Model inference of Mobile Applications with dynamic state abstraction

Sébastien Salva [1]
Patrice Laurençot [2]
Stassia R. Zafimiharisoa [3]

Research Report LIMOS/RR-15-01

12 mars 2015

1. LIMOS, UMR 6158, PRES Clermont-Ferrand University, FRANCE, sebastien.salva@udamail.fr
2. LIMOS, UMR 6158, PRES Clermont-Ferrand University, FRANCE, laurenco@udamail.fr
3. LIMOS, UMR 6158, PRES Clermont-Ferrand University, FRANCE, s.zafimiharisoa@openium.fr

Abstract

We propose an automatic testing method of mobile applications, which also learns formal models expressing navigational paths and application states. We focus on the quality of the models to later perform analysis (verification or test case generation). In this context, our algorithm infers formal and exact models that capture the events applied while testing, the content of the observed screens and the application environment changes. Indeed, the more data we collect, the more precise an analysis can be done later. A key feature of the algorithm is that it avoids the state space explosion problem by dynamically constructing state equivalence classes to slice the state space domain of an application in a finite manner and to explore these equivalence classes. We implemented this algorithm on the tool *MCrawlT* that was used for experimentations. The results show that *MCrawlT* achieves significantly better code coverage than several available tools in a given time budget. We show that the obtained models are expressive, i.e. they hold enough information about the application to later generate detailed test cases.

Keywords: Model inference, Automatic testing, Android applications, State abstraction

1 Introduction

Desktop, Web and more recently mobile applications are becoming increasingly prevalent nowadays and a plethora are now developed for several heterogeneous platforms. All these pieces of software need to be tested to assess the quality of their features in terms of functionalities e.g., conformance, security, performance, etc. Manual testing is the most employed approach for testing them, but manual testing is often error-prone and insufficient to achieve high code coverage. These applications share a common feature that can be used for automatic testing: they expose GUIs (Graphical User Interface) for user interaction which can be automatically experimented and explored. Several works already deal with GUI applications testing e.g., desktop applications [1], Web applications [2] or mobile ones [3]. These approaches interact with applications in an attempt to detect bugs and eventually to record models, but all with the same purpose: to obtain good code coverage quickly.

The work, proposed in this paper, falls under this automatic testing category and tackles the testing of mobile applications but also, and above all the learning of models. Our study of model inference techniques has revealed that they often leave aside the notion of correctness of the learned models. This feature is not required for just detecting bug, but is mandatory if models are later used for analysis. Indeed, false models may easily lead to false positives. The quality of the model with regards to its level of abstraction and the amount of information it captures is important as well. Indeed, the more data we collect, the more precise an analysis can be done thereafter. Nevertheless, large amounts of data often lead to large models, up to a state space explosion problem. Based on these observations, we propose an algorithm that aims at learning exact models of mobile applications. We consider the PLTS model (Parameterised Labelled Transition System) to capture the different events made on GUIs. PLTS states also capture all the observed screen contents and notifications about the modifications of the application environment. These notifications signal system events e.g., local database modifications or remote server calls. All this amount of data provide a rich expressiveness that is used while learning the model and that may be later considered for precise model analysis. To avoid a state space explosion, our algorithm dynamically builds state equivalence classes while testing. Each time a new state is discovered, it dynamically re-adjusts the state equivalence relation and classes to limit the state set. These equivalence classes also help recognise similar states that do not require to be explored. Like some available tools [4, 5], our algorithm can also detect application crashes and create test cases for replaying bugs.

We proceed as follows: Section 2 briefly presents some related work before introducing an overview of our algorithm that we apply on a straightforward Android application example in Section 3. We define the model, the state equivalence relation, and we provide the model inference algorithm in Section 4. We give an empirical evaluation on Android applications in Section 5 and conclude in Section 6.

2 Related Work

Several papers dealing with automatic testing and model generation approaches of black-box systems were issued in the last decade. Due to lack of room, we only present some of them relative to our work. Memon et al. [1] initially presented *GUI Ripper*, 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 *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). Mesbah et al. [6] proposed the tool *Crawljax* specialised in Ajax applications. It produces state machine models to capture the changes of DOM structures of the HTML documents by means of events (click, mouseover,etc.). To avoid the state explosion problem, state abstractions must be given manually to extract a model with a manageable size. Furthermore, the concatenation of identical states proposed in [6] is done in our work by minimisation.

Google's *Monkey* [4] is a random testing tool that 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. *Dynodroid* [5] is an extension of Monkey supporting system events. No model is provided. Amalfitano et al. [7] 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. 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 tree 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* [9] is based

on the learning algorithm L^* [10] 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 eventually regenerate the model w.r.t. all the event and observation sequences.

To prevent from a state space explosion, the approaches [1, 6, 8] require state-abstractions given by users and specified in a high level of abstraction. Choi et al. [9] prefer using the approximate learning algorithm $L*$. These choices are particularly suitable for inferring models for comprehension aid, but these models often are over approximations and given in a high level of abstraction, which may lead to many false positives with test case generation. In this paper, we focus on the inference of exact models. As in [6, 7], we consider the notion of state abstraction that we formally define to limit the state space domain to be explored. But, our algorithm dynamically readjusts state equivalence classes to restrain the exploration and constructs a state abstraction according to the content of the application. Secondly, it produces a more readable model. The obtained PLTSs are more detailed since they capture the tested events, all the Widget properties extracted from screens, and the notifications about application environment changes. All these elements helps derive precise test cases.

3 Overview

In the following, we present an overview on our model inference algorithm. Beforehand, we give some assumptions on mobile applications considered to design our approach:

Mobile application testing: we consider black-box applications which can be exercised through screens. It is possible to dynamically inspect application states to collect Widget properties. The set of UI 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. Furthermore, any new screen can be observed and inspected (including application crashes). The application environment modifications (databases, network traffic, etc.) can be observed with probes,

Application reset: we assume that mobile applications and their environments (database, remote servers or mocked servers, Operating Systems) 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 its last action. 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 (wrong implementation, unreachable state, etc.). Most of the other methods assume that the back mechanism always works as expected [7, 5], but this is frequently not the case.

3.1 Terminology

Mobile applications depict screens which represent application states, the number of states being potentially infinite. Screens are built by application components; here we take back the notation used with Android applications, i.e. *Activities*. The later display screens by instantiating Widgets (buttons, text fields, etc.) which are organised into a tree structure. They also declare the available 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 depict several screens, composed of different Widgets or composed of the same Widgets but having different properties.

Figure 1 depicts the screens of an Android application example used throughout the paper. This application converts colour formats from RGB to HSL (hue-saturation-lightness) and vice-versa by means of two radio buttons *r1* and *r2*. When the button *Convert* is pressed, the value entered in the blank text field *txt* is converted and the result appears in the red text field *result*. The chosen colour is also displayed in a colour-box which is depicted at the screen bottom. This application is composed of one Activity which can display an infinite number of screens composed of different text fields values and colour-boxes.

3.2 Algorithm overview

Figure 2 introduces an overview of our algorithm which is composed of two parts. The algorithm is framed on the task-pool paradigm (Figure 2(a)). Tasks are placed into the task-pool, implemented as an ordered list, and each can be executed in parallel. A task $Explore(q, p)$ corresponds to one screen to explore. A screen is transcribed by the state q gathering all the Widget properties composing the screen and p is a path allowing to reach q from the initial state q_0. When there is no more task to do, the exploration implicitly ends. The resulting model is then minimised to be more readable.

The exploration of one state (Figure 2(b)) is done by the Explore procedure. A set of test events (parameter values combined with an event set),

Figure 1 – Colour Converter Android application

which match the current application state, is firstly generated. The current screen is experimented with every test event to produce new screens. However, this step may lead to an infinite set of states to explore. To avoid this well-known issue, the algorithm slices the state space domain into a finite state equivalence class set by means of an equivalence relation (defined in Section 4.1). A state which belongs to a previously discovered equivalence class is marked as final otherwise it has to be explored. Intuitively, for every new built state q_2 (step 2), the algorithm eventually readjusts the state abstraction to limit the state set size (step 3). It scans the detected equivalence classes and checks if some of them (three or more in the algorithm) are different only on account of one Widget property.

If so, it has detected a Widget property which may lead to the construction of several equivalence classes and states to explore (up to an infinite set of states). Consequently, it readjusts the equivalence relation, classes and the model by masking this Widget property. This means that this property is no more taken into account for the equivalence class computation. Therefore, the new state q_2 belongs automatically to an already discovered equivalence class and so, it will not be explored. No new equivalence class is built either. Then, the algorithm checks if new states have to be explored (step 4). Finally, the algorithm tries to backtrack the application to go back to its previous state by undoing the previous action. If it doesn't work, the application and its environment (OS, databases, etc.) are reset and the previous path p is used to reach the state which is currently under exploration.

Figure 3 illustrates with simplified graphs (no PLTSs) how the algorithm

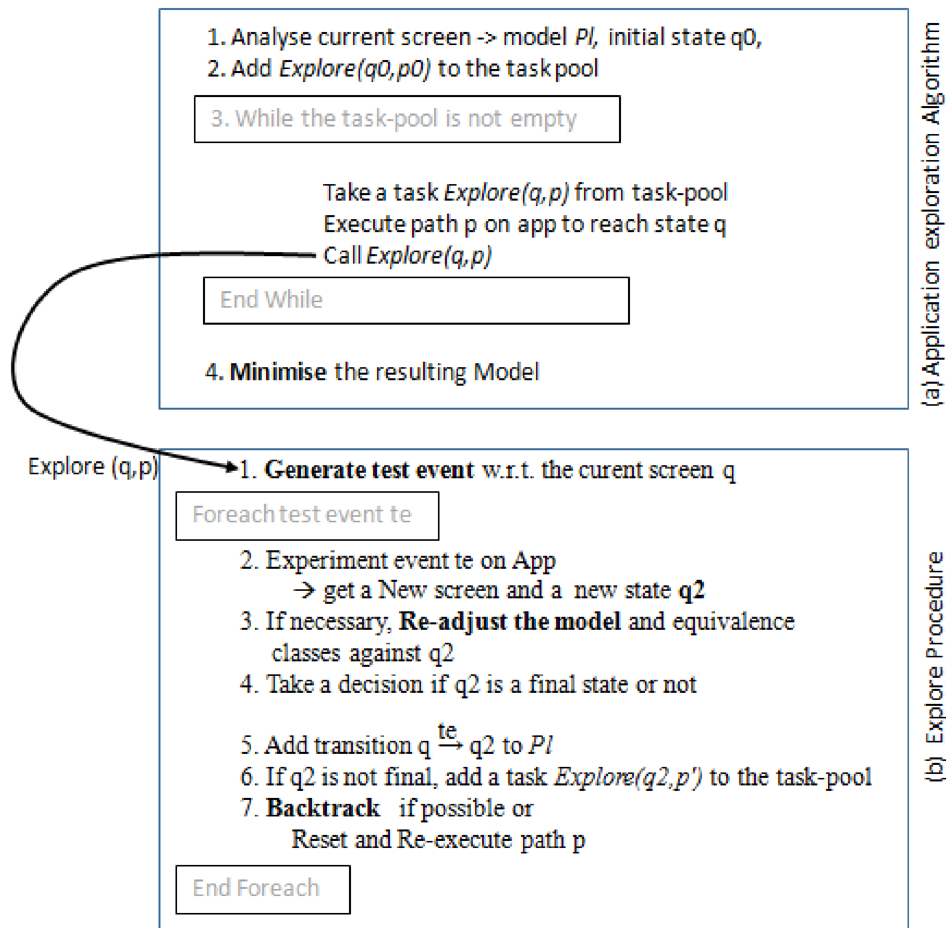

Figure 2 – Overview of the Model inference algorithm

works on the example of Figure 1. For simplicity, only three values are considered for testing: colour red (rgb=255,0,0 or hsv=0,100,50), green and blue. We also assume that these values are always used for testing in the same order. The equivalence relation is: *two states are equivalent if they have the same Widget properties, except those related to text field values.* These last properties are usually not considered for conceiving state abstractions since these often lead to a large potentially infinite set of states. Furthermore, if a Widget property takes more than two values in the different equivalence classes then the relation has to be re-adjusted.

1. Initially, we have a state $q0$ which corresponds to the beginning of the application (Figure 1(a)) and the corresponding equivalence class $[q0]$. A list of test events is generated from $q0$: the events click on

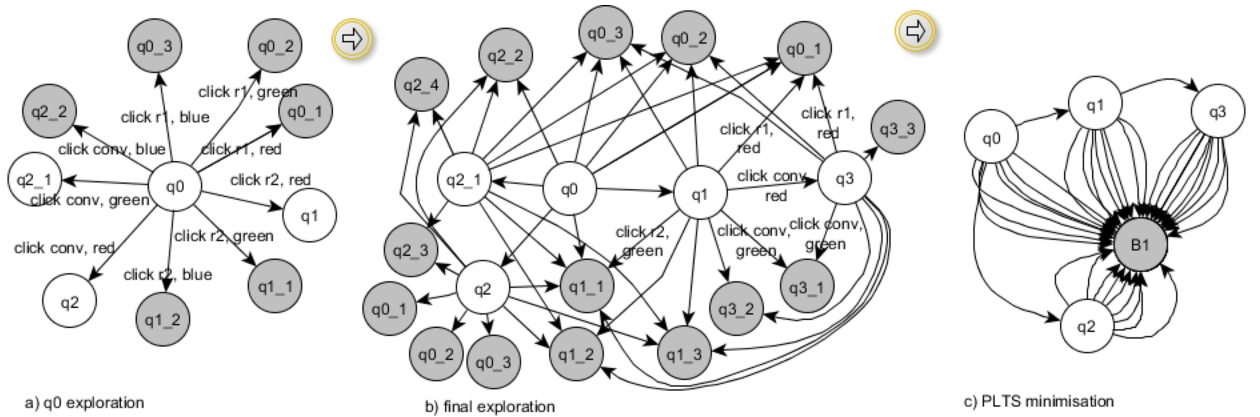

Figure 3 – PLTS generation example

the radio-buttons $r1$, $r2$ or on the button *conv* are combined with the colour values that are injected into the blank text field *txt*. The application is firstly experimented with the click on $r1$ and with the red colour value. This produces a new screen and a state $q0_1$ which belongs to the same equivalence class $[q0]$ because only the text field *txt* is modified. This new state is marked as final (in grey in Figure 3) and is not explored. The application is backtracked to return to $q0$. With the other colours, we also reach final states $q0_2$ and $q0_3$ which are marked as final because they belong to the same equivalence class $[q0]$.

Then, the radio-button $r2$ is clicked, with the red colour. We obtain a new state $q1$ (Figure 1(b)) and a new equivalence class $[q1]$ since $r2$ is now enabled. Therefore, we get a new task $Explore(q1, p')$. Once more, the application is backtracked. When using the other colour values, we obtain the states $q1_1$ and $q1_2$ that are marked as final since these belong to $[q1]$. No task is created.

When the *conv* button is clicked, a value appears in the text field *result* and a colour is depicted in the colour-box (Figure 1(c)). We obtain a state $q2$, which has to be explored, and a new equivalence class $[q2]$. Next, *conv* is clicked with the green colour. The state $q2_1$ is built with a new equivalence class $[q2_1]$. This state is not marked as final since the colour-box displays a new colour. This process should continue for every colour and in particular with the blue one, which produces the state $q2_2$. A state space explosion may happens here. But the algorithm detects that three equivalence classes are different only on account of the same property *colourbox.colour*. The algorithm readjusts the equivalence relation to limit the state

set size. Intuitively, the equivalence relation becomes *two states are equivalent if they have the same Widget properties, except those related to text field values and to the colourbox.colour property.* Then, it updates states and equivalence classes to match this new relation. As a consequence, $[q2]$, $[q2_1]$ and $q2_2]$ are now merged into $[q2]$. The new state $q2_2$ now belongs to an existing equivalence class and is hence marked as final. The first task $Explore(q0, p)$ is finished and we obtain the graph depicted in Figure 3a,

2. we assume that the task $Explore(q1, q0 \xrightarrow{clickr2,txt=red} q1)$ is picked out to explore $q1$. A list of test event, which is the same as previously is constructed. From the state $q1$, when the button *conv* is clicked with the red colour value, a new state $q3$ is added because the colour box appears. When *conv* is clicked with other colour values and events we only obtain final states, since they belong to previously discovered equivalence classes,
3. the same reasoning is followed on states $q2$, $q2_1$ and $q3$, but only final states are added (no task). We obtain the PLTS of Figure 3b,
4. the task-pool is empty. The PLTS is finally minimised [11]. Here, the final states are merged to one unique state as illustrated in Figure 3c.

In this short example, we have shown that our algorithm avoids the state explosion problem and ends once at least one state of all the detected equivalence classes is explored. In the following, we describe formally the model, the equivalence relation, and the algorithm.

4 Model inference Algorithm

4.1 Mobile application modelling with PLTS

We use PLTSs as models for representing mobile applications. A PLTS is a kind of state machine extended with variables and guards on transitions. Beforehand, 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 $\alpha : Y \to D$. We denote D_Y the assignment set over Y.

Definition 1 (PLTS) *A PLTS (Parameterised Labelled Transition System) is a tuple* $< V, I, Q, q0, \Sigma, \to >$ *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 composed of the initial condition D_{V0},*
— *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 $q \xrightarrow{a(\alpha)} q'$.*

The behaviour of a PLTS P is characterised by its sequences of valued actions starting from its initial state $q0$. These sequences are also called the traces of P:

Definition 2 (PLTS Traces) *Let $P =< V, I, Q, q_0, \Sigma, \rightarrow>$ be a PLTS.*
$Traces(P) = Traces(q_0) = \{a_1(\alpha_1)...a_n(\alpha_n) \mid \exists q_1, ...q_n, q0 \xrightarrow{a_1(\alpha_1)} q_1...q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n \in (\rightarrow)^*\}$.

We model mobile application behaviours with PLTSs by encoding (UI) events with actions. We also store the properties collected from screens (Widget properties) and notifications about the application environment changes in states with variable assignments:

UI events representation

we interact with mobile applications by means of events, e.g, a click on a button, and by entering values into editable Widgets. We capture such 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 is applied the event and the remaining variables are assignments of Widget properties. We denote the triggering of the back mechanism with the action $back(\alpha)$ with α an empty assignment.

Application state representation

we specialise PLTS states to store the content of screens (Widget properties) in such a way as to facilitate the construction of equivalence classes of states. We split the set of Widget properties into two categories: we gathers in the set W the Widget properties that indicate a strong application behaviour modification and that take only few values e.g., Widget visibility, size, etc. The others that usually take a lot of different values such as the properties about text field values, are placed into W^c. This separation affects the state representation: we denote wp the assignment composed of properties in W, while the assignment wo is composed of the other Widget properties. A PLTS state q is then a specific assignment of the form $act \cup wp \cup wo \cup env \cup end$ where:

— act is an assignment returning an Activity name,

— (wp, wo) are two Widget property assignments. The union of wp and wo gives all the property values of an application screen displayed by act.

— env is a boolean assignment indicating whether the application environment has been modified,

— end is a boolean assignment marking a state as final or not.

For readability, a state $q = act \cup wp \cup wo \cup env \cup end$ is denoted (act, wp, wo, env, end). This state structure eases the definition of the state equivalence relation given below:

Definition 3 (State equivalence relation) *Let $P =< V, I, Q, q0, \Sigma, \rightarrow>$ be a PLTS and for $i = (1, 2)$ let $q_i = (act_i, wp_i, wo_i, env_i, end_i)$, be two states in Q. We say that q_1 is equivalent to q_2, denoted $q_1 \sim q_2$ iff $act_1 = act_2$, $wp_1 = wp_2$ and $env_1 = env_2$.*
$[q]$ denotes the equivalence class of equivalent states of q. $Q/\sim$ stands for the set of all equivalence classes in Q.

This definition gives a very adaptable state equivalence relation whose meaning can be modified by altering the assignments wp. If we take back our example, one can consider two states $q1 \nsim q2$ which are different only because they do not include the same assignments of the Widget property *colourbox.colour* ($act_1 = act_2$, $env_1 = env_2$, but $wp_1 \neq wp_2$). We have two equivalence classes $[q_1], [q_2]$. The equivalence relation is adaptable in the sense that wp_i can be changed as follows: if we consider that *colourbox.colour* takes too much values and implies to much different equivalence classes, *colourbox.colour* can be shifted from wp_i to $wo_i (i = 1, 2)$ in states. We obtain $wp_1 = wp_2$, $q_1 \sim q_2$ and only one equivalence class $[q_1]$. Intuitively, our algorithm uses this adjustment process to dynamically reduce the equivalence class domain and the state exploration according to the screen content.

The above definition also implies that the equivalence class set is finite. This is captured by the following Proposition:

Proposition 4 *Let $Tree =< V, I, Q, q0, \Sigma, \rightarrow>$ be a PLTS modelling a mobile application App. Let n be the number of the Widget properties of wp. If m is the maximum number of values that any Widget property can take during the testing of the application App then $card(Q/\sim) \leq 2 * 2^n$.*

In short, the proof is based on the fact that at most, we can have two different assignments env ($env := true$, $env := false$) and m^n different assignments of wp. Nonetheless, when a property of wp takes more than two

Algorithm 1: Mobile application exploration

```
input  : Application App
output: PLTS Tree, MTree
  // initialisation performed by one thread
1 Analyse the first screen of App → assignments act, wp, wo;
2 Initialise PLTS P with q0 = (act, wp, wo, env := false, end := false);
3 Q/∼ = {[q0]};
4 Add (Explore(q0, p = ∅)) to the Task-Pool;
  // code performed in parallel, P, Task-Pool, ∼ are shared
5 while the Task-Pool is not empty do
6 |   Take a task (Explore(q, p)) Such that q = (act, wp, wo, env, end) ;
7 |   Reset and Execute App by covering the sequence of actions p;
8 |_  Explore(q, p);
  // code performed by one thread
9 MP:= Minimise(P);
```

values in out algorithm, then, this one is shifted to wo. Consequently, we have at most $2 * 2^n$ equivalence classes. State sets are always finite.

4.2 Model inference Algorithm

Now all ingredients are there to present our algorithms. The first part and the second part of the overview given in Figure 2 are respectively detailed in Algorithms 1 and 2. Algorithm 1 initialises the mobile application App and some variables, the PLTS P, the state equivalence class set $Q/\sim$ and the first state q_0. Then, it implements the main loop composed of the taking of a task ($Explore(q, p)$) and its execution. At the end of the exploration, a second PLTS MP is computed with a minimisation technique.

The pseudo code of the Explore procedure (second part of the overview in Figure 2) is given in Algorithm 2. As stated above, this procedure aims at visiting one state to augment the PLTS under construction, denoted P, with new transitions and states and to eventually produce new tasks $Explore(q, p)$ added to the task-pool. Its steps are explained below:

— Test data generation and execution (lines 4-11): the current screen is analysed to generate a set of events expressing how to complete Widgets with values and to trigger an event. In short, our algorithm generates a set of events of the form $\{event(\alpha) \mid event$ is an event, α is an assignment$\}$. It starts collecting the events that may be applied on the different Widgets of the current screen. Then, it constructs assignments of the form $w_1 = v1 \wedge ... \wedge w_n = vn$, with $(w_1, ..., w_n)$ the list of editable Widget properties found on the screen and $(v_1, ..., v_n)$,

a list of test values. Instead of only using random values, we propose to use several data sets: a set $User$ gathering values manually chosen such as logins and passwords, a set RV composed of values well known for detecting bugs e.g., String values like "&", "", or null, and of random values. A last set, denoted $Fakedata$, is composed of fake user identities. Furthermore, we adopted a Pairwise technique [12] to derive a set of assignment tuples over these data sets. 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. Then, each $event(\alpha)$ is applied on the current screen to produce new ones (application crash included). Each screen is analysed to retrieve Widget properties and the activity which produces this screen. Probes are requested to detect if the application environment were modified. These data are formalised by the state q_2,

— Model readjustment: the Explore procedure now checks whether the re-adjustment of P and of the state equivalence classes is required (lines 9-12). We denote $C^{Wprop}(Q/\sim)$ the number of assignments of the same Widget property $Wprop$ found in the set of equivalence classes $Q/\sim$. $C^{Wprop}(Q/\sim m) = card(\{\alpha = (Wprop := val) \mid [q] \in Q/\sim, q = (act, wp, wo, env, end), \alpha \in wp\})$. For each assignment $\alpha = (Wprop := val)$ in wp_2, we check how much values the Widget property $Wprop$ takes in the equivalence classes: if $Wprop$ takes more than 2 values in $Q/\sim$ (if $card(C^{Wprop)}(Q/\sim) > 2)$, then we re-adjust the state representation. In every state $q = (act, wp, wo, env, end)$ of $Q \cup \{q_2\}$, the assignments of the form $(Wprop := val)$ are shifted from wp to wo (procedure Readjust in Algorithm 2 line 11). The equivalence classes are also transformed in accordance (procedure Readjust line 12),

— PLTS completion: a new transition $q \xrightarrow{event(\alpha)} q_2$ is added to the PLTS P (lines 13-20). q_2 is marked as final if q_2 belongs to an existing equivalence class. Otherwise (line 17), q_2 has the assignment $(end := false)$ and a new task $Explore(q_2, p')$ is added to the task pool. Since the algorithm is highly parallelisable, we use critical sections to modify the PLTS P (which is shared among threads),

— Application backtracking: to apply the next event, the Explore procedure calls the Backtrack one (line 21) to reach the previous screen and state q. Its algorithm is given in Algorithm 3. Here the notion of application environment really makes a difference to achieve an exact model: if the current state q_2 has an assignment $(env := false)$, its

Algorithm 2: Explore Procedure

```
1  Procedure Explore(q, p);
2  Events = GenEvents, analyse the current screen to generate the set of events
3  {event(α) | event is an event, α is an assignment};
4  foreach event(α) ∈ Events do
5      Experiment event(α) on App → new screen Inew;
6      Analyse Inew → assignments act_2, wp_2, wo_2;
7      Analyse the application environment → env_2;
8      q_2 = (act_2, wp_2, wo_2, env_2, end := null);
9      foreach α = {Wprop := val} ∈ wp_2 do
10         if card(C^{Wprop}(Q/∼) ∪ {α}) > 2 then
11             Readjust(Q ∪ {q_2}, Wprop);
12             Readjust(Q/∼, Wprop);
13     if Inew reflects a crash or there exists [q'] ∈ Q/∼ such that q_2 ∈ [q'] then
14         {Add a transition q --event(α)--> q_2 = (act_2, wp_2, wo_2, env_2, end := true) to
           →_P;
15         } (in critical section)
16     else
17         {Add a transition t = q --event(α)--> q_2 = (act_2, wp_2, wo_2, env_2, end := false)
           to →_P;
18         Q/∼ = Q/∼ ∪ {[q_2]};
19         Add the task (Explore(q_2, p.t)) to the task-pool;
20         } (in critical section)
21     Backtrack(q_2, q, p);
```

reflects the fact that the application environment has not be modified, therefore the Backtrack procedure calls the back mechanism to undo the most recent action (if available). We observe a new screen and check whether it is equivalent to the previous screen stored in q (we compare their Widget properties). Otherwise, the application and its environment are reset and we re-execute the path p to reach the state q (Algorithm 3, line 7) (here, we assume that the application is deterministic though).

4.3 PLTS minimisation

Our algorithm performs a minimisation on the first generated PLTS to achieve a more readable model. We have chosen a bisimulation minimisation technique since this one still preserves the functional behaviours represented in the original model while reducing the state space domain. A detailed algorithm can be found in [11]. In short, this algorithm constructs sets

Algorithm 3: Backtrack procedure Algorithm

```
1  Procedure
   Backtrack(q2 = (act2, wp2, wo2, env2, end2), q = (act, wp, wo, env, end), p);
2  if env2 = (env := false) and the back mechanism is available then
3  |  Call the back mechanism → screen INew;
4  |  Analyse Inew → assignments rc', wp', wo';
5  |  Analyse the application environment → env';
6  |  if act ≠ act' or wp ≠ wp' or wo ≠ wo' or env ≠ env' then
7  |  |  Reset and Execute App by covering the actions of p;
8  |  else
9  |  |  Add a transition t = q2 --back(α)--> q to →Tree;
10 else
11 |  Reset and Execute App by covering the actions of p;
```

(blocks) of states that are bisimilar equivalent (any action from one of them can be matched by the same action from the other and the arrival states are again bisimilar). Figure 3c depicts the (simplified) minimised PLTS of the application example. Here, final states are aggregated into one block of states.

4.4 Algorithm correctness, complexity and termination

We express the correctness of our model inference method in term of trace equivalence between the inferred PLTS and the traces of the application under test:

Proposition 5 *Let P be a PLTS constructed with our model inference algorithm from a deterministic mobile application App. We have $Traces(P) \subseteq Traces(App)$.*

The proof is given in Appendix A. Intuitively, our algorithm constructs a PLTS P with these steps:

1. *Generation of PLTS:* from a given state q, every new event applied on the application is modelled with a unique transition whose arrival state q_2 is new or final. We do not merge states and hence we construct a PLTS P,
2. *Correct use of the back mechanism:* we call this mechanism with care: it is called only if the environment of the application (databases, remote servers, etc.) were not modified with the execution of the last

action. Indeed, if we apply the back mechanism even so, we necessarily reach a new state since the application environment is modified. Secondly, we check if the state of the application obtained after the call of the back mechanism is really the previous state of the application. If one of these conditions is not met, we reset the application and its environment and we re-execute the path p to reach the state q,

3. *Minimisation with trace equivalence:* we apply a bisimulation minimisation technique to produce a PTLS MP from P such that the two PLTS are bisimilar and consequently trace equivalent as well.

Complexity and termination of the Algorithm: our algorithm builds at most $2 * 2^n$ equivalence classes, with n the number of Widget properties in W. In short, we can have two different ($env := true$, $env := false$) and m^n different assignments over W if m is the maximum number of values that any Widget property can take. Nonetheless, when a property of W takes more than two values, our algorithm shifts it from the assignment wp to wo in states. Furthermore, since we explore one state per equivalence class, the algorithm ends and we have $2*2^n$ equivalence classes and not final states. We also have at most nm transitions (Pairwise testing [12]) for each. If N and M stand for the number of not final states and transitions, the whole algorithm has a complexity proportional to $\mathcal{O}(M + N + MN + Mlog(N))$. Indeed, the Explore procedure covers every transition twice (one time to execute the event and one time to go back to the previous state) and every not final state is processed once. But, sometimes the back mechanism is not available. In this situation, the application is reset to go back to a state q by executing the events of a path p at worst composed of M transitions. In the worst case, this step is done for every state with a complexity proportional to NM. Furthermore, the minimisation procedure has a complexity proportional to $\mathcal{O}(Mlog(N))$ [11].

5 Empirical Evaluation

We present here some experimentations on Android applications to answer on the following questions: does the algorithm offer good code coverage in a reasonable time delay? How are the models in terms of size and quality for analysis?

We have implemented our algorithm in a tool called *MCrawlT* (Mobile Crawler Tool[4]). It takes packaged applications or source projects and

4. available here https://github.com/statops/mcrawlert.git

Applications	Monkey	Orbit	Guitar	MCrawlT	Swift Hand	Dyno droid
NotePad	60	82		88		crash
Tippy_TipperV1	41	78		79		48
ToDoManager	71	75	71	81		34
OpenManager	29	63		65		crash
HelloAUT	71	86	51	96		76
TomDroid	46	70		76		42
ContactManager	53	91	71	68		28
Aardict	52	65		67		51
Musicnote	69			81	72.2	47
Explorer	58			74	74	crash
Myexpense	25			61	41.8	40
Anynemo	61			54	52.9	crash
Whohas	58			95	59.3	65
Mininote	42			26	34	39
Weight	51			34	62	56
Tippy_TipperV2	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

Table 1 – Code coverage (in %)

user data e.g., logins and passwords required for the application execution. *MCrawlT* is based on the testing framework *Robotium*[5] which retrieves the Widget properties of a screen and simulates events. We chose to compare the effectiveness of *MCrawlT* with several tools: *Monkey* [4], *Orbit* [8], *Guitar* [1] with an extension for mobile applications, *Swifthand* [9] and *Dynodroid* [5]. Nonetheless, we faced many difficulties to launch or configure *Guitar* for mobile applications and *Swifthand* (no documentation showing how to instrument source code). *Orbit* is not available. In this context and to avoid any bias, we chose to apply our tool, *Monkey* and *Dynodroid* on applications taken as reference in these papers and whose source code is available (30 applications). The results for the other tools are taken from [8, 1, 9].

Code coverage and execution time: Table 1 reports the percentages of code coverage obtained with the different tools on 30 applications with a time budget of three hours. Table 2 only exhibits the applications and the

5. https://code.google.com/p/robotium/

Applications	Orbit	Guitar	MCrawlT	SwiftHand
NotePad	102		175	
Tippy TipperV1	198		110	
ToDoManager	121	194	210	
OpenManager	480		489	
HelloAUT	156	117	106	
TomDroid	340		196	
Contact Manager	125	194	135	
Aardict	124		580	
Aagtl			920	
Bomber			3100	
Anycut			8037	
Mirrored			4090	
Nectdroid			8120	
Whohas			5020	10800
Mininote			8230	10800
Tippy TipperV2			1556	10800

Table 2 – Execution time (in seconds)

tools requiring less than three hours. If we do a side by side comparison of *MCrawlT* with the other tools, we observe that *Monkey* provides better code coverage for 8 applications, *SwiftHand* for 2 and *Dynodroid* for 5. In comparison to all the tools together, *MCrawlT* provides better code coverage for 20 applications, the coverage difference being higher than 5% with 13 applications. These results show that *MCrawlT* gives better code coverage than the other tools and even offers good results against all the tools together on half the applications with comparable execution times.

Table 1 also reveals that the obtained code coverage percentage is between 25% and 96%. We manually analysed the 8 applications which yield the less good results with *MCrawlT* to identify the underlying causes behind low coverage. This can be explained at least by these ways:

- Specific functionalities and unreachable code: several applications are 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,
- 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 *Android broadcast messages*. Our tool does not support these events yet. Moreover, *MCrawlT* only supports the event list also supported by the testing tool *Robotium* (viz. click and scroll). The long click is thus not supported but is used in some applications (Mininote and Contact-manager). In contrast, *Orbit* supports this event and therefore offers

Applications	MCrawlT	Monkey	Dynodroid
WordPress	63	3	
Notepad	5		
TomDroid	7	1	
Mirror	25	3	
Mininote	2		
Aagtl	1		2
PasswordForAndroid	1		1
Sanity	5	1	1
Aardict	2		

Table 3 – Application crash detection

a better code coverage with the application Contactmanager.

Crash detection: *MCrawlT*, *Monkey* and *Dynodroid* also detect application crashes. Table 3 reports the 9 applications which crashed while testing with one of the tools. The results of Table 3 correspond to genuine bugs only. We manually ascertained test reports to eventually remove false positives such as emulator misbehaving. We kept only the Exceptions which cause the termination of the applications such as *NullPointerException*. On the 30 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, which is not surprising since *MCrawlT* does stress testing like *Monkey* and *Dynodroid* but it also uses values known for relieving bugs.

Quality and size of the models: Table 4 finally shows the number of states obtained with *MCrawlT*, *Orbit* [8] and *SwitHand* [9] since they produce models as well. Before minimisation, our tool generates larger and tacitly less comprehensive models than those obtained with *Orbit*. In term of quality of the learned models, we do not produce extrapolated models and we believe that those generated by *MCrawlT* offer more testing capabilities. Indeed, these models include states which store all the observed Widget properties (colours, texts, etc.) and notifications about the application environment changes. We have precisely chosen this feature to later perform test case generation. For instance, with this amount of information, we can construct test cases to apply events and to check the content of the resulting screen but also if remote servers are called, etc. Both *Orbit* and *SwiftHand* only store UI events. After minimisation, we obtain more compact and readable models whose sizes are comparable to the sizes of the models obtained with *Orbit*. This tends to show that our approach of producing larger but more detailed models that are after minimised, only offer advantages for model inference. In addition, *MCrawlT* constructs storyboards from these minimized models by replacing states with screen-shots of the application.

All these experimental results on real applications tend to show that our

Applications	#PLTS states (MCrawlT)	#states after minimisation (MCrawlT)	#states (Orbit)	#states (Swift-Hand)
NotePad	13	8	7	
Tippy_TipperV1	37	18	9	
ToDoManager	6	2		
OpenManager	31	12	20	
HelloAUT	8	5	8	
TomDroid	12	6	9	
ContactManager	5	4	5	
Sanity	31	24		78
Musicnote	41	23		46
Explorer	96	74		195
Myexpense	52	37		149
Anynemo	139	106		169
Whohas	36	11		97
Mininote	45	19		169
Tippy_TipperV2	54	26		71
Weight	69	23		109

Table 4 – Inferred model size

tool is effective and can be used in practice since it produces equivalent or higher code coverages than the other tools, it detects crashes and produces detailed and compact models.

6 Conclusion

In this paper, we present an algorithm which infers PLTS models from mobile applications. It constructs PLTSs that capture events and all the Widget properties extracted from the observed screens. Despite the huge amount of collected data, we avoid the state space explosion problem by using an equivalence relation and classes that are dynamically re-adjusted all along the algorithm execution with regards to the screen content. Our experimental results show that our algorithm offers good code coverage quickly and can be used in practice. Furthermore, the generated models can be reused for precise model analysis. An immediate line of future work would be to apply this kind of algorithm for security breach detection. The exploration could be specialised to target some specific application parts (login step, etc.). Then, test cases could be automatically generated from test patterns to further explore specific states with the purpose of improving detection.

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–.

[2] 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.

[3] 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.

[4] Google, "Ui/application exerciser monkey," http://developer.android .com/-tools/help/monkey.html, last accessed jan 2015.

[5] 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.

[6] 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.

[7] 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.

[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.

[9] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *SIGPLAN Not.*, vol. 48, no. 10, pp. 623–640, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544173.2509552

[10] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

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

[12] 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.

A Proof of Proposition A

Proposition 6 *Let $Tree$ be a PLTS constructed with Algorithm 1 from a deterministic GUI application App. We have $Traces(Tree) \subseteq Traces(App)$.*

We shall consider two steps for proving this proposition. Initially, we remove the back mechanism of the Procedure Backtrack (lines 2-8), the Application is therefore always reset before triggering an event on one of its screens. With two first Lemma, we prove by induction that $Traces(Tree) \subseteq Traces(App)$. Intuitively, we construct a PLTS tree such that each branch is unique and reflect a behaviour of *App*. Then, we add the back mechanism, and show the trace inclusion with two other Lemmas.

Part I (without the back mechanism).

Lemma 1 $\forall p_1 = q_0 \xrightarrow{a_1(\alpha_1)} q_1 \in \rightarrow_{Tree}$*, we have* $Traces(p_1) \subseteq Traces(App)$*.*

Proof We have $Traces(p_1) = a_1(\alpha_1)$.
q_0 is the initial state representing the initial screen of *App*. q_0 is explored with the task $Explore(q_0, p_0)$ after resetting *App* ((Algorithm 1 line 8 or Algorithm 3 line 9).
$a_1(\alpha_1)$ is an event applied of the initial screen of *App* (line 6) w.r.t. the Widget properties found in q_0. $q_0 \xrightarrow{a_1(\alpha_1)} q_1$ is a transition added to $\rightarrow_{Tree}$ with q_1 either a final a state or a new state (Algorithm 2, lines 6-19).
Since *App* has been reset before, $a_1(\alpha_1)$ is a trace of *App*.
Consequently, $Traces(p_1) \subseteq Traces(App)$. ■

Lemma 2 $\forall p_n = p_{n-1}.q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n \in \rightarrow_{Tree}$*, such that* $Traces(p_{n-1}) \subseteq Traces(App)$*, we have* $Traces(p_n) \subseteq Traces(App)$*.*

Proof We assume that $p_n \in rightarrow_{Tree}$ and $Traces(p_{n-1}) \subseteq Traces(App)$.
q_{n-1} is explored in Algorithm 1 with the task $Explore(q_{n-1}, p)$. *App* is reset and p_{n-1} is executed on *App* (Algorithm 1 line 8 or Algorithm 3 line 9). The screen modelled with q_{n-1} is reached since *App* is deterministic).
The event $a_n(\alpha_n)$ is executed w.r.t. the Widget properties found in q_{n-1} (Algorithm 2 lines 4-6). $q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n$ is the unique transition modelling the execution of $a_n(\alpha_n)$ on *App*.
$a_1(\alpha_1)...a_{n-1}(\alpha_{n-1}).a_n(\alpha_n) = Traces(p_n) \in Traces(App)$. ■

Without considering the back mechanism (Algorithm 3 lines 2-7), and by applying the same reasoning with $n > 1$ and with Lemma 1, we can say that $\forall p_n \in \rightarrow Tree(n \geq 0)$, $Traces(p_n) \subseteq Traces(App)$, and $Traces(Tree) \subseteq Traces(App)$.

Part II (with the back mechanism, complete algorithm).

Lemma 3 *Let* $p'_1 = q_0 \xrightarrow{a'_1(\alpha'_1)} q'_1 \in \rightarrow_{Tree}$*, with* $q'_1 = (rc'_1, wp'_1, wo'_1, (env := false), end'_1, ph'_1)$ *and* $a'_1 \neq back$*.* $\forall p_1 = p'_1.q'_1 \xrightarrow{back(\alpha)} q'_0 \xrightarrow{a_1(\alpha_1)} q_1 \in \rightarrow_{Tree}$*, we have* $Traces(p_1) \subseteq Traces(App)$ *and* $Traces(q'_0 \xrightarrow{a_1(\alpha_1)} q_1) \subseteq Traces(App)$*.*

Proof

We assume having $p'_1 = q_0 \xrightarrow{a'_1(\alpha'_1)} q'_1 \in \rightarrow_{Tree}$, with $q'_1 = (rc'_1, wp'_1, wo'_1, (env := false), end'_1, ph'_1)$ and $a'_1 \neq back$.
$Traces(p'_1) \subseteq Traces(App)$ (Lemma 1)
.

Let $p_1 = p'_1.q'_1 \xrightarrow{back(\alpha)} q'_0 \xrightarrow{a_1(\alpha_1)} q_1$. We have $Traces(p'_1.q'_1 \xrightarrow{back(\alpha)} q'_0) \subseteq Traces(App)$ and $q'_0 = q_0$.
The back mechanism is applied on an application state from which no application environment change has been observed. We assume here that this environment has not been modified. Consequently, $q'_0 = q_0$. This assignment is checked in the Procedure 3 (lines 6-7). Hence, if our assumption is false, the Algorithm resets the application, like in Lemma 1.

The back mechanism is only called from the Backtrack procedure to test an application screen with a new event (Algorithm 2 line 5). Let $a_1(\alpha_1)$ be the corresponding action. $a_1(\alpha_1)$ is applied w.r.t. the screen of *App* and a unique transition $q_0 \xrightarrow{a_1(\alpha_1)} q_1$ is added to $\rightarrow_{Tree}$ with q_1 either a final a state or a new state (Algorithm 2, lines 6-19).
$Traces(p'_1.q'_1 \xrightarrow{back(\alpha)} q'_0 \xrightarrow{a_1(\alpha_1)} q_1 = Trace(p'_1).back(\alpha).a_1(\alpha_1) \subseteq Traces(App)$ since $Trace(p'_1) \subseteq Traces(App)$.
With the back mechanism, *App* goes back to its previous state $q'_0 = q_0$. $Traces(q_0 \xrightarrow{a_1(\alpha_1)} q_1 = a_1(\alpha_1) \subseteq Traces(App)$ (Lemma 1). ∎

Lemma 4 *Let* $p'_n = q_0 \xrightarrow{a_1(\alpha_1)} q_1...q_{n-1} \xrightarrow{a'_n(\alpha'_n)} q'_n \in \rightarrow_{Tree}$*, with* $a_i(\alpha_i)(0 \leq i \leq n-1) \neq back(\alpha)$*,* $a'_n(\alpha'_n) \neq back(\alpha)$*,* $q'_n = (rc'_n, wp'_n, wo'_n, (env := false), end'_n, ph'_n)$*.*

$\forall p_n = p'_n.q'_n \xrightarrow{back(\alpha)} q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n \in \rightarrow_{Tree}$*, we have* $Traces(p_n) \subseteq Traces(App)$ *and* $Traces(q_0 \xrightarrow{a_1(\alpha_1)} q_1...q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n) \subseteq Traces(App)$*.*

Proof

We assume having $p'_n = q_0 \xrightarrow{a_1(\alpha_1)} q_1...q_{n-1} \xrightarrow{a'_n(\alpha'_n)} q'_n \in \rightarrow_{Tree}$, with $a_i(\alpha_i)(0 \leq i \leq n-1) \neq back(\alpha)$, $a'_n(\alpha'_n) \neq back(\alpha)$, $q'_n = (rc'_n, wp'_n, wo'_n, (env :=$

$false), end'_n, ph'_n)$.
$Traces(p'_n) \subseteq Traces(App)$ (Lemma 2)
.

Let $p_n = p'_n.q'_n \xrightarrow{back(\alpha)} q_{n-1'} \xrightarrow{a_n(\alpha_n)} q_n$, we have $Traces(p'_n.q'_n \xrightarrow{back(\alpha)} q'_{n-1}) \subseteq Traces(App)$ and $q'_{n-1} = q_{n-1}$.
As previously, the back mechanism is applied on an application state from which no application environment change has been observed. We assume here that this environment has not been modified. Consequently, $q'_{n-1} = q_{n-1}$.

The back mechanism is only called from the Backtrack procedure to test an application screen with a new event (Algorithm 2 line 5) modelled with $a_n(\alpha_n)$. $a_n(\alpha_n)$ is applied w.r.t. the screen of *App* and a unique transition $q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n$ is added to $\rightarrow_{Tree}$ with q_n either a final a state or a new and unique state.
$Traces(p'_n.q'_n \xrightarrow{back(\alpha)} q'_{n-1} \xrightarrow{a_n(\alpha_n)} q_n = Trace(p'_n).back(\alpha).a_n(\alpha_n) \subseteq Traces(App)$ since $Trace(p'_n) \subseteq Traces(App)$.

When the back mechanism is triggered, *App* goes back to its previous state $q'_{n-1} = q_{n-1}$ and we also have $Traces(p''_{n-1} = q_0 \xrightarrow{a_1(\alpha_1)} q_1 ... q_{n-1}) \subseteq Traces(App)$ (Lemmas 1 and 2).
$p''_n = p''_{n-1}.q_{n-1} \xrightarrow{a_n(\alpha_n} q_n \in \rightarrow_{Tree}$. According to Lemma 2, $Traces(p''_n = q_0 \xrightarrow{a_1(\alpha_1)} q_1 ... q_{n-1} \xrightarrow{a_n(\alpha_n)} q_n) \subseteq Traces(App)$. ■

By induction on Lemmas 3 and 4, we obtain $\forall p_n \in \rightarrow Tree (n \geq 0)$, $Traces(p_n) \subseteq Traces(App)$, and $Traces(Tree) \subseteq Traces(App)$. Consequently, with the four lemmas, Proposition holds.