Model Inference and Automatic Testing of Mobile Applications *

Sébastien Salva, Patrice Laurencot

Model Inference and Automatic Testing of Mobile Applications*

Sébastien Salva
LIMOS - UMR CNRS 6158
Auvergne University, France
email: sebastien.salva@udamail.fr

Patrice Laurençot
LIMOS - UMR CNRS 6158
Blaise Pascal University, France
email: laurenco@isima.fr

***Abstract*—We consider, in this paper, the problem of automatically testing Mobile applications while inferring formal models expressing their functional behaviours. We propose a framework called *MCrawlT*, which performs automatic testing through application interfaces and collects interface changes to incrementally infer models expressing the navigational paths and states of the applications under test. These models could be later used for comprehension aid or to carry out some tasks automatically, e.g., the test case generation. The main contributions of this paper can be summarised as follows: we introduce a flexible Mobile application model that allows the definition of state abstraction with regard to the application content. This definition also helps define state equivalence classes that segment the state space domain. Our approach supports different exploration strategies by applying the Ant Colony Optimisation technique. This feature offers the advantage to change the exploration strategy by another one as desired. The performances of *MCrawlT* in terms of code coverage, execution time, and bug detection are evaluated on 30 Android applications and compared to other tools found in the literature. The results show that *MCrawlT* achieves significantly better code coverage in a given time budget.**

***Keywords-Model inference; Automatic testing; Android applications*.**

I. Introduction

One of the primary purposes of software testing is to assess the quality of the features offered by an application in terms of conformance, security, performance, etc., to discover and fix its defects. Traditionally, testing is performed by means of test cases written by hands. But manual testing is often tedious and error-prone. Model-based Testing is another well-known approach, which automates the test case generation from a formal model describing the functional behaviours of the application. MbT makes possible the generation of exhaustive test suites (composed of all combinations of input values), but a complete model expressing all the expected behaviours of an application is then required. Unfortunately, writing complete models is often a long, difficult, and tedious task. As a consequence, only partial models are often proposed and available for testing. This makes MbT less interesting and even impractical with many real systems.

This work was co-financed by the Openium company (http://www.openium.fr) and the European Regional Development Fund.

For specific applications, model inference methods based upon automatic testing can strongly help in the design of models. In particular, GUI applications (a.k.a. event-driven applications) belong to this category. Such applications offer a Graphical User Interface (GUI) to interact with and respond to a sequence of events played by the user. Partial models can be inferred by exploring (a.k.a. crawling) interfaces with automatic testing approaches. Furthermore, a large part of the application defects can be detected during the process. Afterwards, these generated models may be manually extended, analysed with verification techniques or employed for generating test cases.

This work falls under the automatic testing category and tackles the testing and the generation of functional models for Mobile applications. It provides additional details over [1] on various aspects, e.g., the use of strategies to explore applications. Several works already dealt with the crawling of GUI application, e.g., Desktop applications [2], Web applications [3], [4], [5] or Mobile ones [6], [7], [8], [9]. These approaches interact with applications in an attempt to either detect bugs or record models or both. These previous works already propose interesting features, such as the test case generation from the inferred models. However, it also emerges that many interesting issues still remain open. Firstly, performing experiments with the GUIs of Web or Mobile applications may lead to a large and potentially unlimited number of states that cannot be all explored. Additionally, the application traversing is usually guided by one of these strategies: DFS (Depth First path Search) or BFS (Breadth First path Search). These are relevant on condition that all the application states would be explored. But when the application state number is large or the execution time is limited, using other strategies could help target the most interesting application features as a first step.

This paper contributes in these issues by proposing a framework called *MCrawlT*. Its goals are to experiment Mobile applications to infer both storyboards and formal models, and to detect bugs. It also aims to achieve good code coverage quickly. The originality of our approach lies in the following features:

- model definition and state abstraction: we use PLTSs (Parametrised Labelled Transition Systems) as models that we specialise to capture the functional behaviours

of Mobile applications. Most of the model inference approaches use state abstractions to produce models. But they often either face the problem of state space explosion or produce too abstract models that do not capture sufficient information to later perform analysis and testing. Both issues often occur because of an inappropriate and unmodifiable state abstraction definition. Here, we propose a flexible PLTS state representation which allows the definition of state abstraction with regard to the application content. This PLTS state definition also helps define state equivalence classes, which slice a potentially infinite state space domain into finite equivalence classes. Our algorithm aims at exploring every discovered state equivalence classes once. As a consequence, our algorithm terminates;

- exploration strategies performed in parallel: we propose a first algorithm which uses exploration strategies to target specific parts of a Mobile application. The algorithm is based upon the Ant Colony Optimisation (ACO) technique and simulates several ants represented by threads, which explore application states and lay down pheromones. These pheromone trails, built in parallel, allow the ants to target the most relevant states w.r.t. a chosen strategy. A strategy can be replaced by another one as desired;
- code coverage enhancement: GUI application testing approaches traditionally start exercising an application from its root interface. Nevertheless, we observed that some application features cannot be automatically tested and hence block the application exploration, which may lead to low code coverage. This is why we propose an extended algorithm which tries to cover an application starting from each of its available interfaces and which infers several PLTSs along the execution. If a blocking feature is bypassed, the application is deeper covered and therefore the code coverage is improved. Furthermore, the algorithm avoids exploring states previously encountered to not build several identical models.

In collaboration with the Openium company, which is specialised in the design of Mobile applications, we have implemented *MCrawlT* for Android applications. The tool is publicly available at https://github.com/statops/MCrawlerT. We applied *MCrawlT* to 30 real-world Android applications and compared its effectiveness against other available automatic testing tool in terms of code coverage, execution time and bug detection.

The paper is structured as follows: Section II surveys related work and introduces our motivations. For expository purposes, we start by presenting an overview of our approach in Section III that we apply on the Ebay Mobile application, taken as example throughout the paper. We also give the assumptions that guided the design of our approach. Section IV gives definitions and notations about the PLTS model. In particular, we specialise the PLTS formalism for Mobile applications and give a state equivalence relation. Our exploration algorithms are detailed in Section V. Section VI presents experimental results. Conclusions are drawn in Section VII along with directions for further research and improvements.

II. Related Work and discussion

Several papers dealing with automatic testing and model inference approaches were issued in the last decade. Here, we present some of them relative to our work.

Several works were proposed for white-box systems. For example, *Contest* [6] is a testing framework which exercises smartphone applications with the generation of input events. This approach relies upon a systematic test generation technique, a.k.a. concolic testing, to explore symbolic execution paths of the application. Artzi et al. [4] proposed an automatic white-box testing approach for finding faults in PHP Web applications. The application code is covered using combined concrete and symbolic (concolic) execution, and constraint solving to detect execution failures and malformed HTML code. These white-box based approaches should theoretically offer a better code coverage than the automatic testing of black-box systems. However, the number of paths being explored concretely limits to short paths only. Furthermore, the constraints have not to be too complex for being solved. As a consequence, the code coverage of these approaches is not high in practice.

On the other hand, many black-box based methods were also proposed. Memon et al. [2] initially presented *GUI Ripper*, a tool for scanning and testing desktop applications. This tool produces event flow graphs and trees showing the GUI execution behaviours. Only the click event can be applied, and *GUI Ripper* produces many false event sequences which may need to be weeded out later. Furthermore, the actions provided in the generated models are quite simple (no parameters). This approach was extended to support Mobile applications in [10] with the tool *Guitar*. This one is based upon *GUI Ripper* but also supports the inference of Event flow graphs and test case generation. Mesbah et al. [3] proposed the tool *Crawljax* specialised in Ajax applications. It produces state machine models which capture the changes of the DOM structures of HTML documents by means of events (click, mouseover, etc.). An interesting feature of *Crawljax* is the concatenation of identical states in the model under construction. If two states, which represent the DOM structures of HTML documents, are similar, they are assembled together. This helps reduce the number of states which may be as large as the DOM modifications. In practice, to limit the state space and to avoid the state explosion problem, state abstractions have to be given manually to extract a model with a manageable size. *Webmate* [5] is another automatic testing tool for Web

applications. It produces graphs showing the observed GUI and events.

Since our approach targets Mobile applications, we explore more cautiously this field in the following. *Monkey* [11] is a random testing tool proposed by Google. It is considered as a reference in many papers dealing with Android application automatic testing. However, it cannot simulate complex workloads such as authentication, hence, it offers light code coverage in such situations. The tool Dynodroid [9] exercises Android applications with UI events like *Monkey* but also with system events to improve code coverage. A similar technique is applied on Android applications in [12]. But, the approach additionally performs static analyses on Android application source codes to later guide the application exploration. No model is provided with these approaches. Amalfitano et al. [7], [13] proposed *AndroidRipper*, a crawler for crash testing and for regression test case generation. A simple model, called GUI tree, depicts the observed screens. Then, paths of the tree not terminated by a crash detection, are used to re-generate regression test cases. Joorabch et al. [14] proposed another crawler, similar to *AndroidRipper*, dedicated to iOS applications. Yang et al. proposed the tool *Orbit* [8] whose novelty lies in the static analysis of Android application source code to infer the events that can be applied on screens. Then, a classical crawling technique is employed to derive a graph labelled by events. This grey-box testing approach should cover an application quicker than our proposal since the events to trigger are listed by the static analysis. But *Orbit* can be applied only when source code is available. This is not the case for many Android applications though. The algorithm implemented in *SwiftHand* [15] is based on the learning algorithm L^* [16] to generate approximate models. The algorithm is composed of a testing engine which executes applications to check if event sequences meet the model under generation until a counterexample is found. An active learning algorithm repeatedly asks the testing engine observation sequences to infer and potentially regenerate the model w.r.t. all the event and observation sequences.

We deduced from these papers the main following key observations:

1) to prevent from a state space explosion, the approaches that infer models, e.g., [2], [8], [17], usually represent application states in a fixed manner and with a high level of abstraction. This choice is particularly suitable for comprehension aid, but these models often lack information for test case generation. In contrast, other approaches try to limit the model size on the fly. The algorithms introduced in [3], [13] concatenate identical states of the model under construction, but the resulting model does not capture the same behaviours as those expressed in the original model. Such an extrapolated model may lead to false positives if used for test case generation. We propose here another solution based upon the PLTS formalism and the definition of state equivalence classes. We specialise the PLTS for Mobile applications to ease the definition of state abstraction. Users can modify the latter to build models as desired. We define state equivalence classes to segment the potentially infinite state space domain of an application in a finite manner. As a consequence, we show that our algorithm terminates. Finally, we use a bisimulation minimisation technique [18] to reduce the PLTS size. This technique offers the advantage to preserve behavioural equivalence between models;
2) many inference model methods consist in analysing and completing interfaces with random test data and triggering events to discover new interfaces that are recursively explored in an in-depth manner. As a consequence, the application exploration is usually guided with a DFS strategy. When an application returns a high number of new interfaces, the graph to be explored may become too large to visit in a reasonable time delay. The search is only performed to a limited depth, and the explored section of the application is not necessarily the most interesting one. We believe that a strategy choice is relevant when the execution time is limited, for instance or when an insight of the application functioning (code structure) is known or both. Indeed, strategies allow to quicker target some application features. Our algorithm is based upon the ACO technique in order to accept a large strategy set. For instance, our algorithm supports semantics-based strategies, i.e., strategies guided by the content found in applications screens. Furthermore, the ACO technique is known as a good heuristic to cover paths through graphs in parallel;
3) crash reporting is another feature supported by some of these methods. Stress testing is performed in [11], [19], [9] for trying to reveal more bugs, for instance by using random sequences of events. Besides, the tool *AndroidRipper* [7] generates test cases for each observed crash. Our approach also performs stress testing: like Monkey [11], random sequences of events are applied on screens. We also use well-known values for revealing bugs for testing. Our tool reports the observed bugs and generates one test case for each as well. These ease the analysis of the detected errors and help deduce whether some errors are false positives. As crash reporting and detection are not original features, we do not detail this part in the paper. We only discuss about crash detection in the section dealing with the evaluation of our framework (Section VI).

We presented in [1] a rudimentary introduction of this work describing an initial algorithm based upon the ACO technique. In this paper, we define another model, state equivalence classes, and we revisit the exploration algorithm

to better match the concept of the ACO technique. Then, we propose a second Exploration algorithm to enhance code coverage and we show that our algorithms terminate. Finally, our evaluation focuses on much more applications and criteria.

III. Overview

In this section, we introduce the terminology used throughout the paper and a motivating example on which we apply our framework *MCrawlT*. The formal details and *MCrawlT* algorithms are introduced in Section V.

A. Terminology and assumptions

Generally speaking, we say that users expect screens to interact with Mobile applications. We consider that a screen represents one application state, the number of states being potentially infinite. A screen is built by a GUI application component, e.g., a class. We call them *Activities* (in reference to Android applications). These components display screens by instantiating *Widgets* (buttons, text fields, etc.) which are often organised into a tree structure. They also declare the available (UI) events that may be triggered by users (click, swipe, etc.). A Widget is characterised by a set of properties (colour, text values, etc.). Hence, one Activity can depicts several screens, composed of different Widgets or composed of the same Widgets but having different properties.

Figure 1 depicts some screen examples of the *Ebay Mobile* application, which is available on the *Google Play* store (https://play.google.com/store). This complex application includes 135 Activities and we only depict five of them in Figure 1. The initial screen is loaded by the Activity *eBay* (i0). A user may choose to search for an item by clicking on the editable text field Widget. In this case, the Activity *RefineSearch* is reached (i1). For instance, if the user enters the keyword "shoes", the search result list is displayed in the screen i2; the Activity *RefineSearch* is unchanged but its content (Widgets) is. Then, three new Activities may be reached: 1) an Activity called *SegmentSearchResult* (i3) displays a result when one element of the proposed list in i2 is chosen, 2) a *Scanner* Activity is started when the text field "Scan" is clicked (i4) and 3) a log-in process is performed when the "saved searches" item is selected (Activity *SignIn*, i5). Now if we replace the value "shoes" by any other String value, one can easily deduce that this application can yield a huge state number.

B. Assumptions

The purpose of our algorithm is to generate input events in order to feed a Mobile application with respect to an exploration strategy to achieve formal models and good code coverage quickly. To design this approach, we had to assert the following hypotheses:

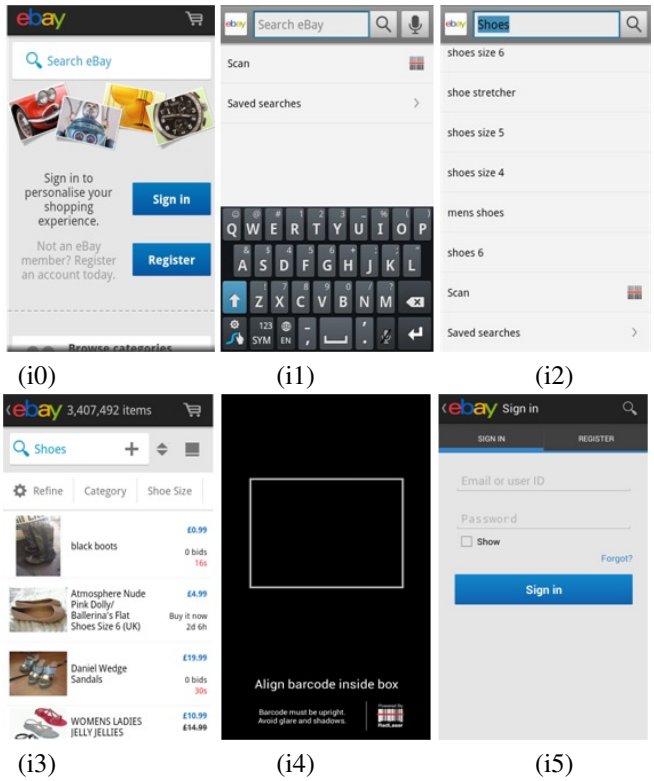

Figure 1: Ebay Mobile Screen examples

Mobile application testing: we consider black box applications which can be exercised through screens. It is possible to dynamically inspect the states of the running application (to collect Widget properties). This assumption holds with many recent GUI applications (Web applications, Mobile applications, etc.). The set of user events enabled on a screen should be collected as well. If not, Widgets provide enough information (type, etc.) to determine the set of events that may be triggered. Otherwise, our algorithm considers them all for testing an application. Furthermore, any new screen can be observed and inspected (including application crashes),

Application reset: we assume that Mobile applications and their environments (database, Operating System) can be reset,

Back mechanism availability: several operating systems or applications (Web navigators, etc.) also propose a specialised mechanism, called the *back mechanism* to let users going back to the previous state of an application by undoing the last event. We do not consider that this mechanism is necessarily available and, if available, we assume that it does not always allow to go back to the previous state of an application (modified implementation, unreachable state, etc.). Most of the other methods assume that the back mechanism always works as expected, but this is frequently not the case.

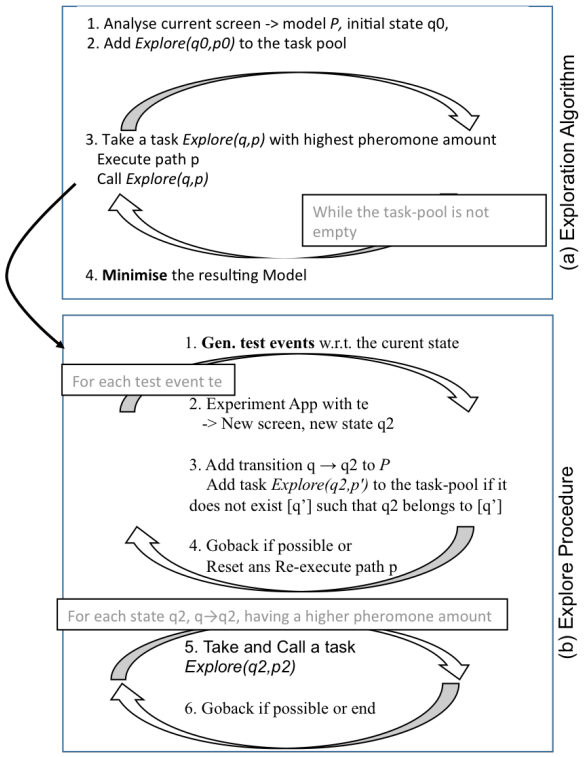

Figure 2: Algorithm overview

C. Exploration algorithm overview

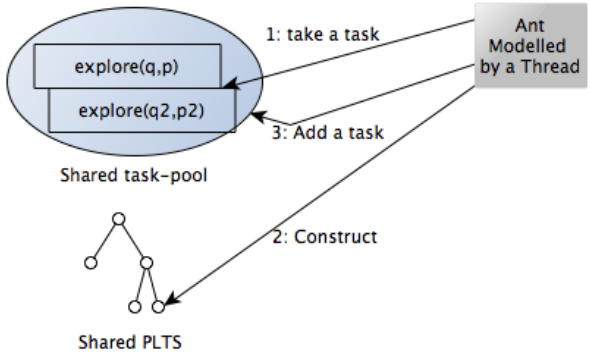

Figure 3: Parallel exploration functioning

An overview of our algorithm is depicted in Figure 2. It is framed upon the Ant Colony Optimisation (ACO) technique to support the definition and the use of exploration strategies, which can be applied with concurrent threads. With the ACO technique, the optimal path search in a graph is performed by simulating the behaviour of ants seeking a path between their colony and a source of food: firstly, ants explore randomly and lay down little by little pheromone trails that are finally followed by other ants.

Ants are here modelled with threads that explore application states having the highest pheromone amount, earlier put down by other ants. This part is implemented, as described in Figures 2(a) and 3 by using the task-pool paradigm associated with tasks of the form $Explore(q, p)$ with q the state to visit and p the path allowing to reach q from the initial state $q0$ of the application. Intuitively, this path corresponds to a trail previously constructed by ants. Initially, the first screen of the application under test is modelled with an initial PLTS state q_0. The exploration of a screen, modelled with a PLTS state q, is conveyed with a task $Explore(q, p)$, which is placed into the task-pool, implemented as an ordered list in descending order. A thread picks out a task having the highest pheromone amount, reaches the state q and starts the exploration. Once the task-pool is empty, the application exploration is over and a PLTS $\mathcal{P}$ is achieved. This one is minimised to reduce the PLTS state set.

The execution of a task $Explore(q, p)$ is achieved by the *Explore* procedure, illustrated in Figure 2(b)). The latter consists in generating a set of test events (parameter values combined with an event set) w.r.t. the current application state. Each test event is applied on the application to reach a new screen. This one is interpreted and modelled as a new state q_2. However, this step may lead to infinite state space domains and endless explorations. To avoid this issue, the algorithm slices state space domains into finite equivalence class sets by means of a relation defined upon the state content (see Section IV). We have chosen to explore one state per equivalence class to keep a reasonable model size but this could be modified. The algorithm completes the model with a transition $q \xrightarrow{q_2} 2$ and finally tries to backtrack the application to go back to its previous state by undoing the previous action. If the back mechanism cannot be triggered, the application is restarted from its initial state $q0$. Once the state q is explored, the thread can explore a next state iff it includes a pheromone amount higher than the one found in q. Likewise, if the back mechanism cannot be applied, the *Explore* procedure execution terminates here. The thread continues its execution in Algorithm 1 by taking another task in the task-pool.

Figure 4 illustrates how our algorithm works on the *Ebay Mobile* application. We have chosen the DFS exploration strategy whose implementation is detailed in Section V-B. With this strategy, the deeper a state is in the model from the initial state, the higher the pheromone amount is. In order to show a comprehensive but yet concise illustration, we use only two text field values ”shoes” and ”All shoes” to fill the editable text fields found in screens. Furthermore, we consider that the back mechanism is available and that the state equivalence relation is: *two states are similar if they have the same Widget properties except those related to the text field values*. These last Widget properties are usually not considered for conceiving state abstractions since these often lead to a large and potentially infinite set of states.

1) initially, the first screen of the application (Fig-

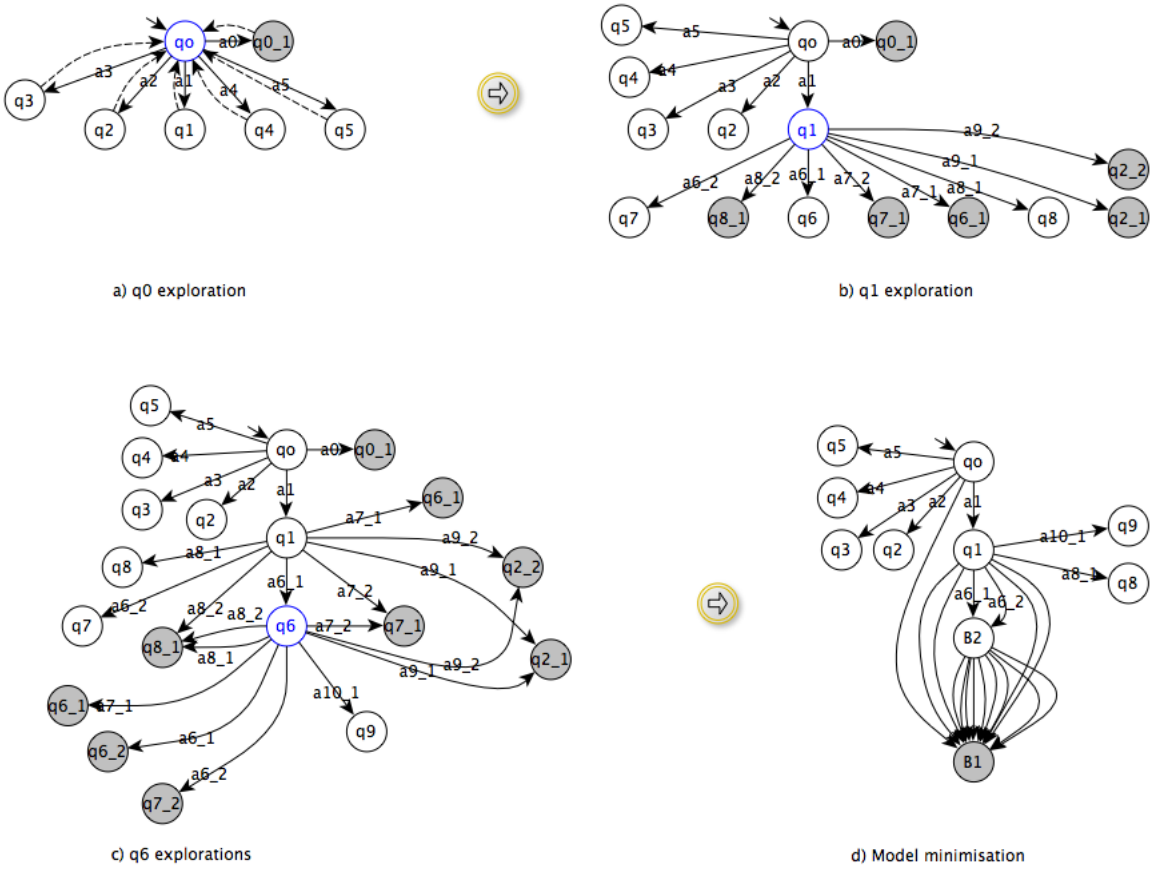

Figure 4: Model inference progress on the Ebay application

ure 1(i0)) is inspected to derive a first task $Explore(q0, p0 = \emptyset)$. $q0$ is derived from the Widget properties extracted from the screen. It also includes a pheromone amount equal to 0. $[q0]$ is the initial state equivalence class. Then, the task $Explore(q0, p0 = \emptyset)$ is chosen by the algorithm. The outcome of this task is depicted in Figure 4(a). A list of test events that can be applied on the current screen is constructed: intuitively, these events aims at clicking on the 2 buttons, the 2 images, or the text field *home_search_text* found in the current screen. Some events are detailed in Figure 5. When the test event $a0$ (click on the Widget id/home) is executed, a new screen is observed. The Widget properties are extracted to construct the state $q0\_1$. This state is marked as final (in grey in the figure) since it has the same Widget properties as $q0$, except for the text field values. In other words, $q0\_1$ belongs to the state equivalence class $[q0]$. The transition $q0 \xrightarrow{a0} q0\_1$ is added to the model. Then, we call the back mechanism to go back to $q0$. This event is illustrated with dashed transitions in Figure 4(a). For the other test events, every time a new screen is found, a new state and a new transition are added to the model. These states include a pheromone amount increased by one unit to meet the DFS strategy. For each state, a new equivalence class is created, and a new task is also put into the task-pool;

2) the algorithm now takes the next task having a state with the highest pheromone amount. In this case, the task $Explore(q1, q0 \xrightarrow{clickhome\_search\_text} q1)$ is picked out. $q1$ represents the "RefineSearchAct" Activity (Figure 1(i1)). This task gives the PLTS of Figure 4(b) but to keep this figure readable, we intentionally do not add the transitions which express the calls of the back mechanism. As before, a list of test events is generated. $q1$ is experimented with these and new states, e.g., $q6, q7, q8$, are observed. For instance, $q6$ is obtained by clicking on the Widget *up* and by filling the editable Widget *search_text* with the value "shoes". $q7$ is reached in the same way but by using the value "All shoes". The two states $q6$ and $q7$ are obtained from the Activity "RefineSearch" but they differ from each other on the Widget *listview* which is a container. In $q6$, *listview* has more items than in $q7$, and consequently, they are not similar and do not belong to the same equivalence class.

The events $a8\_1$ or $a8\_2$ (clik:widget=id/text2) lead to the Scanner Activity (Figure 1 (i4)). The event $a8\_1$ is firstly executed. We observe a new state $q8$, which has to be explored. The event $a8\_2$ leads to the same screen from which a state $q8\_1$ is derived. $q8\_1$ exactly includes the same Widget properties as $q8$ but it belongs to the equivalence class $[q8]$ and is then marked as final. The new states discovered during this step include a pheromone amount increased by one unit;

3) the task $Explore(q6, q0 \xrightarrow{clickhome\_search\_text} q1 \xrightarrow{clickup,search\_text="shoes"} q6)$ is now chosen in the task-pool. This state expresses the Activity of Figure 1(i3). As in the previous steps, test events are generated to experiment the current screen. For sake of readability, we chose to only consider the event a_10 with the container "listview", which stands for "the click on the first item of listview". When the event a_10 is triggered, a new state $q9$ is found and another equivalence class is created. In contrast, for the other events, all the arrival states belong to an existing equivalent class and are marked as final. We obtain the model illustrated in Figure 4(c);
4) even though the algorithm should continue with the task composed of the state $q9$, we assume here that the task-pool is empty to keep the example concise. The model of Figure 4(c) is finally minimised. All the final states are merged to one unique state B1 as illustrated in Figure 4(d). States $q6$ and $q7$ are aggregated into $B2$ since the same behaviours can be observed from both states. The minimisation process is detailed in Section V.

In this example, we have shown that the algorithm discovers trails into the applications by laying down pheromone amounts. The generated models contain the events and screens observed while testing. As described previously, this algorithm works with one thread. But, the task-pool paradigm is particularly suitable to run a group of threads exploring states in parallel.

In the following, we describe the functionalities, the model and equivalence relation definitions, succinctly suggested previously. Algorithm 1, given in Section V, implements with details this overview.

IV. Mobile application modelling

In this section, we introduce a few definitions and notations to be used throughout the paper.

We use PLTS (Parameterised Labelled Transition System) as models that we specialise to represent Mobile application behaviours. The PLTS is a kind of automata model extended with variable sets. The use of variables helps describe valued actions composed of parameter values and to encode the states of a system.

Before giving the model definition, we give some notations on variables. We assume that there exist a domain of values denoted D and a variable set X taking values in D. The assignment of variables in $Y \subseteq X$ to elements of D is denoted with a mapping $\alpha : Y \rightarrow D$. We denote D_Y the assignment set over Y. Given two assignments $\alpha_1 \in D_Y$ and $\alpha_2 \in D_Z$ with $Y \cap Z = \emptyset$, their union is defined as $\alpha_1 \cup \alpha_2(x) = \alpha_1(x)$ iff $x \in Y, \alpha_2(x)$ iff $x \in Z$. An example of assignment is $\alpha = \{x := "blue", y := 1\} \in D_{x,y}$.

Definition 1 (PLTS) *A PLTS (Parameterised Labelled Transition System) is a tuple $< V, I, Q, q0, \Sigma, \rightarrow >$ where:*

- *$V \subseteq X$ is the finite set of variables, $I \subseteq X$ is the finite set of parameters used with actions,*
- *Q is the finite set of states, such that a state $q \in Q$ is an assignment over D_V,*
- *q_0 is the initial state,*
- *Σ is the finite set of valued actions $a(\alpha)$ with $\alpha \subseteq D_I$,*
- *$\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation. A transition $(q, a(\alpha), q')$ is also denoted $q \xrightarrow{a(\alpha)} q'$.*

Below, we adapt this generalised PLTS definition to express Mobile application properties, i.e., screens and events.

UI event representation: We interact with Mobile applications by means of events, e.g., a click, applied on Widgets. Furthermore, editable Widgets are possibly completed before triggering events. We capture these events with PLTS actions of the form $event(\alpha)$ with $\alpha = \{widget := w, w_1 := val_1, ..., w_n := val_n\}$ an assignment over D_I; the parameter $widget$ denotes the Widget name on which the event is applied, and the remaining variables are assignments on Widget properties. We also denote the triggering of the back mechanism with the action $back(\alpha)$ with α an empty assignment.

Mobile application state representation: We concluded from the literature that some Widget properties are considered as more important than others to encode Mobile application states. These properties usually indicate a strong application behaviour modification and take only a few values to prevent from a state space explosion. We denote WP the set of these Widget properties. It often gathers properties related to the Widget visibility, size, position, colour, etc. The properties that usually take a lot of different values, e.g., the properties about text field values, are not chosen to identify Mobile application states. Consequently, in the remainder of the paper, we consider that WP is composed of all Widget properties except those related to text field values.

We specialise PLTS states to store the content of screens (Widget properties) in such a way as to later facilitate the construction of state equivalence classes. We define a PLTS state q as a specific assignment of the form $act \cup wp \cup wo \cup end \cup ph$ where:

- act is an assignment returning an Activity name,

- (wp, wo) are two sets of Widget property assignments. The union of wp and wo gives all the Widget property values found in an application screen displayed by act. We keep in wp the Widget properties of WP that indicate a strong application behaviour modification and that take only a few values. The other property assignments are placed into wo,
- end is a boolean assignment marking a state as final,
- ph is an assignment related to the exploration strategy, which stores a pheromone amount.

For readability, a state $q = act \cup wp \cup wo \cup end \cup ph$ is denoted (act, wp, wo, end, ph).

This state structure greatly eases the definition of the state equivalence relation given below. This one shall be particularly useful to determine if a state belongs to an existing equivalent class and requires to be explored or not.

Definition 2 (State equivalence relation) *Let $\mathcal{P} =< V, I, Q, q0, \Sigma, \rightarrow >$ be a PLTS and for $i = 1, 2$ let $q_i = (act_i, wp_i, wo_i, end_i, ph_i)$, be a state in Q. We say that q_1 is equivalent to q_2, denoted $q_1 \sim q_2$ iff $act_1 = act_2$ and $wp_1 = wp_2$. $[q]$ denotes the equivalence class of equivalent states of q. $Q/\sim$ is the set of equivalence classes in Q.*

This definition, combined with our algorithm, gives a very adaptable state equivalence relation which can be modified according to the WP set. As stated previously, we consider that WP is initially composed of all Widget properties except those related to text field values. But if, for an application, a Widget property takes a large number of values in WP, this one can be removed from WP to obtain a constricted set of equivalence classes and to achieve a finite exploration.

Let us consider an application including advertising strips that are continuously updated. We assume that the Widget property related to the advertising display is denoted $w.content$. This property takes a potentially infinite number of values and may lead to the state space explosion problem while generating the PLTS. Indeed, for an Activity act which holds $w.content$, the exploration algorithm shall reach several states $q_i = (act, wp_i, wo_i, end_i, ph_i)_{(i>1)}$ that are almost similar except that they contain in wp_i different assignments $w.content := val_i$ related to the different advertisings. Each state q_i involves a new equivalence class $[q_i]_{(i>1)}$. The application exploration will likely not terminate. The removal of the property $w.content$ in WP fixes this problem. In this case, the algorithm now constructs different states $q_i = (act, wp, wo_i, end_i, ph_i)_{(i>1)}$ such that the different assignments $w.content := val_i$ are now placed into wo_i. But, the algorithm builds one equivalence class $[q]$ since the assignment wp is unchanged. Therefore, when the algorithm reaches a state $q_2 = (act, wp, wo_2, end_2, ph_2)$, q_2 belongs to $[q]$ and is thus marked as final. Only one state of $[q]$ is explored, hence, the exploration is finite.

Label	Action
$a0$	click(widget:=id/home)
$a1$	click(widget:=id/home_search_text)
$a2$	click(widget:=id/button_sign_in)
$a3$	click(widget:=id/button_register)
$a4$	click(widget:=id/rtmImageView)
$a5$	click(widget:=id/home_settings)
$a6\_1$	click(widget:=id/up, search_src_text:=”All shoes”)
$a6\_2$	click(widget:=id/up, search_src_text:=”shoes”)
$a7\_1$	click(widget:=id/search_button, search_src_text:=”All shoes”)
$a7\_2$	click(widget:=id/search_button, search_src_text:=”shoes”)
$a8\_1$	click(widget:=id/text1, search_src_text:=”All shoes”)
$a8\_2$	click(widget:=id/text1, search_src_text:=”shoes”
$a9\_1$	click(widget=id/text2, search_src_text:=”All shoes”)
$a9\_2$	click(widget:=id/text2, search_src_text=”shoes”)
$a10\_1$	click(widget:=”listview at position 1”, search_src_text:=”shoes”)

Figure 5: Actions and Guards of the PLTS

This example also shows that if WP is only composed of discrete variables taking a finite number of values, then the number of state equivalence classes in a PLTS is bounded. This is captured by the following Proposition, which shall be particularly useful to prove the termination of our algorithms.

Proposition 3 *Let $\mathcal{P} =< V, I, Q, q0, \Sigma, \rightarrow >$ be a PLTS modelling a Mobile application App. Let n be the number of Widget properties of WP. If m is the maximum number of values that any Widget property can take during the testing of App, then $card(Q/\sim) \leq m^n$.*

Sketch of proof: all the screens of App are expressed with states of the form $q = (act, wp, wo, end, ph)$. At most, we have m^n different assignments wp. Two states q_1, q_2 are equivalent iff $act_1 = act_2$ and $wp_1 = wp_2$ (Definition 2). But if there is no equivalent state, we have at most m^n equivalence classes including one state.

The choice of the Widget properties to keep in WP is left to users. Intuitively, the more a Widget property of WP takes values, the larger the generated models is. We observed that ignoring the Widget properties related to text field values is mostly sufficient. But sometimes, the first exploration of an application makes emerge some properties that need to be removed to achieve a finite model in a reasonable time delay.

Figures 4(c), 5, and 6 illustrate a PLTS example derived from the Ebay Mobile application, after covering only 5% of its Activities. We detail some PLTS states in a reduced form in Figure 6: we give the Activity name, the numbers of Widget properties (wp and wo), and the assignments end and ph. The PLTS actions are given in Figure 5. For instance, the state $q0$ represents the initial Activity *eBay* of the application, which includes 2 buttons, 6 images, and 7 text fields . $q1$ is reached from $q0$ by executing the action $a1$, i.e., by clicking on the $home\_search\_text$ Widget.

State	Activity	#wp	#wt	end	ph
$q0$	eBay	2b,2im	6t,1e	false	0
$q1$	RefineSearchAct	2b,4im	3t,1e	false	1
$q2$	SignAct	2b,4im	2t,1e	false	1
$q6$	RefineSearchAct	2b,3im,40l_e	3t,1e	false	2
$q6\_1$	RefineSearchAct	2b,3im,40l_e	3t,1e	true	2
$q7$	RefineSearchAct	2b,3im,105l_e	3t,1e	false	2
$q7\_1$	RefineSearchAct	2b,3im,105l_e	3t,1e	true	2

b: button e: editable text field t: text field im: image
l_e: # elements in the listview Widget

Figure 6: Summary of some states of the PLTS

V. Automatic testing and model inference with ACO

In this section, we formally describe the algorithms implemented in *MCrawlT*. We firstly detail the algorithm considered in the overview. Then, we propose an extended version, which aims at improving code coverage. We also provide (time) complexity results.

A. Model inference Algorithm 1

Our solution is framed upon the PLTS formalism to infer formal models. The combination of the PLTS state definition with the state equivalence relation segments the potentially infinite state space domain into a finite set of equivalence classes and every class is visited once. Our algorithm is also based upon the ACO technique to perform explorations in parallel and to support different application strategies. Algorithm 1 implements the initial part of this solution. The ACO technique is implemented with the task-pool paradigm where the tasks of the pool are executed in parallel on condition that the tasks are independent. This is the case here since several application instances can be experimented into independent test environments (smartphones or emulators). All the threads share the same PLTS $\mathcal{P}$ and the same task-pool implemented as an ordered list in descending order. For sake of readability, we assume that these shared resources are protected against concurrent accesses.

Algorithm 1 takes a Mobile application App as input and launches it to analyse its first screen and to initialise the first state $q0 = (act, wp, wo, end := false, ph_0)$ of the PLTS $\mathcal{P}$. q_0 is obviously not marked as final and includes a pheromone amount related to the chosen strategy. This initial step is carried out by one thread only. Afterwards, the interface exploration begins: each available thread executes the loop of Algorithm 1 (line 7): it picks out a task $Explore(q, p)$, which corresponds to the exploration of the state q, such that q holds the highest pheromone amount. Before exploring q, an instance of the application is launched in a re-initialised test environment and q is reached from q_0 by covering and executing the actions of the PLTS path p. Once there is no more task to perform, a second PLTS $\mathcal{MP}$ is computed with a minimisation technique. This PLTS minimisation aims to yield more compact and readable models for comprehension aid.

Algorithm 1: Mobile application exploration V1

input : Application App
output: PLTS $\mathcal{P}$, $\mathcal{MP}$

```
   // Initialisation performed by one thread only
1  Start the application App;
2  Analyse the current screen → Activity act, the Widget property
     lists wp, wt;
3  Initialise ph_0 (depends on the chosen strategy);
4  Initialise PLTS P with q0 = (act, wp, wo, end := false, ph_0);
5  Q/~ = {[q0]};
6  Add (Explore(q0, p = ∅)) to the task-pool;
   // code executed in parallel, P, task-pool,
      Q/~ are shared
7  while the task pool is not empty do
8      Take a task (Explore(q, p)) such that
         q = (act, wp, wt, end, ph) includes the highest pheromone
         amount ph;
9      Reset and Execute app by covering the sequence of actions
         of p;
10     Call Explore(q,p);
   // code executed by one thread
11 MP:= Minimise(P);
```

One task, pulled from the task-pool, is now executed by calling the *Explore* procedure, which somehow simulates an ant exploring a state and laying down pheromones. Initially, we added a stopping condition limiting the execution time. This condition was used in the experiments presented in Section VI. The *GenEvents* procedure is called and generates test events used to feed the application. It starts by analysing the current screen, extracts the editable Widgets, and produces a set of assignments expressing how completing these editable Widgets with values. Similarly, the events that can be triggered on the Widgets are dynamically detected. We obtain a set $Events$ composed of $event(\alpha)$ with α an assignment of the form $\{widget := w, w_1 := val_1, ..., w_n := val_n\}$. Then, the exploration of the current state q begins (line 6). The editable Widgets are completed, and an event is triggered with respect to the test event $event(\alpha)$. It results in a new screen $Inew$ (line 7), which is analysed to extract the assignments constituting the state q_2. The pheromone amount, which is laid down in q_2, is computed with the *Ph_Deposit* procedure. This one implements the exploration strategies. The algorithm now checks whether this new screen and its corresponding state q_2 have to be explored. Naturally, if $Inew$ reflects the termination of the application (exception, crash), q_2 must not be explored. As stated previously, we have chosen to explore one state for each equivalence class. Hence, if q_2 belongs to a previously discovered equivalence class $[q']$ in $Q/\sim$ then q_2 is marked as final with the assignment $end := true$ and is not explored. Otherwise, q_2 has to be explored and a new task $Explore(q_2, p.t)$ is added to the task-pool (lines 14-16). In both cases, a new transition carrying $event(\alpha)$ and leading to q_2 is added to the PLTS $\mathcal{P}$. To apply the next input event $event(\alpha)$, the application has to go back to its previous

state by undoing the previous interaction. This is done with the *Backtrack* procedure (line 17) whose role is to undo the most recent action. When the direct interface restoration is not possible, the *Backtrack* procedure returns false and *Explore* has to reset the application and to incrementally replay the actions of the path p before experimenting the state q.

Once the exploration of the state q is finished, the *Explore* procedure now simulates an ant which pursues its trail. The application exploration is indeed extended at the state q_2, on condition that q_2 is directly reachable and that q_2 contains an assignment of the ph variable higher than the one found in q (line 19). If there is no such state q_2 or if the back mechanism cannot be applied, then the *Explore* procedure terminates. The current thread goes back to the task-pool, and picks out a task previously built by any other thread (in Algorithm 1).

Remark 4 *MCrawlT supports both deterministic and non-deterministic applications. For sake of readability, we have concealed this feature in the previous algorithms in the line "Reset and Execute App by covering the action sequence of p". Given a task $Explore(q, p)$, when the path p is replayed to reach the state q, MCrawlT continuously checks if the arrival state is the one expected in the path p or a new state q'(indeterministic case). If this is a new state, MCrawlT adds a new transition leading to q' and carries on the execution of p.*

The above algorithms rely upon some procedures that are summarised below:

1) PLTS minimisation: We have chosen a bisimulation minimisation technique [18] to make minimised PLTSs. Given a PLTS $\mathcal{P}$, this technique offers the strong advantage to generate a minimised model $\mathcal{MP}$, which is behavioural equivalent to $\mathcal{P}$. In short, this algorithm constructs the state sets (blocks) that are bisimilar equivalent (every state can fire the same actions and the arrival states have to be bisimilar again). A detailed algorithm can be found in [18]. The time complexity of this minimisation technique is also reasonable (proportional to $\mathcal{O}(mlog(n))$ with m the transition number and n the state number).

Figures 4(d) and 7 depict the minimised PLTS obtained with the *Ebay Mobile* application. Some locations are now grouped into blocks. All the final states are bisimilar and grouped into the block $B1$. Furthermore, the states $q6$ and $q7$ are grouped into the Block $B2$ because the same action sequences leading to bisimilar states can be executed from both $q6$ and $q7$.

2) Test event generation : The *Explore* procedure calls *GenEvents*, which constructs test events expressing how to interact with screens. Since this part is already presented in [1], we only briefly introduce it here.

Our algorithm generates a set of test events of the form $\{event(\alpha) \mid event$ is an event, α is an assignment$\}$.

Procedure Explore

```
1  Procedure Explore(q, p);
2  if [processing time > T] then
3      stop;
4  Events = GenEvents, analyse the current screen to generate
     the set of test events
5  {event(α) | event is a UI event, α is an assignment};
6  foreach event(α) ∈ Events do
7      Experiment event(α) on App → new screen Inew;
8      Analyse Inew → assignments act2, wp2, wo2;
9      ph2= Ph_Deposit(q, act2, wp2, wo2);
10     q2 = (act2, wp2, wo2, end := null, ph2);
11     if Inew reflects a crash or there exists [q'] ∈ Q/∼ such that
         q2 ∈ [q'] then
12         Add a transition q --event(α)--> q2 =
             (act2, wp2, wo2, end := true, ph := 0) to →P;
13     else
14         Add a transition t = q --event(α)--> q2 =
             (act2, wp2, wo2, end := false, ph2) to →P;
15         Q/∼ = Q/∼ ∪{[q2]};
16         Add the task (Explore(q2, p.t)) to the task-pool;
17     if Backtrack(q, p)==false then
18         Reset and Execute App by covering the sequence of
             actions p;
19 foreach q --event(α)--> q2 such that ph2 > ph and Explore(q2, p2)
     in the task-pool) do
20     Experiment event(α) on App;
21     Take and Execute Explore(q2, p2);
22     if Backtrack(q, q2)==false then
23         End;
```

Block	States
$B2$	$q6, q7$
$B1$	$q0\_1, q2\_1, q2\_2, q6\_1, q6\_2, q7\_1, q7\_2, q8\_1$

Figure 7: Blocks of states of the minimised PLTS

It starts collecting the events that may be applied on the different Widgets of the current screen. Then, it constructs assignments of the form $\{w_1.value := v1, ..., w_n.value := vn\}$, with $(w_1, ..., w_n)$ the list of editable Widgets found on the screen and $(v_1, ..., v_n)$ a list of test values.

Instead of using random values, we propose to use several data sets, which can be completed before starting the exploration algorithm (the algorithm does not ask for values) The first one, denoted $User$, is completed with values provided by users. If required, this set should hold the logins and passwords needed to access to the application features relative to user accounts. This implies that the user knows some features of the application. To reduce the test event set, if a user value is devoted to some specific Widgets, this value can be accompanied with Widget names.

The set RV is composed of values well known for detecting bugs, e.g., String values like "&", "", or null, completed with random values. A last set, denoted $Fakedata$, is composed of fake user identities. An identity gathers a list of

Procedure Backtrack

```
1  Procedure Backtrack(q = (act, wp, wo, end, ph), q_2);
2  if the back mechanism is available then
3      Call the back mechanism → screen INew;
4      Analyse INew → assignments act', wp', wo';
5      if act ≠ act' or wp ≠ wp' or wo ≠ wo' then
6          return false;
7      else
8          Add a transition t = q_2 --back(α)--> q to →_P;
9          return true;
10 else
11     return false;
```

parameters $(p_1, ..., p_m)$, such as (name, age, email, address, gender), which are correlated together to form realistic identities. Both $User$ and RV sets are segmented per type (String, Integer, etc.). During the analysis of the current screen, we collect the types and names of the Widget properties. Then, we search for the largest subset of properties that form an identity with respect to the parameters of $Fakedata$ (e.g., name, age, email, address, gender). We obtain a list of Widget properties $(w_1, ..., w_n)$ that we bind with a set of value lists extracted from $Fakedata$. For instance, if two Widgets called *name* and *email* are found, the fake identities of $Fakedata$ are parsed to remove the undesired parameters and to return a set of identities composed only of a name and an email. Each remaining Widget property is associated to the set $User \cup RV$. For instance, if an editable Widget takes String values, we bind this Widget with the set $String(User \cup RV)$. Now, given a list of Widget properties $(w_1, ..., w_n)$, we have a corresponding list of value sets $(V_1, ..., V_n)$. It remains to generate a set of assignments of the form $\alpha = \{w_1.value := v1, ..., w_n.value := vn\}$. Instead of computing the Cartesian product of $(V_1, ..., V_n)$, we adopted a Pairwise technique [20] to build these assignments. Assuming that errors can be revealed by modifying pairs of variables, this technique strongly reduces the coverage of variable domains by constructing discrete combinations for pairs of parameters only.

Finally, every UI event $event$ is associated to an assignment $\alpha = \{w_1.value := v1, ..., w_n.value := vn\}$, which is added to the set $Events$ and returned to the *Explore* procedure.

3) Call of the back mechanism: Based upon preliminary studies, we observed that the back mechanism does not always allow to go back to the previous state of an application. Actually, this mechanism is sometimes considered as an event allowing to reach a new application state. As a consequence, we always check whether the state, reached after calling this mechanism, is the expected one. The pseudo-code is given in the *Backtrack* procedure.

This procedure calls the back mechanism to undo the most recent action if available and to go back to the state q (line 3). A new screen is observed, and *Backtrack* checks whether this screen is equivalent to the expected one, modelled with q (we compare their Widget properties). If we observe a state different from q or if the back mechanism is not available, *Backtrack* returns "false" (line 6). On the contrary, a transition $q_2 \xrightarrow{back(\alpha)} q$ is added to $\mathcal{P}$ and the procedure returns "true" (line 9).

B. Exploration strategies

Different strategies can be applied to cover an application. These are mainly implemented by means of the $Ph\_Deposit$ procedure which is called to return pheromone amounts but also with the task-pool paradigm. Independently of the chosen strategy, the threads, which are executed to explore an application, always pick out the first task of the task-pool composed of a state having the highest pheromone amount. The task-pool is implemented as an ordered list in descending order.

We succinctly present how to implement some of strategy examples below:

- **BFS strategy:** the classical breadth-first search strategy is the easiest one to put in practice. Indeed, our algorithm is tacitly based upon it. Whenever a new state q_2 is built, it is only needed to set its pheromone amount to 0. In our algorithm, a state is tested in a breadth-wise order and each new task $Explore(q_2, p_2)$ composed of a new state q_2 to visit, is added to the task-pool. The threads shall only take the tasks in the task-pool in the same order as they have been submitted. As a consequence, the PLTS $\mathcal{P}$ is conceived in breadth-first order;
- **DFS strategy:** the depth-first search strategy can be implemented as follows: the initial state q_0 is initialised with a pheromone amount equal to 0. Afterwards, whenever a new state q_2 is detected from another one q, it is completed with the pheromone amount found in q increased by 1. In this case, the next task $Explore(q, p)$ chosen by a thread shall be composed by the last detected state. Tacitly, a DFS strategy is followed;
- **Crash-driven strategy:** the number of detected bugs could also be considered in a strategy: when the number of bugs detected from the states of a path p is higher than the one detected from the states of another path p', it may be more interesting to continue to cover the former for trying to detect the highest number of application defects. We call this strategy crash-driven exploration. This can be conducted by initialising the pheromone amount to 0 in q_0. Next, given a task $Explore(q, p)$, whenever a new state q_2 is detected, it is completed with a pheromone amount equal to the number of bugs detected from all the states of the path p;
- **Semantics-driven strategy:** this kind of strategy denotes an exploration guided by the recognition of the

meaning of some Widget properties (text field values, etc.). Here, the pheromone deposit mainly depends on the number of recognised Widget properties and on their relevance. It is manifest that the semantic-driven strategy domain can be tremendously vast. For e-commerce applications, the login step and the term "buy" are usually important. A strategy example could be then conducted as follows: an authentication process is detected when a text field Widget has the type "passwdtype". In this case, the pheromone amount considered is set to 10, otherwise it is equal to 1. When a Widget name is composed of the term "buy", the pheromone amount added in a new state could be equal to 5, etc.

Many other strategies could be defined to meet the user requirements. Some of them could be defined to target specific application states or features. Others could be conceived in accordance with the intended usage of the inferred models. For instance, if models are later used for generating security test cases, the exploration strategy should be defined to cover the most sensitive features of the application. Other criteria could also be considered, e.g., the number of Widgets found in screens. Furthermore, the previous strategies could also be mixed together.

The PLTS of Figure 4(c) is built with a DFS strategy. Our algorithm starts by visiting the state $q0$ which holds a pheromone amount equal to 0. The actions $a0$ to $a5$ lead to new screens and states $q0\_1, q1, ..., q5$, which have a pheromone amount equal to 1 and have to be explored. Here, the state $q1$ is chosen since it is the first not final state encountered during the exploration of $q0$ and has the highest pheromone amount. From $q1$, the execution of actions leads to new states, e.g., $q6$, $q7$, or $q8$. These states have a pheromone amount equal to 2. The next state having the highest pheromone amount is $q6$. Therefore, this one is explored, and so on.

C. Code coverage enhancement, Exploration Algorithm 2

After the evaluation of the previous algorithm, we observed that the code coverages obtained with some applications was lower than expected. After investigation, we discovered that *MCrawlT* was actually unable to launch some specific features of these applications. As a consequence, several screens were not displayed and explored, which explains low code coverages. For instance, a text editor can delete a document if and only if a document is available. At the moment, no testing tool is able to automatically deduce such a scenario since it belongs to the logic of the application.

To solve an aspect of this problem and to enhance code coverage, we propose a straightforward and general solution that tries to bypass the blocking features in order to deeper explore an application. Intuitively, it is technically possible with Mobile applications to directly instantiate any Activity instead of the initial ones. Doing this sometimes allows to bypass a blocking Activity that cannot be automatically tested. The pseudo-code of this solution is given in Algorithm 2. The latter tries to directly launch every Activity of an application instead of only considering the initial one to infer models. We do not provide the details of the *Explore* procedure since it only requires slightly modifications. For an application App, this algorithm tries to instantiate each Activity and builds a PTLS for each. We obtain the PLTS set $\{\mathcal{P}_1, ..., \mathcal{P}_n\}$ and the respective set of minimised PLTS $\{\mathcal{MP}_1, ..., \mathcal{MP}_n\}$. The algorithm starts by analysing App and extracts its Activity list; act_1 represents the initial Activity of App. As in the previous algorithm version, a first PLTS $\mathcal{P}_1$ is generated from the initial Activity act_1 (line 1). Then, each Activity act_i is launched, and a corresponding PLTS $\mathcal{P}_i$ is built (line 3). But the algorithm is designed to inspect new states only, i.e., to prevent from exploring several times a state early encountered during the generation of another PLTS. To do so, the state equivalence classes are now kept in the set EQ, which is used all along the execution. Given a model $\mathcal{P}_i (i > 1)$ under construction, the exploration of a state q is done in the *Explore* procedure if and only if q has not been previously encountered in one of the previous PLTSs. In other words, q is marked as final if q belongs to an equivalence class of EQ.

With this algorithm, we switch the initial Activity to start an application from different entry-points and to potentially scan deeper an application. This process does not always deliver the intended outcomes though, since an Activity, which was not designed to be launched at the beginning of the application, may crash. We show in Section VI that

Algorithm 2: Mobile application exploration v2

input : Application App
output: PLTS $\{\mathcal{P}_1, ..., \mathcal{P}_n\}$, $\{\mathcal{MP}_1, ..., \mathcal{MP}_n\}$

```
// initialisation performed by one thread only
1  Analyse App → list of activities LAct = {act_1, ..., act_n}, list of
     PLTSs {P_1, ..., P_n} with act_1 this root Activity of App;
2  EQ = ∅;
3  foreach Activity act_i in LAct such that act_i has not been
     encountered do
4      Start the application App and Launch act_i;
5      Analyse the current screen → Widget property lists wp, wo;
6      Initialise ph_0;
7      Initialise PLTS P_i with
         q0_{P_i} = (act_i, wp, wo, end := false, ph_0);
8      EQ = EQ ∪ {[q0_{P_i}]};
9      Add (Explore(q0_{P_i}, p = ∅), P_i) to the task-pool;
10     while the task-pool is not empty do
11         Take a task (Explore(q, p, P_i)) such that
             q = (act, wp, wt, end, ph) includes the highest
             pheromone amount ph;
12         Reset and Execute App by executing the sequence of
             actions of p;
13         Explore(q, p, P_i);
       // code executed by one thread
14     MP_i := Minimise(P_i);
```

this algorithm achieves better code coverage on 1/3 of the experimented applications.

D. Algorithm complexities and termination

Both Algorithms 1 and 2 run in linear time. Theoretically, for an application *App*, the number of states to visit may be infinite. But our algorithm covers one state per equivalence class. The number of equivalence classes is finite (Proposition 3), hence, the algorithm is finite. If we denote the number of states and transitions by N and M, Algorithm 1 has a complexity proportional to $\mathcal{O}(M+N+MN+Mlog(N))$. Indeed, the *Explore* procedure covers every transition twice (one time to execute an event and one time to go back to the previous state), and every state is processed once. Hence, the complexity should be proportional to $O(M+N)$. But, sometimes the back mechanism is not available. In this situation, the application is reset and the event sequence of a path p is executed from the initial state $q0$. This path is at worst composed of M transitions and, in the worst case, this step is done for every state with a complexity proportional to NM. Furthermore, the minimisation technique has a complexity proportional to $\mathcal{O}(Mlog(N))$ [18].

More precisely, the number of states N is at most equal to $2m^n$ with m is the maximum number of values that any Widget property can take during the testing of *App*, and n the number of Widget properties in WP. Indeed, the number of equivalence classes is bounded to m^n (Proposition 3). Additionally, a state has either an assignment $end := true$ or $end := false$ (2 further possibilities). Regarding the number of transitions M, it is finite and depends on the number of test events executed on the application. If each state is at most tested with e events and has k editable Widgets that are iteratively fulfilled nb times with test values, then M is equal to $N * e * nb^2$ (nb^2 is the maximum number of test value tuples returned by the Pairwise technique [20]).

Algorithm 2 is designed as Algorithm 1 except that it infers one model for each Activity of an application *App*. The Activity number, denoted K is always finite, therefore Algorithm 2 terminates as well. It complexity is proportional to $\mathcal{O}K(M+N+MN+Mlog(N))$.

VI. Implementation and Evaluation

A. Implementation for Android Applications

With the collaboration of the Openium company, we have implemented our solution in a prototype tool, called *MCrawlT* (Mobile Crawler Tool). *MCrawlT* is publicly available in a GitHub repository (https://github.com/statops/MCrawlerT), and is accompanied with a detailed user guide. The tool is specialised for Android applications: in short, these applications are typically Mobile GUI applications built over a set of reusable components. For instance, Activities are components that display screens whereas Service components are used to call remote servers.

As detailed in the above sections, *MCrawlT* infers models from Android applications, it reports code coverage percentages and execution times. Besides, it tries to detect bugs with stress testing (use of values known for revealing bugs such as unexpected values (wrong types), execution of large random event flows on screens). It reports the detected bugs, and generates test cases to replay them. Finally, it displays lightweight or complete storyboards (graphs of screen shots) to simplify the understanding of the application behaviours. Figures 8 and 9 depict two storyboard examples derived from the Ebay Mobile application with different execution times.

Figure 8: Ebay Mobile storyboard

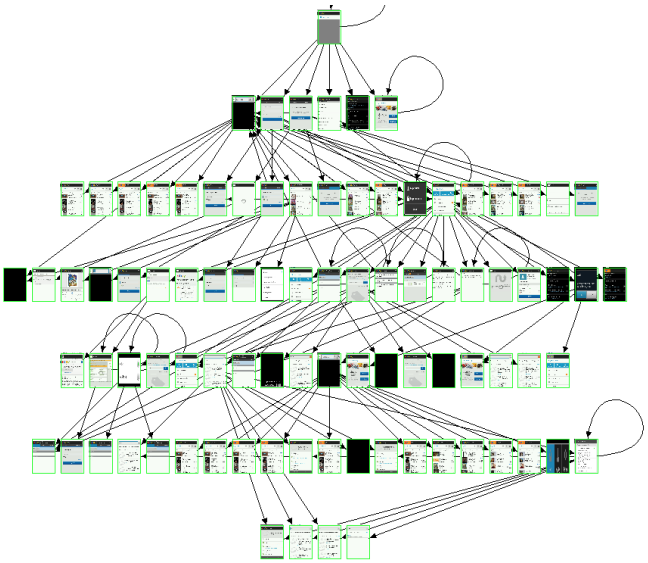

Figure 9: Ebay Mobile storyboard 2

MCrawlT expects packaged Android applications or source projects, a testing strategy and a delay for testing

```
1 public class DFSStrategy extends AntStrategy {

3 public int getRank(State st, ScenarioData path) {
   if (st != null) {
5    State last = path.get(path.size-1));
     return last.getPh()+1;
7    }
   else return defaultRank;
9 } }
```

Figure 10: Implementation of the DFS strategy

(this delay can be set to some seconds up to several days). A user may add some specific test data (login, password, etc.) and prepare local and/or remote databases.

MCrawlT is composed by two main modules. *MCrawlT-Desktop* corresponds to Algorithms 1, 2 and essentially aims to construct PLTS and to manage the task-pool. The second module, *MCrawlTMobile*, corresponds to the *Explore* procedure. It is deployed on the smartphone side to exercise application screens and to generate PLTS transitions. The second module is implemented using the testing framework *Robotium* [21]. Robotium is used to extract the Widget properties found in screens. It also provides functionalities for editing Widgets and simulating user events (click, scroll). Additionally, we have extended the instrumentation package of Android (InstrumentationTestRunner class) to detect and observe application crashes, and periodically compute the code coverage percentage by means of the tool *Emma* [22]. The communication between the modules is ensured by the *Android Debug Bridge (adb)* tool, which is available in the Android tool kit. *MCrawlT* can exercise applications in parallel by launching several *MCrawlerTMobile* modules on emulators or smartphones (Android versions from 2.3 to 4.2.2).

MCrawlT supports the strategies presented in Section V-B and can be upgraded with additional ones. A strategy is implemented in a Java class inherited from the class "AntStrategy", which must have a method *public int getRank(State q, ScenarioData path)*. This one returns the pheromone amount which is put down in a state q. Figure 10 illustrates the Java code used to implement the DFS strategy.

B. Limitations

The *MCrawlT* implementation has three main limitations:

- remote servers cannot be reset, so the tool violates an assumption of the algorithm related to the application environment reset. This limitation can be eliminated by mocking remote servers. This can be done with the SOAPUI framework [23];
- *MCrawlT* supports the following UI events: click and scroll. This is a limitation imposed by *Robotium*. But this tool is updated continuously, therefore, more events should be available in the future;
- in the paper, we focus on UI events but Android proposes a set of system events (sms calls, battery notifications, etc.). We do not consider them yet as inputs.

C. Empirical Evaluation

In this section, we evaluate *MCrawlT* on several real-world Android applications. We chose to compare the effectiveness of several recent tools in terms of execution time, code coverage and crash detection. We tried executing the following tools *Monkey* [11], *Guitar* [19], *AndroidRipper* [7], *SwiftHand* [15] and *Dynodroid* [9]. The others are not available. Unfortunately, we faced many difficulties to use some of them. In summary, we do not know how *Guitar* works with Mobile applications due to lack of documentation; we were unable to launch *AndroidRipper*; *SwiftHand* works well with the proposed examples but, for new applications, source codes need to be instrumented (with a lot of classes) and we do not know how to do this.

In this context, and to avoid any bias, we chose to apply our tool, *Monkey* and *Dynodroid* on all the applications whose source code is available and taken for experimentation in the papers [7], [15], [9]. We have also taken the applications and experimental results found in [8] although the corresponding tool *Orbit* is not available. This corresponds to 30 applications. It is important to note that *Monkey* is taken as a reference in most of the papers dealing with Android testing. Thereby, our results can be compared with other studies related to Android testing.

1) Code coverage and execution time: We compare here the effectiveness, relating to code coverage and execution time, of *MCrawlT* with the other recent Android testing tools. Most of them explore these applications in an in-depth manner. So, *MCrawlT* was executed only with this strategy to carry out a fair comparison.

Figure 11 reports the percentages of code coverage obtained with the different tools on 30 applications with a time budget of three hours. If we do a side by side comparison of *MCrawlT* with the other tools, we observe that *MCrawlT* provides better code coverage than *Monkey* for 23 applications, than *SwiftHand* for 29 applications, than *Orbit* for 29 applications, and than *Dynodroid* for 24 applications. In comparison to all the tools, *MCrawlT* provides better code coverage with 20 applications, the coverage difference being higher than 5% with 14 applications and higher than 10% with 10. This comparison is more explicitly given in the radar chart of Figure 12, which depicts the code coverage percentages obtained with *Monkey*, *MCrawlT* and *Dynodroid*. When *Monkey* is confronted with all the other tools, it offers better results for 5 applications and *Dynodroid* for 3 applications.

Consequently, these results show that *MCrawlT* gives better code coverage than each tool taken one by one and overall offers good results against all the tools on half the applications. Figure 12 clearly illustrates this claim.

Application	Mon key	Orbit	Gui tar	And. Rip- per	MC rawlT	Swift Hand	Dyno droid
NotePad	60	82			88		
TippyTipperV1	41	78			79		48
ToDoManager	71	75	71		81		34
OpenManager	29	63			65		
HelloAUT	71	86	51		96		76
TomDroid	46	70		40	76		42
ContactManager	53	91	71		68		28
Aardict	52	65		27	67		51
Musicnote	69				81	72.2	47
Explorer	58				74	74	
Myexpense	25				61	41.8	40
Anynemo	61				54	52.9	
Whohas	58				95	59.3	65
Mininote	42				26	34	39
Weight	51				34	62	56
TippyTipperV2	49				74	68	12
Sanity	8				26	19.6	1
Nectdroid	70.7				54		68.6
Alogcat	66.6				66		67.2
ACal	14				46		23
Anycut	67				71		69.7
Mirrored	63				76		60
Jamendo	64				46		3.9
Netcounter	47				56		70
Multisms	65				73		77
Alarm	77				72		55
Bomber	79				75		70
Adsdroid	72				83		80
Aagtl	18				25		17
PasswordFor Android	58				61		58

Figure 11: Code coverage comparison (in %)

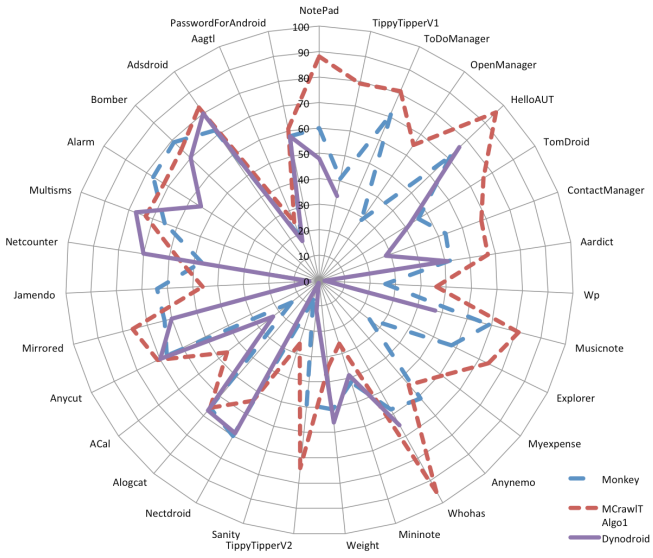

Figure 12: Code coverage comparison (in %)

Figure 11 shows that the code coverage percentage obtained with *MCrawlT* is between 25% and 96%. We manually analysed the 10 applications that provide the lower code coverage percentages with *MCrawlT* to identify the causes behind low coverage.

These can be explained as follows:

- specific functionalities and unreachable code: several applications are incompletely covered either on account of unused code parts (libraries, packages, etc.) that are not called by the application, or on account of functionalities difficult to start automatically. For instance, at least one stored audio file is required for *OpenManager* before testing the functionalities related to the audio file management,
- unsupported events: Several applications, e.g., Nectdroid, Multism, Acal or Alogcat chosen for experimentation with *Dynodroid* take UI events as inputs but also system events such as broadcast messages from other applications or from the Android system. Our tool does not support these events yet. Moreover, MCrawlT only supports the event list also supported by the testing tool *Robotium* (click, scroll). The long click event does not belong to this list but it is used in some applications (Mininote and Contactmanager). In contrast, *Orbit* supports this event and therefore offers a better code coverage with the application Contactmanager.

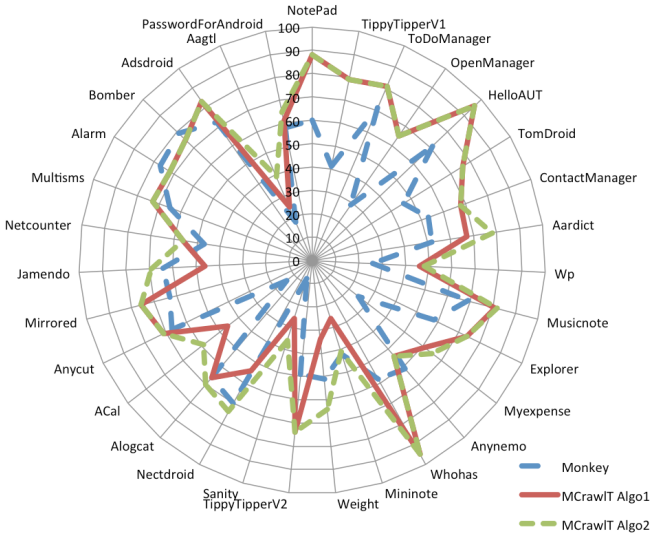

Figure 13: Code coverage comparison (in %)

We also experimented the 30 Android applications with the second version of our algorithm, which tries to infer models for every Activity (Algorithm 2). We kept the same time budget of three hours. The radar chart of Figure 13 gives the code coverage percentages obtained with *Monkey* (for comparison purposes), *MCrawlT* Algorithm 1 and Algorithm 2. It illustrates that our extended algorithm visibly offers better code coverage. More precisely, 13 applications are more covered and 8 have a code coverage increased by 10 %. With some applications the code coverage difference become significant. For instance, we observe a code coverage increased by 23 % with Jamento and by 30 % with Weight. In comparison to all the tools, *MCrawlT* provides now better code coverage with 25 applications.

Application	Orbit	Guitar	Android Ripper	MCrawlT	Swift Hand
NotePad	102			268	
TippyTipperV1	198			251	
ToDoManager	121	194		551	
OpenManager	480			696	
HelloAUT	156	117		106	
TomDroid	340		529	235	
Contact Manager	125	194		233	
Aardict	124		694	580	
Musicnote				10696	10800
Explorer				10800	10800
Myexpense				10800	10800
Anynemo				10800	10800
Whohas				9260	10800
Mininote				8230	10800
TippyTipperV2				1556	10800
Weight				10800	10800
Sanity				10800	10800
Nectdroid				8120	
AlogCat				10800	
ACal				10800	
Anycut				8037	
Mirrored				6020	
Jamendo				10800	
Netcounter				10800	
Multisms				10800	
Alarm				10040	
Bomber				4800	
Adsdroid				10800	
Aagtl				920	
Password ForAndroid				10800	

Figure 14: Execution time (in seconds)

Regarding execution time, the evaluated tools work differently. *Monkey* and *Dynodroid* take a number of events, and perform fuzzy testing independently of the application coverage. We set an event number sufficiently high so as to let the tools perform testing during three hours or more (more than 10,000 events for *Monkey* and more than 600 events for *Dynodroid*). The other tools explore applications with a delay of three hours or until all the application states are explored. Figure 14 reports the execution times obtained with the second category of tools (in seconds) on the same application list. These results reflect the fact that *MCrawlT* is comparable to the others despite using strategies, state equivalence classes and a state minimisation technique. For small applications (first eight lines), we obtain roughly similar time executions as *Orbit*, *Guitar* and *AndroidRipper*. *SwiftHand* always need more than three hours to explore applications, whereas 18 applications are completely explored with *MCrawlT* in less time.

2) Crash detection: MCrawlT, *Monkey*, *Dynodroid* and *AndroidRipper* also detect application crashes. Figure 15 reports the 9 applications that crashed while testing with one of third first tools. We also added the empirical results obtained with *AndroidRipper* found in [7]. Figure 15 exposes genuine bugs only. We manually ascertained test reports to eventually remove false positives such as emulator misbehaving. We kept only the Exceptions that cause the termination of the applications such as *NullPointerException*. On the 30

Application	MCrawlT	Monkey	Dynodroid	Android Ripper
WordPress	63	3		37
Notepad	5			
TomDroid	7	1		14
Mirror	25	3		
Mininote	2			
Aagtl	1		2	
PasswordForAndroid	1		1	
Sanity	5	1	1	
Aardict	2			

Figure 15: Application crash detection

applications, *MCrawlT* revealed that 9 of them have bugs and detected all the applications also found with *Monkey* and *Dynodroid*. We deduce from these results that our tool outperforms the others in automatic crash detection. *MCrawlT* does stress testing like *Monkey* and *Dynodroid* but it also uses values known for detecting bugs. This probably explains the better performance.

3) Impact of the strategy choice and parallelism gain: To illustrate the benefits of using different strategies, we applied on the Ebay Mobile application, the DFS strategy exposed in the previous section and a semantics-driven strategy. This strategy aims to target the account management part of the application and was applied by deposing a higher pheromone amount in states including Widgets of type "passwdtype" or Widget properties composed of the terms "account" or "sign in".

For readability and comparison purposes, we illustrate in Figure 16 a simplified graph showing the visited Activities with the DFS strategy. The application is explored independently of the meaning of the Widgets and the Activities. Here, *MCrawlT* has mostly covered the "Refine search" feature of the Ebay Mobile application. Figure 17 illustrates the simplified graph achieved after applying the second one. Here, the Activity $SignIn$, allowing to log in to user accounts, was firstly visited instead of the Activity *RefineSearch*. Then, the second strategy has guided the exploration on the Activity *SavedSellerList*, which allows to manage the favourite seller list and on *SellItem*, which shows the sold items. With the same time budget, the account management part of the application has been explored with the second strategy instead of the "Refine search" feature. As a consequence, since security vulnerabilities on the user account management affect users and may lead to serious consequences, this strategy makes the generated PLTS more interesting to later analyse the security of the application.

The strategy choice also impacts the time required to explore an application. Figure 18 shows the execution times (in seconds) obtained with two strategies for completely exploring 10 applications. *MCrawlT* were applied with a DFS strategy (1 thread using 1 Android emulator), a BFS (with 1 and 3 threads in parallel). These results show that 7 out of 10 applications are more rapidly covered with BFS traversing. For instance, with *toDoManager*, using the BFS

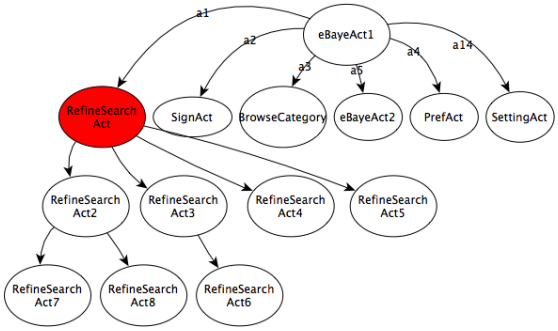

Figure 16: Ebay Mobile simplified graph obtained with a DFS strategy

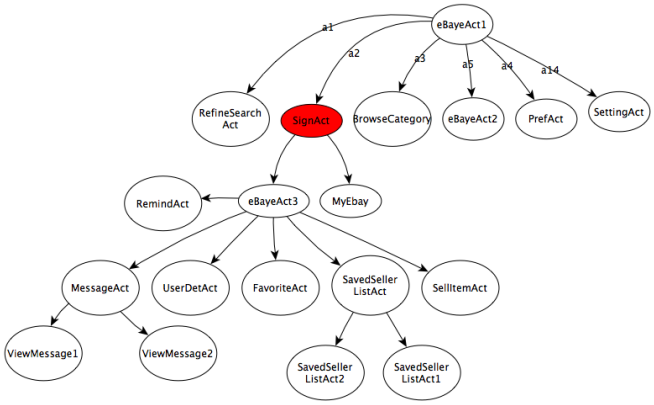

Figure 17: Ebay Mobile simplified graph obtained with a semantics-driven strategy

Application	DFS(1)	BFS(1)	BFS(3)
NotePad	268	310	175
TippyTipperv1	251	210	110
ToDoManager	551	410	210
OpenManager	696	560	489
HelloAUT	106	216	201
TomDroid	235	256	196
ContactManager	233	216	135
Bomber	6120	4800	3100
Mirrored	6690	6020	4090
Nectdroid	10650	8120	5020

Figure 18: Execution time with different strategies (in seconds)

strategy instead of the DFS one, reduces the exploration time by 140 seconds because all of its Activities are directly accessible from the initial one. Actually, when a user has knowledge of the code structure or of how the Activities are composed together, he can choose the most appropriate strategy to speed up the exploration.

Figure 18 also shows that the parallelization of our algorithm is effective. With three emulators, the execution time is always reduced. For instance, the parallel exploration of TippyTipperV1 is achieved with a time almost divided by two.

All these experimental results on real applications tend to show that our tool is effective and leads to substantial improvements in the automatic testing of Mobile applications. Indeed, on the 30 applications taken for evaluation in [7], [15], [9], *MCrawlT* gives better code coverage for 25 applications with the same time budget and detects more bugs. Furthermore, different exploration strategies can be applied to directly target the most relevant application features.

VII. Conclusion

Automatic testing of GUI applications is an interesting solution that complements other testing techniques, e.g., Model-based testing. This approach may be used to generate partial models, which can be later completed or reused for test case generation. This paper brings some original contributions by proposing: 1) a formal model definition that helps limit the application exploration by segmenting state space domains into finite sets of equivalence classes, 2) the use of exploration strategies to cover applications by applying the ACO technique, 3) a code coverage enhancement method which infers sets of models.

The evaluation of *MCrawlT* against other recent tools shows that our approach can be used in practice. It automates testing tasks that users usually consider tedious. Furthermore, it generates models and storyboards that can be used for model analysis and comprehension aid. *MCrawlT* also provides good code coverage quickly and detects more bugs than those exposed by the other tools.

The initial purpose of this work was to generate partial models given to an automatic security testing method for Mobile applications. Based upon this framework, we intend to design this security testing method by developing these future research directions: 1) define an exploration strategy in order to automatically detect the highest number of security issues while the model generation, 2) devise or reuse verification methods on inferred models to detect security vulnerabilities, 3) generate test cases to extend the inferred models from some specific states in an attempt to expose further security vulnerabilities.

References

[1] S. Salva and S. R. Zafimiharisoa, "Model reverse-engineering of mobile applications with exploration strategies," in *The Ninth International Conference on Software Engineering Advances, ICSEA 2014*, Nice, France, 10 2014, pp. 396–403.

[2] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–269. [Online]. Available: http://dl.acm.org/citation.cfm?id=950792.951350

[3] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 1–30, 2012.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *Software Engineering, IEEE Transactions on*, vol. 36, no. 4, pp. 474–494, 2010.

[5] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools*, ser. JSTools '12. New York, NY, USA: ACM, 2012, pp. 11–15. [Online]. Available: http://doi.acm.org/10.1145/2307720.2307722

[6] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[8] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[9] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar – a tool for automated model-based testing of mobile apps," *IEEE Software*, vol. 99, no. PrePrints, pp. 1–6, 2014.

[11] Google. Ui/application exerciser monkey. Accessed: 2015-03-01. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[12] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[13] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, Oct 2008, pp. 69–73.

[14] M. E. Joorabchi and A. Mesbah, "Reverse engineering ios mobile applications," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, ser. WCRE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 177–186. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2012.27

[15] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[16] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(87)90052-6

[17] D. Amalfitano, A. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 252–261.

[18] J.-C. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Science of Computer Programming*, vol. 13, pp. 13–219, 1989.

[19] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10515-013-0128-9

[20] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proc. of the 25th International Conference on Software Engineering*, 2003, pp. 38–48.

[21] Robotium, user scenario testing for android. Accessed: 2015-03-01. [Online]. Available: http:/code.google.com/p/robotium/

[22] Emma, a free java code coverage tool. Accessed: 2015-03-01. [Online]. Available: http://emma.sourceforge.net

[23] Soapui. Accessed: 2015-03-01. [Online]. Available: http://www.soapui.org