ITC 2/48
Journal of Information Technology and Control
Vol. 48 / No. 2 / 2019
pp. 316-334
DOI 10.5755/j01.itc.48.2.21725

Automatic Test Set Generation for Event-Driven Systems in the Absence of Specifications Combining Testing with Model Inference

Received 2018/09/28 | Accepted after revision 2019/04/08

crossref **http://dx.doi.org/10.5755/j01.itc.48.2.21725**

Automatic Test Set Generation for Event-Driven Systems in the Absence of Specifications Combining Testing with Model Inference

Luigi Novella, Manuela Tufo, Giovanni Fiengo

Università degli Studi del Sannio, Department of Engineering, Piazza Roma 21, 82100 Benevento, Italy,
e-mail: {luigi.novella,manuela.tufo,gifiengo}@unisannio.it

Corresponding author: luigi.novella@unisannio.it

The growing dependency of human activities on software technologies is leading to the need for designing more and more accurate testing techniques to ensure the quality and reliability of software components. A recent literature review of software testing methodologies reveals that several new approaches, which differ in the way test inputs are generated to efficiently explore systems behaviour, have been proposed. This paper is concerned with the challenge of automatically generating test input sets for Event-Driven Systems (EDS) for which neither source code nor specifications are available, therefore we propose an innovative fully automatic testing with model learning technique. It basically involves *active* learning to automatically infer a behavioural model of the System Under Test (SUT) using tests as queries, generates further tests based on the learned model to systematically explore unseen parts of the subject system, and makes use of *passive* learning to refine the current model hypothesis as soon as an inconsistency is found with the observed behaviour. Our passive learning algorithm uses the basic steps of Evidence-Driven State Merging (EDSM) and introduces an effective heuristic for choosing the pair of states to merge to obtain the target machine. Finally, the effectiveness of the proposed testing technique is demonstrated within the context of event-based functional testing of Android Graphical User Interface (GUI) applications and compared with that of existing baseline approaches.

KEYWORDS: Event-Based testing, Android GUI testing, Model Learning, Model-Based exploration, Automata.

1. Introduction

Guaranteeing the quality and reliability of software components is often a key factor for business. It becomes even crucial for human safety in the context of safety-critical systems. To achieve qualitative and robust software, an intensive testing phase must be performed. Our interest concerns with testing black-box event-driven systems at the system level, where systems behaviour is checked with specific input event sequences. This testing process requires SUT formal specifications or test models to automatically generate test cases that capture the system behaviour and to determine whether a test has passed or failed (test *oracle* problem). However, such specifications are rarely written out because developers cannot design and maintain system models in the actual agile context in which requirements and implementation are constantly in flux. In this scenario, testing software elements becomes very challenging because there is no basis upon which to select suitable test inputs for the subject system. To deal with this problem, test engineers can use both their intuition and background to drive the testing process or, more often, a random test input generation approach. Both of these approaches have several disadvantages in the sense that they often cannot lead to build a representative set of test input event sequences capable of investigating all the facets of software behaviour. Moreover, the absence of specifications and test models makes it difficult to estimate the quality and the adequacy of a test set. To address this problem, new academic research has focused on several learning-based approaches that investigate the use of Machine Learning in the software testing domain [17, 18, 9, 1, 33, 37, 16]. These techniques automatically infer models from the observed system behaviour and generate new tests based on the learned model.

The idea of combining model inference with software testing was first introduced by Weyuker [42], but it is only recently that the topic has become very popular. Remarkable success has been obtained in the area of testing reactive systems by combining *active* learning and model checking to infer a black-box SUT model using tests as queries [17, 30]. An alternative to this approach is to use model synthesis techniques to automatically derive behavioural models via *passive* learning from pre-recorded execution traces and to guide the generation of new tests based on the inferred models [37, 39].

In this paper, we address the problem of automatically generating test input sets for event-based Graphical User Interface (GUI) systems, for which we have neither an existing model of the GUI nor a complete specification, to achieve significant *functional coverage*. In recent years, both researchers and practitioners have investigated different approaches for testing event-based systems through their GUI. These techniques mainly differ in the way they generate test inputs and for the strategy they adopt to discover the SUT behaviour. In particular, much effort has been focused on applying these techniques on mobile applications. In this field, automated GUI exploration techniques aim to detect the set of events that can be fired at each screen to guide the execution of the application. These events may be chosen either randomly or via a systematic exploration strategy of the GUI.

In this context, this paper proposes a fully automatic *online* black-box testing technique that combines active learning, a systematic exploration of the GUI and passive learning. The testing approach effectiveness is demonstrated by applying it to Android applications. The proposed approach proceeds in a similar way as described in [9], however it introduces some relevant innovations that will be highlighted in the rest of the paper.

Based on SUT interface descriptions and domain knowledge, our testing technique probes the system behaviour with tests, uses the test results to on-the-fly learn an Extended Labelled Transition System (ELTS) model of the SUT, generates further tests on the learned model and refines it via inductive inference as soon as an inconsistency is found between the current ELTS model hypothesis and the set of execution traces observed so far. With reference to the proposed ELTS inductive algorithm, it adopts the basic steps of the Evidence-Driven State Merging (EDSM) algorithm in the "blue fringe" framework [24, 39] and introduces a novel heuristic approach according to the order in which state pairs are chosen for merging during the generalization process.

In this setting the learned model cannot be validity checked against a formal specification of the SUT. As

a consequence, the proposed testing technique terminates by approximating the equivalence check between the ELTS model and the SUT.

The effectiveness of our testing with model learning technique is demonstrated by measuring the extent to which the generated test set covers the functionalities of the SUT in terms of the observed behaviour. The learned model represents the result of what we have observed during the test execution process, therefore we use it as a basis for estimating the functional coverage in a similar way as in [37]. To confirm the analysis, we also measure the method coverage via app bytecode instrumentation. We highlight that our testing technique does not need app bytecode instrumentation to work; this is implemented just to perform this kind of measurement.

The experimental results show that our solution can outperform current baseline approaches in terms of test set adequacy. Moreover, the statistical significance of the results has been evaluated via rank-based statistical testing. The rest of the paper is organized as follows. Section 2 gives a brief overview of the existing automated GUI exploration strategies for Android GUI apps. The third section provides the mathematical definitions related to model inference and introduces the trace encoding process used to infer the sequential behaviour model of an Android GUI App. In Section 4 the proposed fully automatic *online* black-box testing technique is described. Section 5 and 6 give an overview of the Deterministic Finite Automaton induction problem in the *learning from an informant* setting and describe the proposed passive learning algorithm used to refine the ELTS model respectively. Technical details regarding the implementation of our solution and the experimental results obtained in the context of event-based functional testing of Android applications are given in Section 7. Finally, our conclusions are drawn in the last section.

2. Related Work

In the last few years, the ever-growing demand for mobile applications has led to the development of a set of frameworks and methods for automating GUI testing. In particular, several studies have been conducted to highlight the strengths and weaknesses of different testing frameworks and automated GUI exploration strategies for Android apps [2, 10, 29, 11, 21, 26, 34].

Android mobile applications are Event-Driven Systems (EDS) that can sense and react to events of different types. They may be either events generated through the application user interface, or system events produced by the device hardware platform and other running applications. Existing test input generation approaches are able to generate such events either randomly or systematically. In the first case, a random event is chosen at each GUI screen to navigate the app [20, 3, 44, 41, 28, 31, 27]. On the other hand, systematic exploration strategies select new actions to be executed based on the app GUI model [9, 5, 4, 6, 43, 7, 40, 25, 8, 32].

The remainder of the current section briefly describes these two main approaches along with the most popular Android GUI testing tools.

2.1. Random Exploration Strategy

Random testing represents the most popular approach to automatically explore GUI-based software systems in the absence of specifications and test models. It consists of randomly selecting events (GUI events or system events) during the execution of the application to probe the system behaviour. However, the approach effectiveness strictly depends on the characteristics of the SUT; therefore, a random exploration strategy may be generally weak at selecting specific inputs resulting in a poor or partial coverage of the system behaviour. Moreover, when both specifications and source code are missing, it becomes really difficult to determine a stop termination criterion and define the success or not of the exploration strategy. The common practice is to manually specify a time budget limit for the testing procedure.

The first random automatic test input generation technique was proposed by Hu et al. [20]. It uses the Monkey tool [35] which comes together with the Android developer toolkit. The approach consists of generating and sending random events such as clicks, scroll, system-level events to the SUT in order to detect GUI bugs. Relying on the Monkey tool, many other random GUI exploration tools have been developed [3, 44, 41]. Adaptive Random Testing (ART) [27] is a different random testing approach. The technique consists of selecting random sequences of events

that, according to a distance metric, are farthest from the already executed ones. The approach results in a better distributed generation of events sequences if compared with pure random testing. Dynodroid, proposed by Machiry et al. [28], performs a random selection of events considered to be relevant for the app. The authors state that an event is relevant to an application if it registers a listener for that event by means of the Android framework.

Finally, Morgado et al. [31] propose a random testing approach within the iMPAcT tool that automatically analyses a mobile application with the aim of identifying and testing its recurring behaviour (UI Patterns). To achieve the goal, the iMPAcT tool explores the current state of the application, identifies all the events that can be fired and then uniformly selects at random one of them to verify if the specific UI Pattern is correctly implemented.

2.2. Systematic Exploration Strategy

Most of the systematic exploration approaches generally build a behavioural model of the application under test during its activity. Then, based on the obtained model, new test input events are generated and executed to discover the app behaviour. According to various systematic traversal strategies, several Android GUI exploration tools have been developed.

In [5, 4, 6], Amalfitano et al. propose several contributions to the field of Android GUI app exploration. The designed approach builds a GUI model of the app and uses *ripping* to automatically explore it. For each newly visited GUI state, the procedure keeps all the events that could be fired in the current state and systematically executes them. The process terminates as soon as all the GUI app states have been explored. Yang at al. [43] implement a grey-box exploration strategy for automatically obtaining a model of the app under test in the Orbit tool. It is composed by an action detector module and a dynamic crawler. The first module automatically extracts the relevant GUI events by statically analysing the app source code and the manifest file. The second module builds a model of the GUI app by exercising the detected events on the live application. The systematic exploration of the GUI app is obtained via a modified depth-first strategy. In [7], the authors describe Automatic Android App Explorer (A^3E). The approach consists of two different exploration strategies, a *Targeted Exploration* and a *Depth-First* one. The Targeted Exploration first performs a static bytecode analysis to build a Static Activity Transition Graph. Such graph is then used to systematically explore the running app. Differently from the first one, the second approach automatically explores all the activities and the Android GUI elements in a depth-first manner to infer a dynamic Activity Transition Graph model. For each activity, the procedure extracts the GUI components and systematically exercises them by firing their corresponding event handlers. The procedure stops when no more activities are found.

A different systematic GUI exploration strategy is the one proposed by Choi at al. in the SwiftHand tool [9]. It aims to learn a behavioural model of the GUI app in the form of a deterministic Extended Label Transition System (ELTS) using tests as queries. Further tests are then generated based on the learned model to discover new states of the application. If an inconsistency between the learned model and the app is found, the ELTS is rebuilt from scratch using the set of execution traces observed so far. Wang et al. [40] introduce the DroidCrawle tool to automatically traverse an application's GUI and achieve high GUI coverage. The crawler technique consists of automatically exploring the application under test via a depth-first approach and of inferring a GUI tree model at the same time. In [25], the authors introduce DroidBot, a lightweight UI-guided test input generator. It is able to generate UI-guided test inputs based on a transition model generated on-the-fly, and allows users to integrate their own testing strategies.

More recently, new techniques have been introduced with the aim of improving the effectiveness of model-based testing. Cao et al. [8] present the CrawlDroid tool that via a novel feedback based exploration strategy, allows to dynamically adjust the priority of the actions to execute. With this approach, actions that potentially have more chances to expose new states of the GUI app can be selected. In [32], the authors design a behavioral-based GUI testing approach in order to create a behavioural model based on usage logs by applying a statistical model. The approach consists of dynamically updating the model to increase the probability of selecting an event that rarely or never occurs when users use the application.

3. Background

The automatic inference of state-machine models has been intensively investigated within the machine learning domain of grammar inference. The problem of grammar inference is concerned with the process of identifying a language from positive (valid) strings that belong to the language and negative (invalid) ones that do not. Several inference techniques have been developed to reduce human effort in automatically generating state machine models from examples of software behaviours. These examples can either be in the form of scenarios extracted from models created during the development stage of a software system, or execution traces from the current implementation of a program.

The aim of this section is to provide the reader with the mathematical definitions related to model inference. It also introduces an example of the specific trace-encoding process used to infer the sequential behaviour model of an Android GUI application.

3.1. Definitions

In this paper, we restrict the discussion to those systems that can be modelled as an Extended deterministic Labelled Transition System (ELTS) [33]. A Labelled Transition System (LTS) [37] is an instance of a state machine often used to represent the behaviour of software systems. In intuitive terms, an ELTS model augments a conventional LTS with a state labelling function λ defining the set of enabled transitions at each model-state.

Most of the techniques used to infer software behaviour models take as input a set of program execution traces that consist of sequences of input events to which the system responds with actions. In this work, we adopt the same trace-encoding process described in [33] in which we assume that the interactions with the SUT can be characterised in terms of *event* labels belonging to the ELTS finite alphabet Σ. The definitions of *trace* and *trace projection* are formally given below.

Definition 1 [*Trace*]. An execution trace, or simply a *trace t* is a finite sequence of pairs of input events and sets of possible events at each traversed state *q*, starting from the initial state q_0. Formally a trace *t* is an element of $\left(\Sigma \times \wp(\Sigma)\right)^*$ where $\wp(\Sigma)$ is a subset of Σ.

Definition 2 [*Trace Projection*]. A *trace projection* $\pi(t)$denotes the sequence of input event labels in the trace *t*. Formally, if $t = \left(a_1, \Sigma_1\right),\ldots,\left(a_n, \Sigma_n\right)$, then $\pi(t) = a_1,\ldots, a_n$. The set of traces projections is denoted by $\pi(T)$.

We define a trace *t* to be *consistent* with an ELTS if and only if it can trigger in order the input event labels of the trace and passes through the same states labelled as in the trace. This concept is formally defined below.

Definition 3 [*Consistency*]. A trace $t = \left(a_1, \Sigma_1\right),\ldots, \left(a_n, \Sigma_n\right)$ is consistent with a given ELTS $M = \left(Q, q_0, \Sigma, \delta, \lambda\right)$ if and only if:

$$\bigwedge_{i\in[1,n]} q_{i-1} \xrightarrow{a_i} q_i \wedge \lambda\left(q_i\right) = \Sigma_i,\ q_1, \ldots, q_n \in Q. \quad (1)$$

Moreover, a trace *t* is said to be a *positive* trace of a system if it represents a feasible behaviour that the given system may exhibit, otherwise it is said to be *negative*. The rest of the manuscript will also refer to the language of the ELTS. It can be defined as the set of labels sequences accepted by *M* and represents the behaviour permitted by *M*.

Definition 4 [*The Language of an ELTS*]. Given some ELTS *M*, for a given state $q \in Q$, $L(M,q)$, is the language of *M* in state *q*, and can be defined as: $L(M,q) = \left\{w \mid \hat{\delta} \text{ is defined for } (q,w)\right\}$. The language of an ELTS *M* is defined as: $L(M) = \left\{w \mid \hat{\delta} \text{ is defined for } (q_0,w)\right\}$.

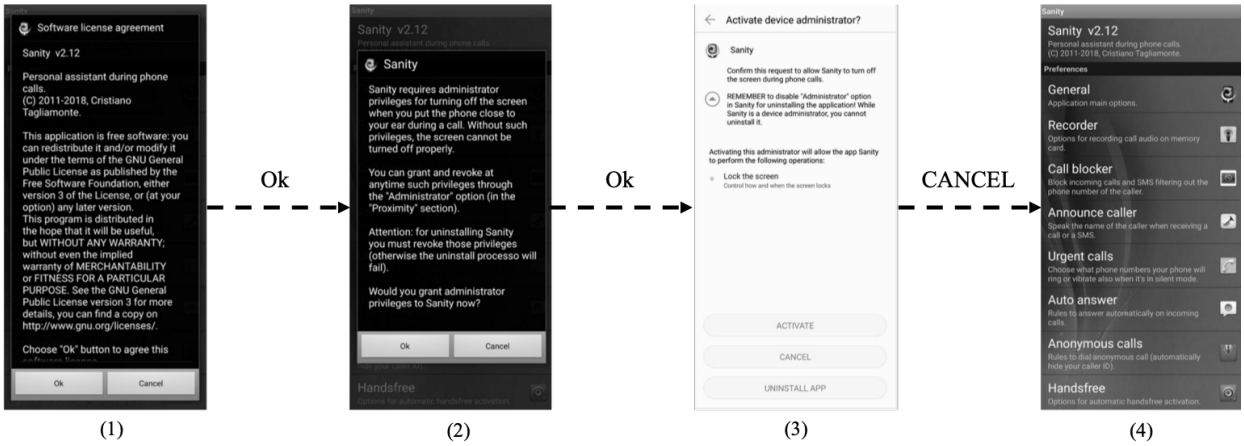
3.2. Illustrative Example

Let us consider the scenario where we are required to learn a model of the sequential behaviour exhibited by an Android GUI application without the availability of the app source code. In this setting, we assume that we are able to detect the GUI components at each visited app screen and, as a consequence, to automatically navigate the application through the generation of GUI events [2]. It is during such exploration that both the given input event sequences and the corresponding app behaviour are encoded in a set of execution traces.

As a running example we use the Sanity Android application to better explain the trace-encoding process needed to set up the behavioural model inference challenge.

Figure 1 shows one of the possible input sequences that, once given to the SUT, let the automatic exploration reach the main Sanity app screen starting from the initial one. This scenario of execution is encoded in the trace below:

Figure 1
An automatic GUI exploration



...
(a, set of available GUI events from screen 2)
(b, set of available GUI events from screen 3)
(c, set of available GUI events from screen 4)
...

Table 1
GUI events labels

GUI event label	GUI input action
'a'	*Click* on Button Ok of screen 1
'b'	*Click* on Button Ok of screen 2
'c'	*Click* on Button CANCEL of screen 3

According to Definition 1 this execution trace consists of a sequence of pairs (shown on a different line) of input event labels and GUI events available in the current app screen. Each event label represents a specific GUI event sent to the SUT as detailed in Table 1.

4. Testing with Model Learning

In this Section, the proposed fully automatic *online* black-box testing technique for GUI applications is described. The key idea is to combine testing with model learning to obtain a test input set capable of intensively exercising the functional behaviour of the SUT. To reach this goal, an *active* learning algorithm is used in conjunction with a systematic exploration of the GUI to build an ELTS behavioural model of the SUT. The learned model is then used with the aim of guiding the generation of new test inputs to discover as quickly as possible unseen parts of the subject system. Generally speaking, an active learning algorithm for inferring finite state machines iteratively asks for additional observations to complete its task and often implies a teacher-student relationship [12]. In our scenario, indeed, the designed testing with model learning procedure is thought to act as a teacher. It executes the SUT on specific test event sequences to get answers to two kind of different queries, *membership* and *equivalence* queries. Membership queries aim to realize whether the GUI app can trigger in order a particular sequence of events, whereas equivalence queries are intended to check if the current model hypothesis is consistent with the observed system behaviour, without the aid of formal specifications. Moreover, according to the taxonomy presented by Utting et al. [36], the overall testing procedure is defined as *online* in the sense that test generation and test execution are iteratively performed on-the-fly.

The testing with model learning process is described by Algorithm 1. It is based on the following practical assumptions:

- it is possible to detect the set of available user inputs at each app screen,
- the testing algorithm is able to restart the SUT, send events to the subject system and record its reaction.

Algorithm 1: Testing with Model Learning Algorithm

Input: the *SUT*
Output: the set of execution traces T, the ELTS M modelling the sequential behaviour of the *SUT*

```
1.  M, p, s, T, t ← initialisation()
2.  stop ← false
3.  while ¬ (timeBudget() ∨ stop) do
4.      if q ← findFrontierState(M) then
5.          if l ← isReachable(q, p, M) then
6.              (s, t) ← execute(s, t, l)
7.              if t is consistent with M then
8.                  a ← selectNextInput( λ(q) \ Σ_q^out )
9.                  (s, t) ← execute(s, t, a)
10.                 if there exists r in M s.t. λ(r) = λ(s) then
11.                     M ← addTransition(q, a, r)
12.                 else
13.                     M ← addState(r)
14.                     M ← addTransition(q, a, r)
15.                 end if
16.             else
17.                 (T, t) ← updateTraces(t)
18.                 M ← inferELTS(T)
19.             end if
20.         else
21.             (T, t) ← updateTraces(t)
22.             (p, s) ← restart()
23.         end if
24.     else if (l, stop) ← equivalenceCheck(projections(T)) then
25.         (s, t) ← execute(s, t, l)
26.         if t is not consistent with M then
27.             (T, t) ← updateTraces(t)
28.             M ← inferELTS(T)
29.         end if
30.     else
31.         return T, M
32.     end if
33. end while
34. return T, M
```

Due to the fact that no prior knowledge is available about the hidden state transition structure of SUT, an initial ELTS model of the Android GUI app is needed to start with the the model-based test input generation process. Therefore, the initialisation function launches the app to reach the initial system-state and queries the set of available GUI actions. The initial app-state s is abstracted as the initial ELTS model M containing the corresponding model-state p, whereas the detected set of available GUI events is modelled as the state labelling function value in p ($\lambda(p)$) (line 1). We use throughout the rest of this Section s to refer to the current app-state (current GUI screen) and p to indicate its corresponding abstract state in the model. At each iteration, the testing algorithm exercises new functionalities of the GUI via a model-based exploration strategy.

Definition 5 [*Frontier model-state*]. A state q in an ELTS model is a *frontier-state* if and only if there exists $a \in \lambda(q)$ such that $q \xrightarrow{a} u$ is not true for any $u \in Q$ [33].

The *findFrontierState* function is responsible of choosing a *frontier-state* q in the ELTS model that can be reached starting from the current model-state p, then the algorithm selects a sequence of event labels l that leads to q starting from p (lines 4-5). We heuristically pick the frontier-state q as follows:

- if during the exploration an app-state s is reached whose corresponding model-state p is a frontier-state in M, then p is selected as the next frontier-state to explore,
- otherwise, the algorithm selects the frontier-state q for which the number of diverse states to traverse, in the path that leads from p to q, is the greatest one.

Starting from s, the algorithm executes the SUT on the sequence of GUI events corresponding to l and obtains a trace of execution t (line 6). If t is consistent with the current model hypothesis M (Definition 3), the algorithm randomly picks an unexplored outgoing transition a from q ($a \in \lambda(q) \setminus \Sigma_q^{out}$), executes the app on the corresponding GUI action and updates both the current app-state s and the actual execution trace t (lines 7-9). If the state labelling function of an existing M state agrees with the set of available GUI events of the newly reached app-state s, a merge operation is performed in the model, otherwise a new fresh model-state p that abstracts s is added to M (lines 10-14). Two app-states are considered equivalent if they exhibit the same set of GUI components (enabled in the same boxes of screen coordinates). This is formally reflected in the ELTS state labelling function. Therefore, two model-states are compatible if they agree on λ.

The above mentioned *approximate* check of equivalence between two app-states and their corresponding model-states results in an aggressive merging strategy because it cannot take into account any future behaviour of the SUT starting from the states considered for merging. This approach may introduce an inconsistency between the model and the be-

haviour of the subject system observed so far. Indeed, if an inconsistency is found in the current ELTS model, the algorithm adds the current execution trace t to the set of traces T and refines M via passive learning (lines 17-18). The *inferELTS* function is exhaustively described by Algorithm 3 of Section 6. Whenever the selected frontier-state q is not reachable from the current model-state p, the testing algorithm updates the set of traces T with t, initialises t and restarts the app under test (lines 21-22) to select a new frontier-state and continue with the GUI exploration. The systematic exploration goes on until no more frontier-states are available in M, meaning that every transition from each state in the model has been taken (*model-based coverage termination criterion*), or that the predefined testing time budget is expired. In the first case, it is needed to check whether the resulting ELTS M is equivalent to the SUT. This is performed by generating a predefined number of random walks on the ELTS model that do not represent a subsequence of any $\pi(t)$. The SUT is then executed on these sequences of events and if a counterexample is found, the model is refined using the set of traces T observed so far to continue with the exploration.

4.1. Comparison with SwiftHand

Our proposed testing with model learning algorithm exploits the basic steps of the learning-guided testing algorithm designed by Choi et al. in [9]. In this paper, the authors present an automated testing technique called *SwiftHand* for generating sequences of test inputs for Android apps. A key feature of *SwiftHand* is that it achieves significantly better code coverage than traditional random testing and active learning-based testing by reducing the number of restarts needed to complete the systematic GUI app exploration process.

Here we want to highlight the main differences between our testing technique and *SwiftHand*. First of all *SwiftHand* randomly chooses the next frontier-state to explore, whereas our heuristic GUI exploration strategy moves in the direction of enhancing both the *test depth* and the ELTS state-coverage while minimizing the number of app restarts needed to complete the learning task. Indeed, we decide to make the exploration process as fluid as possible by privileging the choice of the current model-state as the next frontier-state to explore if possible or to maximize, with a different selection, the number of model states to traverse when it is not. Another relevant distinction concerns with the merging strategy adopted in the active model learning stage. Whenever more than one model-state is compatible for merging with the current one, then *SwiftHand* heuristically selects the nearest ancestor between them. The authors claim that this choice often avoids refining the model in future. However, in order to generalize the testing procedure, we randomly pick a model-state for merging and whenever an inconsistency is introduced into the model, due to an aggressive merging operation, we use the proposed passive learning algorithm to refine it. Both the testing strategies terminate by means of an equivalence check between the ELTS inferred model and the SUT but they differ in the way they implement it. In absence of specifications, as is the case here, the equivalence check is implemented executing untried scenarios until a counter-example is found because the ELTS model cannot be validity checked against formal specifications of the SUT. As soon as the ELTS model is complete, *SwiftHand* performs this check by executing a sequence of events l starting from the current model-state and by ensuring that l is not a subsequence of any trace projection $\pi(t) \in \pi(T)$. Moreover, if multiple transition sequences are available, *SwiftHand* uses a random walk strategy to select one of them. This heuristic to approximate the equivalence query suffers of the following drawback: if the current model-state corresponds to a terminal state, no subsequence would be found and *SwiftHand* terminates without executing any equivalence check. Our approximate check of equivalence between the ELTS model and the SUT consists of executing a predefined number of acceptance tests. Starting from the current model-state, we generate a set of random walks then we randomly pick a sequence of events that is not a subsequence for any trace projection. In the case the current model state is a terminal state, we restart the app under test and iteratively evaluate new paths.

Finally, the two strategies are totally different regarding the inductive inference algorithm used to re-learn the model whenever an inconsistency is found between the learned model and the SUT. *SwiftHand* refines the model via the Evidence-Driven State Merging (EDSM) algorithm with the Blue-fringe control strategy and exploits the idea of blocking constraints introduced by [23]. Our passive learning algorithm

adopts the basic steps of the EDSM algorithm in the "blue fringe" framework and introduces a novel heuristic approach according to the order in which state pairs are chosen for merging as described in Section 6.

5. The Synthesis of Software Behaviour Models

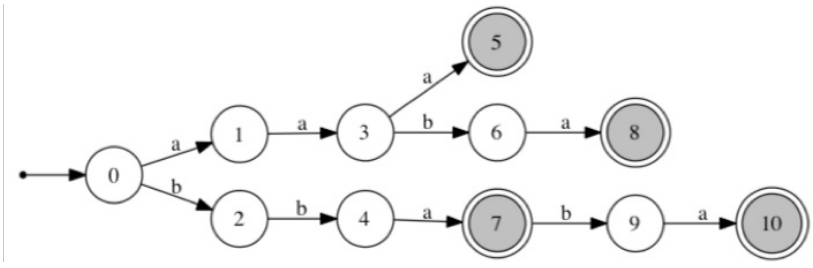
To better understand our inductive learning algorithm, we first give a brief review of the DFA induction problem in the *learning from an informant* setting [12]. The automatic learning of behaviour models from scenarios of interaction between the SUT and its environment can be interpreted as a Deterministic Finite Automaton (DFA) induction problem [15]. Starting from a set of system execution traces, the derived scenarios of interaction can be represented as strings over a finite alphabet of events (Σ) and they can be generalized to form a language of acceptable behaviours. Indeed, whenever behaviours are represented as finite-state machines, the problem is equivalent to induce a DFA from positive and negative strings. State-merging is the foundation for most successful techniques in inferring DFA from positive (S_+) and negative ($S\_$) examples. These algorithms start by building an initial automaton called *Prefix Tree Acceptor* (PTA) [12] accepting exactly S_+ and successively merge states to generalize the induced language.

5.1. State-Merging and Quotient Automaton

The generalization operation obtained by merging states of an original automaton A is formally defined through the concept of *quotient automaton* [15]. The set of possible generalizations which can be obtained by merging states of the original automaton can be searched through a lattice of partitions *Lat(A)* [14]. Figure 2 shows how, starting from the $PTA(S_+)$ (Figure 2a), states belonging to the same subset, or block, of π are merged in the quotient automaton (Figure 2b). Accepting states are represented as light grey nodes.

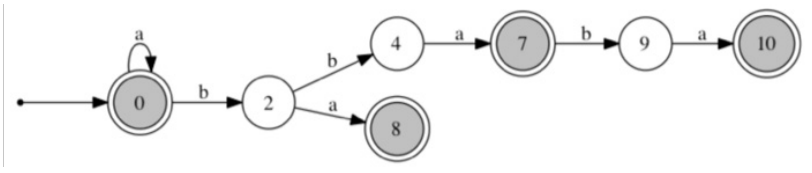
Merging blocks $B(0, \pi)$ and $B(1, \pi)$ belonging to the initial partition π defined on the *PTA* states set (Figure 2a) leads to a nondetermistic quotient automaton. Therefore, a recursive process of merging blocks is needed to obtain the deterministic quotient automaton shown in Figure 2b.

Figure 2
Generalization process



(a)

$PTA(S_+)$ with $S_+ = \{\varepsilon, aaa, aaba, bba, bbaba\}$ and ε denoting the empty string. The initial partition $\pi = \{\{0\}, \{1\}, \{2\}, \ldots, \{10\}\}$ is defined on the PTA states set



(b)

A quotient automaton where $A = PTA/\pi$ and $\pi = \{\{0,1,3,5\}, \{2,6\}, \{4\}, \{7\}, \{8\}, \{9\}, \{10\}\}$

5.2. Evidence-Driven State Merging

Evidence-Driven State Merging (EDSM) is considered to be the *state of the art* with respect to the inference of DFAs from positive and negative examples. It won the *Abbadingo* competition [24] and was used as a baseline for the *STAMINA* one [38]. The idea behind the EDSM approach is fairly straightforward. Given a sample, a PTA is built based on positive examples, then two states are iteratively selected and merged unless compatibility is broken. The state-merging challenge is to identify pairs of states in the current DFA hypothesis that represent equivalent states. It is critically important to obtain correct algorithm's early decisions, and hence a good strategy is to first perform those merges that are supported by the most *evidence*. A heuristic for choosing the pair of states to merge, can be realized in many ways. We show in Algorithm 2, the implementation of EDSM using the "blue-fringe" control strategy (state-merging ordering) [24].

Algorithm 2: EDSM Algorithm with Blue-fringe control strategy

Input: $S=(S_+,S_-)$
Output: $A=(\Sigma,Q,q_0,\mathbb{F}_A,\mathbb{F}_R,\delta)$

1. $A \leftarrow buildPTA(S_+)$
2. $Red \leftarrow \{q_0\}$; $Blue \leftarrow \{q_a : a \in \Sigma \cap Pref(S_+)\}$
3. **while** $Blue \neq 0$ **do**
4. $\quad promotion \leftarrow false$; $bs \leftarrow -\infty$
5. $\quad$ **for** $q_b \in Blue$ **do**
6. $\quad\quad$ **if** $\neg\, promotion$ **then**
7. $\quad\quad\quad atleastonemerge \leftarrow false$
8. $\quad\quad\quad$ **for** $q_r \in Red$ **do**
9. $\quad\quad\quad\quad s \leftarrow calculateScore(merge(q_r,q_b,A),S_+,S_-)$
10. $\quad\quad\quad\quad$ **if** $s > -\infty$ **then**
11. $\quad\quad\quad\quad\quad atleastonemerge \leftarrow true$
12. $\quad\quad\quad\quad$ **end if**
13. $\quad\quad\quad\quad$ **if** $s > bs$ **then**
14. $\quad\quad\quad\quad\quad bs \leftarrow s$; $\overline{q_r} \leftarrow q_r$; $\overline{q_b} \leftarrow q_b$
15. $\quad\quad\quad\quad$ **end if**
16. $\quad\quad\quad$ **end for**
17. $\quad\quad\quad$ **if** $\neg\, atleastonemerge$ **then**
18. $\quad\quad\quad\quad promotion \leftarrow true$
19. $\quad\quad\quad\quad Red, Blue \leftarrow promote(q_b, A, Red, Blue)$
20. $\quad\quad\quad$ **end if**
21. $\quad\quad$ **end if**
22. $\quad$ **end for**
23. $\quad$ **if** $\neg\, promotion$ **then**
24. $\quad\quad Blue \leftarrow Blue \setminus \{\overline{q_b}\}$; $A \leftarrow merge(\overline{q_r}, \overline{q_b}, A)$
25. $\quad$ **end if**
26. **end while**
27. **for** $x \in S_+$ **do**
28. $\quad \mathbb{F}_A \leftarrow \mathbb{F}_A \cup \{\hat{\delta}(q_o,x)\}$
29. **end for**
30. **for** $x \in S_-$ **do**
31. $\quad \mathbb{F}_R \leftarrow \mathbb{F}_R \cup \{\hat{\delta}(q_o,x)\}$
32. **end for**
33. **return** A

It begins by composing the positive sample (S_+) in the form of a PTA (line 1) and then continues merging states in order to produce a compact and generalized final DFA hypothesis. Starting from the PTA, the root is *coloured* red, its children are blue and the remaining states are white (line 2). At each iteration, the algorithm selects a set of pairs of states (one red, the other blue) as candidates for merging. The *calculateScore* function (line 9) then computes, for every selected pair of states, the score of that merge as the number of strings that end in the same state if that merge is done. To do that, the strings from S_+ and S_- have to be parsed. If by doing that merge (line 9), a conflict arises (a negative string ends in a final accepting state or a positive string is rejected) the score is equal to $-\infty$. Finally, the merge with the highest score is chosen (line 24). If during the generalization process, a blue node is unmergeable with a red one then it is promoted to the red set (line 17-20) and the algorithm continues by considering new candidates for merging until no further pairs of equivalent states can be found, which indicates convergence at the final DFA hypothesis. At the end of Algorithm 2, the final accepting and rejecting states are marked by parsing strings from S_+ and S_- and the final DFA hypothesis is returned (lines 27-33). The overall generalization process is controlled by the negative sample S_- to prevent merging those states that would lead to build an inconsistent machine, that is a DFA which accepts at least one negative string [12]. The availability of negative information is theoretically motivated since positive and negative samples are required to identify in the limit any super-finite class of languages, including the regular language class [19]. We use the basic steps of Algorithm 2 as a basis for our ELTS passive learning algorithm; however, a novel heuristic approach according to the order in which pairs of states are chosen for merging is introduced.

6. ELTS Model Refinement via Inductive Inference

This Section describes the state-merging algorithm used to refine the ELTS model from scratch as soon as an inconsistency between the learned model and the SUT is found. Unlike many similar machine learning techniques, we cannot presume the presence of negative examples. In our setting, we only have execution traces to work from which represent possible behaviours (positive examples) for the SUT. In case of few or no negative examples, one can sometimes rely on another kind of knowledge to prevent merging incompatible states, which is typically provided by the application domain [23].

When learning a GUI model, two user interface states can be considered equivalent if they have the same set of enabled user inputs. This information is modelled by the state labelling function (λ) in our ELTS model. In the absence of negative examples, the key idea of our approach is to let the generalization process be controlled by λ and to perform those merges that are supported by the most evidence. This aspect is dealt with in more detail in Subsection 6.1.

Algorithm 3 describes the proposed ELTS inductive learning algorithm. It begins by arranging the set of traces projections $\pi(T)$ (Definition 2) into an initial PTA M. It is a tree-shaped ELTS whose states are the set of all traces prefixes in $\pi(T)$ (line 1). Starting from the PTA, the algorithm iteratively chooses a pair of states deemed to be a suitable merge-candidate in the blue-fringe framework (line 14). Each selected pair is then scored by the *calculateScore* function (line 9). The higher is this score, the higher the evidence that the candidate states for merging are equivalent. Therefore, the pair of states with the highest score is deemed to be most likely to be equivalent, and is merged (line 24). In this setting, the merge operation always occurs between a red state and a blue state (one of the two candidate nodes is always the root of a tree, resulting in a simple algorithm for merging two nodes).

Algorithm 3: ELTS Inductive Learning Algorithm

Input: T
Output: M

1. $M \leftarrow buildPTA(\pi(T))$
2. $Red \leftarrow \{q_0\}$; $Blue \leftarrow \{q_a : a \in \Sigma \cap Pref(\pi(T))\}$
3. **while** $Blue \neq 0$ **do**
4. $promotion \leftarrow false$; $bs \leftarrow -\infty$
5. **for** $q_b \in Blue$ **do**
6. **if** $\neg$ $promotion$ **then**
7. $atleastonemerge \leftarrow false$
8. **for** $q_r \in Red$ **do**
9. $score \leftarrow calculateScore(M, q_r, q_b)$
10. **if** $score > -\infty$ **then**
11. $atleastonemerge \leftarrow true$
12. **end if**
13. **if** $score > bs$ **then**
14. $bs \leftarrow score$; $\overline{q_r} \leftarrow q_r$; $\overline{q_b} \leftarrow q_b$
15. **end if**
16. **end for**
17. **if** $\neg$ $atleastonemerge$ **then**
18. $promotion \leftarrow true$
19. $Red, Blue \leftarrow promote(q_b, M, Red, Blue)$
20. **end if**
21. **end if**
22. **end for**
23. **if** $\neg$ $promotion$ **then**
24. $Blue \leftarrow Blue \setminus \{\overline{q_b}\}$; $M \leftarrow merge(\overline{q_r}, \overline{q_b}, M)$
25. **end if**
26. **end while**
27. **return** M

The state-merging operation frequently introduces non-determinism into the ELTS hypothesis, which can then be removed by the classical *determinisation* procedure (as shown in Figure 2). The symmetrical merging operation, which requires determinisation through a cascade of merges is here replaced by the simpler asymmetric folding operation as in [12]. As described in [24], red states correspond to those states that have already been analysed by the generalization procedure and will be the states of the final ELTS. Blue states, instead, are the candidate states to consider for merging with a red one. If during the generalization process, the algorithm discovers that a blue state is unmergeable with any red node, it is promoted to the red states set (line 19). The generalization process continues until no further pairs of equivalent states can be found, which indicates convergence at the final ELTS hypothesis.

6.1. Heuristic State-Merging Ordering

Due to the lack of negative examples, the generalization operation performed by Algorithm 3 cannot be controlled in the same way as Algorithm 2 does to pre-

vent merging incompatible states. In our scenario, two ELTS states are considered to be incompatible if they disagree on the state labelling function (λ). Therefore, states having the same λ may be merged but states that disagree on λ must not be merged. This clearly explains how the generalization process is possible even if there is no notion of negative information. In order to obtain correct algorithm's early decisions, we propose a novel heuristic approach according to the order in which state pairs are chosen for merging that aims to identify as soon as possible incompatible states in the ELTS hypothesis. For each red-blue pair of states a score is computed measuring the evidence that the two candidates states are equivalent. This is done without actually performing the merge operation as classical state-merging algorithms do.

Let (q_r, q_b) be a red-blue pair of candidate states for merging in the current ELTS hypothesis M. According to Definition 4, we define $L(M, q_r)$ and $L(M, q_b)$ as the languages of M in q_r and q_b respectively, therefore $L_i = L(M, q_r) \cap L(M, q_b)$ represents the set of common words w accepted by M when starting from states q_r and q_b. Formally, the measure of evidence associated to (q_r, q_b) is computed as follows:

- a null score is associated if L_i is the empty set,
- provided that states q_r and q_b have the same λs, a positive score is associated if, when processing each $w \in L_i$ starting from q_r and q_b, no target states with different λs are encountered in the outgoing paths. In this case the score is equal to the L_i cardinality ($|L_i|$),
- a negative score is associated if states q_r and q_b have different λs or in the case that, when processing at least one word $w \in L_i$ starting from q_r and q_b, two target states that disagree on λ are encountered in the outgoing paths. In this case the score is equal to $-\infty$.

For example, let $M = (Q, q_o, \Sigma, \delta, \lambda)$ be the current ELTS hypothesis shown in Figure 3 and $\{0\}, \{1,5\}$, the current red and blue states sets respectively. Let $(q_r, q_b) = (0,1)$ be a candidate pair of states for merging. They agree on λ as shown in Table 2 and the initial score is set equal to 0. The set of common words L_i recognized by M when starting from states 0 and 1 is $L_i = \{'a', 'aa', 'aae', 'aaee', 'e', 'ee', 'b'\}$. In order to compute the score for $(q_r, q_b) = (0,1)$, each word w in L_i is evaluated starting from both q_r and q_b. Let us suppose we are interested in processing the word *w='aae'*, we obtain $\hat{\delta}(0, 'aae') = 5$, $\hat{\delta}(1, 'aae') = 11$. All the traversed pair of states in the outgoing paths from states 0 and 1, leading to states 1 and 11, agree on λ as it is shown in Table 2, therefore the score is incremented by one.

Table 2
ELTS M state labelling function (λ)

M states	λ
{0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 12}	{'a', 'b', 'c', 'd', 'e', 'f'}
{8, 10, 14}	{'a', 'h', 'b', 'g'}
{13, 15}	{'a', 'b'}

Figure 3
Current ELTS hypothesis

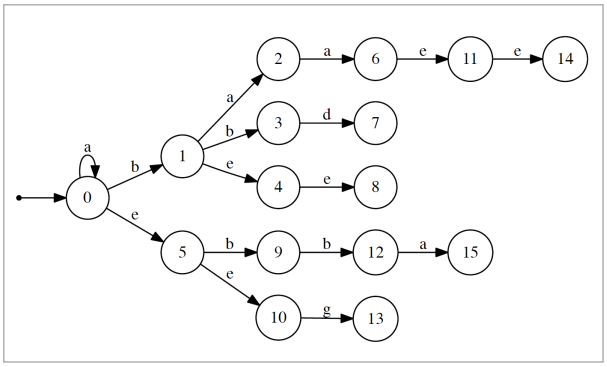
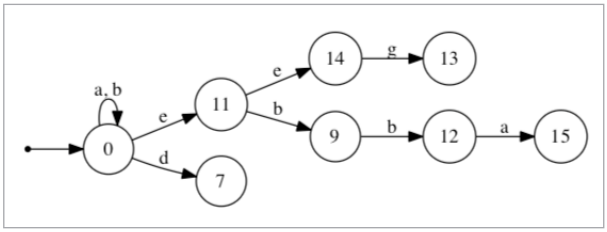


Figure 4
Updated ELTS hypothesis



This evaluation process is performed for each word w in L_i. In this case, no conflict arises on λ when processing all $w \in L_i$, then a score equal to $|L_i| = 7$ is associated to the pair of states $(q_r, q_b) = (0,1)$. Let $(q_r, q_b) = (0,5)$ be the other candidate pair of states for merging. The set of common words L_i accepted by M when starting from states 0 and 5 is $L_i = \{'bb', 'e'\}$. A score of $-\infty$ is computed for this pair of states because when processing *w='e'* starting from 0 and 5 two traversed states (5 and 10) that disagree on λ are encountered.

Finally, the pair of states $(0,1)$ is chosen for merging, resulting in the updated ELTS hypothesis of Figure 4. The scoring procedure is described in Algorithm 4. From an implementation point of view, the *calculateScore* function operates by counting the number of transitions in the outgoing paths from the two argument states (q_r, q_b) that share the same labels, (this is always finite because one of the states is always the root of a tree [39, 23]) provided that the target states of such equivalent transitions agree on λ.

The strength of our approach is that it can evaluate a priori the evidence that two states are equivalent or incompatible, differently from the control strategy described in [9] where the authors claim to use the classical blue-fringe ordering [24] with the idea of blocking constraints introduced in [23]. Due to the lack of negative examples, the proposed control strategy strongly takes into account the future behaviour in the outgoing paths of the candidates states for merging to evaluate the evidence that they are equivalent. Moreover, it does not need an unroll procedure and drastically reduces the total number of merges operation needed to obtain the target machine because the merge of a pair of states is executed if and only if the resulting score is positive.

Algorithm 4: CalculateScore

Input: q_r, q_b, M
Output: *score*

1. $score \leftarrow 0$
2. **if** $\lambda(q_r) \neq \lambda(q_b)$ **then**
3. **return** $score \leftarrow -\infty$
4. **else**
5. **for** $(t_1,t_2) \in equivalentTransitions\ (q_r,q_b)$ **do**
6. $score \leftarrow score + 1$
7. $d_1 \leftarrow \delta(q_r,t_1)$; $d_2 \leftarrow \delta(q_b,t_2)$
8. $score \leftarrow score + calculateScore\ (M,\ d_1,\ d_2)$
9. **end for**
10. **end if**
11. **return** *score*

7. Implementation and Experimental Results

Conventional coverage-driven testing approaches fully exercise the SUT in terms of its source code or specification. In our case, we address the problem of producing a test input set for EDS that are black-boxes, and for which neither formal specifications nor source code are available. Exhaustively executing the SUT on every test input is infeasible for most realistic software systems, and conventional random testing often fails to reach those system-states that are most likely to elicit an unexpected behaviour. Our goal is not to demonstrate that the behaviour of the subject system is functionally correct. This is practically impossible without a complete specification or a reliable test model. Instead, here the aim is to generate and execute a test input set that fully exercises the SUT in terms of its observable behaviour. Conceptually, a success in this direction allows us to infer a model of the sequential behavioural of the SUT using tests as queries. The definition of "observable behaviour" is strictly related with the characteristics of the SUT. For instance, the behaviour of a GUI mobile app is manifested in the variation of the GUI state in response to different GUI actions. The inferred model is then the result of what we learn from test execution, therefore it provides a functional perspective on the test set. Having said that, in this work we use the ELTS model as a basis for measuring the functional coverage in a similar way as in [37]. The reason behind this is that a larger model implies that a broad range of SUT behaviour is exercised. Therefore, we estimate the functional coverage by counting the number of individual transitions in the model. Moreover, it is generally acknowledged that longer test sequences tend to lead to a higher level of coverage. They are able to reach system-states that would remain unreachable otherwise. For this reason, the average length of the generated tests is recorded at the end of the testing procedure.

In this Section, technical details regarding the implementation of our testing with model learning technique are provided. Moreover, the experimental results obtained in the context of event-based functional testing of Android applications are compared to those achieved by a random testing approach and *SwiftHand*.

7.1. Implementation

The proposed testing with model learning algorithm (Algorithm 1) is written in the Python language. It does not need Android bytecode instrumentation for

the app under test to work and only uses the Android Debug Bridge (ADB) command tool to interact with the SUT. To proceed with the systematic exploration of the app, we need to dynamically extract the set of GUI components at each screen (*GUI Tree*), send events to the SUT and restart the app whenever it is required. These three activities are performed using the ADB tool. More in detail, the restart operation is implemented as follows: first, all data associated with the app package is deleted, then the app under test is launched again. To demonstrate the effectiveness of our solution, a comparison with two baseline approaches is performed. The first one is a random testing technique that is aware of the set of available GUI actions at each app-state. Indeed, during the app exploration, this strategy randomly executes a GUI action from the set of detected GUI events and only restarts the SUT when a terminal app-state is reached. All the scenarios of interaction between the automatic procedure and the SUT are recorded in a set of execution traces that is returned at the end of the testing procedure. The second testing technique we compare to is a Python implementation of the *SwiftHand* tool. *SwiftHand* [9] consists of a front-end module which performs a bytecode instrumentation of the app under test, and a back-end one which is responsible for the test input generation. Due to the technical limitations of its basic framework, we cannot use the original version available at https://github.com/wtchoi/SwiftHand, therefore we have implemented its back-end module. In order to perform an objective comparison of the three testing techniques and due to the fact that the random testing approach is not based on model learning, a time budget is chosen as the common termination criterion. As soon as the time-budget expires, the execution traces collected by the random testing algorithm are given to Algorithm 3 with the aim of inferring an ELTS model and estimate the functional coverage also in this case.

7.2. Experimental Setup

Our benchmark-suite consists of twelve Android apps available on the Google Play Store (see Table 3). All the experiments have been performed on a Samsung smartphone running the Android operating system (version 8.0.0) and using a 2.6 GHz Intel Core i5 Mac OS machine with 8Gb RAM. Moreover, we adopted 0.5 and 1 hours test budgets per app for each strategy.

Table 3
Benchmark Apps

Name	Category	Size (MB)
Sanity	communication	0.63
Alarm Klock	tools	0.59
EP Mobile	medical	2.2
FillUp	maps and navigation	0.93
TomDroid	productivity	1.1
TippyTipper	finance	0.8
Pedometer	health and fitness	7.5
My Expenses	finance	6.9
Weight Chart	health and fitness	3.3
Timetable	education	4.5
ToDoList	productivity	4.3
BMI	health and fitness	3.4

7.3. Experimental Results

Tables 4 and 5 summarize the results of applying the three different testing strategies to the selected Android apps with a time budget of 0.5 and 1 hours respectively. All values are the average of 10 experiments run for each testing algorithm. In the tables we use *TML*, *R* and *SH* to respectively denote the designed testing with model learning technique, random testing and our implementation of *SwiftHand*. For each app in the list, *#Functional Coverage* columns report the model-based estimation of the functional coverage, *#Test Depth* columns show the average length of the generated test set, *#Restarts* columns report the number of the app resets executed during the automatic exploration and finally *#Model-States* columns show the number of states discovered in the ELTS model.

The proposed testing with model learning technique always achieves better functional coverage than both random testing and *SwiftHand* within a time budget of 0.5 hour (Table 4). The gap even increases when the apps are executed within a time budget of 1 hour (Table 5), meaning that the designed testing technique achieves functional coverage at a faster rate than that of random testing and *SwiftHand*. This is an important aspect to consider when the available time budget for testing is strictly limited. For all the tested apps,

Table 4

Comparison Experiments (*time budget = 0.5 hour*)

App	# Functional Coverage			#Test Depth			#Restarts			#Model-States		
	TML	R	SH	TML	R	SH	TML	R	SH	TML	R	SH
Sanity	50.3	39.4	28.1	12.7	7.6	4.6	6.1	10.0	13.7	17.8	16.0	12.7
Alarm Klock	46.6	45.2	33.8	23.3	14.5	6.4	3.6	5.5	11.4	11.8	13.0	11.8
EP Mobile	72.0	64.25	34.0	26.5	52.5	5.0	2.5	1.1	14.5	28.0	23.7	16.5
FillUp	55.8	50.7	29.2	24.2	18.7	4.9	2.9	4.6	13.7	18.7	17.3	10.4
TomDroid	57.6	54.7	29.8	24.8	17.0	4.9	2.8	4.4	14.9	21.2	20.6	13.3
TippyTipper	48.6	40.6	33.9	20.7	15.7	6.0	3.4	4.5	11.5	11.6	9.7	9.9
Pedometer	41.7	38.7	32.3	8.7	3.7	4.3	8.0	15.7	14.7	24.3	17.3	13.7
My Expenses	57.8	56.9	33.5	23.4	21.9	6.5	2.8	3.5	10.5	25.4	23.2	16.0
Weight Chart	51.0	49.0	34.3	20.0	15.6	5.6	3.5	6.8	12.7	14.9	14.0	13.3
Timetable	50.4	48.6	27.1	13.5	9.6	3.9	5.0	9.0	15.6	19.4	17.3	13.4
ToDoList	56.8	52.8	24.0	9.8	7.6	3.0	6.8	12.0	17.0	22.2	19.2	10.5
BMI	40.6	29.3	37.5	15.0	9.7	9.7	4.8	8.3	11.0	12.8	9.7	11.0

Table 5

Comparison Experiments (*time budget = 1 hour*)

App	# Functional Coverage			#Test Depth			#Restarts			#Model-States		
	TML	R	SH	TML	R	SH	TML	R	SH	TML	R	SH
Sanity	88.3	59.2	48.6	14.1	7.5	4.9	11.3	22.2	26.1	30.0	21.0	18.0
Alarm Klock	86.9	69.6	64.0	29.5	14.3	8.7	4.6	12.6	17.5	21.4	20.1	22.6
EP Mobile	125.3	114.6	55.0	31.0	62.0	6.0	4.6	2.2	25.0	44.6	40.0	24.0
FillUp	86.1	73.7	52.7	27.4	15.9	5.8	5.1	11.8	23.7	25.2	20.7	15.7
TomDroid	87.1	83.2	54.9	28.5	21.2	6.7	5.1	8.6	22.9	30.0	28.6	21.5
TippyTipper	82.8	71.6	58.6	23.6	16.1	6.4	6.9	9.4	22.4	17.5	16.4	15.0
Pedometer	91.3	60.3	48.7	9.7	4.5	4.7	15.7	27.3	28.3	52.3	25.0	22.0
My Expenses	102.2	98.3	51.0	30.3	28.6	6.7	4.6	5.6	27.0	38.3	35.2	24.0
Weight Chart	84.3	75.8	58.0	29.9	26.3	6.3	4.8	9.7	22.3	16.7	16.1	16.0
Timetable	88.7	85.5	46.3	19.0	11.6	5.0	6.3	17.7	24.7	29.5	27.2	17.3
ToDoList	97.0	95.3	47.7	17.0	10.3	3.8	8.4	16.3	31.7	32.8	30.3	18.5
BMI	82.0	80.3	67.0	16.5	10.3	10.6	9.5	22.3	24	21.0	19.6	21.5

our solution performs a deeper exploration of the SUT than that of *SwiftHand* by generating longer test event sequences as it is shown by the *#Test Depth* columns values. As a consequence, *SwiftHand* restarts the app under test more frequently than our testing algorithm. The same analysis holds for the random testing strategy except if we consider the *EP Mobile* app. This app has few terminal states, therefore the random testing algorithm rarely performs resets. However, due to the GUI app structure this aspect is not reflected in the estimated functional coverage. In addition, to confirm the above analysis, we also consider measuring the method coverage. Indeed, Table 6 shows a further comparison in terms of *method coverage* between the different testing strategies. Even if the proposed testing with model learning technique does not need app bytecode instrumentation to work, it is only applied for the purpose of measuring the method coverage during the test execution. Given an Android application package (apk) file, the process consists of injecting debugging statements into it and prints a log message whenever an instrumented method is executed. We use the Soot library [22] to perform the instrumentation process. During the testing process, all the log messages are collected and then used to evaluate the method coverage. However, Table 6 measurements are coherent with the trend of previous analysis. This also implies the goodness of the criterion adopted to estimate the functional coverage based on the learned ELTS model.

Table 6
Method Coverage (*time budget = 1 hour*)

App	# IM	% Method Coverage		
		TML	R	SH
Sanity	1408	19.4	16.8	16.2
Alarm Klock	7856	2.8	2.5	2.0
EP Mobile	9561	8.0	7.8	6.9
FillUp	1783	36.7	28.8	17.3
TomDroid	1186	29.1	28.7	24.1
TippyTipper	5473	12.1	10.9	10.8
Pedometer	39594	17.0	16.3	16.8
My Expenses	47412	10.9	9.9	9.5
Weight Chart	2991	41.2	34.7	7.7
Timetable	23908	18.3	17.0	14.4
ToDoList	34682	12.8	12.3	12.3
BMI	31138	19.4	18.6	16.6

Table 7
Pairwise Comparison: *p-values*

	TML	R	SH
TML	-	0.03805	0.00100
R	-	-	0.44041

(a) Model-States

	TML	R	SH
TML	-	0.03805	0.00100
R	-	-	0.15772

(b) Test Depth

	TML	R	SH
TML	-	0.03805	0.00100
R	-	-	0.03805

(c) Functional Coverage

To ensure that the result is significant from a statistical point of view, we performed the Friedman test with the Nemenyi post-hoc test ($\alpha = 0.05$) [13]. First, Friedman test checks the hypothesis of "*no difference*" (null hypothesis) among the compared testing techniques, then the Nemenyi post-hoc analysis is performed to detect which of the considered techniques significantly differs from the others. Table 7 lists the results. The analysis, which has been carried out on the *#Functional Coverage*, *#Test Depth* and *#Model-States* columns values of Table 5 (1 hour test budget), confirms that the proposed approach is statistically significant in terms of functional coverage, test depth and number of model-states discovered during the testing process.

8. Conclusions

High-quality and reliable software has nowadays become the exception rather than the rule. This is especially the case of mobile apps whose demand is growing faster and faster. A knee-jerk reaction to guarantee qualitative and robust software is often to add resources to testing teams and perform intensive

testing. However, testing black-box event-driven systems require formal specifications or models that are rarely written out by developers in the current agile context. When neither complete specifications nor reliable test models are available, the test input generation problem becomes a very challenging task to solve because there is no basis upon which to select suitable inputs that properly investigate the subject system behaviour. To tackle this issue, we proposed an innovative learning-based testing technique for automatically generating test input sets for event-based driven systems and we demonstrated its effectiveness in the context of event-based functional testing of Android GUI applications. Our idea is to combine active learning with a systematic exploration strategy of the GUI and inductive inference. The proposed testing algorithm probes the system behaviour with tests and uses the test results to automatically learn a behavioural model of the SUT in the form of a deterministic ELTS. Based on the learned model, the procedure generates further tests to reach unseen parts of the running app. As soon as an inconsistency between the current model hypothesis and the observed behaviour of the subject system is discovered, the testing algorithm refines the inferred model via a state-merging based learning algorithm using the set of execution traces observed so far. With reference to the passive learning algorithm designed to refine the ELTS model, we introduced an effective heuristic according to the order in which state pairs are chosen for merging that drastically reduces the number of required merge operations to reach the target machine. We highlight that our goal is not to demonstrate that the subject system behaviour is functionally correct since the test oracle problem cannot automatically be solved in the absence of formal specifications. Instead, the aim is to generate and execute a test input set that fully exercises the SUT in terms of its observable behaviour. Therefore, the effectiveness of the testing technique is measured in terms of the test set adequacy and compared with that of two baseline testing strategies, SwiftHand and random testing. The test adequacy has been expressed via model-based estimation of functional coverage, test depth and discovered ELTS model states during the testing process. Moreover, measurements of the app restarts needed by the different approaches to explore the SUT and the method coverage have been preformed. The experimental results show that the presented testing with model learning approach is better than the compared ones at exploring the SUT behaviour within the same time budget. The reason behind this has to be found in the designed GUI exploration strategy that systematically tries to discover as soon as possible unseen system states. It also results in a better test depth and in few restarts when compared with the considered approaches. An evaluation of the statistical significance of the experimental results has been performed via rank-based statistical testing and presented. First, Friedman test has been carried out to test hypothesis of no differences among the compared testing techniques, then the Nemenyi post-hoc analysis has been performed to detect which of the considered techniques significantly differs from the others. The analysis confirms that the proposed approach is statistically significant in terms of functional coverage, test depth and number of model-states discovered during the testing process.

References

1. Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F., Verwer, S. Learning and Testing the Bounded Retransmission Protocol. Proceedings of the Eleventh International Conference on Grammatical Inference, 2012, 21, 4-18.
2. Amalfitano, D., Amatucci, N., Memon, A. M., Tramontana, P., Fasolino, A. R. A General Framework for Comparing Automatic Testing Techniques of Android Mobile Apps. Journal of Systems and Software, 2017, 125, 322-343. https://doi.org/10.1016/j.jss.2016.12.017
3. Amalfitano, D., Amatucci, N., Fasolino, A. R., Tramontana, P., Kowalczyk, E., Memon, A. M. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. 2nd ACM International Conference on Mobile Software Engineering and Systems, 2015, 33-43. https://doi.org/10.1109/MobileSoft.2015.11
4. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., Imparato, G. A toolset for GUI testing of Android applications. 28th IEEE International Conference on Software Maintenance (ICSM), 2012, 650-653. https://doi.org/10.1109/ICSM.2012.6405345

5. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., Memon, A. M. Using GUI Ripping for Automated Testing of Android Applications. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, 258-261. https://doi.org/10.1145/2351676.2351717
6. Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., Memon, A. M. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. IEEE Software, 2015, 32(5), 53-59. https://doi.org/10.1109/MS.2014.55
7. Azim, T., Neamtiu, I. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38. Applications, 2013, 641-660. http://doi.acm.org/10.1145/2509136.2509549.
8. Cao, Y., Wu, G., Chen, W., Wei, J. CrawlDroid: Effective Model-based GUI Testing of Android Apps. Proceedings of the Tenth Asia-Pacific Symposium on Internetware, 2018, 19:1-19:6. https://doi.org/10.1145/3275219.3275238
9. Choi, W., Necula, G., Sen, K. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. SIGPLAN Not., 2013, 623-640. https://doi.org/10.1145/2544173.2509552
10. Choudhary, S. R., Gorla, A., Orso, A. Automated Test Input Generation for Android: Are We There Yet? (E). Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, 429-440. https://doi.org/10.1109/ASE.2015.89
11. Coppola, R., Morisio, M., Torchiano, M. Mobile GUI Testing Fragility: A Study on Open-Source Android Applications. IEEE Transactions on Reliability, 2018, 1-24. https://doi.org/10.1109/TR.2018.2869227
12. de la Higuera, C. Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, 2010. https://doi.org/10.1017/CBO9781139194655
13. Demšar, J. Statistical Comparisons of Classifiers over Multiple Data Sets. Journal of Machine Learning Research, 2006, 7, 1-30.
14. Dupont, P., Miclet, L., Vidal, E. What is the Search Space of the Regular Inference? Grammatical Inference and Applications. Springer Berlin Heidelberg, 1994, 25-37. https://doi.org/10.1007/3-540-58473-0_134
15. Dupont, P., Lambeau, B., Damas, C., van Lamsweerde, A. The QSM Algorithm and Its Application to Software Behavior Model Induction. Applied Artificial Intelligence, 2008, 22(1-2), 77-115. https://doi.org/10.1080/08839510701853200
16. Esparcia-Alcázar, A. I., Almenar, F., Martínez, M., Rueda, U., Vos, T. E. J. Q-Learning Strategies for Action Selection in the TESTAR Automated Testing Tool. 6th International Conference on Metaheuristics and Nature Inspired Computing (META 2016), 2016, 130-137.
17. Feng, L., Lundmark, S., Meinke, K., Niu, F., Sindhu, M. A., Wong, P. Y. H., Case Studies in Learning-Based Testing. Testing Software and Systems, Springer Berlin Heidelberg, 2013, 164-179. https://doi.org/10.1007/978-3-642-41707-8_11
18. Fraser, G., Walkinshaw, N. Assessing and Generating Test Sets in Terms of Behavioural Adequacy. Software Testing, Verification and Reliability, John Wiley and Sons Ltd., 2015, 25(8), 749-780. https://doi.org/10.1002/stvr.1575
19. Gold, E. M. Language Identification in the Limit. Information and Control, 1967, 10(5), 447-474. https://doi.org/10.1016/S0019-9958(67)91165-5
20. Hu, C., Neamtiu, I. Automating GUI Testing for Android Applications. Proceedings of the 6th International Workshop on Automation of Software Test, ACM, 2011, 77-83. https://doi.org/10.1145/1982595.1982612
21. Kong, P., Li, L., Gao, J., Liu, K. , Bissyandé, T. F., Klein, J. Automated Testing of Android Apps: A Systematic Literature Review. IEEE Transactions on Reliability, 2018, 1-22. https://doi.org/10.1109/TR.2018.2865733
22. Lam, P., Bodden, E., Hendren L., Technische Universität Darmstadt. The Soot Framework for Java Program Analysis: a Retrospective, 2011
23. Lambeau, B., Damas, C., Dupont, P. State-Merging DFA Induction Algorithms with Mandatory Merge Constraints. Grammatical Inference: Algorithms and Applications: 9th International Colloquium, ICGI 2008 Saint-Malo, France, September 22-24, 2008 Proceedings,Springer Berlin Heidelberg, 2008, 139-153. https://doi.org/10.1007/978-3-540-88009-7_11
24. Lang, K. J., Pearlmutter, B. A., Price, R. A. Results of the Abbadingo one DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. Grammatical Inference: 4th International Colloquium, ICGI-98 Ames, Iowa, USA, July 12-14, 1998 Proceedings, Springer Berlin Heidelberg, 1998, 1-12. https://doi.org/10.1007/BFb0054059
25. Li, Y., Yang, Z., Guo, Y., Chen, X. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17), IEEE Press, 2017, 23-26. https://doi.org/10.1109/ICSE-C.2017.8

26. Liu, C. A Compatibility Testing Platform for Android Multimedia Applications. Multimedia Tools and Applications, 2018. https://doi.org/10.1007/s11042-018-6268-y
27. Liu, Z., Gao, X., Long, X. Adaptive Random Testing of Mobile Application. 2nd International Conference on Computer Engineering and Technology, 2010, 2, 297-301. 10.1109/ICCET.2010.5485442.
28. Machiry, A., Tahiliani, R., Naik, M. Dynodroid: An Input Generation System for Android Apps. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), ACM, 2013, 224-234. https://doi.org/10.1145/2491411.2491450
29. Meiliana, Septian, I., Alianto, R. S., Daniel. Comparison Analysis of Android GUI Testing Frameworks by Using an Experimental Study. Procedia Computer Science, 2018, 135, 736 - 748. https://doi.org/10.1016/j.procs.2018.08.211
30. Meinke, K., Sindhu, M. A. Incremental Learning-Based Testing for Reactive Systems. Tests and Proofs, Springer Berlin Heidelberg, 2011, 134-151. https://doi.org/10.1007/978-3-642-21768-5_11
31. Morgado, I. C., Paiva, A. C. R. The iMPAcT Tool: Testing UI Patterns on Mobile Applications. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, 876-881. https://doi.org/10.1109/ASE.2015.96
32. Muangsiri, W., Takada, S. Random GUI Testing of Android Application Using Behavioral Model. The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017, 266-271.https://doi.org/10.18293/SEKE2017-099
33. Novella, L., Tufo, M., Fiengo, G. Improving Test Suites via a Novel Testing with Model Learning Approach. IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2018. https://doi.org/10.1109/WETICE.2018.00051
34. Packevičius, Š., Barisas, D., Ušaniov, A., Guogis, E., Bareiša, E. Text Semantics and Layout Defects Detection in Android Apps Using Dynamic Execution and Screenshot Analysis. Information and Software Technologies, Springer International Publishing, 2018, 279-292. https://doi.org/10.1007/978-3-319-99972-2_22
35. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.
36. Utting, M., Pretschner, A., Legeard, B. A Taxonomy of Model-based Testing Approaches. Software Testing, Verification and Reliability, John Wiley and Sons Ltd, 2012, 22(5), 297-312. https://doi.org/10.1002/stvr.456
37. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J. Increasing Functional Coverage by Inductive Testing: A Case Study. Testing Software and Systems, Springer Berlin Heidelberg, 2010, 126-141. https://doi.org/10.1007/978-3-642-16573-3_10
38. Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., Dupont, P. STAMINA: A Competition to Encourage the Development and Assessment of Software Model Inference Techniques. Empirical Software Engineering, Kluwer Academic Publishers, 2013, 18(4), 791-824. https://doi.org/10.1007/s10664-012-9210-3
39. Walkinshaw, N., Taylor, R., Derrick, J. Inferring Extended Finite State Machine Models from Software Executions. Empirical Software Engineering, 2016, 21(3), 811-853. https://doi.org/10.1007/s10664-015-9367-7
40. Wang, P., Liang, B., You, W., Li, J., Shi, W. Automatic Android GUI Traversal with High Coverage. Fourth International Conference on Communication Systems and Network Technologies. 2014, 1161-1166. https://doi.org/10.1109/CSNT.2014.236
41. Wetzlmaier, T., Ramler, R., Putschögl, W. A Framework for Monkey GUI Testing. IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, 416-423. https://doi.org/10.1109/ICST.2016.51
42. Weyuker, E. J. Assessing Test Data Adequacy Through Program Inference. ACM Transactions on Programming Languages and Systems, 1983, 5(4), 641-655. https://doi.org/10.1145/69575.357231
43. Yang, W., Prasad, M. R., Xie, T. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, 2013, 250-265. https://doi.org/10.1007/978-3-642-37057-1_19
44. Zhauniarovich, Y., Philippov, A., Gadyatskaya, O., Crispo, B., Massacci, F. Towards Black Box Testing of Android Apps. 10th International Conference on Availability, Reliability and Security, 2015, 501-510. https://doi.org/10.1109/ARES.2015.70