the app can react when the SMS is received or a new connection is available. In addition to the given information, structured data in the file system or a database is managed by a content provider. The applications may have their own content providers and share them with other applications by making a content provider available. All primary components are managed by the ADF, even activities and services. Each activity has its own XML file to store controls and components of an activity, as mentioned previously. The XML-formatted file plays an important role in our research as explained later.

2.2 Combinatorial Interaction Testing and Covering Arrays

Combinatorial Interaction Testing (CIT) is an effective testing technique to address the interaction of input parameters in software systems. CIT-based approaches systematically generate samples with the configuration space and test only selected configurations (C. Yilmaz & Koc, 2014; Nie & Leung, 2011). The approach takes a configuration space as input. The configuration space model includes a set of parameters such as configuration options, constraints that affect the configuration, and the settings of options. With the given configuration space model, CIT approaches to generate a set of configurations, known as *t-way covering array* (D. M. Cohen & Patton, 1997), which include all possible combinations of the option settings that appear at least once in every combination of t settings. After the generation of a covering array, the system is tested by executing the test cases in the covering array.

A covering array, denoted by $CA(N;t,k,s)$, is a Nxk array on s symbols that contains all t combinations of the symbols, since k is the number of options. As mentioned in a previous paragraph, a configuration space model includes a set of options $O = \{o_1, o_2, o_3, ..., o_n\}$ and their possible settings $V = \{v_1, v_2, v_3, ..., v_n\}$. In our approach, each option o stands for an input field on the screen and each setting v represents the discrete test values that need to be fed to the given option. In order to clearly understand how the covering arrays work, we present an example related to the approach.

In our illustrative example (*Table **2.1**, **2.2***), we suppose that an application has a screen and that screen has four editable input fields and one button. The test values of the input fields are produced. *Table **2.1*** demonstrates the input fields of a given Android screen represented as **options** and the test values, that need to be

Table 2.1 The Options and Settings of An Example

Options	Settings
O_1 (Input-1)	$< Summer, Winter >$
O_2 (Input-2)	$< Turkey, France, Italy >$
O_3 (Input-3)	$< Male, Female >$
O_4 (Input-4)	$< 18, 26, 45 >$

Table 2.2 An Illustrative Example of CA when $t = 2$

Input-1	Input-2	Input-3	Input-4
Summer	Turkey	Male	26
Winter	Turkey	Female	18
Winter	Turkey	Male	45
Summer	France	Female	18
Winter	France	Male	26
Summer	France	Female	45
Summer	Italy	Female	26
Winter	Italy	Male	18
Winter	Italy	Female	45

fed to the input fields, represented as **settings**. The first option O_1 gets two test values *Summer* and *Winter*, respectively. The second option O_2 takes three values *Turkey*, *France* and *Italy*. As a third option, it takes two values *Male* and *Female*. Last option gets three integer values *18, 26* and *45*. The strength of a covering array is represented by t. We set the strength of a covering array $t = 2$ to cover the interactions of the all *2-way* combinations of the options on a current Android screen. So, we may generate a covering array $CA(9; 2, 4, 3)$. Once the covering array is generated, the length of test cases is 9 and the generated configurations are shown in *Table **2.2***. If the strength of a covering array t is increased, we then generate more systematic samples as test cases.

2.3 Domain and Equivalence Classes

One assumption behind the combinatorial covering arrays is that each option takes its values from a discrete domain. In this section, we present two terminology called as *Domain* and *Equivalence Class* in the approach. As we explained more details in

Table 2.3 An example demonstrating the relationship between domains and equivalence classes

Domain	Equivalence Class	Values
Email	Valid Email	{test@hotmail.com, test@outlook.com}
Email	Invalid Email	{abc@def, test@!xyz}
Age	Infant	$[0,1]$
Age	Toddler	$[2,3]$
Age	Teenager	$[4,18]$
Age	Young Adult	$[19,25]$

Section 4, the goal of the domain and equivalence classes is to produce appropriate test values for the input fields of the screens so that the approach can generate the covering arrays for the screens using the input fields and their test values.

Domain of an input field is the set of all possible test values related to the input field. Also, a domain represents the information that explains what kind of test values should be fed for a given input field (e.g., email, age). In the approach, we divide each domain into the partitions represented as equivalence classes.

Equivalence Class is a partition or group of the test input values that can be used to derive the test cases and reduce the time required for testing. We create the domains and equivalence classes manually and pick a test value randomly from each equivalence class of a respective domain so that we can generate a covering array for a given screen.

As a structure of domain and equivalence classes, they are used together in the approach. If we do not know the domain of a given input field, the equivalence classes cannot be generated showing that the approach cannot produce discrete test values for the input fields. Therefore, the domain is detected by using the attributes, provided by Android (AndroidDeveloper, 2018), of each input field (e.g., resource-id, class, description). We then pick appropriate test values randomly from the equivalence classes of a respective domain. In this way, the test values of the input fields are produced to be covered by the covering arrays as systematic sampling.

*Table **2.3*** demonstrates the relationship between the domain and its equivalence classes. In the table, there are equivalence classes and discrete test values for each equivalence class of a given domain. For example, we consider that for a given input field, the input domain is an email. We partition the email domain into two equivalence classes as valid email and invalid email, respectively. Each equivalence class has test input values that covers the email domain. We pick a random value from each equivalence class. For a valid email test and an invalid email test of the given input field, we pick discrete test values from their equivalence classes as shown in *Table **2.3***. At the end, a given input field has two discrete test values taken from the invalid mail and valid mail equivalence classes, and these values cover the specifications of the email domain. The process of producing test values for the input fields is the same for the age domain as given in *Table **2.3*** and other domains, too. We will give more details and discuss the approach regarding the domain and equivalence classes detection in the *Section **4.3***.

3. RELATED WORK

As mobile applications have rapidly become more complex, the testing procedure has been crucial providing the development of high-quality applications. Since the complexity of the applications may cause more failures that users encounter, the researchers and practitioners have typically studied test automation approaches and tools. In the literature, the recent works of the automated testing have mostly focused on various exploration strategies (e.g., *random testing, model-based testing* and *systematic testing*) (A. Machiry & Naik., 2013; AndroidMonkey, 2018; D. Amalfitano & Memon., 2015,1; H. van der Merwe & Visser, 2014; L. Mariani & Santoro, 2012; R. Mahmood & Malek., 2014,1; S. Anand & Yang., 2012; S. Hao & Govindan., 2014; W. Yang & Xie., 2013; Z. Liu, 2010) for different purposes such as crawling and testing the applications, generating test cases or detecting the bugs.

In random testing (A. Machiry & Naik., 2013; AndroidMonkey, 2018; Claessen & Hughes, 2000; Hu & Neamtiu, 2011), which is known as one of the black-box software testing techniques where the software systems are tested with a random generation, proposed approaches generate the test inputs with a random strategy for mobile applications, indicating that they produce UI (*User Interface*) and system events as test cases. Since *Monkey* is the most frequently used tool as a random testing tool, it randomly generates a limited number of UI events with a black-box strategy. Also, Hu and Neamtiu (Hu & Neamtiu, 2011) proposed a random approach in order to generate GUI tests with Monkey (AndroidMonkey, 2018). On the other hand, *Dynodroid* (A. Machiry & Naik., 2013) is another random exploration tool using several features as Monkey. Basically, Dynodroid generates the test events randomly or the users produce test values manually. However, random testing approaches are not efficient to get high code coverage because of the random generation. In other words, random testing may not satisfy most of the all conditions that implemented in the code-base and not execute most of the code lines during testing. Thus, we focus on systematic sampling using the covering arrays.

As model-based testing approaches (D. Amalfitano & Memon., 2015,1; Nariman Mirzaei & Malek, 2016; S. Hao & Govindan., 2014; W. Yang & Xie., 2013), following the web crawlers (S. Roy Choudhary & Orso, 2013; V. Dallmeier & Zeller, 2013; van Deursen & Lenselin, 2012), they have been proposed to generate events and explore the behaviors of an application building the model. Model-based testing represents a software testing technique where the behaviors of a given software are checked against the predictions of the model while the system is under test. The models generated by the approaches may also be produced manually (Takala, 2011), since other approaches may build the model dynamically (D. Amalfitano & Memon., 2015,1). *GUIRipper* (D. Amalfitano & Memon., 2012), which is known as *MobiGUITAR* (D. Amalfitano & Memon., 2015) later, builds the model of an application dynamically by crawling the application from a start state. While the approach is implemented by DFS (Depth-First Search) strategy and generates only UI events, the approach cannot observe the interactions of the input fields systematically in terms of the guard conditions of an application. Also, *PUMA* (S. Hao & Govindan., 2014) is another model-based testing tool that consists of generic UI automator and random exploration implemented by Monkey (AndroidMonkey, 2018). It is also implemented by a dynamic analysis with the basis of the Monkey approach. Since these approaches use DFS in their structures, *Trimdroid* (Nariman Mirzaei & Malek, 2016) uses combinatorial fashion instead of using randomly generated inputs. On the other hand, *ORBIT* (W. Yang & Xie., 2013) is another model-based strategy that uses static analysis instead of generating a model dynamically to discover suitable UI events for a particular screen of a mobile application. Even if the use of models achieve higher code coverage than random testing approaches (S. R. Choudhary & Orso, 2015), a model must be provided as input in some cases (Takala, 2011) or the models are not discovered by using covering arrays as systematic sampling; hence the states of a model are not tested systematically. Here, our focus is to dynamically discover the model of an application with systematic sampling.

In terms of systematic exploration strategies (H. van der Merwe & Visser, 2014; R. Mahmood & Malek., 2014; S. Anand & Yang., 2012), basically, the system/systematic testing, known as planned and ordered testing, is a software testing technique that evaluate the end-to-end system specifications in the literature. The researchers have developed different approaches that crawl the application in a systematic way. Also, the inputs and system events are generated systematically (S. Anand & Yang., 2012). In addition, *EvoDroid* (R. Mahmood & Malek., 2014) is based on evolutionary algorithms to produce relevant inputs. In the framework, EvoDroid presents the sequences of test inputs in order to maximize the code coverage. *ACTEve* (S. Anand & Yang., 2012) is a concolic-testing tool that triggers the events in the framework so that it may instrument the application and the framework. Moreover, *JPF-Android* (H. van der Merwe & Visser, 2014) is another systematic exploration strategy that extends Java Path Finder (JPF), which lets to verify the applications systematically against the specific properties. However, either the input and system events are generated systematically or the specifications are verified with a systematic manner, the weakness of systematic testing is to ignore the interactions of the input fields. This weakness causes to not cover the interactions of the input fields of an application systematically. Here, in our approach, we cover the interactions between the inputs fields of the screens by using covering arrays as systematic sampling so that we can discover the model systematically and automatically.

In our approach, we get higher code coverage than the existing tools (A. Machiry & Naik., 2013; AndroidMonkey, 2018) with the implementation of systematic sampling in the model discovery process. Since the model-based testing approaches need to build a model with a dynamic or static analysis, the approach automatically crawls the application and generates dynamically a model providing test samples generated systematically by the covering arrays. The approach we propose prevents randomness to crawl the application entirely applying systematic sampling when compared to random testing strategies. In the approach, we use covering arrays to systematically generate appropriate test cases and the approach predicts the guard conditions of the model interacting the input fields systematically. Moreover, when compared to random sampling in *Section **5***, under the equal conditions as systematic sampling (e.g., same number of test cases, domains and equivalence classes), the systematic sampling is better than the random sampling in terms of various evaluation metrics like state, transition and code coverage, and accuracy of predicted guard conditions of the model.

4. APPROACH

In this section, we present our approach and explain different algorithms to generate test cases systematically, crawl the application automatically and discover the model by predicting the guard-conditions.

4.1 General Overview of Approach

In this part, we define the general characteristics of the approach and express the relationship between the main steps. We basically develop an automated model discovery approach for mobile applications by applying systematic sampling generated by the covering arrays during the test process. *Figure **4.1*** demonstrates the general overview of the framework by subdividing the approach to explain how the system works.

According to the *Figure **4.1***, we use Android mobile applications as an input in the testing procedure. Although we use Android platform in this approach, the proposed approach are readily available to other platforms, such as iOS and Web. In general, the approach starts to crawl the application by detecting the screens of the application while the system is under test and build a model at run time. In the model, we represent the *nodes* as discovered distinct screens of the application and the *edges* as transitions between the screens. In the approach, we call the nodes as **states** and edges as **transitions** In addition, there are *guard conditions* on the transitions, which are the conditions that need to be satisfied before the transitions are taken. In the approach, the guard conditions consist of the test values of the input fields positioned on the transition of a source state. If a test case satisfies a guard condition, the approach arrives at a target state from a current source state.

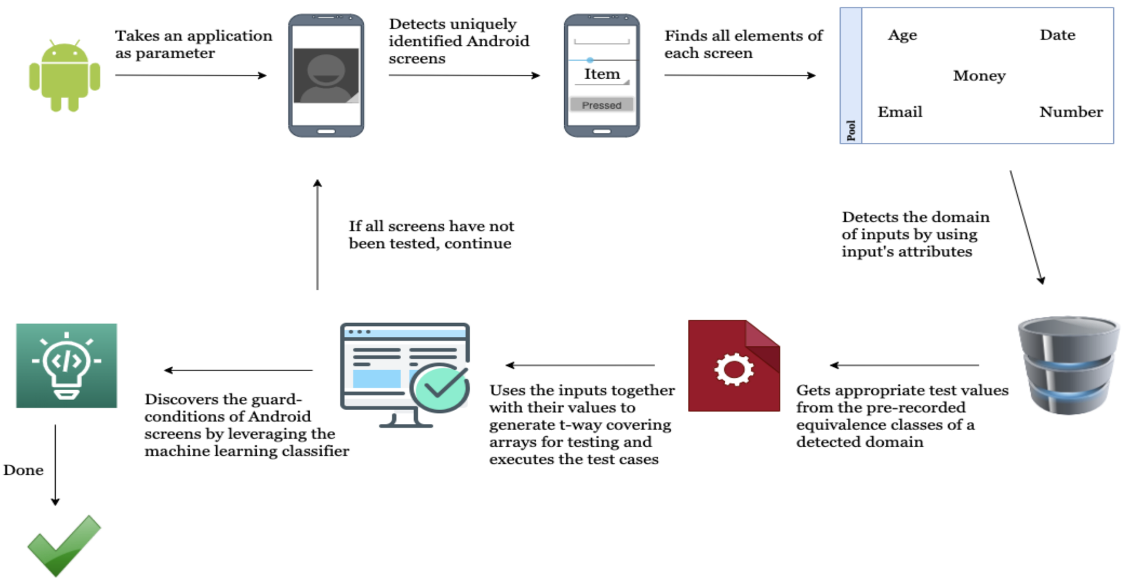
In order to define the screens, we first crawl a screen of the application and check the distinctness of the screen with its specific attributes provided by Android (e.g., class name, resource-id, or package). At the end of the distinctness decision of a screen, if a screen is distinctly identified, meaning that it has been discovered for the first time, the approach starts to collect the input fields of a given screen by using an XML file. For a given screen, the file basically includes the input fields and the attributes of the input fields in a special formatted way so that we can parse the file and use the input fields and their attributes easily in the approach.

After collecting the all inputs fields with their attributes of a current screen, we initiate the domain detection process for each collected input field. At this point, the approach needs to determine the input type to produce test values for the input fields. The input domain is detected by matching the information taken from the attributes of the input field with the keywords of the domains stored in the database. Later, we pick a value randomly from each pre-recorded equivalence classes of a detected domain for a given input field. We then generate test cases systematically via covering arrays with selected test values of the equivalence classes in accordance with the input fields.

For the test case generation process, we generate a t-way covering array, as systematic samples, with the input fields and their produced test values for each screen. We then start to execute the generated test cases sequentially. In the approach, we proceed in a depth-search manner. After each execution, we begin to check the status of the application. If the approach detects a new screen, the test cases are generated by the covering arrays and the approach executes them. On the other hand, when we move to a discovered screen, we check whether there are test cases remaining the covering array for the screen. If so, we pick the unexecuted test cases and execute them. At the end of the test executions, the test case generation process is completed and we then start to predict the guard conditions of the model.

As a last process of the approach, we discover the guard conditions of the discovered model by making a prediction on the results of test executions. In the guard condition discovery process, we make a binary classification leveraging a machine learning approach (e.g., decision tree classifier). The results of the test executions, as the data that will be trained, are prepared by labeling the data as 1 for each target state that we want to predict the guard condition and as 0 for the rest. We then execute the decision tree classifier on the prepared data and predict the guard conditions. This process is repeated to discover the guard conditions between the source state and each distinct target state.

Figure 4.1 General overview of the approach



Takes an application as parameter
Detects uniquely identified Android screens
Item
Pressed
Finds all elements of each screen
Pool
Age
Date
Money
Email
Number
Detects the domain of inputs by using input's attributes
Gets appropriate test values from the pre-recorded equivalence classes of a detected domain
Uses the inputs together with their values to generate t-way covering arrays for testing and executes the test cases
Discovers the guard-conditions of Android screens by leveraging the machine learning classifier
If all screens have not been tested, continue
Done

Algorithm ***1*** expresses the general crawling flow of the approach in an algorithmic manner. As explained in *Figure* ***4.1***, the approach takes a mobile application A as input. Then, the initialization processes are started. In lines *2-3*, $isFinished$ is set to $False$ and $coveringarray\ c_a$, $domainDispatcher\ d_d$, $inputFieldDispatcher\ f_d$ are declared. For each iteration, lines *5-6* show the initialization of a screen. The algorithm takes a current screen (state) of an application as *XML* file, which includes formatted information regarding Android screen, and transforms it to *screen* class in line *5*. Later, the input fields of a given screen are collected in line *6*. After screen initialization steps, the approach is ready to generate test cases systematically and execute them in an automated way. In line *7*, a current screen is checked to see whether it is known by using $isScreenKnown()$ method. If it is known, clarifying that the screen has not been discovered before, the domain, equivalence classes, and the input type detection processes are executed for each element in lines *8-12*. d_d is a service that detects the input domain and its equivalence classes for a given screen.

On the other hand, f_d a detector service that determines the input type (e.g., Edit-Text, Checkbox, List) of a given input field. This service finds the type of any input by using its attributes (e.g., class name, clickable, touchable). At the end of the loop, all test cases of a screen are generated by c_a with a systematic manner in line *13*. Then, we trigger to execute each test case t_c. After each execution, the algorithm needs to check the state of a current screen. If the application is still on the

Algorithm 1 General crawling algorithm with the main steps as pseudocode.

```
1: Input: A mobile application $A$
2: Initialize $isFinished = False$
3: Initialize $domainDispatcher\ d_d$, $inputFieldDispatcher\ f_d$, $coveringArray\ c_a$
4: repeat
5:    Initialize $screen = getScreen(A.currentState)$
6:    Initialize $screen.Elements = findElements(screen)$
7:    if $isScreenKnown(screen)$ then
8:       for $el$ in $screen.Elements$ do
9:          $el.Domain = d_d.findDomain(el)$
10:         $el.Values = d_d.findEquivalenceClasses(el.Domain)$
11:         $el.Type = f_d.findInputType(el)$
12:      end for
13:      Initialize $testCases = c_a.GenerateTestCases(screen.Elements)$
14:      for $testCase\ t_c$ in $testCases$ do
15:         Execute $t_c$
16:         if $getScreen(A.currentState) == screen$ then
               continue
17:         else
18:            break
19:         end if
20:      end for
21:   else
22:      Initialize $paths = ShortestPathToMoveScreen(screen)$
23:      $Move(paths)$
24:      $testCases = GetTestCases(screen)$
25:      $ExecuteTestCases(testCases)$
26:   end if
27:   if $isTestProgressDone(A)$ then
28:      $isFinished = True$
29:   end if
30: until $isFinished$ is not $True$
```

same screen, we then continue to execute the next test case. If not, it means that we find a new screen or observe a previously detected screen. In both conditions, the test case execution process of a screen is stopped and the application is restarted.

A new current screen is selected and all input fields of a given screen are detected. If the screen is known, the remaining process is the same as explained above. If not, we move to a previously detected screen. However, there might be different paths that allow to move a target screen from a current screen. In this situation, the approach uses a shortest path algorithm to arrive at a target screen quicker. Thus, $ShortestPathToMoveScreen()$ function finds the shortest path that includes minimum number of test cases to move a target screen in line *22*. $Move()$ function is then executed to move to the target screen by restarting the application. In lines

24-25, the test cases of a screen are selected from a database and execute them sequentially. At the end of all test case executions and checking conditions, in lines *27-29* $isTestProgressDone()$ method is triggered to check whether all test cases of all screens are executed. If the function returns $True$, the approach then finishes the procedure. If not, the algorithm selects a screen whose test cases have not been finished and continues to execute the test cases.

When finishing the procedure, we start to discover the model of an application by using stored details of the executions and predicting the guard-conditions between the states known as Android screens. When we analyze the the execution details stored in the database (e.g., source states, executed test cases, target states), we know the distinct states on the model of an application. After the approach generates a covering array as systematic samples for each state and execute all systematic samples of a given state, we then discover the guard-conditions of the models by training the executed systematic samples and making predictions on the trained data. In the prediction of the guard-conditions, we use decision-tree classifier as binary classifier and execute the classifier on the trained data of each state. At the end of the guard-condition discovery process, we build the model of an application by discovering the states and the guard-conditions.

4.2 Screen and Input Detection

To crawl the application in a systematic manner, we first discover the screens (e.g., the states) of a given application and collect the input fields (e.g., EditText, Button) in order to build test cases for each screen. Basically, we provide an algorithm that consists of different functionalities for the screen and input detection. As we focus on Android mobile applications, we use *XML* file of each screen in order to discover the Android screens with formatted information. The XML file includes the elements of a screen with their attributes provided by Android (e.g., class name, resource id, bounds), so that a developer can catch the elements easily in the codebase via the attributes. The attributes we use are shown in *Table **4.1*** with their sample values. *The algorithm **2*** demonstrates how to detect the input types using their attributes as the input detection approach.

Table 4.1 The attributes of Android elements used in both Input Detection and Domain Detection

Attributes	Sample Attribute Values
Class Name	*android.widget.Button*
Resource-id	*com.sample.android:id/LoginButton*
Text	*Login*
Content-desc	*Login the system*
Clickable	*True*
Long-Clickable	*True*
Checkable	*False*
Scrollable	*False*
Editable	*False*
Bounds	*[10,360][172,426]*

4.2.1 Input Detection

In *Algorithm **2***, we first take an XML file of given screen as input. Then, the given XML file is converted to a tree in order to iterate each child of a tree inside itself. In a tree, each child represents an input field with its attributes.

In line *3*, there is a list called *actions* that stores possible actions attributes of an input field that can be taken by a user. In the approach, we use the action attributes *clickable*, $long-clickable$, *checkable*, *scrollable* and *editable*, respectively as major action attributes. Line *4* shows the list variable which returns the input fields with the attributes of a given screen at the end of the execution. In lines *5-10*, the approach iterates each child of a tree generated by an XML file. For each child (e.g., an input field), we make sure that the action attributes of a given input field need to be matched with at least one of the *actions* list defined in line *3*. If there is no match, the approach cannot generate a test action for a given input field, since it does not know how to interact with the input field. If an action is matched with a defined *actions* list, meaning that a given child of the tree has an action, then, we collect the attributes of an input field with *getAttributes*() method in line *7*. Since the *tree* is not easily readable to get the attributes, we develop a *getAttributes*() method which takes a child as a tree member and converts it to the information including the attributes as a class called *elementAttributes*. In Android applications, various attributes are provided for the use of different purposes (e.g., writing a test case, catching an input field). In the approach, we choose some of the attributes to use in our approach and they are explained in *Table **4.1***. We collect these attributes and store them in a database with the input field of a screen. In addition to the flow of the actions, we use an attribute called as *class name* to specifically determine what

Algorithm 2 General input detection algorithm as pseudocode.

```
1: Input: XML file of the Android Screen $XML_a$
2: Initialize $tree = ParseXMLFile(XML_a)$, $actions$
3: $actions = ["clickable", "long-clickable", "checkable", "scrollable", "editable"]$
4: Initialize $elementsList = []$
5: for $child$ in $tree$ do
6:    if $child.actions()$ in $actions$ then
7:       Initialize $elementAttributes = getAttributes(child)$
8:       $elementsList.append(elementAttributes)$
9:    end if
10: end for
11: return $elementsList$
```

Algorithm 3 General screen detection algorithm as pseudocode.

```
1: Input: an XML file of a screen $XML_a$, discovered screens' hash values $H_a$
2: Initialize $tree = ParseXMLFile(XML_a)$
3: Initialize $elementsList = []$
4: for $child$ in $tree$ do
5:    Initialize $resourceId = child.get("resource-id")$
6:    Initialize $className = child.get("className")$
7:    $elementsList.append(resourceId + "-" + className)$
8: end for
9: $elementsList = sort(elementsList)$
10: Initialize $listHashValue = HashList(elementsList)$
11: if $listHashValue$ not in $H_a$ then
12:    $H_a.append(listHashValue)$
13:    return $True$
14: else
15:    return $False$
16: end if
```

the input field is. At the end of the *Algorithm 2*, we detect the input fields with their attributes of a given screen by checking the actions that an input field may take and write the test cases depending on the input types. Also, the attributes we use in *Input Detection Algorithm* are used in *Screen Detection Algorithm* as explained in ***Algorithm 3***.

4.2.2 Screen Detection

Algorithm 3 is simply developed to detect the screens and determine the distinct screens. In addition, the workflow of the screen detection algorithm is similar to the input detection approach. Similarly, we take an XML file of a screen XML_a

as input and also a list H_a that stores the hash values of the discovered screens. As explained clearly in *Algorithm 2*, we again parse an XML file as *tree* and get the input fields of a given screen with their attributes as *childs*. After collecting the input fields together with their attributes, we then hash the input fields of a given screen combining the specific attributes. To detect the screen as a new or a previously discovered screen, we compare the hash value of a given screen with the hash values of the previously discovered screens that stored in the database. At the end of this comparison, the screen detection process is completed. Most importantly, for a screen detection process, any screen detection logic can be implemented. In other words, our approach is convenient for other screen detection algorithms.

The major difference between screen detection and input detection algorithms begins in lines between *5-7*. Since the approach selects only the action attributes of the input fields to detect the input types in the input detection process, the input fields of a given screen are collected from *tree* based on specific attributes called as $resourceId$ and $className$ in a screen detection algorithm. The reason to use these attributes is that they are typically not changed in the applications. When compared with other attributes of an input field (e.g., bounds, text, contentDesc), these attributes may easily be modified in the new versions of the applications. The reason for this decision is related to the change in the value of a screen. When an element is relocated (e.g., bounds like x-y positions) or a text on the input field is changed, a hash value of the given screen will change, although the input field is the exactly the same as itself. Therefore, because of the changes in the values of the input fields, the approach may discover the screen as a new one.

In line *7*, we combine the values of the attributes and store in $elementsList$. At the end of the loop iteration, We first sort the input fields by combining in an ordered agnostic way and then calculate the hash value of a screen via $HashList$ function. Line *11* checks the distinctness of a given screen comparing the has values of the screens stored in the database to see whether a given screen has been discovered before. If a calculated hash value $listHashValue$ is matched with a value stored in the hash values of discovered screens H_a, it means that the screen has been discovered previously and the algorithm returns $False$. If the hash value is not matched, the approach shows that the current screen is distinct. Therefore, the algorithm stores a new hash value into H_a and returns $True$.

Algorithm 4 Domain and equivalence classes detection algorithm as pseudocode.

```
1: Input: An input element $el$
2: Initialize $d_d = domainDispatcher$
3: Initialize $domains = d_d.getAllDomains()$
4: Initialize $matchedDomain, matchedEquivalenceClasses = null, null$
5: Initialize $attr = el.attributes$
6: for $domain$ in $domains$ do
7:   if ($attr.ResourceId$ in $domain.Keywords$) or ($attr.Text$ in $domain.Keywords$) then
8:     $matchedDomain = domain$
9:     $matchedEquivalenceClasses = d_d.findEquivalenceClasses(matchedDomain)$
10:    break
11:  end if
12: end for
13: return $matchedDomain, matchedEquivalenceClasses$
```

4.3 Domain Detection and Equivalence Classes

To systematically sample the input space by using covering arrays, the approach needs to produce coherent test values for each input field of the screen. In addition, one assumption behind the combinatorial covering arrays is that each input field, as an *option* parameter of the covering arrays, takes its discrete values from the domains. In this process, we detect the input domains, we divide the domains into the partitions as equivalence classes so that the approach can produce test values for the input fields from the equivalence classes of a detected domain.

Algorithm 4 demonstrates the domain detection and pre-recorded equivalence classes of the approach. Basically, the algorithm takes an input field *el* as input and returns *matchedDomain* and *matchedEquivalenceClasses* (e.g., the domain and test values from the equivalence classes) at the end of the execution. As initialization, we declare *domainDispatcher*, the variables that will return the values, the attributes of an input field in *attr*, and the *domains* that stores all the domains of the approach in lines between *2-5.* In the approach, we generally define helpers that manage the functionalities for the detection approaches.

We have mainly three dispatchers called as *DomainDispatcher*, *EquivalenceClassDispatcher* and *InputFieldDisPatcher*. As we mentioned in *Algorithm 1*, *InputFieldDispatcher* determines the type of a given input field (e.g., Button, EditText, CheckBox) by analyzing the input fields stored in the database.

DomainDispatcher and *EquivalenceClassDispatcher* work with the same logic as *InputFieldDisPatcher*. Since *EquivalenceClassDispatcher* is responsible for the equivalence classes, the *DomainDispatcher* manages a domain detection logic and communicates with *EquivalenceClassDispatcher* to produce test values for the input fields. All domains and equivalence classes are stored in the database and the dispatchers have access to use the database.

In line *2*, all input domains are selected from the database by d_d and are stored in the *domains* variable. Each domain includes *name* and *keywords* attributes. The *keywords* attribute represents a set of words that identify the input domain. For instance, if we have a *login domain*, the keywords might be *mail*, *email*, *e-mail* or *username*. For this reason, we use three of the attributes of an input field called as ***resourceId***, ***contentDesc*** and ***text*** to detect the input domain by matching the attributes with the keywords of the domains, because other attributes do not contain eligible contexts (e.g., Bounds, ClassName) for the input domain detection.

If one of the selected attributes contains a keyword from *keywords*, we detect the domain in lines between *6-12*. Then, the pre-reecorded equivalence classes of a detected domain are selected from the database with the $findEquivalenceClasses()$ function. *EquivalenceClassDispatcher* is triggered inside d_d to collect all test input values from the equivalence classes. Here, the approach picks a value randomly from each pre-recorded equivalence classes of a given domain. If there is no match with the keywords, it means that there is no suitable domain stored in the database for a given input field, clarifying that the input domain must be added into the database. At the end, for each input field, the input domain is detected and the discrete test values are produced from the equivalence classes of a given domain so that the test values of the input fields can be covered by using covering arrays for each screen.

In the domain detection process, we could have used a semantic similarity approach (Islam, 2008) to determine the domains of the input fields. We however opted not to do so in this work as our ultimate goal is demonstrate that systematic sampling can do better when it comes to model discovery.

4.4 Covering Array Generation

In this step, for each screen, we generate a covering array by using the test values that collected from the equivalence classes for each input field of a given screen

Algorithm 5 Covering array and test cases generation algorithm as pseudocode.

```
1: **Input:** Elements of a screen $elements_a$, the strength of a covering array $t$
2: Initialize $c_a = coveringArrayGenerator(ACTS)$
3: Initialize $file$, $testCaseCombinations$, $testCases$
4: Initialize $actionElements = c_a.GetActionElements(elements_a)$
5: Initialize $otherElements = c_a.GetNotActionElements(elements_a)$
6: for $el$ in $otherElements$ do
7:     $file.write(el.InputName + "(enum) : " + el.Values)$
8: end for
9: Initialize $tempList = []$
10: for $actionEl$ in $actionElements$ do
11:     $tempList.append(actionEl.Values)$
12: end for
13: $file.write("Actions(enum) : " + tempList)$
14: $testCaseCombinations = c_a.Generate(t, file)$
15: Initialize $tempCombinationList$
16: for $combination$ in $testCaseCombinations$ do
17:     for $combinationValue$ in $combination$ do
18:         $case = c_a.WriteTest(combinationValue.InputField, combinationValue.Value)$
19:         $tempCombinationList.append(case)$
20:     end for
21:     $testCases.append(tempCombinationList)$
22:     $tempCombinationList = []$
23: end for
24: return $testCases$
```

in order to execute all t-way combinations of the input fields, as test cases, in a systematic manner. At the end of the covering array generation process, we execute all the test cases in the computed covering array sequentially for each screen. In order to generate the test cases with the combinations of the input fields together with their test values, we used *ACTS* (ACTS, 2018) framework, known as a covering array generator, in the approach.

In *Algorithm* ***5***, the workflow of a covering array and test case generation procedure are explained in detail. The algorithm first takes the input fields of a screen with their test values, the input types $elements_a$ and the strength of a covering array t as parameters to generate t-way combinations of input fields. An instance of a covering array generator *ACTS* is taken and initialized with c_a. In order to generate a covering array, we need to write all input fields with their values into a $file$ and execute the file.

In lines between *4-5*, we have a separation process based on the input fields (e.g., EditText, Button, RadioButton). Since the input fields have different action attributes (e.g., clickable, editable), we need to divide the input fields according to

Table 4.2 An example of a register screen showing the generated covering array with four options and their test values where t=3 (Option 1:Pharmacy Number, Option 2:Username, Option 3:Password, Option 4:Agreement, A1:Register Button, A2:Login Button)

Option 1	Option 2	Option 3	Option 4	Actions
Set 100716	Set john@gmail.com	Set Passw0rd	Set checked	Click A1
Set 100716	Set john@gmail.com	Set ????	Set unchecked	Click A1
Set 100716	Set qy@11.com	Set Passw0rd	Set unchecked	Click A1
Set 100716	Set qy@11.com	Set ????	Set checked	Click A1
Set 000000	Set john@gmail.com	Set Passw0rd	Set unchecked	Click A1
Set 000000	Set john@gmail.com	Set ????	Set checked	Click A1
Set 000000	Set qy@11.com	Set Passw0rd	Set checked	Click A1
Set 000000	Set qy@11.com	Set ????	Set unchecked	Click A1
Set 100716	Set john@gmail.com	Set Passw0rd	Set checked	Click A2
Set 100716	Set john@gmail.com	Set ????	Set unchecked	Click A2
Set 100716	Set qy@11.com	Set Passw0rd	Set unchecked	Click A2
Set 100716	Set qy@11.com	Set ????	Set checked	Click A2
Set 000000	Set john@gmail.com	Set Passw0rd	Set unchecked	Click A2
Set 000000	Set john@gmail.com	Set ????	Set checked	Click A2
Set 000000	Set qy@11.com	Set Passw0rd	Set checked	Click A2
Set 000000	Set qy@11.com	Set ????	Set unchecked	Click A2

their action attributes before the test cases are produced.

GetActionElements() method is used to select the input fields *actionElements* that may take the actions (e.g., click, double-click) and change the state of the screen. Therefore, we determine these input fields by checking the action attributes. For instance, if the input field has the actions such as click and double-click, these two actions are selected and combined with test values of the input fields systematically.

On the other hand, $GetNotActionElements$() function determines all input fields $otherElements$ that have test values. After the separation procedure of the input fields according to the input types, the algorithm begins to store each input field in the $otherElements$ and writes the test values in the $file$. While choosing test values for the input fields, we use equivalence classes as explained in *Approach Section 4.3* to generate systematic samples with the covering arrays. In this situation, we randomly take a test value from each equivalence class of each input field to generate a covering array.

Since there are no test input values of *actionElements* for testing, we store action input fields in a list *tempList*. Then, they are written in a file as *Actions*. In line *14*, the covering array is generated by *Generate*() function depends on a strength of a covering array t. At the end of the covering array generation process, we parse each combination of *testCaseCombinations* and converts the combined values to the test cases in lines between *16-23*.

When compared to *Table **4.2*** , each column refers to an option represented as an input field of a register screen and each row gives a combination of the input fields as a test case. In addition, each cell represents a test action including the test value of an input field. After getting value from each cell, $WriteTest()$ function is used to generate an executable test code by using the input field with a selected value. As a last step, all test cases *testCases* are systematically generated and the approach starts to execute the generated test cases sequentially.

4.5 Guard-Condition Discovery

In this step, we discover the guard conditions between the states on the discovered model of an application. After generating systematic samples with the covering arrays for each state, the approach executes all samples sequentially on the given state. After each execution, we store the details regarding the execution such as source state, executed sample as test case, and target state. When the approach finishes all test executions for each state, we begin to predict the guard conditions between current state and the target states. For instance, if we generate a covering array as systematic samples for state $s1$, execute all test cases and move to the state $s2$ and state $s3$ from state $s1$, the approach predicts the guard conditions between state $s1$ and $s2$, and state $s1$ and $s3$. To discover each guard condition of the given state, we leverage the decision-tree classifier as binary classifier. For each target state, we apply a binary classification to predict the guard condition between the source state and target state.

In *Algorithm **6***, we illustrate the classification process of guard condition discovery. In line *1*, we collect the executed test cases, that satisfied to move from the given state to each target state, from the database. We use the collected data as train data in the classification. In lines between *2-3*, we find the target states arrived by the state S' analyzing the collected data D' and initialize the $predictionresults$ that returns the prediction results of the state S'.

In lines between *4-9*, we predict the guard conditions of the state S' for each arrived target state iteratively. In line *5*, for each target state, we first label the data that will be trained by the classifier. We label the data whose target state we want

Algorithm 6 General guard-condition discovery algorithm as pseudocode.

```
1: Input: Data $D'$ stored in the database for state $S'$
2: Initialize $targetStates = FindTargets(D')$
3: Initialize $predictionResults = []$
4: for $state$ in $targetStates$ do
5:     $LabelData(D', state)$
6:     Initialize $prediction = RunClassifier(D', state)$
7:     Initialize $predictedTargetState = Execute(prediction)$
8:     Initialize $result = CheckPrediction(D', predictedTargetState)$
9:     $predictionResults.append(result)$
10: end for
11: return $predictionResults$
```

to predict as **1**, and label others as **0**. After labeling the data, the approach then predicts the guard condition of each target state labeled as **1** and finds the discovered guard condition in line *6*.

To satisfy the accuracy of predicted guard condition, the approach executes the predicted guard condition once on the given screen and finds the predicted target state in line *7*. We then check the accuracy of the prediction with *CheckPrediction* method in line *8*.

If the predicted guard condition is the same as the one before the prediction and the given state S' cannot move to the other target states with the predicted guard condition, the approach approves that the guard condition is discovered correctly. Otherwise, the prediction of a guard condition is marked as incorrect.

This process is repeated for each target state of each screen sequentially. After each prediction, the approach inserts the prediction result into the *predictionResults* variable in line *9*. At the end of the guard condition discovery process, the approach discovers the guard conditions on the discovered model of an application and returns the results in line *11*.

5. EXPERIMENTS

We have conducted a series of experiments to evaluate the proposed approach. In the first set of experiments (*Section* ***5.1***), we have evaluated the sensitivity of the approach to various model parameters, including the number of states, density, the level of determinism, and the complexity of the guard conditions. To this end, we have used simulations as it was not possible for us to systematically vary these parameters on real subject applications, on which we had no control over. In the second set of experiments (*Section* ***5.2***), we have evaluated the proposed approach by conducting comparative studies using real subject applications.

5.1 Evaluating sensitivity to model parameters

In this set of experiments, we evaluate the sensitivity of the proposed approach to the model parameters by systematically varying these parameters on the simulations.

5.1.1 Setup

In particular, we manipulate the following parameter:

- **states**: the number of states in the graph-based model.
- **density:** the density of the graph-based model (Ahuja, 2017), which is used to compute the number of edges in the graph-based model.
- **parameters:** the number of parameters defined in a state, i.e., the number of input fields on a screen.

Table 5.1 Model parameters manipulated in the experiments.

Parameter	Values
number of states	$\{10, 20, 50\}$
density	$\{0.4, 0.6\}$
number of parameters per state	$\{[5, 10], [16, 20]\}$
number of equivalence classes per input	$\{[3, 6]\}$
number of distinct parameters involved in guard conditions	$\{1, 2, 3, 4, [1, 5]\}$
covering array strength	$\{2, 3, 4\}$
level of determinism	$\{0, 0.01, 0.05, 0.1\}$

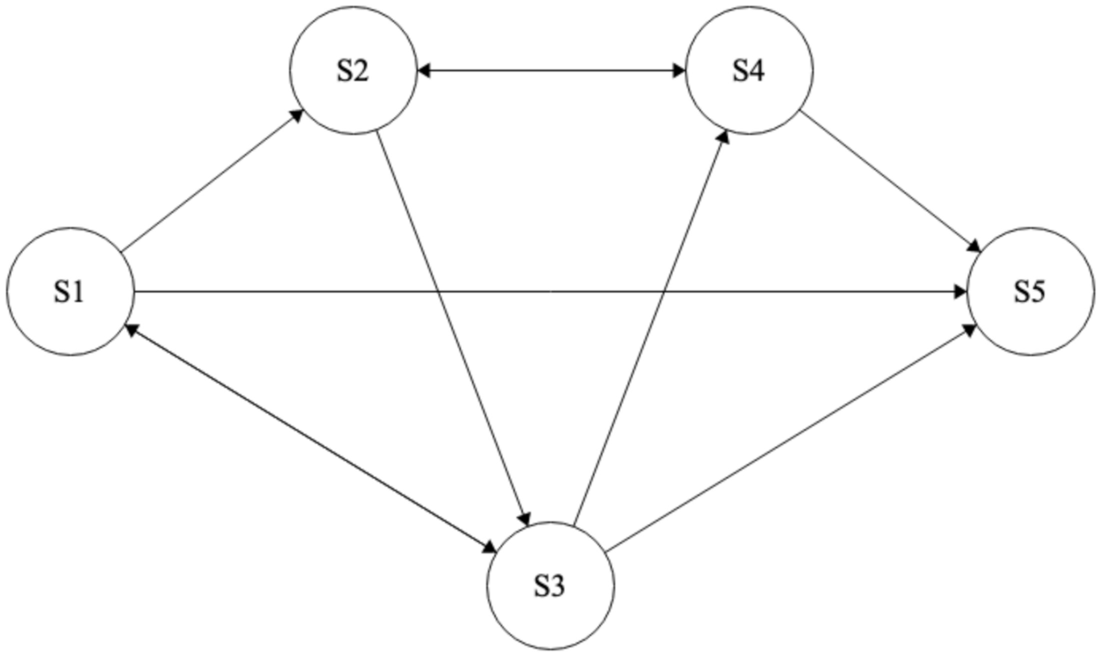
- **settings:** the number of equivalence classes for a parameter.
- **guard-complexity:** the number of distinct parameters involved in a guard condition associated with a transition.
- **t:** the coverage strength of the covering arrays used for sampling.
- **determinism:** the level of determinism in the model, depicting the probability of taking a transition given that the guard condition of the transition is satisfied. When *determinism*= 1.0, all the transitions are deterministic; given a transition, when the system is currently in the source state and the guard condition of the transition is satisfied, the transition is guaranteed to be taken and the system moves to the target state.

*Table **5.1*** presents the values used for these parameters in the experiments. The range values, which are given in the form of $[min, max]$, indicate that the actual values are randomly chosen, such that they are between min and max, inclusive. For example, when $settings = [3, 6]$, each state parameter in the model will have in between 3 and 6 randomly chosen equivalence classes. More specifically, for each state parameter, a number is randomly picked from the range 3 through 6 and used as the number of equivalence classes that the parameter has. For each configuration in the Cartesian product of the settings given in *Table **5.1***, we randomly generated 100 models and stored the test results of the simulations in the database.

5.1.1.1 Models and Simulations

Since we did not know true guard conditions and had no control over the real applications, we used the simulations to evaluate the sensitivity of the approach varying the model parameters systematically. For this reason, in this subsection, we

Figure 5.1 An example of a model used in the simulations where the number of states = 5, the density = 0.4, and the number of edges = 10.

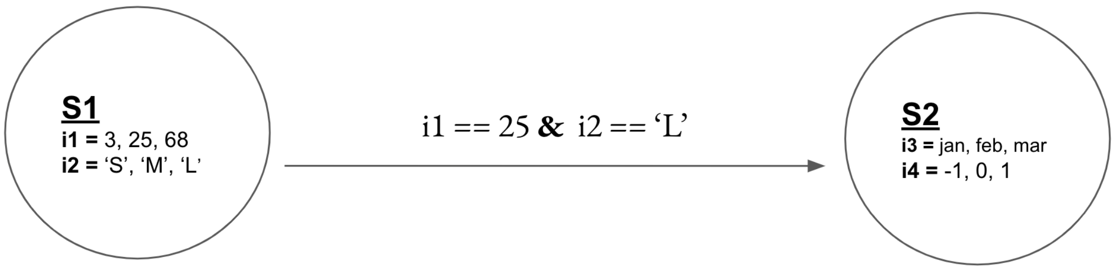


explained what the models are and how the models are used in the simulations. To understand clearly, we presented an example model used in the simulations with the given model parameters in *Figure* ***5.1***.

In the figure, the model impersonated real Android applications during the simulations. Each node of a given model was represented as Android screen including the input fields with their test values. The nodes between the states were used as the transitions having the guard conditions. In a given model, we had 5 distinct states and 10 transitions calculated with the density between the states where the density was 0.4. The number of transitions on the model was calculated with a given formula; $transitions = (numberOfStates)^2 * density$. In the simulations, each model was generated in a such way that it is a strongly connected directed graph so that the automaton does not get stuck at a state. This is also consistent with the general behavior of mobile applications; every screen is reachable from the start screen and the start screen is typically reachable from every other screen.

In addition, we presented a detailed example of a given model, demonstrated in *Figure* ***5.1***, to clarify how the model is simulated by analyzing two specific states of a model with a transition in *Figure* ***5.2***. In a given part of the model, $S1$ and $S2$ were the states represented as a source state and target state, respectively. Each state had 2 parameters, implied as input fields, $i1$, $i2$, $i3$ and $i4$, respectively.

Figure 5.2 An example of a given model in detail with the model parameters (parameters=2, settings=3, guard-complexity=2).



For each parameter, 3 settings, known as equivalence classes, were produced. On the other hand, we had a guard condition on the transition between $S1$ and $S2$. The guard condition included two parameters of the state $S1$, where the guard-complexity was 2. Here, the parameters of a guard condition were linked with the AND (&) operator. If a guard condition was satisfied, the state $S1$ moved to the state $S2$ taking the respective transition. Furthermore, for each state in a model, the guard conditions for the transitions originating from the state were guaranteed to be mutually exclusive, so that $determinism = 1.0$ makes sense in the experiments.

In the simulations, for each model, a randomly chosen state was marked as the start state. On a given model, we operated as if it were operating on a subject application. In particular, when a state was visited, we obtained the parameters associated with the state and generated a covering array. And, when a test case is executed, the model was fed with the respective values, the transition to be taken was determined, and the system was moved from the current state to the target state.

5.1.2 Evaluation Framework

For the evaluations, we used the following metrics:

- **state coverage:** percentage of the states visited.
- **transition coverage:** percentage of the transitions exercised.
- **accuracy:** accuracy of the guard conditions predicted.

In terms of coverage criteria evaluation, we evaluated the states and transitions in all simulations. As state coverage, we confirmed that the state s' was covered if the simulation satisfied the state s' at least once. On the other hand, as transition coverage, we confirmed that the transition t' was covered if the simulation verified the guard condition of the given transition t' with its parameters at least once. In other words, the simulation moved to the target state from the source state when the transition between the source and target state was satisfied.

We, furthermore, compared the results obtained from the proposed approach to those obtained from random testing. To this end, we used the same models and for every state that the random testing strategy visited in these models, we randomly generated the same number of test cases with the proposed approach by using exactly the same equivalence classes used by the proposed approach. Everything else was kept the same, i.e., the way the models were simulated and the way the guard conditions were predicted and evaluated.

5.1.3 Operational Framework

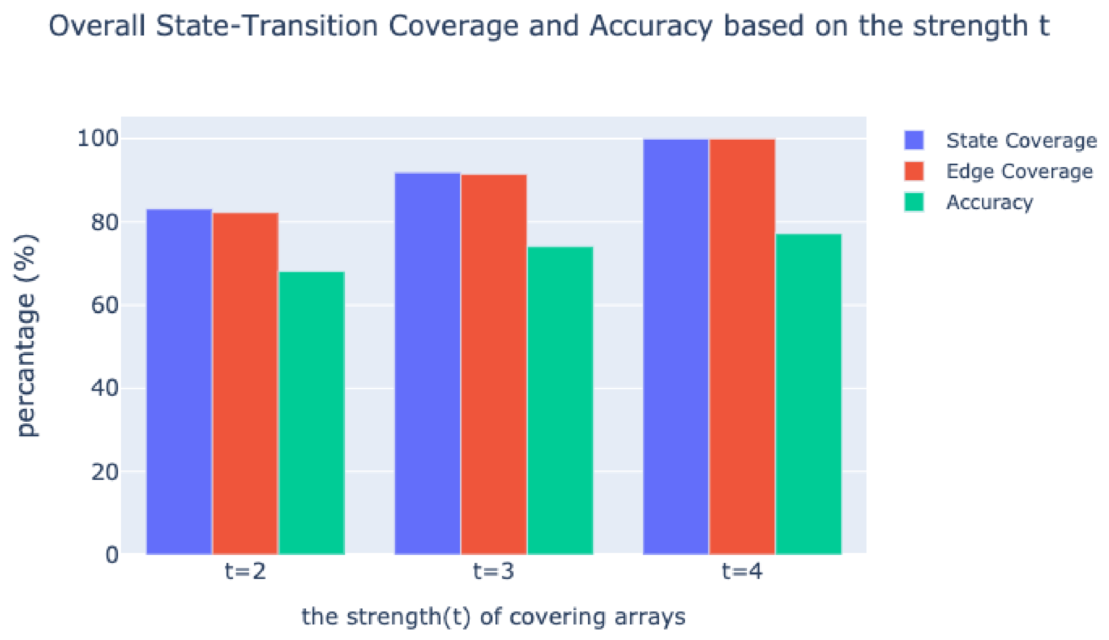
All the experiments were carried out on an Intel I7 6700HQ machine with 16 GB of RAM, running Windows 10 as the operating system. The classification models were trained by using the Decision Tree classifier implemented in scikit-learn (Scikit-learn, 2018). Furthermore, we used ACTS (ACTS, 2018) to generate the covering arrays used in the experiments.

5.1.4 Data and Analysis

In total, we generated 22,997,103 test cases and trained 7200 classification models by using the proposed approach. The results we obtained are summarized in *Figures **5.3**, **5.4**, **5.5**, **5.6***, and ***5.7***.

We first observed that as the strength of the covering arrays used for discovery increased, state coverage, transition coverage, and accuracy of the predicted guard conditions for all guard complexity (e.g., 1, 2, 3, and 4) increased (*Figure **5.3***). This is, indeed, to be expected as higher strength covering arrays cover more distinct combinations of parameter values using additional configurations. The average number of test cases generated by 2-, 3- and 4-way covering arrays on a per-state basis, were 818, $5,477$ and $38,662$, respectively. Overall, while the average state coverage, transition coverage, and accuracy of predicted guard condiitons obtained from 2-

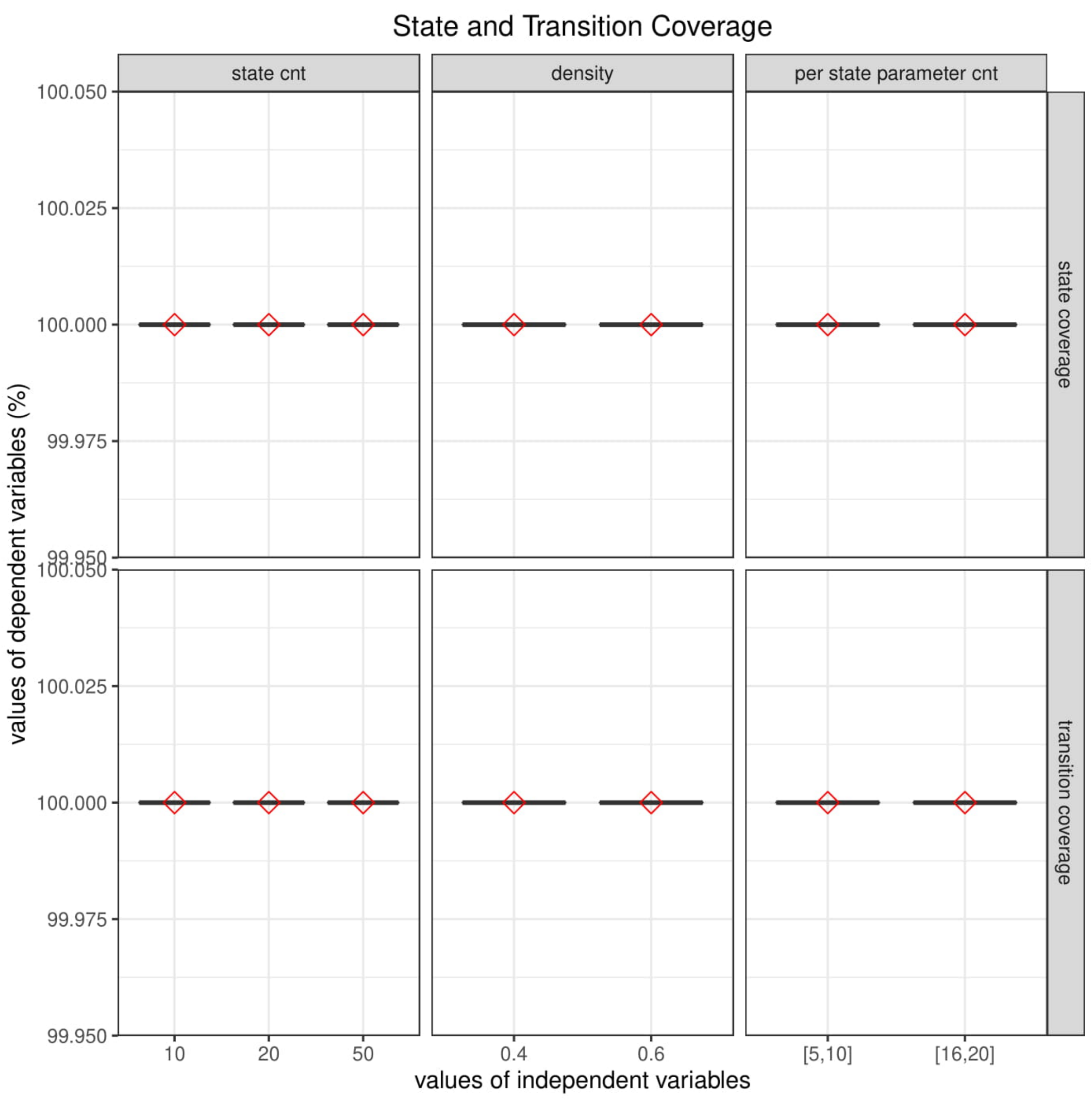
Figure 5.3 Overall State Coverage, Transition Coverage and Accuracy comparison based on the strength t of covering arrays.



way coverings arrays were 83.10%, 82.28%, and 68.07%, those obtained from 3-way covering arrays were 91.90%, 91.48%, and 73.12%, respectively. As 4-way covering arrays, we obtained 100% state and transition coverage, and 77.20% accuracy.

In *Figure* ***5.4*** and ***5.5***, we made a couple of comparisons in terms of the values of independent variables and the accuracy of predicted guard conditions where the strength of covering arrays t is greater than the guard-complexity. As we expected in *Figure* ***5.4***, the state coverage and transition coverage were 100% and 100% respectively for all independent variables used in the evaluation. The results showed that systematic sampling is an effective sampling approach to get complete state and transition coverage so that we can visit the entire system by satisfying all guard conditions on the transitions. Even if the number of states, the density which affects the number of transitions, and the number of parameters per state increased, the systematic sampling kept 100% state and transition coverage by showing that independent variables do not have impact on the coverage, if the system has a condition where $t > guard-complexity$. On the other side, for all simulations without having a specific condition (e.g., $t > guard-complexity$), we observed that as the complexity of the models increased, i.e., as the number of states, transitions (i.e., density), state parameters, or equivalence classes increased and/or the guard conditions became more complex, the state coverage, transition coverage, and the accuracy of the

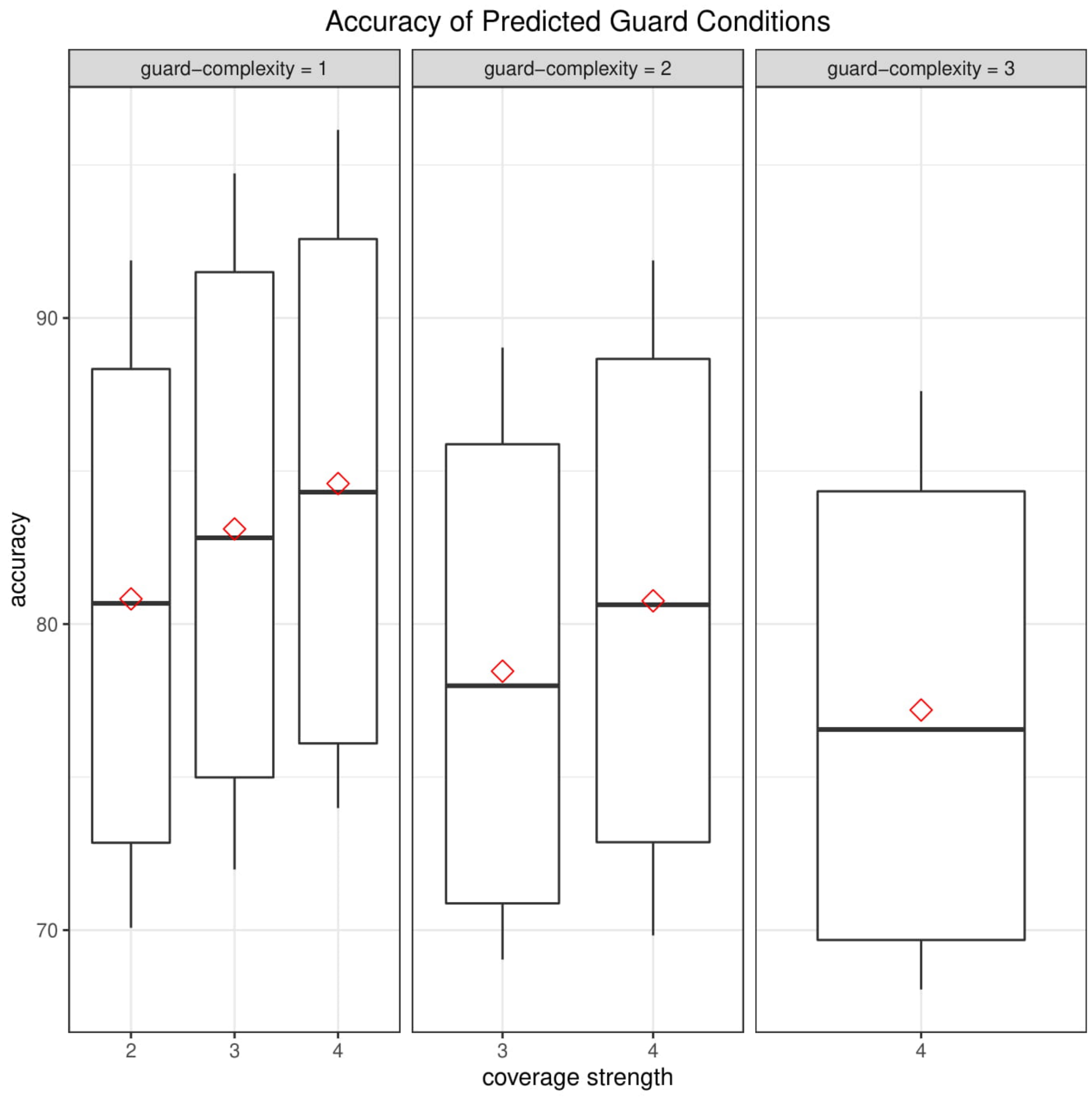
Figure 5.4 State and Transition Coverage based on values of independent variables (e.g., state count, density and per state parameter count).



predicted guard conditions tented to decrease, especially when lower strength covering arrays were used. However, the smallest state coverage, transition coverage, and accuracy obtained in all the experiments when $t = 3$, $density = 0.6$, $stateCnt = 50$, $stateParams = [16, 20]$, and $guard - complexity = 4$ for example, were still 57.70%, 56.05%, and 67.36%.

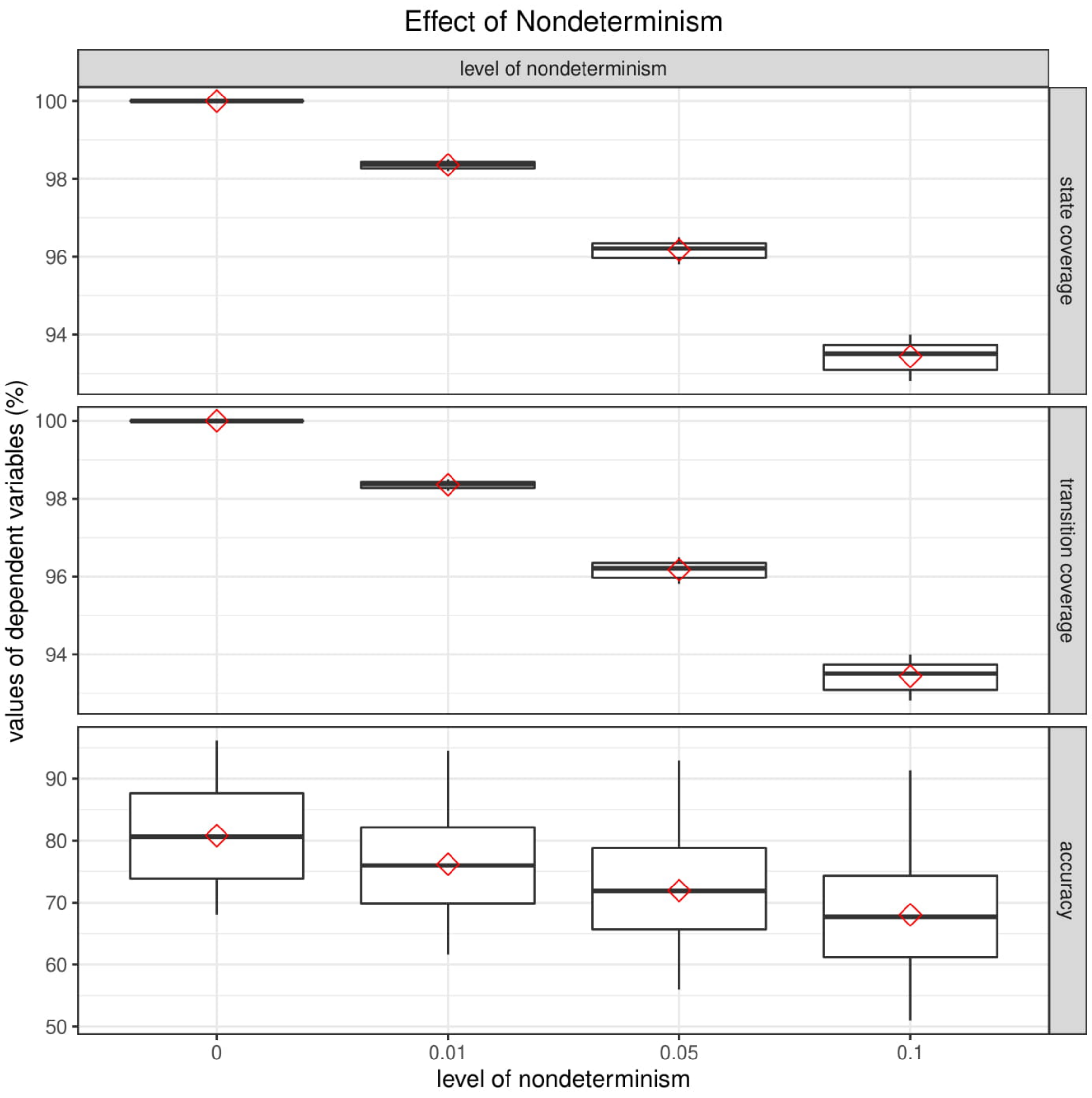
In addition, in *Figure* ***5.5***, we again observed that the accuracy of predicted guard conditions on the transitions increased when the strength t of covering arrays increased for each guard complexity (e.g., guard-complexity=1). For example, in the figure, when we kept the guard-complexity same as 1, if the strength of the covering arrays increased, the accuracy of predicted guard conditions increased as well. On the other hand, if the guard complexity of the transitions increased, then, the results

Figure 5.5 Accuracy of predicted guard conditions based on the guard complexity and the strength t of covering arrays.



obtained from the figure demonstrated that the accuracy of predicted guard conditions decreased for each strength t of the covering arrays as we expected, because the complexity makes the approach difficult to detect the guard conditions accurately. For instance, in *Figure* ***5.5***, when we kept the strength of the covering arrays same as 4, if the guard-complexity increased, the accuracy of predicted guard conditions of the models decreased correlatively. While the accuracy for $guard-complexity = 1$ obtained from 2-, 3- and 4-way covering arrays were 80.82%, 82.10%, and 84.89%, the accuracy for $guard-complexity = 2$ obtained from 3- and 4-way covering arrays were 77.45%, 80.75%, respectively. As $guard-complexity = 3$, we obtained 77.13% accuracy.

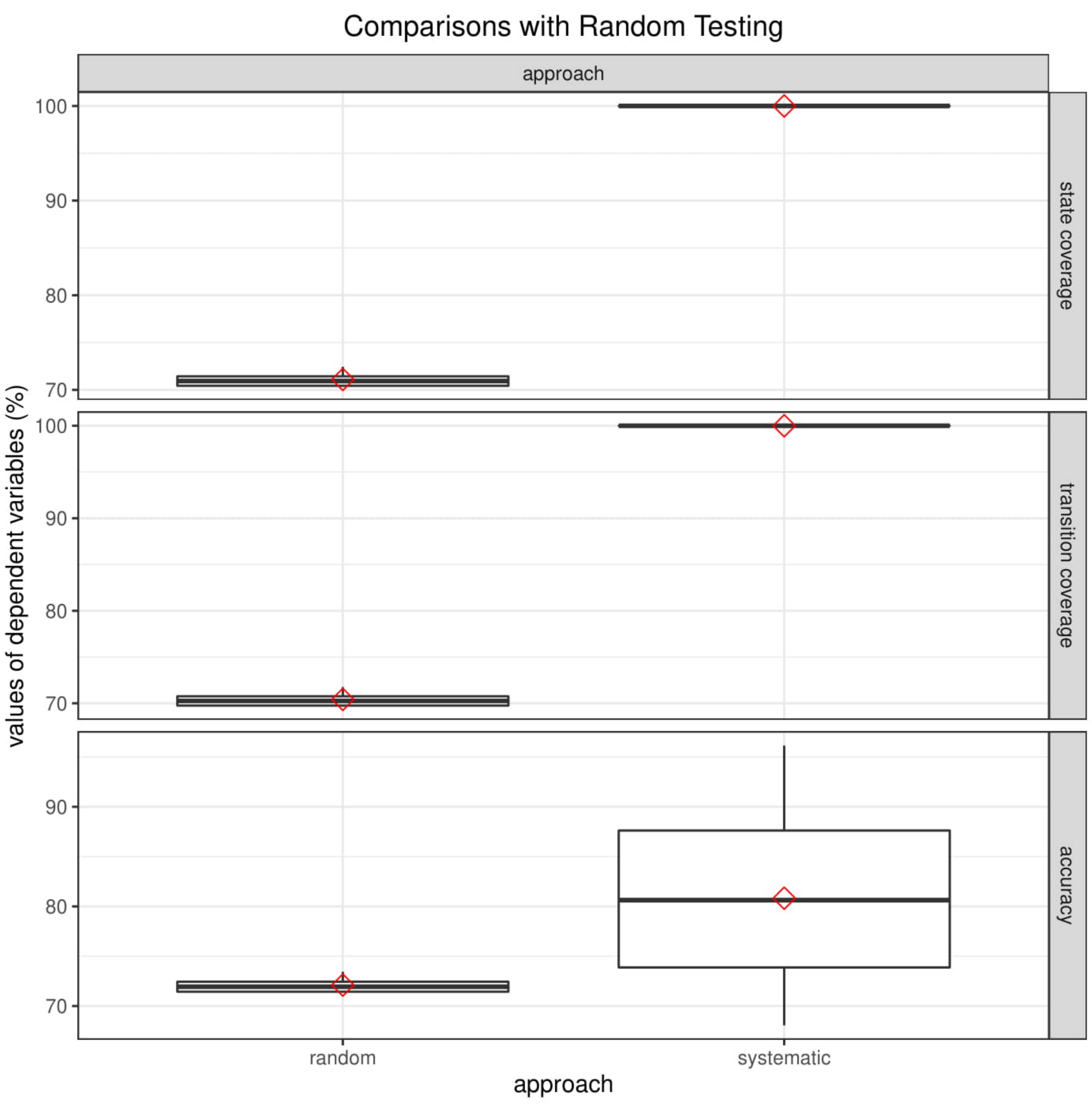
Figure 5.6 Effect of Non-determinism comparison in terms of State Coverage, Transition Coverage and Accuracy.



In *Figure **5.6***, as we expected, the results indicated that if the determinism level of the model increased, the state coverage, transition coverage, and accuracy of predicted guard conditions decreased. For example, while the state coverage, transition coverage and accuracy of predicted guard conditions obtained from the model that worked as fully deterministic (e.g., level=0) were 100%, 100% and 81.89%, respectively, those obtained from the model where $level = 0.1$ were 93.56%, 92.23% and 68.81%, respectively. Thus, we proved that the rate of determinism for each simulation model affects the coverage and accuracy.

Last but not least, comparing the results obtained from the proposed approach to those obtained from random testing in *Figure **5.7***, we observed that our proposed approach prominently offers higher state coverage, transition coverage and accu-

Figure 5.7 Comparisons between systematic and random testing in terms of state coverage, transition coverage, and accuracy.



racy of predicted guard conditions than the random testing under the equal conditions (e.g., same number of systematic samples, domains and equivalence classes) in terms of the number of generated test cases and model parameters (e.g., density, guard-complexity, number of states). Since the state coverage, transition coverage and accuracy of predicted guard conditions obtained from proposed approach were 100%, 100%, and 81.67%, respectively, those obtained from the random testing were 71.34%, 70.89% and 73.78%, respectively. At the end, the results obtained from *Figure **5.7*** showed that the systamtic sampling is better than the random sampling in terms of different evaluation metrics.

5.2 Evaluations on Real Subject Applications

In this section, we evaluate the proposed approach by conducting comparative studies using real subject applications. Also, we applied the random testing with the proposed approach and compared it with other approaches.

5.2.1 Setup

In particular, we used 10 subject applications from Google Play Store (Store, 2018). Information about these applications can be found in *Table **5.2***. We chose these applications as they had been also used to evaluate related approaches (A. Machiry & Naik., 2013; D. Amalfitano & Memon., 2015; R. Mahmood & Malek., 2014; W. Yang & Xie., 2013).

Also, we used the following terms in the evaluations of real applications:

- **screen:** a page of Android mobile applications. It might be an Android activity or a different page in the same activity (e.g., pop-up, modal). Each page which consists of UI elements is called as screen.
- **test action:** one of the executable tests in test suites. For example, if we have a test suite that includes 3 executable tests, each of them is called as test action.

5.2.2 Evaluation Framework

For the evaluations, as the models of the subject applications were not known, we could not use the state and transition coverage metrics *(Section **5.1**)*. Instead, we used the *code* and *screen coverage* metrics, depicting the percentages of the source code statements and the screens visited during testing, respectively. Given a subject application, we determined the number of screens by first performing a static analysis of the binaries for the application (i.e., by analyzing the *apk* file). Then, as we discovered new screens during testing (either by the proposed approach or by the existing testing tools used for comparative studies) we updated the set of

Table 5.2 Information about the subject applications used in the study (e.g., C1:application, C2:description and C3:number of activities).

C1	C2	C3
Tureng	A Dictionary for Turkish-English translations	11
Tippy Tipper	A simple and open source Tip Calculator	5
Munchlife	A counter app to keep track of your character level	2
Contact Manager	A tool for managing the contacts	10
To Do Manager	A task manager to track and organize to-do tasks	18
Alarm Klock	An alarm tool to create and manage the alarms	7
Habit App	A tool to keep track of your habits and routines	8
Any Memo	An open-sourced flashcard learning software	15
Simple Weight Tracker	A body weight watcher to analyze weight loss	3
BBC News	An app to bring the news from BBC News	40

known screens.

Furthermore, as the true guard conditions for the transitions were not known, we computed the accuracy of the predicted conditions by using n-fold cross validation, where $3 \leq n \leq 5$ depending on the number of samples available in the training data. That is, after the covering array generated for a state was tested, the results obtained were used to carry out the cross validation (*Section **5.2.4***).

Last but not least, we compared the results obtained from the proposed approach to those obtained from existing approaches, namely Dynodroid (A. Machiry & Naik., 2013) and Monkey (AndroidMonkey, 2018), as well as from random testing. For the latter, as was the case in *Section **5.1***, for every state visited by the random testing strategy, the same number of test cases with the proposed approach was randomly generated by using the same equivalence classes. For the former, we ran the existing tools on the same experimental platform with the proposed approach (*Section **5.2.3***). We, furthermore, ensured that the existing approaches performed similar or more actions, compared to the proposed approach. In this context, an action corresponds to a collection of related events for performing a task from the perspective of end-user, such as tapping on a button or entering a string into a text field. We counted the number of actions, rather than the number of test cases because the existing tools used in the study do not have a notion of the test case.

5.2.3 Operational Framework

In the experiments, we used 1) ACTS (ACTS, 2018) to generate the covering arrays, 2) Appium (Appium, 2018) to execute the generated test cases of real applications, 3) ACVTool (AcvTool, 2018) to measure the code coverage without having the source codes of the applications, and 4) Decision Tree classifier in scikit-learn (Scikit-learn, 2018) to predict the guard conditions. All the experiments were carried out on the same computing platform used in *Section **5.1***.

5.2.4 Data and Analysis

We first run all generated test cases for each mobile application and collected data from the test results. *Table **5.3*** shows that general information regarding test results on the subjects applications. The average execution time of 10 applications is 40.3 minutes. In the table, the number of domains demonstrates how many input domains the approach detected for the applications. Since the **EC** stands for *Equivalence Classes* in the table, the number of equivalence classes emphasized how many equivalence classes for the respective domains the approach produced. The data from general information on the test results show that the average number of input domains and equivalence classes generated by the approach for all input fields were 2.9 and 6.4, respectively. On the other hand, the average number of test cases generated by covering arrays for systematic sampling was 89.2. In terms of executed test cases during the experiments, the average number of actions taken by the approach was 257.

According to the analysis of general test results of real applications, the execution times of the approach were quite high. The reason was about the test automation framework we used in the approach called as *Appium*. Since Appium has a client-server architecture, there were sometimes network problems as expected during the testing procedure, indicating that the network problem caused the delays in the responses coming from the server to the client. So, it took more time to handle the problems and continues to the test process of subject applications.

In terms of the coverage criteria, the screen coverage was 93.3% on average which shows that the approach detected almost all distinct screens of the applications with the given input domains and their equivalence classes. However, there were a couple of reasons that made 100% screen coverage difficult. One reason was

Table 5.3 Overall test results of subject applications (e.g., $C1$:application, $C2$:execution time, $C3$:number of domains, $C4$:number of equivalence classes, $C5$:number of inputs, $C6$:number of test cases and $C7$:number of Android screens, $C8$:screen coverage).

$C1$	$C2$	$C3$	$C4$	$C5$	$C6$	$C7$	$C8$
Tureng	39 min	5	11	40	184	11	81.88%
To Do Manager	57 min	7	23	52	104	18	100%
Tippy Tipper	21 min	4	15	10	72	5	100%
Munchlife	15 min	2	5	8	21	2	100%
Alarm Klock	47 min	2	6	16	89	7	100%
Habit App	65 min	6	11	30	100	8	81.88%
Simple Weight Tracker	56 min	3	7	12	76	3	100%
BBC News	35 min	6	16	25	165	40	70%
Any Memo	43 min	4	12	21	101	15	100%
Contact Manager	25 min	4	11	29	128	10	100%

regarding the system bugs of the applications. When an application was suddenly crashed because of a failure (e.g., network connection, deficient development, infinite loops), the path our approach traverses was broken, showing that Android screen became unreachable. Another reason was related to the behaviors of the applications (e.g., HabitApp, BBC News). In some situations, when a test case was executed, an application did not react or exit itself while the system was under test. Since the approach used the shortest path to move a specific screen during testing, the application repeatedly stayed in the same screen or suddenly exited itself. These issues showed that even if all test cases are executed, there may be the screens that have not been tested yet.

In the second analysis, the comparisons related to the code coverage results with other approaches in *Table **5.4*** demonstrated that proposed approach offers higher code coverage than Monkey (AndroidMonkey, 2018), Dynodroid (A. Machiry & Naik., 2013) and random sampling indicating that the systematic sampling covers more functionalities and conditions implemented in the application than other approaches. Instead of using covering arrays, since random sampling is one of the sampling approaches that generates the test cases randomly by using equivalence classes related to the input domains under the equal conditions of the proposed approach, random sampling still provides higher code coverage for each real application than Monkey and Dynodroid.

When we compared the random and systematic sampling under the equal conditions (e..g, equal number of test cases, same domains and equivalence classes), the comparison obviously indicated that the test cases generated systematically provide

Table 5.4 Code coverage comparison with other approaches (e.g., Random sampling, Monkey and Dynodroid).

App Name	Approach	Random Sampling	Monkey	Dynodroid
Tureng	53%	46%	32%	40%
To Do Manager	66%	57%	38%	45%
Tippy Tipper	78%	65%	51%	-
Munchlife	78%	66%	45%	59%
Alarm Klock	65%	58%	44%	52%
Habit App	62%	49%	39%	32%
Simple Weight Tracker	65%	58%	45%	48%
BBC News	52%	43%	28%	35%
Any Memo	53%	44%	38%	33%
Contact Manager	69%	60%	51%	53%

Table 5.5 The comparisons between proposed approach and other approaches in terms of executed test actions for subject applications (e.g., Monkey, Dynodroid).

App Name	Approach	Monkey	Dynodroid
Tureng	418	2000	2000
To Do Manager	365	2000	2000
Tippy Tipper	180	2000	2000
Munchlife	65	2000	2000
Alarm Klock	207	2000	2000
Habit App	245	2000	2000
Simple Weight Tracker	202	2000	2000
BBC News	380	2000	2000
Any Memo	237	2000	2000
Contact Manager	271	2000	2000

higher code coverage than the test cases generated randomly. In the comparisons between our approach and other approaches (e.g., random sampling, Monkey, Dynodroid), we showed that the use of covering arrays to generate systematic samples is an effective factor in terms of getting higher code coverage.

In the third analysis, we stored the number of test actions executed by both our approach and other approaches during testing and compared the results. We determined the equal numbers of test actions to be executed by other approaches because of the limitation of initializing the maximum number. In *Table **5.5***, the comparison showed how the number of executed test actions affected the coverage of the applications. The data in the table indicated that proposed approach provides higher code coverage with less number of executed test actions when compared to other approaches. Moreover, the number of Android screens in the applications has an effect on the test actions executed by the approach, showing that there is a correlation be-

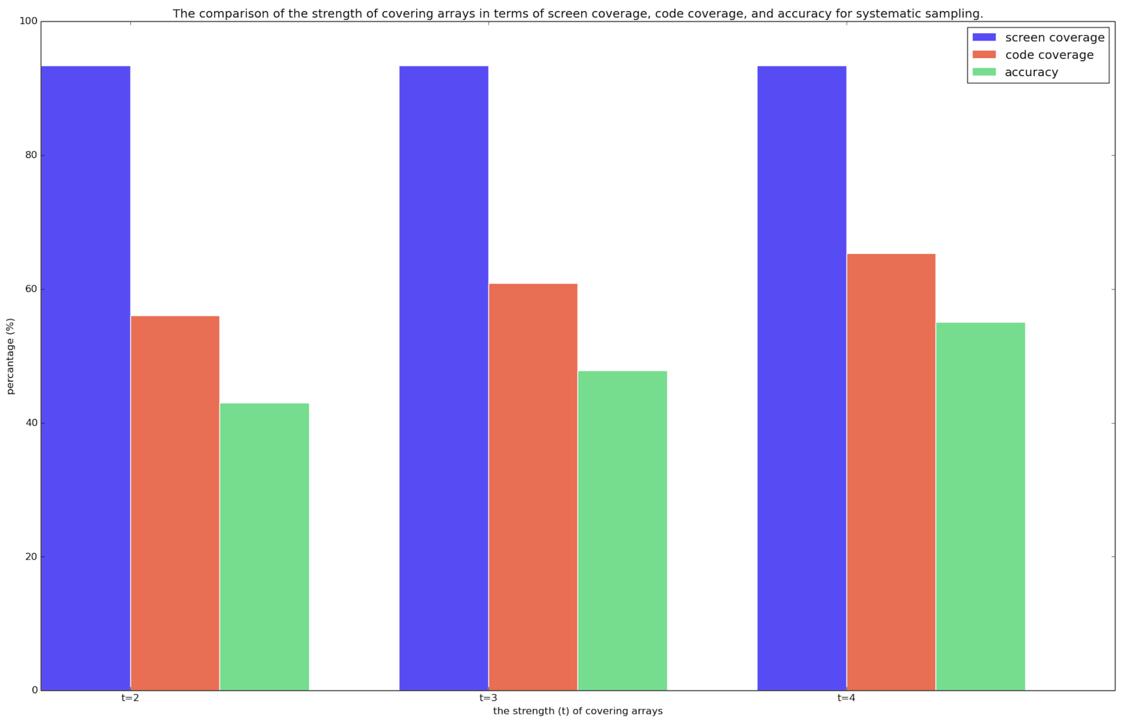
Table 5.6 Average cross-validation results based on guard conditions of subject applications ($C1$:application, $C2$:discovered number of screens, $C3$:discovered number of guard conditions and $C4$:cross-validation accuracy).

$C1$	$C2$	$C3$	$C4$ (%)
Tureng	30	89	43%
To Do Manager	15	45	49%
Tippy Tipper	4	6	68%
Munchlife	5	10	70%
Alarm Klock	19	41	51%
Habit App	8	12	60%
Simple Weight Tracker	6	11	54%
BBC News	14	37	38%
Any Memo	15	34	58%
Contact Manager	9	19	45%

tween the number of executed test actions and the number of Android screens in the application. However, we transparently know that the number of screens is not the only case. As we demonstrated in the second analysis (*Table*-***5.4***), the applications that have more complex screens and highly constrained conditions were covered less such as Habit App and BBC News. The complexity of the subject applications is related to the guard-complexity of the conditions between the screens. Then, we observed that the complexity of an application is a significant factor in this analysis.

In the fourth analysis, *Table* ***5.6*** demonstrated the cross-validation results of predictions with respect to the guard conditions of subject applications on average. In terms of given subject applications as parameters, the number of screens and the number of guard conditions of the transitions discovered by the approach were clearly measured. Since the number of screens showed how many each distinct screen discovered, the number of guard conditions represented how many transitions were satisfied between the screens. Basically, the data in *Table* ***5.6*** indicated that more complex applications have a huge number of guard conditions (e.g., Tureng, Alarm Klock or To Do Manager). As shown in *CV Accuracy* column, the predicted accuracy of guard conditions were not enough high. The major reason was the mechanism of real applications. As researchers, we did not know the expected behaviors of the the subject applications when compared to the simulations in (*Study* ***5.1***. Since the comparisons of the prediction results with the guard conditions were quite difficult because of a reason as mentioned above and the size of the train data was not much enough to build a better classifier, we, therefore, measure cross-validation accuracy for the predictions of the guard conditions.

Figure 5.8 The comparison of the strength t of covering arrays in terms of screen coverage, code coverage, and accuracy for systematic sampling.

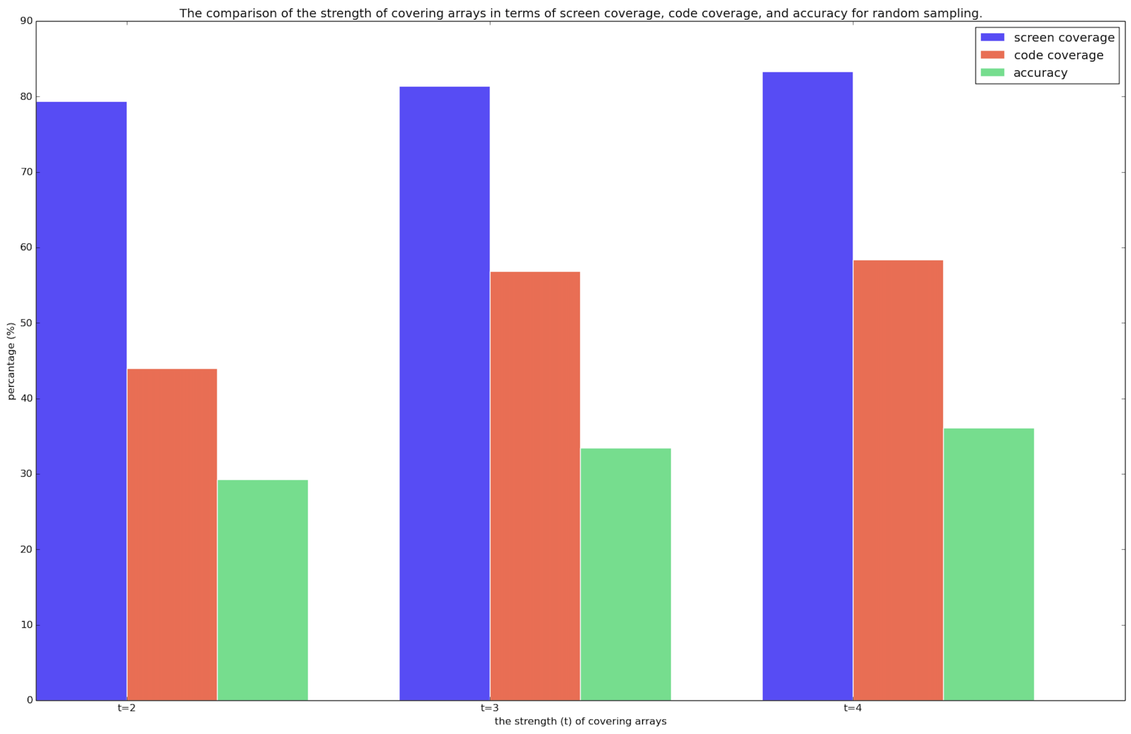


In addition, in *Figure* ***5.8***, we observed that if the strength of covering arrays increased, the code coverage and accuracy of predicted guard conditions increased for the subject applications such that we observed in the simulation study (Section **5.1**). Although the screen coverage was same 93.37% for $t = 2$, $t = 3$ and $t = 4$, the code coverage and accuracy of predicted guard conditions obtained from $t = 2$ were 56.02% and 43.02%, those obtained from $t = 3$ were 60.86% and 47.87%, and those obtained from $t = 4$ were 65.37% and 55.02%, respectively.

Last but not least, in *Figure* ***5.9***, we compared the strength of covering arrays in term of the state, code coverage, and the accuracy of predicted guard conditions when we applied random sapling for the testing process of mobile applications. We first observed that if the strength of covering arrays increased, the state, code coverage and accuracy of predicted guard conditions increased for the subject applications.

While the screen coverage obtained from *Figure* ***5.9*** were 79.37%, 81.33%, and 83.21%, respectively where $t = 2$, $t = 3$ and $t = 4$, the code coverage and accuracy of predicted guard conditions obtained from $t = 2$ were 44.02% and 29.32%, those obtained from $t = 3$ were 56.86% and 33.45%, and those obtained from $t = 4$ were 58.35% and 36.12%, respectively.

Figure 5.9 The comparison of the strength t of covering arrays in terms of screen coverage, code coverage, and accuracy for random sampling.



In the random sampling comparison, we then observed that the random sampling gets higher code coverage than the existing tools (A. Machiry & Naik., 2013; AndroidMonkey, 2018), as systematic sampling provided, indicating that even if the test cases are produced randomly in the same approach, the random sampling is better than the existing tools in terms of the code coverage.

Also, the results obtained from the random sampling comparison showed that when we compared systematic sampling with random sampling, under the equal conditions (e.g., same number of test cases, same input domains, and associated equivalence classes), systematic sampling is better than random sampling, as we also observed in the simulation study (*Study* ***5.1***), in terms of the state and code coverage, and the accuracy of predicted guard conditions, while the real applications under the test.

6. DISCUSSION

In this section, we will discuss the assumption of the proposed approach related to our research questions and clarify a solution for a given assumption. In particular, the approach assumes that the input domains and equivalence classes are given. Now, the question investigates whether we can use the same input domains and equivalence classes between the applications and domains.

6.1 Analyzing the feasibility of the input domains from the clusters

In this part, we will cluster the inputs to see the feasibility of using the input domains and their associated equivalence classes across different applications and domains. In other words, we want to see whether different applications and categories can share the input domains between each other, once we create the equivalence classes for each input domain, when new applications are tested, the input fields of these applications can use the shared input domains so that the manual effort of producing the domains and equivalence classes for each application will reduce.

6.1.1 Setup

In particular, we used 100 Android applications from F-Droid (Fdroid, 2020), which is a free and open source Android repository. In total, we selected 5 different categories and used 20 unique Android applications for each category. Information about these categories can be found in *Table **6.1***.

Also, we used the following term in the evaluation of clustering:

- **silhouette score:** a technique that provides a graphical representation of how well each object has been clustered. The range of score is between -1 and 1. If the score is near to 1, it indicates that the object is well matched to its own cluster. Otherwise, the clusters are not appropriate.

6.1.2 Approach and Evaluation Framework

In the approach, our aim is to analyze the feasibility of using the input domains and their associated equivalence classes to see whether the applications can share common input domains. In this case, we collect 100 applications from different 5 categories and applied the Agglomerative Hierarchical Clustering Algorithm (K.Sasirekha, 2013) for each category to make clusters by collecting the values used in Android applications.

As the first step, we used Apktool (Apktool, 2020) to extract the Android app files and get the resources. For each application, we took a *strings.xml* file, provided by Android, to get all the explanations used in the entire application. After collecting all strings from 100 Android applications, we used our algorithm developed for preprocessing the data (e.g., strings, values) and calculated the similarity scores between the values before clustering. In the preprocessing of the string values, we used the following techniques; *lower-casing, stemming, lemmatization,* and *stop words removal* (Extraction, 2018). In addition, we removed the non-English words to make the similarity more appropriate.

In the similarity calculations, we used a semantic similarity metric (Islam, 2008; Rau, Hotzkow & Zeller, 2018) to calculate the similarity score between two values. In semantic similarity, the similarity score was calculated by training *word2vec* (word vector model) on a large set of documents (Islam, 2008) and using a cosine similarity (Rau et al., 2018). If the similarity of the two values was near to 1, it showed that these values are very similar. On the other hand, they were far from each other in terms of the similarity. At the end of the similarity calculation process, we created the similarity matrix to be clustered. In the similarity matrix, the rows and columns represented the strings (e.g., please enter username, please type email) used in the applications and the cells were used as the similarity scores between each row and column. As a last step of the clustering process, we executed the clustering algorithm (e.g., Agglomerative Hierarchical Algorithm) on the similarity matrix of

Table 6.1 The information about the categories of Android applications.

Category	Descriptive Keywords
Finance	applications related to money, budget, economy and finance.
Sport & Health	applications related to sport, health, tracking and person.
Internet	applications related to internet, cloud, client and network.
Security	applications related to security, encryption and privacy.
Science	applications related to science, education and learning.

each category, and then analyzed the results in terms of how appropriate the clusters were.

For the evaluations, we used the silhouette score to see the distribution of the data and clusters. Also, we checked the predicted clusters manually to see whether the values in each cluster were similar or not. In addition to these evaluation metrics, we calculated our metric scores to support to the evaluation results coming from the silhouette scores. In our metric, we measured *in-cluster similarity* and iterated the measurement process as the number of clusters. In this evaluation, we checked the words in the predicted clusters by calculating the similarities between them once again. If the similarity s' of two words in the cluster was lower than the threshold, we labeled the s' as $false$. In this calculation, we counted the numbers of $false$ and measured the average in all clusters. The threshold t was $0.9 \leq t \leq 1.0$.

6.1.3 Operational Framework

In the clustering executions, we used 1) Agglomerative Hierarchical Algorithm, as a clustering algorithm, provided by scikit-learn (Scikit-learn, 2018), 2) the silhouette score, as a evaluation metric for the predicted clusters, provided by again scikit-learn (Scikit-learn, 2018), 3) Apktool (Apktool, 2020) as an extractor of Android applications. All the executions were carried out on the same computing platform used in *Section **5.1.3***

6.1.4 Data, Analysis and Discussion

We first applied Agglomerative Hierarchical Algorithm to Android applications from 5 different categories for clustering. In total, we used 24.345 string values taken

Table 6.2 The information about the results of clustering based on the categories of Android applications. ($C1$:Category, $C2$:Total Values, $C3$:Preprocessed Total Values, $C4$:Intra-Cluster Similarity, $C5$:Average Number of Apps per cluster, $C6$:Number of Clusters)

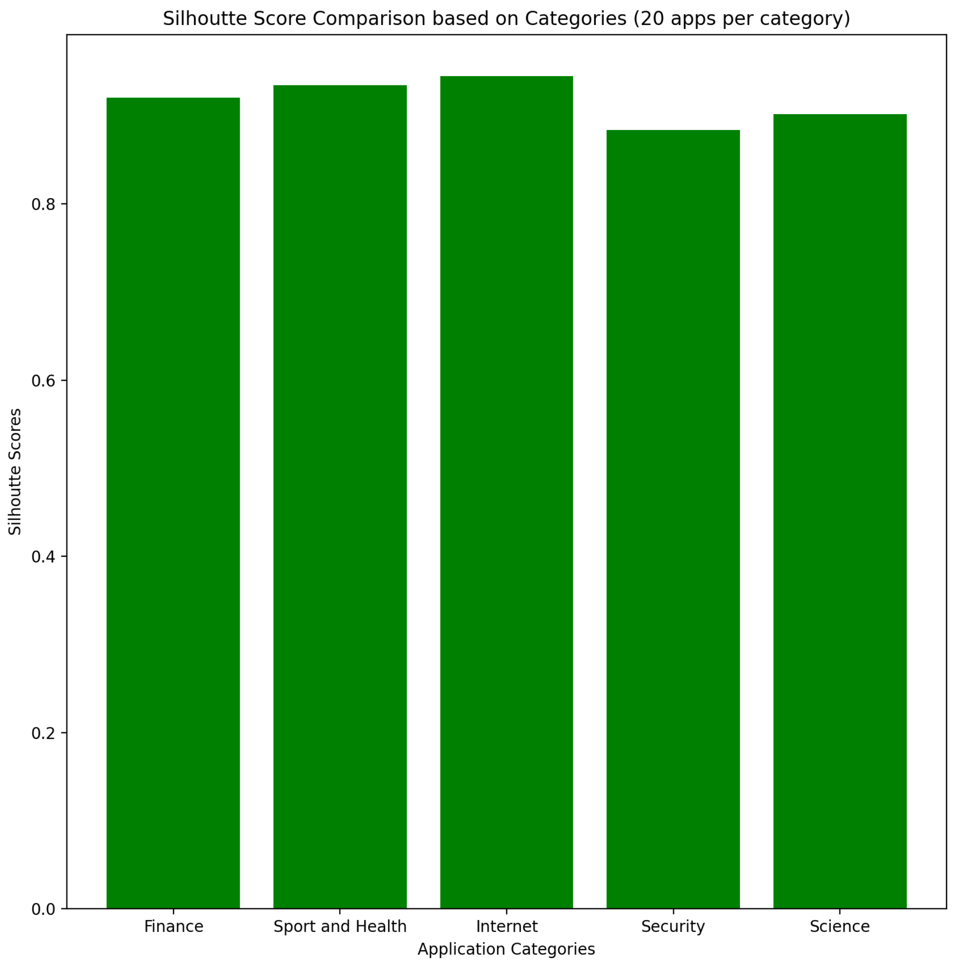
$C1$	$C2$	$C3$	$C4$	$C5$	$C6$
Finance	6235	6120	94.65%	3.4	502
Sport & Health	5369	4320	93.59%	4.3	431
Internet	5434	4738	95.32%	3.1	387
Security	4504	3813	89.92%	4.7	302
Science	5672	5210	91.39%	3.7	493

from *strings.xml* of each application and those values were stored in 134 different Android screens. Moreover, in total, we achieved 2.242 different clusters for all applications, indicating that each cluster was a distinct input domain. Thus, we observed that different applications can commonly share the input domains together. On the average, we observed that the values of each cluster were provided by 3.84 different Android applications in each category. In *Table **6.2***, we clarified the general information regarding the results of clustering based on the categories of Android applications. $C1$ represents the category of the applications used in the evaluation. $C2$ shows the total values collected from each category. $C3$ demonstrates the values that preprocessed from the total values of the applications to be clustered. $C4$ represents the intra-cluster similarity as silhouette score. $C5$ clarifies the number of different applications that trained in each cluster. Lastly, $C6$ shows the number of clusters obtained from the execution of clustering algorithm.

In the second analysis, we evaluated the clusters with their silhouette scores. In *Figure **6.1***, we compared the the silhouette scores in terms of 5 different categories. In each category, we used 20 Android applications. The silhouette scores obtained from *Figure **6.1*** were 0.921, 0.935, 0.945, 0.884 and 0.902, respectively. The data obtained from the evaluation of silhouette scores easily showed that the clustering approach we applied achieves high scores, indicating that the values taken from each category were well-matched in their clusters. In other words, each cluster, as an input domain, can be shared between the applications.

The results obtained from the clustering for 5 different categories in *Table **6.2*** showed that we can achieve high intra-cluster similarity scores, which indicates that the clusters are well-separated, for all categories. In other words, we observed that the values of each cluster are provided from different applications. Also, the results showed that the applications from the same category or from different categories can use the input domains as shared so that we do not need to make an effort to

Figure 6.1 Silhouette score comparisons based on the categories.



create the input domains and their associated equivalence classes for each application. On the other hand, since the values of the predicted clusters were the same or very similar to each other, we clarified that these values were used by more than 1 different Android application in the given category. The analysis of an average number of Android applications used in per cluster once again showed that the input domains of the different applications are typically similar and the input domains can be shared between each other. Once such clusters are obtained, the equivalence classes can be produced for each cluster and then given a previously unseen input field, the field can automatically be mapped to a cluster, and then the equivalence classes associated with the cluster can be automatically leveraged.

In terms of all evaluations in the discussion, we clarified that there are generally similarity between the domains of inputs from different applications and those input domains can be shared between the applications when we analyzed the results. Once the approach wants to detect the domain of a new input field that not clustered before, the input domain might easily be detected by using the clusters so that the manual effort of a domain detection will be decreased. In other words, we do not have to repeat the generation process of the domains and equivalence classes for the testing of each application.

In addition, we obtained that the input domains have as many similarities as the number of clusters between applications. If the number of clusters is high, the applications can share more input domains between each other. Thus, once we can generate the equivalence classes for each cluster, we can use them for many applications so that we can not make an effort for the manual process.

Also, we can observe the domain of the given input field from predicted clusters. In both solutions, we can choose test values from Android applications by applying a clustering approach, which is one of the machine learning approaches, instead of choosing the test values from the databases generated by us making searches and providing the domains manually. Also, what we demonstrated that as domains seem to be shared across different applications the manual effort required to come up with equivalence may not need to be duplicated, which increase the practicality of the proposed approach.

7. THREATS TO VALIDITY

All empirical studies suffer from threats to their internal and external validity. For this work, we were concerned with threats to both of the internal and external validity since they limit our ability to generalize the results of our experiment to industrial practice.

7.1 External Validity

One threat concerns the representativeness of the subject applications used in the experiments. There was a limited number of mobile applications used in the experiments. The reason was that they have been used and well-known by many research papers in the literature (A. Machiry & Naik., 2013; AndroidMonkey, 2018; R. Mahmood & Malek., 2014; S. R. Choudhary & Orso, 2015; W. Yang & Xie., 2013) so that we could easily compare our approach with another approaches (A. Machiry & Naik., 2013; AndroidMonkey, 2018) using the same applications in terms of different evaluation metrics.

Yet another threat concerns the representativeness of the traditional covering array generator used in the experiments, namely, ACTS (ACTS, 2018). In the literature, ACTS is a well-known and widely used generator. We opted to use only one covering array generator as ACTS since it offered the best run-time performance among the generators we experimented with, and other generators could also provide same properties to generate the covering arrays of the screens for mobile applications and the models in the simulations.

7.2 Internal Validity

Firstly, a potential internal threat is that the proposed approach assumes that all the input domains and their associated equivalence classes (e.g., test values) are given. In other words, the input domains and equivalence classes are produced for each application. It means that if the approach cannot match the domain with a given input field by checking its attributes, the equivalence classes cannot be produced. For this reason, we should add a new input domain and its equivalence classes in the database indicating that there needs to be manual effort for producing.

A second internal threat is about the structure of the guard conditions. In the approach, we assumed that the guard conditions are not like the arbitrary functions of the state parameters. Since we interacted with only UI of the applications and applied no static analysis, it was typically not possible to predict the guard conditions between the states, if a guard of the conditions was structured with the system conditions (e..g, the level of a battery, GPS, WIFI, Cellular) or created with the operational conditions implemented in the source code (e.g., a + b > c, number % 2 == 0, city like '%Istanbul%').

To this end, we need to apply the approach to a larger number of the applications with real faults to analyze how well the approach works and further automate the clustering approach, explained in *Section **6***, in future work.

8. CONCLUSION

Exploration strategies such as model-based and random testing are very common for the automated testing of mobile applications. While the system is under test, the model of an application has an impact on the different use of purposes (e.g., generating test cases, representing test strategies, detecting bugs).

In many researches, model-based and random testing have been used to build the model of an application and/or generate test cases for the testing procedures, and also have been combined together. Since the input space is not provided by the systematic sampling in random testing and the model must be provided for the model-based testing approaches (Takala, 2011), the model is not discovered systematically. In addition, even if divergent systematic exploration strategies have been proposed in the literature, the systematicity is not applied by the covering arrays, indicating that they ignore the interactions of the input fields. In this paper, we have presented a novel tool that discovers the model of mobile applications automatically with systematic sampling as a black-box approach. The key contributions of our approach are (1) discovering the model of mobile applications with systematic samples generated by covering arrays and (2) predicting the guard conditions between the states, represented as Android screens, on the discovered models by leveraging a machine learning approach, known as a decision-tree classifier, on the results of the test executions.

In this work, we have developed a couple of algorithms. As a first algorithm, we have enhanced a mechanism that detects each distinct Android screen, the input fields (e.g., Buttons, EditTexts) of a screen, the input domain and its associated equivalence classes (e.g., test values) of the input fields for a given application as a parameter, and discovers the model of an application via systematic sampling. The second algorithm aims to systematically sample the input space for each distinct screen that discovered by the approach using covering arrays, a part of *CIT* combinatorial interaction testing, in order to execute all combinations of test values of a screen. In the last algorithm, we have leveraged a machine learning approach called as a decision-tree classifier to predict the guard conditions of the discovered model by making binary classification.

Lastly and most importantly, we have evaluated the approach with different case studies. In the first study, we have simply observed the approach using simulations so that we can control the model parameters and analyze the effects of these parameters on the performance of the proposed approach. In the simulations, we have randomly generated different state machines by creating the states and guard conditions on the transitions with the configuration parameters of the system. The simulation test results have demonstrated that the test cases generated in a systematic manner are an effective factor in order to predict the guard conditions and visit the states by satisfying the transitions. Also, under the equal conditions (e.g., same number of test cases, same domains and equivalence classes), the systematic sampling is better than the random sampling. In the second case study, we have systematically generated test cases and executed them on the subject applications by comparing with previously proposed tools in terms of the screen and code coverage, the accuracy of predicted guard conditions, and the number of executed test actions of different applications. The results of subject applications comparison have indicated that the approach provides higher code coverage than previously proposed tools in the literature (A. Machiry & Naik., 2013; AndroidMonkey, 2018), showing that it is more powerful by executing less number of test actions to crawl the application and discover the model. In general, the evaluation results of the approach have significantly shown that the approach is better than existing tools (A. Machiry & Naik., 2013; AndroidMonkey, 2018) and random sampling in terms of the code and state coverage of the applications and the accuracy of predicted guard conditions.

As future work, we plan to apply the proposed approach to a large number of Android applications to analyze how the approach works better. Secondly, we aim to automate the clustering approach discussed in *Section **6***. Lastly, as developed for Android application, we aim to enhance the proposed approach to test IOS applications and also discover the models of those applications by making infrastructural changes in the algorithms.

BIBLIOGRAPHY

A. Machiry, R. T. & Naik., M. (2013). Dynodroid: An input generation system for android apps. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 224–234.

ACTS (2018). Automated combinatorial testing for software. *https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software.*

AcvTool (2018). *https://github.com/pilgun/acvtool.*

Ahuja, R. K. (2017). Network flows: Theory, algorithms, and applications (1st ed.). *Pearson Education.*

AndroidDeveloper (2018). The guide for developers about android. *http://developer.android.com/guide/topics/fundamentals.html.*

AndroidMonkey (2018). *http://developer.android.com/guide/developing/tools/monkey.html.*

Apktool (2020). *https://ibotpeaches.github.io/Apktool/.*

Appium (2018). *http://appium.io/.*

Bhat, A. & Quadri, S. M. K. (2015). Equivalence class partitioning and boundary value analysis - a review, 1557–1562.

C. Yilmaz, S. Fouche, M. B. C. A. P. G. D. & Koc, U. (2014). Moving forward with combinatorial interaction testing. *Computer*, (2), 37–45.

Claessen, K. & Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, *35*(9), 268–279.

D. Amalfitano, A. R. Fasolino, P. T. B. D. T. & Memon., A. M. (2015). Mobiguitar: a tool for automated model-based testing of mobile apps. *IEEE Software*, *32*(5), 53–59.

D. Amalfitano, A. R. Fasolino, P. T. S. D. C. & Memon., A. M. (2012). Using gui ripping for automated testing of android applications. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, 258–261.

D. M. Cohen, S. R. Dalal, M. L. F. & Patton, G. C. (1997). The aetg system:

an approach to testing based on combinatorial design. *IEEE Trans. on Soft. Eng.*, *23*(7), 437–444.

Dalvik (2018). Code and documentation from android's vm team. *http://code.google.com/p/dalvik/.*

Extraction, F. (2018). *https://scikit-learn.org/stable/modules/feature$_e$xtraction.html.*

Fang, L. & Li, G. (2015). Test selection with equivalence class partitioning, 40–49.

Fdroid (2020). *https://f-droid.org/.*

H. van der Merwe, B. v. d. M. & Visser, W. (2014). Execution and property specifications for jpf-android.. *SIGSOFT Software Engineering Notes*, 1–5.

Hu, C. & Neamtiu, I. (2011). Automating gui testing for android applications. *International Workshop on Automation of Software Test, AST'11*, 77–83.

Islam, A., I. D. (2008). Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Disc. Data*, 1–25.

K.Sasirekha, P. (2013). Agglomerative hierarchical clustering algorithm - a review. *International Journal of Scientific and Research Publications.*

L. Mariani, M. Pezze, O. R. & Santoro, M. (2012). Autoblacktest: Automatic black-box testing of interactive applications. *International Conference on Software Testing, Verification and Validation, ICST'12*, 81–90.

Nariman Mirzaei, Joshua Garcia, H. B. A. S. & Malek, S. (2016). Reducing combinatorics in gui testing of android applications. *International Conference on Software Engineering,ICSE'16.*

Nie, C. & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, *43*(2).

Pradhan, S., R. M. . P. S. (2019). Coverage criteria for state-based testing: A systematic review. *International Journal of Information Technology Project Management (IJITPM).*

R. Mahmood, N. M. & Malek., S. (2014). Evodroid: Segmented evolutionary testing of android apps. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'14.*

Rau, A., Hotzkow, J., & Zeller, A. (2018). Transferring tests across web applications, 50–64.

S. Anand, M. Naik, M. J. H. & Yang., H. (2012). Automated concolic testing of smartphone apps. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, *59*, 1–11.

S. Hao, B. Liu, S. N. W. G. H. & Govindan., R. (2014). Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14*, 204–217.

S. R. Choudhary, A. G. & Orso, A. (2015). Automated test input generation for android: Are we there yet? *Proc. of ASE'15*, 429–440.

S. Roy Choudhary, M. R. P. & Orso, A. (2013). X-pert: Accurate identification of cross-browser issues in web applications. *In Proceedings of the 2013 International Conference on Software Engineering, ICSE*, 702–711.

Scikit-learn (2018). *https://scikit-learn.org/stable/.*

Shafique, M., . L. Y. (2010). A systematic review of model based testing tool support (technical report sce-10-04), carleton university, canada.

Store, G. P. (2018). *https://play.google.com/store.*

Takala, T., K. M. H. J. (2011). Experiences of system-level model-based gui testing of an android application. *Proc. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST'11*, 377–386.

V. Dallmeier, M. Burger, T. O. & Zeller, A. (2013). Webmate: Generating test cases for web 2.0. *In Software Quality. Increasing Value in Software and Systems Development, Springer*, 55–69.

van Deursen, A. & Lenselin, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web.*

W. Yang, M. R. P. & Xie., T. (2013). A grey-box approach for automated gui-model generation of mobile applications. *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13*, 250–265.

Z. Liu, X. Gao, X. (2010). Adaptive random testing of mobile application. *International Conference on Computer Engineering and Technology, ICSET*, 297–301.