Model Reverse-engineering of Mobile Applications with Exploration Strategies

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

Stassia R. Zafimiharisoa
LIMOS - UMR CNRS 6158
Blaise Pascal University, France
email: s.zafimiharisoa@openium.fr

***Abstract*—This paper presents a model reverse-engineering approach for mobile applications that belong to the Graphical User Interface (GUI) application category. This approach covers the interfaces of an application with automatic testing to incrementally infer a formal model expressing the navigational paths and states of the application. We propose the definition of a specialised GUI application model which stores the discovered interfaces and helps limit the application exploration. Then, we present an algorithm based upon the Ant Colony Optimisation technique which offers the possibility to parallelise the exploration and to conceive any application exploration strategy. Finally, our approach is experimented on Android applications and compared to other tools available in the literature.**

Keywords–model generation; automatic testing; android applications.

I. Introduction

Many software engineering approaches rely upon models to automate some steps of the software development life cycle. Unfortunately, these kind of approaches suffer from an indisputable problem which often makes them impractical with many real world systems: writing models, especially exhaustive ones, is often a tedious and error-prone task. As a consequence, only partial models are often available which makes model-based approaches less interesting. For instance, Model-based testing is an approach which takes formal specifications to generate test cases, but the former have to be complete.

Model inference or model reverse-engineering is a recent research field that partially address this issue. Indeed, models can be inferred from application documentation or execution traces (sequences of actions given or observed from the application) for comprehension or to automatically carry out some tasks, e.g., the test case generation. Most of the model generation approaches, available in the literature, focus on GUI applications (a.k.a. event-driven applications), which offer a Graphical User Interface (GUI) to interact with and which respond to a sequence defined by the user. In short, these applications are explored (a.k.a. crawled) with automatic testing techniques for extracting traces to derive a model. Furthermore, a large part of the application defects can eventually be detected during the process. Afterwards, these generated models may be manually extended, analysed with verification techniques or employed for generating test cases.

In this paper, we propose a model reverse-engineering approach, combined with automatic testing, which is dedicated to mobile applications. These GUI applications for smartphones, are usually poorly documented and are often manually tested. From a mobile application, our solution generates two STS (Symbolic Transition System) specifications, which can be seen as documentation either useful for maintaining the application or for comprehension, or for performing automatic model analyses and test case generation (verification with model-checkers, etc.).

Several works already deal with the crawling of GUI applications e.g., desktop applications [1], Web applications [2][3][4] or mobile ones [5][6][7]. These approaches interact with applications in an attempt to either detect bugs or record a model or both. These previous works already propose interesting features, such as the test case generation from the inferred models. Nonetheless, it also emerges that many interesting issues still remain open. Firstly, experimenting the GUIs of Web or mobile applications may lead to a large and potentially unlimited number of states that cannot be all explored. Furthermore, 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 processing time is limited, using other strategies could help in the exploration of the most interesting features of the application as a first step.

This paper presents an innovative model generation approach which overcomes the previous problems by putting forth the following features:

- model definition and compactness: we propose an original model definition specialised to mobile applications. Combined with our application exploration algorithm, this model especially offers the advantage to help limit the exploration and to prevent from a state space explosion. But, this model can still store the discovered interfaces and their properties instead of resorting abstract event-based descriptions only. These detailed information are particularly relevant to later perform precise analyses. A bisimulation minimisation technique is also applied to yield a second reduced STS which can be more easily interpreted,
- test data generation: instead of using random test values, the values used to fulfil the application interfaces are constructed from several data sets, and in particular from a set of fake identities. Furthermore, for one interface, the set of test value tuples are constructed

by means of a Pairwise technique to reduce the testing cost,

- strategy choice: the application exploration is here guided by strategies that are applied on the model under generation by means of the Ant Colony Optimisation (ACO) technique. We also show that our exploration algorithm, based upon the ACO heuristic, is highly parallelisable.

The paper is structured as follows: Section II sets down the terminology of mobile applications used throughout the paper and particularly presents our model definition. We present, in Section III, our mobile application exploration algorithm based upon the ACO heuristic. We give some experimental results and compare our approach with available tools in Section IV. We briefly present some related work and discuss about our proposal in Section V and we conclude in Section VI.

II. Mobile application modelling

A. Terminology

We say that a mobile application displays (graphical user) interfaces, each representing one application state (the number of states being potentially infinite). An interface is generated by a component of the application. Here, we take back the notation used in the Android Operating System (OS) where such a component is called an *Activity*. These instantiate *Widgets* (buttons, text fields, etc.) and declare the available events that may be triggered by the user (click, swipe, etc.). A Widget is characterised by a set of properties (colour, text values, etc.); some of them are said *editable*, which means that their values can be provided by users at runtime.

We take as example the *Ebay Mobile* application, which is available on the *Google Play* store[8]. Since this complex application owns 135 Activities, we only depict a part of its storyboard in Figure 1. The launcher interface is loaded by the first 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 *MainSearchActivity* is reached (i1). For instance, if the user enters the keyword "shoes", the search result list is displayed (i2); the Activity is unchanged. Then, three new Activities may be reached: 1) an Activity called *SegmentSearchResultActivity* (i3) displays a result when one element of the proposed list 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 *SignInActivity*, i5).

B. The STS model

To represent the behaviours of mobile applications, we shall consider the Symbolic Transition System (STS) model, which is a kind of automata model extended with variables that encode the state of the system. Transitions also carry actions combined with parameters, guards and assignments. We chose the STS definition proposed in [9] which does not explicitly represent states in transitions. Instead, *(control) locations* are encoded with variables taking values in finite domains. This definition offers more flexibility to represent locations that have a precise meaning by means of variables.

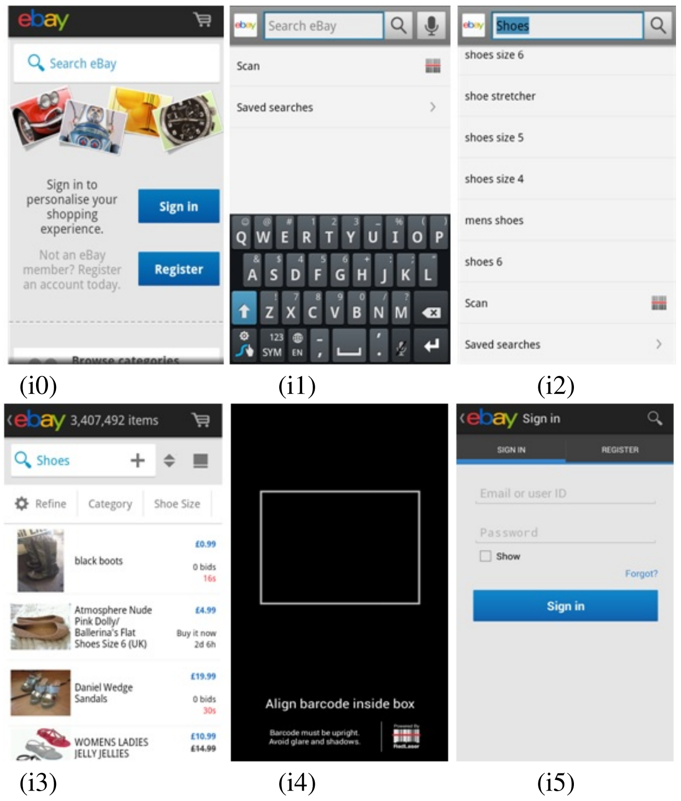

Figure 1. Ebay Mobile Storyboard

Definition 1 (STS) *A STS $\mathcal{S}$ is a tuple $< V, V0, I, \Lambda, \rightarrow>$, where:*

- *V is the finite set of internal variables and I is the finite set of parameter variables. A variable can have a simple type (Integer, String, etc.) or a complex type (List, etc.). We denote D_v the domain in which a variable v takes values. The internal variables are initialised with the initial condition $V0 \subseteq D_v$, which is assumed to be unique,*
- *Λ is the finite set of symbolic actions $a(p)$, with $p = (p_1, ..., p_k)$ a finite list of parameters in $I^k (k \in \mathbb{N})$,*
- *$\rightarrow$ is the finite transition set. A transition $(a(p), G(p, v), T(v, p)), A(v, p, T(v, p)))$ is labelled by an action $a(p) \in \Lambda$. $G \subseteq D_p \times D_V \times D_{T(p \cup V)}$ is a guard on internal variables, parameters and $T(p \cup V)$ a set of functions that return boolean values only (a.k.a. predicates) over $p \cup V$. Internal variables are updated with the assignment function $A : D_V \times D_p \times D_{T(p \cup V)} \rightarrow D_V$ once the transition is fired.*

Below, we adapt this generalised STS definition to express mobile application properties, i.e., interfaces and events.

C. Mobile application modelling with STS

We propose a STS-based model definition allowing to stock complete mobile application interfaces to yield rich models, which may be analysed afterwards. Nonetheless, a GUI application may produce a potentially infinite set of interfaces

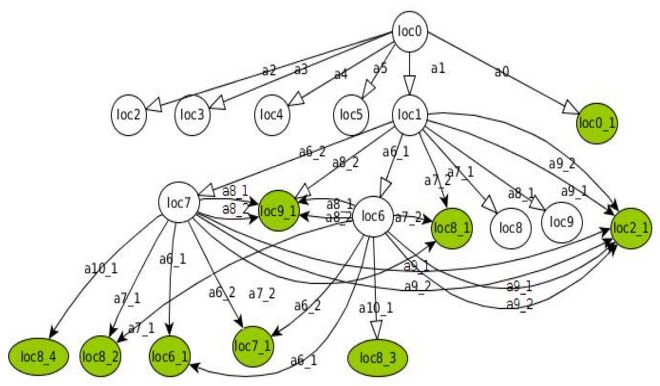

Figure 2. Ebay application STS tree

TABLE I. Actions and Guards of the STS Tree

Label	Action[Guard]
$a1$	click(widget)[widget=id/$home\_search\_text$]
$a6\_1$	click(widget)[widget=$id/up \wedge search\_src\_text$=All shoes]
$a6\_2$	click(widget)[widget=$id/up \wedge search\_src\_text$=shoes]
$a7\_1$	click(widget)[widget=$id/search\_button$ $\wedge$ $search\_src\_text$=All shoes]
$a7\_2$	click(widget)[widget=$id/search\_button$ $\wedge$ $search\_src\_text$=shoes]
$a8\_1$	click(widget)[widget=$id/text1 \wedge search\_src\_text$=All shoes]
$a8\_2$	click(widget)[widget=$id/text1 \wedge search\_src\_text$=shoes]
$a9\_1$	click(widget)[widget=$id/text2 \wedge search\_src\_text$=All shoes]
$a9\_2$	click(widget)[widget=$id/text2 \wedge search\_src\_text$=shoes]
$a10\_1$	click(widget)[widget=$listview$ at position 1 $\wedge$ $search\_src\_text$=shoes]

[6][10] and may lead to a state space explosion problem. We propose to get around this strong issue by focusing on the following idea: many of these interfaces are almost identical in term of content and often display different text field values. For a set of almost identical interfaces, we propose to only explore one interface in this set. To this end, we express an interface by the tuple (wp, wt) where wt is the list of Widget properties related to the text field values found in the interface and wp the remaining list of Widget properties. We define that a STS location is encoded by the variable loc, and captures a value list of the form (act, wp, wt, end, ph) with act an Activity name (or URI), accompanied by the Widget property lists wp and wt. Furthermore, these locations are completed with a boolean value denoted end indicating whether the application has to be explored from this location. Finally, the positive value ph denotes a pheromone amount that shall be used by apply the ACO technique. The purpose of this value is explained in the next Section.

We also interact with mobile applications by means of events, e.g., a click, applied on Widgets. Some editable Widgets are eventually completed before triggering the event. We capture these events with STS transitions of the form $(event(widget), G, A)$. The guard G is composed of conjunctions which show the initial location of the transition, a constraint over editable Widgets expressing their completion with user values, and the value of $widget$, giving the Widget name on which is applied the event. The assignment A gives the final location of the transition. It results that we express the functioning of a mobile application with the following STS model, called the STS Tree of an application:

Definition 2 *A mobile application is modelled by the STS Tree* $< V, V0, I, \Lambda, \rightarrow >$ *where:*

- Λ *gathers the available actions of the form* $event(widget)$*,*
- $\rightarrow$ *is composed of transitions* $(event(widget), G, A)$ *with a guard* G *of the form* $[loc == (act, wp, wt, end, ph) \wedge editable\_constraint \wedge Widget == wn]$ *and an assignment* A *of the form* $loc := (act2, wp2, wt2, end2, ph2)$*:*
    - *the expression* $loc == (act, wp, wt, end, ph)$ *gives the initial location of the transition, while the assignment* $loc := (act2, wp2, wt2, end2, ph2)$ *gives the final location.* act *is an Activity name,* wt *is a list of Widget properties relative to text field values,* wp *is a list of Widget properties excluding* wt*,* end *and* ph *are boolean values,*
    - $editable\_constraint$ *is a conjunction of atomic expressions of the form* $widgetprop == v$ *with* v *a value and the variable* $widgetprop$ *corresponding to an editable Widget property.*
    - $widget == wn$ *denotes the Widget name on which is applied the event.*
- $V0$ *denotes the initialisation of the* loc *variable.*

Figure 2 illustrates an example of STS Tree derived from the Ebay Mobile application. For readability, the locations are not given in the transitions but some of them are presented in a reduced form in Table II: we give the Activity name, the numbers of Widget properties (wt and wp), and the values end and ph. The actions and guards are showed in Table I. The STS Tree is composed of several "click" actions applied on different Widgets (buttons, elements of listView Widgets, etc.). The location $loc0$ represents the initial interface *eBay* of the application, which includes 2 buttons, 6 images and 16 text fields. $loc1$ is reached from $loc0$ by executing the action $a1$, i.e., by clicking on the $home\_search\_text$ Widget. The locations $loc6$ and $loc7$ are respectively reached after the completion of the $search\_src\_text$ Widget with the "All shoes" resp. "shoes" text values and the click on the Widget up (actions $a6\_1$ or $a6\_2$). These two locations correspond to two different interfaces which differ from each other on the value of the $search\_src\_text$ field and on the Widget *listview* which is a Widget container: the latter has 1 element for $loc6$ and 10 elements for $loc7$. The locations $loc8\_i (1 \leq i \leq 4)$, reached from $loc6$ or $loc7$, express 4 interfaces which only differ from the interface stored in $loc8$ by some text field values. As a consequence, they are marked by end to stop the exploration.

After covering only 5% of the *Ebay Mobile* Activities, we already obtain 19 locations in the STS Tree. This is why our approach, explained below, relies upon a minimisation technique to reduce this location number.

III. Automatic testing and model generation with ACO

Intuitively, 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 either a

TABLE II. Summary of some locations of the STS Tree

Loc	act	#wp	#wt	end	ph
$loc0$	eBay	2b,6im	16t	false	0
$loc1$	MainSearchAct	1b,4im	3t,1e	false	1
$loc6$	MainSearchAct	1b,3im,1l_e	3t,1e	false	2
$loc6\_1$	MainSearchAct	1b,3im,1l_e	3t,1e	true	3
$loc7$	MainSearchAct	1b,3im,10l_e	3t,1e	false	2
$loc7\_1$	MainSearchAct	1b,3im,10l_e	3t,1e	true	3

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

DFS (Depth First path Search) or a BFS (Breadth First path Search) strategy. Nonetheless, 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 part of the application is not necessarily the most interesting one. In this section, we address this issue and we propose an algorithm which includes the possibility to define an exploration strategy.

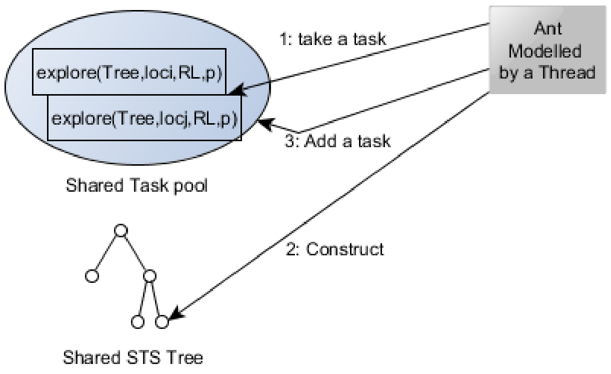

Figure 3. Parallel exploration functioning

Our proposal applies strategies by means of the Ant Colony Optimisation (ACO) technique. With ACO, 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, the ants explore randomly and lay down little by little pheromone trails that are finally followed by all the ants. In our case, this solution leads to the architecture illustrated in Figure 3. The STS construction is guided by laying down in locations an amount of pheromone with regards to the chosen strategy. Each location exploration is considered as a task that is placed into a task pool, implemented as an ordered list, and executed by threads simulating ants. Then, our algorithm proceeds by exploring first the locations having the highest pheromone amount. The process ends when the task pool is empty. These steps are explained below:

A. Application exploration

Algorithm 1 takes as input a mobile application app and starts it to analyse its first interface and to initialise the first location loc_0 of the STS Tree $Tree$. This step is carried out by one thread only. The analysis of an interface does not rise any technical difficulty with mobile applications (Android and iOS). Indeed, it is always possible to retrieve the Activity and the Widget properties of the current interface with testing tools such as Robotium [11]. Afterwards, the interface exploration can begin: each thread (ant) executes the loop of

Algorithm 1: Mobile application exploration simplified Algorithm

```
input : Application $app$
output: STS $Tree$, $MTree$
// initialisation performed by one ant only
1  Start the application $app$;
2  Analyse the current interface $\rightarrow$ Activity $act$, the Widget property
   lists $wp$, $wt$;
3  Initialise $ph_0$ (depends of the chosen strategy);
4  $loc_0$:= $(act, wp, wt, false, ph_0)$;
5  Initialise STS $Tree$ ($V0_{Tree} = loc_0$);
6  Add $(Explore(Tree, loc_0, RL = \{(act, wp)\}, p = \emptyset))$ to the task
   pool;
// code performed by all the ants
7  while the task pool is not empty do
8      Take a task $(Explore(Tree, loc_i, RL, p))$ having a location
       $(act, wp, wt, end, ph)$ with the highest pheromone amount
       $ph$;
9      Reset and Execute $app$ by covering $p$;
10     Explore($Tree$,$loc_i$,$RL$,$p$);
// code performed by one ant only
11 $MTree$:= Minimise($Tree$);
```

TABLE III. Location blocks of the minimised STS Tree

block	locations
$B1$	$loc6, loc7$
$B2$	$loc6\_1, loc7\_1$
$B3$	$loc8\_1, loc8\_2, loc8\_3, loc8\_4$

Algorithm 1: while there is a task to do, an instance of the application is launched in a re-initialised test environment and a task $(Explore(Tree, loc_i, RL, p))$ having a location loc_i, composed of the highest pheromone amount, is picked out. This task aims at exploring one interface only and may produce other tasks. The set RL, used by Explore, stores the discovered locations in a reduced form (act, wp).

After the end of the exploration, a second STS $MTree$ is computed with a minimisation technique. The STS minimisation aims to yield a more compact STS in term of STS location number and more readable for application comprehension. Here, we have chosen a bisimulation minimisation technique since this one preserves the functional behaviours represented in the original model. 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). A detailed algorithm can be found in [12]. This algorithm constructs the location sets (blocks) that are bisimilar equivalent. Due to lack of room, we only present in Figure 4 and in Table III, the minimised STS obtained from the STS Tree of Figure 2. Some locations are now grouped into blocks: for instance, the locations $loc6$ and $loc7$ are grouped into the Block $B1$ because the same action sequences leading to bisimilar locations can be executed from both $loc6$ and $loc7$.

One task, pulled from the task pool, is now performed by calling the Explore procedure given in Algorithm 2. It takes the STS under construction $Tree$, a location loc_i, a path p and the set RL of discovered locations stored in a reduced form. Initially, the procedure ends if a stopping condition, based upon the code coverage and on the processing time, holds. This condition allows to stop the exploration after a reasonable time delay. Otherwise, the Explore procedure calls *GenConstraints* to analyse the current interface, extract the editable Widgets

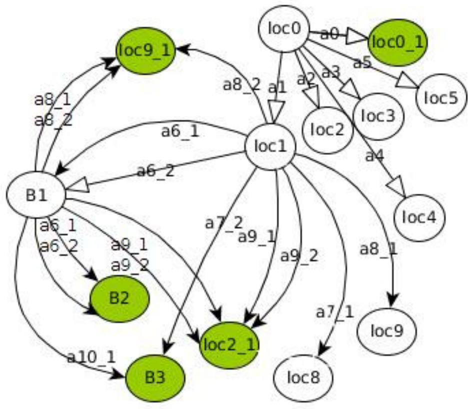

Figure 4. Minimised STS Tree

and to produce a set of constraints expressing how fulfilling these editable Widgets with test values. Similarly, the events that can be triggered on the Widgets are dynamically detected (with testing tools). It results a set of couples $(event, w)$ with $event$ the event to apply on the Widget w. Then, the exploration of the current interface begins. Its editable Widgets are completed in accordance with a constraint c (line 7). A Widget w is stimulated with an event in reference to a couple $(event, w)$ found in the Events set. This results in a new interface $Inew$ (line 9). A $Ph\_Deposit$ procedure is called to compute the pheromone amount that shall be deposed in the arrival location of the transition constructed in the next step. This amount is computed with regards to the chosen strategy. The algorithm now checks whether this interface and its corresponding location have to be explored. Naturally, if $Inew$ reflects the end of the application (exception, crash), $Inew$ must not be explored. Furthermore, if $Inew$ only differs from a previously encountered interface by its text field values, we also stop the exploration. This is done in the algorithm by checking if the list (act_j, wp_j), extracted from $Inew$, which excludes the Widget properties related to text field values, belongs to the set RL. If one of these conditions hold then a new transition carrying the arrival location $(act_j, wp_j, wt_j, true, ph_j)$ is added to the STS $Tree$. The boolean value $true$ denotes that this location must not be explored. On the contrary, a new transition is added (with a location loc_j whose last boolean value is set to false). This location loc_j has to be explored. Therefore, a new task is added to the task pool (line 18). To apply the next constraint and event, the application has to go back to its previous interface by undoing the previous interaction. This is done with the Backtrack procedure (line 20) whose role is to undo the most recent action. When the direct interface restoration is not possible (when the backtrack mechanism is not implemented or when the application crashed), the Backtrack procedure resets the application and incrementally replays the actions of the path p.

This algorithm also relies upon the procedure *GenConstraints* to construct constraints expressing how to fulfil an interface under test with values. Due to lack of room, we present it succinctly. The *GenConstraints* procedure aims to generate constraints of the form $w_1.value = v1 \wedge \ ... \ \wedge w_n.value = vn$, with $(w_1, ..., w_n)$ the list of editable Widgets of an interface and $(v_1, ..., v_n)$, a list of test values. Instead of using random values like in many model inference approaches, we propose

Algorithm 2: Explore Procedure

```
1  Procedure $Explore(Tree, loc_i, RL, p)$;
2  if [$processing time > T$ or $code coverage > CC$] then
3  stop;
4  Generate constraints with GenConstraints $\rightarrow C$;
5  Analyse the current interface $\rightarrow$ Events;
6  foreach $c \in C \wedge (event, w) \in Events$ do
7      fulfil the editable Widgets with $c$;
8      Apply $event$ on the Widget $w \rightarrow$ new interface $Inew$;
9      Analyse the interface $Inew \rightarrow act_j, wp_j, wt_j$;
10     $ph_j$:= $Ph\_Deposit(loc_i, act_j, wp_j, wt_j)$;
11     if $Inew$ is empty or $Inew$ reflects a crash or there exists
       $(act_j, wp_j) \in RL$ then
12         {Add a transition $(event(widget)$,
           $G = [loc == loc_i \wedge c \wedge widget == w], A = (loc :=$
           $(act_j, wp_j, wt_j, true, ph_j)))$ to $\rightarrow_{Tree}$;
13         } (in critical section)
14     else
15         $loc_j := (act_j, wp_j, wt_j, false, ph_j)$;
16         {Add a transition $t = (event(widget)$,
           $G = [loc == loc_i \wedge c \wedge widget == w], A = (loc := loc_j))$
           to $\rightarrow_{Tree}$;
17         $RL := RL \cup \{(act_j, wp_j)\}$
18         Add the task (Explore($Tree, loc_j, RL, p.t$)) to the task
           pool;
19         } (in critical section)
20     Backtrack($loc_i, p$);
```

to use several data sets: a set $User$ of values, eventually composed of logins and passwords, provided by a user, a set RV composed of values well known for detecting bugs, e.g., String values like "&", "", or null. A last set, denoted $Fakedata$, is composed of fake user identities. An identity is itself a list of parameters $(p_1, ..., p_m)$, such as (name, age, email, address, gender), that are correlated together to form realistic identities. Both $User$ and RV sets are segmented per type (String, Integer, etc.). We denote $type(User \cup RV) \subset User \cup RV$ the subset of values having the type $type$. The *GenConstraints* procedure starts collecting the editable Widget list $(w_1, ..., w_n)$. Every w_i is then associated to a specific data set as follows:

1) *GenConstraints* extracts the larger subset $(w_1, ..., w_k)$ which is also a subset of the parameter list $(p_1, ..., p_m)$ of $Fakedata$ (we try to find a correlation between the Widget names and the identity parameters with regular expressions). This subset of Widgets is then associated to a list of "reduced" identities where the parameters which do not belong to $(w_1, ..., w_k)$ are removed. 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 the set of identities composed only of a name and an email,
2) each remaining Widget, is associated to the value set $t(User \cup RV)$ with t the type of data expected by the Widget (usually String). We obtain a list of value sets $\{V_1, ..., V_n\}$ linked to the Widgets $(w_1, ..., w_n)$

Now, instead of using a cartesian product to derive a set of tuple of values denoted V, we adopted a Pairwise technique [13]. 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 pair of parameters only. Finally, the set of constraints C is derived from V.

Last but not least, our proposal also offers the advantage of being highly parallelisable. Indeed, the task pool is a known paradigm of parallel computing where the tasks of the pool are executed in parallel on condition that the tasks are independent. This is the case in our Algorithms since several application instances are experimented into independent test environments. All the threads share the same STS $Tree$, the same discovered location set RL and the same task pool implemented as an ordered list. This is why we added three critical sections in the Explore procedure to prevent concurrent accesses when transitions are added to the STS (lines 12,13,16), or when a task is added into the pool (line 18).

Complexity and termination of Algorithm 1: theoretically, this algorithm does not end if the number of new interfaces to visit is infinite. This is why we added a stopping condition in the Explore procedure. But, our algorithm only explores the interfaces, which have new Widget properties (in excluding those related to text field values), and we have observed in practice that the number of these interfaces is often bounded. Consequently, our algorithm ends with most of the applications. If we assume that the number of locations to visit is then bounded to n, Algorithm 1 has a complexity proportional to $\mathcal{O}(m+n+mn+2mlog(n))$ with m the number of transitions.

B. Exploration strategies

Different strategies can be now used to cover an application. We succinctly present some of them below. These have to be implemented in the $Ph\_Deposit$ procedure.

- **DFS-BSF strategy:** a combination of both DFS and BFS strategies can be easily put into practice as follows: the location loc_0 is initialised with a pheromone amount equal to 0. Afterwards, whenever a new location loc_j is detected from an initial one loc_i, it is completed with the pheromone amount found in loc_i increased by 1. In this case, the next task chosen in the task pool shall be the one including the first discovered location from loc_i. Tacitly, a DFS strategy is followed. But, the current location being explored, is also completely covered in a breadth-wise order first,
- **crash-driven strategy:** the number of observed crashes could also be considered in a strategy: when the number of crashes detected from the locations of a path p is higher than the crash number detected from the locations of another path p', it may be more interesting to continue to cover the former for trying to detect the highest number of crashes. We call this strategy crash-driven exploration. This can be conducted as follows: the pheromone amount is initialised to 0 in loc_0. Whenever a new location loc_j is built, it is completed with a pheromone amount equal to the addition of the pheromone amount found in the preceding location loc_i with the number of crashes (or exceptions) detected from loc_i,
- **semantics-driven strategy:** these strategies denote 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 X, otherwise it is equal to 1. When a Widget name is composed of the term ”buy”, the pheromone amount added in the location could be $Y < X$, etc.

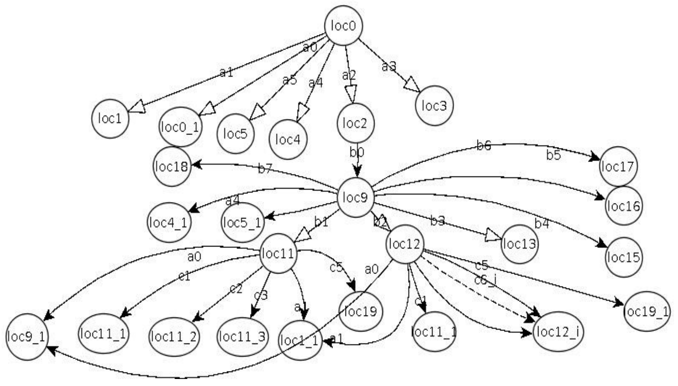

Figure 5. Ebay Mobile STS Tree obtained with a semantics-driven strategy

Many other strategies could be defined in relation to the desired result in terms of model generation and test coverage. Other criteria, e.g., the number of Widgets, could also be taken into consideration. The strategies, succinctly described above, could also be mixed together.

The STS Tree of Figure 2 is constructed with Algorithm 1 related to the DFS-BFS strategy. The Explore procedure starts the exploration from $loc0$, which holds a pheromone amount equal to 0. The actions $a0$ to $a5$ lead to new interfaces and locations $loc1, ..., loc5$ that have to be explored. Here, the location $loc1$ is chosen since it is the first new encountered location and has the highest pheromone amount. From $loc1$, the execution of actions leads to new locations: for instance the locations $loc8$ and $loc8\_1$ are reached with the actions $a7\_1$ and $a7\_2$. These locations only differ by their text field values. Hence, the arrival location $loc8\_1$ is not explored and marked by end. The next location having the highest pheromone amount is $loc6$. Therefore, this one is explored. And so on.

We also applied a semantics-driven strategy on this application to illustrate the different STS Trees which may be generated. This strategy aims to target the account management part of the application and was applied by deposing a higher pheromone amount in locations including Widgets of type ”passwdtype” or Widget properties composed of the terms ”account” or ”sign in”. Figure 5 illustrates the resulting STS Tree after applying this strategy: here the Activity $SignInActivity$ ($loc2$), allowing to manage user accounts, was targeted instead of the Activity $MainSearchAct$ ($loc1$). This strategy makes the generated STS more interesting to later analyse the security of the application or to generate security test cases.

TABLE IV. Processing time to explore all the locations with different strategies

Application	DFS(1)	DFS-BFS(1)	DFS-BFS(3)
Converter	478	435	295
NotePad	268	310	175
Tippy Tipper	251	210	110
ToDoManager	551	410	210
LotsA	70	83	48
OpenManager	696	560	489
HelloAUT	106	216	201
TomDroid	235	256	196
ContactManager	233	216	135
OpenSudoku	434	456	411

TABLE V. Code and Activity coverage

Applications	Mon key	Orbit	GUI TAR	GUI Rip-per	MCrawlT	
					Code cov.	Act. cov.
NotePad	60	82	-	-	88	100
ToDoManager	71	75	71	-	81	100
HelloAUT	71	86	51	-	96	100
TomDroid	46	70	-	40	76	100
Youtube	-	-	-	-	-	54.5
CNN	-	-	-	-	-	73
TaskKiller	-	-	-	-	-	57.1
Ebay	-	-	-	-	-	19
WordPress	-	-	-	39	-	47
CatLog	-	-	-	-	77	80
DiskToFon	-	-	-	-	42	67
SipDroid	-	-	-	-	-	11

IV. Experimentations

We conducted several empirical studies to assess the overall results of our approach applied on Android mobile applications. Our prototype tool, called *MCrawlT (Mobile Crawler Tool)*, is publicly available in a Github repository [14]. It takes packaged Android applications (apk files) or source projects and stimulates them by calling the testing framework *Robotium*. This one is also employed to analyse interfaces. An application can be experimented in parallel by launching several Android emulators. We randomly chose some Android applications of the *Google Play* store and some applications taken as examples in other papers dealing with Android application automated testing for comparison purposes.

Table IV reports the processing time required for completely exploring these applications with a Mid 2011 computer including a CPU 2.1Ghz Core i5 and 4GB of RAM. The tool were applied with a DFS strategy (1 emulator), a mixed DFS-BFS (with 1 and 3 emulators in parallel). Our results firstly show that the chosen strategy has a direct impact on the processing time required to cover an application. In this experimentation, half of the applications are more rapidly covered with DFS-BFS traversing. For instance, with *toDoManager*, using a DFS-BFS strategy instead of a DFS one, reduces the exploration delay by 140 seconds because all of its Activities are directly accessible from the initial one. These results depend mainly on the application structure though. When the insight of the application structure is known, our tool offers the advantage of choosing the most appropriate strategy. Table IV also shows that the parallelisation of our algorithm is effective. With three emulators, the processing time is always reduced. For instance, the parallel exploration of *Tippy Tipper* is achieved with a processing time almost divided by two.

TABLE VI. Crash detection

Applications	MCrawlerT	Monkey	GUI Rip-per
Converter	9	4	
Notepad	2		
TomDroid	3	1	14
WordPress	51	3	37
CatLog	17	0	
DiskToFon	2	0	
Sipdroid	1	1	

Table V shows the resulting code coverage obtained with our tool and other crawlers available in the literature: Monkey [15], Orbit [7], GUITAR [1], GUI Ripper [6]. With our tool, we provide the code coverage that is obtained for the applications whose source code is available (small open source applications). For the others, we can only give the Activity coverage (explored Activities). Most of the other tools explore Android applications in an in-depth manner. Therefore, MCrawlT was executed only with this strategy to carry out a fair comparison. These results show that the code coverage is between 42% and 96%. An application is incompletely covered either on account of unused code parts (libraries, packages, etc.) that are not called, or on account of functionalities difficult to start automatically. The code coverage achieved with MCrawlT is either equivalent or higher than the one given by the other tools. For instance with *TomDroid*, we obtained 76 %, whereas ORBIT covers 70%, Monkey 46% and GUI Ripper 40% of the code. ORBIT offers a better code coverage with *Contactmanager* though. Indeed, users interact on this application with long click events that are supported by Orbit but not yet by our tool. The last lines of Table V show the results obtained with larger applications (*Youtube* to *Sipdroid*). Since the time required to discover these applications may be long, we have limited the exploration time to 30 minutes. Without limitation, the coverage should strongly augment. Surprisingly, this kind of application is not considered by the other tools.

Finally, Table VI illustrates the number of observed crashes on some Android applications with our tool *MCrawlT*, *Monkey* and *GUI Ripper*. We only show the applications for which at least one error has been detected with one of the tools. The processing time was limited to 30 minutes for the two first tools. For GUI Ripper, we have taken back the experimental results given in [6] that were obtained with a processing time varying between 3 and 5 hours. *MCrawlT* outperforms *Monkey* in automatic crash detection, which is not surprising since the former covers deeper Android applications. The comparison with GUI Ripper is less obvious since the authors only provide two detailed results with this tool. For *WordPress*, *MCrawlT* detects more crashes than *GUI Ripper*, and on the contrary, more crashes are detected with *TomDroid*. But, the processing time of GUI Ripper is twelve times more long.

All these experimental results on real applications tend to show that our tool is effective and leads to substantial improvement in the automatic testing and model inference of GUI applications.

V. Related Work and discussion

Several papers dealing with automatic testing and model generation approaches of black-box systems were issued in the last decade. Here, we present some of them relative to our work:

Memon et al. [1] initially presented GUITAR, a tool for scanning desktop applications. This tool produces event flow graphs and trees showing the GUI execution behaviours. Only the click event can be applied and GUITAR 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). Mesbah et al. [2] proposed the tool Crawljax specialised in Ajax applications. It produces a state machine model to capture the changes of DOM structures of the HTML documents by means of events (click, mouseover, etc.). To limit the state space and to avoid a state explosion problem, state abstractions should be given manually to extract a model with a manageable size. The concatenation of identical states proposed in [2] is done in our work by minimisation.

Google's Monkey [15] is a random testing tool (events and data) offering light coverage especially when an authentication is required in the application. No model is provided. Amalfitano et al. [6] proposed GUI Ripper, a crawler for crash testing and for regression test case generation. A simple model, called GUI tree, depicts the observed GUI. Then, paths of the tree not terminated by a crash detection, are used to re-generate regression test cases. Joorabch et al. [10] proposed another crawler, similar to GUI Ripper, dedicated to iOS applications. In comparison to these works, our generated models are much more detailed and can be used to derive new test cases since all the actions and parameters can directly be found in STS Trees. We also consider several exploration strategies. The novelty of the work proposed by Yang et al. [7] lies in the static analysis of Android application codes to infer the events that can be applied to the GUI. Then, a classical crawling technique is employed to derive a tree composed of events. This grey-box testing approach was implemented in the Orbit tool. When only one emulator is used, this approach should cover an application quicker than our proposal since the events to trigger are listed by the static analysis whereas we try to detect them dynamically. But, Orbit can be applied only when the application source code is available. This is not the case for many Android applications. Furthermore, our tool should cover an application quicker than Orbit since the former can be experimented in parallel with several emulators. Another strong advantage proposed in our approach, is the support of different exploration strategies. These can reduce the exploration time when the application structure is known or can guide the exploration when the application interface number is large.

VI. Conclusion

This paper presents a formal model inference approach for mobile applications, which performs automatic testing through application interfaces and which explores applications by means of strategies. For one application, two STS models are generated by this approach. Both express the functional behaviours of the application, but the second one is reduced with a bisimulation technique for readability.

In comparison to the application crawlers available in the literature, this approach takes another direction by proposing the two following main contributions. We propose a formal model definition whose aims are to store rich details about the encountered interfaces and to help reduce the application exploration. Our algorithms are based upon the application of the ACO technique to guide the application exploration with strategies that can be modified by managing differently the pheromone deposit in locations. Our experimental results show that this approach can be used in practice: the prototype tool provides a good application code coverage in a reasonable time delay.

Acknowledgment

This reasearch was conducted with the support of the " Digital Trust" Chair from the University of Auvergne Foundation.

References

[1] 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.

[2] 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, 2012, pp. 1–30.

[3] 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, 2010, pp. 474–494.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Google, "Android google play store," last accessed August 2014. [Online]. Available: https://play.google.com/store

[9] T. Jéron, "Symbolic model-based test selection," Electronic Notes in Theoretical Computer Science, vol. 240, no. 0, 2009, pp. 167 – 184. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106610900173X

[10] 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.

[11] Robotium, "Robotium tool," last accessed August 2014. [Online]. Available: https://code.google.com/p/robotium/

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

[13] 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.

[14] MCrawlerT, "Mobile crawler tool," last accessed August 2014. [Online]. Available: https://github.com/statops/MCrawlerT

[15] Google, "Ui/application exerciser monkey," last accessed August 2014. [Online]. Available: http://developer.android.com/tools/help/monkey.html