Paper Title: A formal approach to automatic analysis of extra-functional properties in mobile apps

Authors: Ana-Rosario Espada, María-del-Mar Gallardo, Alberto Salmerón, Pedro[1]

1. University of Malaga, Spain, {anarosario, gallardo, salmeron, pedro}@lcc.uma.es

Presented at: XXIV Jornadas de Concurrencia y Sistemas Distribuídos (JCSD), 15-17 June 2016, Granada, Spain
Published in: Actas de las XXIV Jornadas de Concurrencia y Sistemas Distribuídos (JCSD)
DOI (link to publication from Publisher):
Publication date: 2016
Document Version Accepted author manuscript, peer reviewed version.

A formal approach to automatic analysis of extra-functional properties in mobile apps*

Ana-Rosario Espada, María-del-Mar Gallardo,
Alberto Salmerón, and Pedro Merino

Dept. Lenguajes y Ciencias de la Computación
E.T.S.I. Informática University of Málaga
[anarosario,gallardo,salmeron,pedro]@lcc.uma.es

Abstract. This paper presents an integrated approach to runtime verification of extra functional properties for mobile applications. The tool chain starts with a formal model of the behaviors of the apps with respect to the user, so all potential actions from the user are represented. Then, test cases are exhaustively generated using model checking. At the last step each test case is used to control the execution the application in parallel with a runtime verification engine that checks the satisfaction of the extra functional properties. The whole approach has been extended to several applications and several devices. The paper presents the definition and formalization of both the modelling language for the applications and the specification language to represent the extra-functional properties as well as a first implementation focussed to Android devices in order to check properties related to network traffic or energy consumption. The work is part of the European Project Triangle devoted to 5G Applications and Devices Benchmarking.

1 Introduction

Automated analysis of the behavior of applications running in smartphones is an emerging trend due to the increasing role of such platforms as the main way for users to connect to the Internet. Execution errors or under-performance in mobile apps have a great impact over the user experience, over the overall behavior of the smartphone and over the mobile communication network. This potential negative impact is not negligible, if we think on more than 2 billions of devices running connected apps every day. In addition to the functional properties of the software, the analysis of extra functional properties (EFPs) like energy consumption, response time, traffic generation, memory use, etc. is a central issue in the new techniques to verify mobile apps.

The current application of formal methods, like some variant of static analysis or model checking, to predict the behavior of mobile apps regarding EFPs is not effective enough due to difficulties to construct a realistic model of the whole

* Work supported by the Andalusian Excellence Project P11-TIC-7659, and by the H2020 European Project TRIANGLE ref. 688712

Actas de las XXIV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2016), pp. 225-239.
15-17 junio 2016, Granada, España. ISBN: 978-84-16478-90-3

environment where the application is running: the user interaction, the rest of apps running in the same device, the operating system, the interaction with the mobile networks, etc. Approaches working with models, like App Explorer [1] or PerfChecker [2], still need more accurate information on delays or energy consumption that can be only obtained from real executions. In the opposite site, approaches based only on runtime monitoring to characterize the apps behavior lack the mechanics to generate all realistic scenarios and/or to formally ensure the correctness or coverage of the analysis. For instance, tools like AntMonitor, NetworkProfiler, ProfileDroid or Automatic Android App Explorer support methods to explore several executions in order to produce statistics, to identify reference patterns in the traffic or to locate potential suspicious behaviors, but they lack a proper formal framework to control the coverage of the executions and to describe the EFPs to be analyzed.

In this paper, we present a new model-checking based approach to verify whether a single running mobile app or a combination of apps meet a given set of EFPs. In case of violation of EPFs, it is possible to locate the execution traces of the app (or interaction of apps) that violate such EFPs. The main goal is to design a method that can be fully automated over real hardware composed by smartphones and monitoring equipment. The generation of the app executions is driven by the realistic interactions of the user with the apps, while the verification is performed by a runtime monitor that compares events and states in the traces wrt formal descriptions of the EFPs. We have designed and implemented the engine to synchronize the observations in the device with the expected reference measurements included in the formalization of the EFPs.

The rest of the paper is organized as follows. Section 2 presents the work flow and architecture for our framework. Sections 3 and 4 describe the modeling and specification languages. In Section 5, we outline the implementation of the proposal. Finally, in Section 6, we give the conclusions.

2 Overview

The main goal of our proposal is to help developers/testers to analyze the correct behaviour of mobile applications with respect to some extra functional properties of interest. To this end, we have built a verification tool which combines Model Based Testing [3], Model Checking [4], and Runtime Verification [5] techniques. The architecture of our tools are shown on Figure 1, divided in two levels. The user level represents the user of the tool, which initiates the testing process and later expects its results, while the architecture level shows the actual components of the tool. The *Mobile Verification Engine* component is the core of the tool, while the components below it correspond to external tools and services used by the engine.

2.1 User level

From the user point of view, the tool is used is as follows. First, the user must provide the so-called *user behaviour model* (UBM) using the modeling language

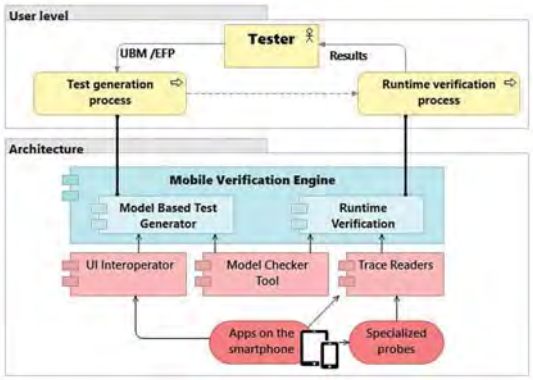


Fig. 1. Design of the Mobile Verification Engine Architecture

discussed in Section 3. This model should describe the way the application is commonly handled by the final users, and will be used to generate a set of test cases. This model-based approach is focused on generating realistic test cases, in contrast with other tools which produce random test cases. The sequences generated from the user behavior model will represent typical application usage scenarios, over which relevant properties can be analyzed. However, it can also be used to model malicious or incorrect uses, and check the reliability of the application in those cases.

The tester should also provide the properties to be checked on the execution traces generated by running the tests cases. In particular, we are interested in the analysis of extra-functional properties, such as energy consumption, described using the specification language presented in Section 4.

Given these two user inputs, the Mobile Verification Engine returns whether the generated test cases satisfy the given properties or not. In the latter, the tool also provides the trace that lead to the property violation, which can be used to aid in the debugging of the application. One of the main benefits of the approach is that, due to the modularity of the modeling language, the tester can build upon the initial user behavior model, adding new user behaviors incrementally.

2.2 Architecture

The hight-level architecture of the tool itself is as follows. As shown in Figure 1, the Mobile Verification Engine is composed of two modules. The first component, the *Model Based Test Generator*, is in charge of generating the test cases. The second one, the *Runtime Verification* component, monitors the execution of the tests cases, and extracts the execution traces for the analysis of the extra-functional properties.

The Model Based Test Generator uses a model checker tool to generate the test cases. This model is indeterministic in nature, e.g. the user may interact

with certain elements or navigate through the application in different order. Therefore, the model checker must explore all the possible paths allowed by the model in order to generate the set of realistic test cases. Each test case is a sequence of user events that can be remotely executed in the mobile device using the UI Interoperator.

The Model Based Test Generator and Runtime Verification modules run simultaneously. While the former executes the test cases on the mobile device, the latter collects runtime information from the device and other sources to build the so-called *enriched trace*. This trace combines and sorts static and dynamic information from several sources, in order to enable the verification tasks.

The extra-functional properties to be analyzed are translated into Linear Temporal Logic formulas, in order to be understood by the model checker that performs the analysis over the enriched trace. The enriched trace is first filtered to provide only information that is relevant to the property of interest [6]. Finally, the model checker provides a verdict for each of the analyzed traces, and these results are given back to the user.

3 Modeling expected user behaviour

Applications are defined by their behavior. However, this behavior is mainly triggered through user interactions with the application's interface. Thus, we model applications from the user's perspective, describing actions and sequences that make sense to them. This model will be used to extract realistic test cases, i.e. sequences of user actions that correspond to typical user behaviors. In this section, we describe the concepts we want to model from mobile applications, present our modeling language and its formalization.

3.1 Elements of mobile applications

Users interact with a mobile application mainly through graphical elements called controls, e.g. buttons, text fields, and lists. In a touch-based interface these controls can be used in several ways, from simple gestures such as tapping, to more complex ones such as pinch-to-zoom. Not all controls respond to these gestures, e.g. a button may react to taps, but not to swipes. In addition, some user actions may also depend on events which they do not directly control. For instance, a "play" button may be disabled while a song is being downloaded.

The navigation between screens presents interesting challenges. Many applications organize their screens in a hierarchical way: new screens go deeper in the hierarchy, and the user can also navigate back to the previous one at any time. This "go back" action is usually supported directly by the mobile device, such as ANDROID's system-level "back" button, or the application framework, such as the IOS navigation bar.

We can picture this navigation model as a stack of screens, where the screen at the top of the stack is the one currently being displayed. When the user navigates to a new screen, that screen is placed on top of the stack. When the

user goes back, the top screen is popped off the stack. The "home screen" of the mobile device is always at the bottom of this stack.

In addition, the same screen may be reached through more than one path. When users press back, they expect to see the previous screen, not one of the other possible "previous screens" from other paths. However, in other cases past screens may be removed from the stack at a certain point. For instance, after completing a multi-screen "wizard", users may not be able to return to the wizard by pressing back.

New applications can also be started from others, e.g. an e-mail application may start the web browser application to load a website when a link is tapped. It may be possible that the new application does not start at its "main" screen, but rather at another screen, depending on the request made by the first application.

Not all behaviors in mobile applications can be described only through user actions. Some depend at some point on events that are not directly controlled by the user, such as the reception of an e-mail or an alarm going off. We call these system events, and must be taken into account in our modeling language.

3.2 Modeling language

Our modeling language is based on state machines. Although it is not a strict subset, we take many elements from UML state machines [7] and Harel statecharts [8], including the graphical notation, and introduce additional ones that model concepts from mobile applications.

State machines are composed of states connected through labeled transitions. While the states themselves have no direct relation with the application's interface, the transitions represent user actions over the controls. Each transition is labeled with the action that the user would perform in order to progress to a new state. These actions include pressing a button, entering text in a field or scrolling in a list. Transitions may also be labeled with system events, which represents some event or condition not controlled by the user. This distinction is important for test case generation, in the sense that system events are not translated into actions performed on the screen.

Figure 2 shows a part of the model used in our case studies that contains most of the concepts from our modeling language. This example models the behavior of a user that searches and plays songs from two pre-defined sets: popular and non-popular songs. This example contains five state machines: "PrincipalStateMachine", "SearchPopularStateMachine", etc. Each one of them contains one or more states, and several labeled transitions. In this example, "SWIPE" in "PrincipalStateMachine" means scrolling on a list present in the screen, while "clicConfiguration" is tied to tapping on a button. Notice also the use of initial and final states, and the presence of more than one outgoing transition in a state. "PopularStateMachine" also contains several transitions with timing information, with user actions to press the "Play" and "Pause" buttons. When taking the transition to left of "S0", the user waits for 300 seconds before finishing with that state machine. On the other branch, however, the song is played for 200 seconds, then "Pause" is pressed, then after 10 seconds the song is resumed again.

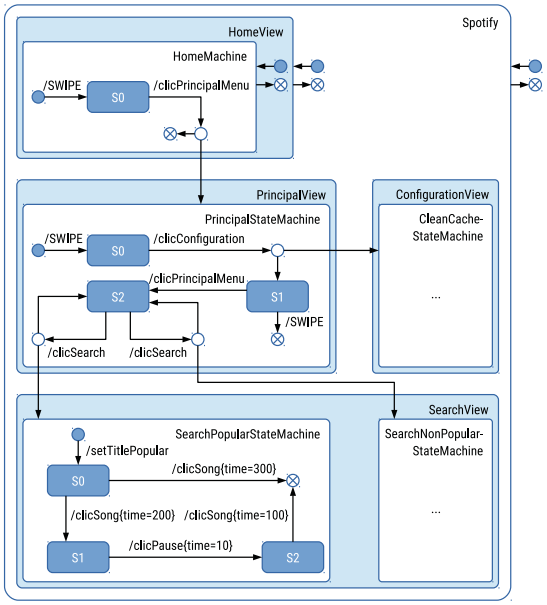


Fig. 2. Part of Spotify user behavior model

By following the available transitions in a state machine, starting from the initial state, we produce a sequence of user actions. This sequence, which we call flow, represents a possible user behavior in a application. If there is more than one available transition from a state, each one will lead to a different flow. Since we are only interested in realistic user behaviors, the state machines should be modeled with this in mind, and not add transitions for the sake of completeness.

Describing almost any application with a single state machine is unpractical. Thus, we promote organization through composition of state machines in two ways. First, a state machine may "call" another state machine, e.g. each one representing the user behavior on a different application screen. Second, state machines are contained in a hierarchy that mimics the elements identified before: devices, applications and screens.

A device can be described through one or more state machines that represents behaviors in the screens from the applications it contains. To provide a more structured visualization, our modeling language allows additional levels to group state machines. Starting from the top, we define the following hierarchy. At the top level we have devices. Each device contains one or more applications. In turn, each application may be composed of one or more *views*, i.e. application screens. Finally, each view may contain one or more state machines. Each state machine within a view may be used to define a different set of user behaviors. This may be useful for composition, as we will see next. This type of hierarchical

composition is purely for convenience, as we could have just devices and state machines.

The example shown in Figure 2 has one application, with four different views. All views have a single state machine except for "SearchPlayView", which has two. Altough they are not shown completely in the figure, these two state machines model slightly different behaviors on the same screen.

A special type of state, called *connection state*, can be used to call another state machine on the same device (but not necessarily within the same application). Connection states have two outgoing unlabeled transitions: one to a state within the same state machine (as usual), and another one to a different state machine. When a connection state is reached, the flow continues with the state machine referenced from the connection state. When the second state machine finishes, the flow continues on the first state machine, taking the transition to another state. "PrincipalStateMachine" in Figure 2 contains three connection states, reachable from state "S0". For instance, after performing the "clicPopular" user action, the flow continues on "PopularStateMachine". When this state machine finishes, the flow returns to the "S0" state in "PrincipalStateMachine", which was the next state from the connection state.

Connection states can also reference a view instead of a state machine. Thus, any *externally accessible* state machine within the view can be executed next. External accessibility is a feature of applications, views and state machines, represented with a pair of initial and final states pointing at them. This resembles the representation used in state machines, although they are not full-featured state machines. The execution of the whole model start with a state machine that can be reached through externally accessible elements, i.e. applications and views. This organization allows any state machine to be called explicitly from another one, while not being available from the start of the flow. The example in Figure 2 contains only one externally accessible state machine: "PrincipalStateMachine". The others can only be accessed through connection states.

Due to space restrictions, we are not able to include the formalization of the modelling approach described above [9]. Nevertheless, the key point to understand how the test cases are generated is that the view and device machines naturally behave as transition systems producing sequences of states/events which can be interpreted as test cases. The role of the model checker tool is to explore all the possible paths determined by the machines with the aim of being as exhaustive as possible. To force the termination of the exploration, we limit the length of the test cases up to a fixed value.

4 Specification and analysis of extra-functional properties

Verification techniques, such as model checking, evaluate properties over traces abstracting the real time when each state occurs. This abstraction is adequate, for instance, to analyze functional (safety and liveness) properties. However, the analysis of some non-functional properties such as the energy consumption requires to take into account (track and measure) the values of some non-discrete

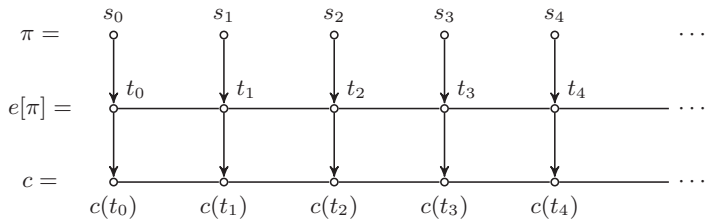


Fig. 3. Synchronization of trace π and continuous variable c using $e[\pi]$

variables which *evolve with time*. For instance, to analyze the energy consumed by a device to download a file, we should be able to detect in which states of the traces starts and finishes the download, and measure the consumed energy by the device during this period.

We use interval formulas to specify extra functional properties. In these formulae, there exists an *implicit synchronization* between the discrete evolution of traces and the continuous evolution of the magnitudes to be checked on traces. .

We assume that traces $\pi = s_0 \longmapsto \cdots$ are described as maps $\pi : \mathbb{N} \to \Sigma$ that associate each natural number with the corresponding state in the trace, that is, $\pi(i) = s_i$. Since the traces provided by the test cases are *finite*, we suppose that each trace π has an *ending state* o that repeats infinitely often. Hence, we assume that, for each trace π, there exists a natural number $n > 0$ (the length of the trace, denoted as $length(\pi)$) such that (1) $\pi(n-1) \neq o$ and (2) $\forall k \geq n.\pi(k) = o$.

Although execution time is abstracted in operational semantics, it is clear that the execution of each trace takes time, and that during this time many other things may occur which influence or are affected by the trace execution.

Definition 1. *Given a trace* $\pi \in \mathcal{O}(P)$, an execution e of π *is a function* $e[\pi] : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *that associates each state* $\pi(i)$ *of* π *with the time instant* $e[\pi](i) \in \mathbb{R}_{\geq 0}$ *in which it occurs.*

Note that each trace π may have many executions $e[\pi]$, each one relating states to different time instants. Once the execution of a trace $e[\pi]$ has been fixed, we can observe the values of continuous magnitudes that evolve synchronously with trace. The diagram in Figure 3 illustrates this synchronization. The upper row shows the states of trace $\pi = s_0 \longmapsto \cdots$. The middle row shows the time passing, with each state s_i associated by means of $e[\pi]$ with the time instant when it occurs. Finally, the lowest row shows the evolution of continuous variable c in time, and its synchronization with $e[\pi]$.

We use intervals of states (inside the traces) to determine the periods of time during continuous variables should be observed. To do this, we use interval calculus introduced by [10] which will allow us to give formal semantics to the language for extra-functional properties. The domain of interval logic is the set of time intervals $\mathbb{I}ntv$ defined as $\{[t_1, t_2] | t_1, t_2 \in \mathbb{R}, t_1 \leq t_2\}$. An *interval variable* v is a function $v : \mathbb{I}ntv \to \mathbb{R}$ that associates each interval with a real number.

For instance, a continuous variable $c : \mathbb{R}_{\geq 0} \to \mathbb{R}$ can be used to define interval variables, such as $\mathit{diff}\_c : \mathbb{I}ntv \to \mathbb{R}$ given as $\mathit{diff}\_c([t_1, t_2]) = c(t_2) - c(t_1)$.

We can construct *interval expressions* describing properties on intervals by using a set of interval variables, relational, boolean operators and real constants. For instance, if K is a constant, $\mathit{diff}\_c \leq K : \mathbb{I}ntv \to \{\mathit{true}, \mathit{false}\}$ defines the property on time intervals $[t_1, t_2]$ that is true iff $c(t_2) - c(t_1) \leq K$.

Given a trace π and an interval of natural numbers $[i, j]$, we denote with $\pi \downarrow [i, j]$ the state interval/subtrace of π from state $\pi(i)$ to $\pi(j)$. Similarly, given an execution e of π, $e[\pi] \downarrow [i, j]$ represents the time interval $[e[\pi](i), e[\pi](j)]$ from the creation of state $\pi(i)$ to the creation of state $\pi(j)$ in execution $e[\pi]$. Thus, state intervals and executions of traces provide time intervals on which we can evaluate interval expressions such as $\mathit{diff}\_c \leq K$.

We now explain how *state formulae* are used to construct state intervals. Given the set of state formulae Prop, we call *proposition intervals* to expressions such as $[p, q]$ with $p, q \in \mathit{Prop}$. We extend the satisfaction relation $\models$ on state intervals as follows.

Definition 2. *Given a trace π and an interval of natural numbers $I = [i, j]$, we say that the state interval $\pi \downarrow I$ satisfies $[p, q]$, written as $\pi \downarrow I \models [p, q]$, iff the following conditions hold: (1) $\pi(i) \models p$; (2) $\mathit{foralli} < k < j.\pi(k) \not\models q$; and (3) $\pi(j) \models q$. That is, $[i, j]$ is a state interval of π such that $\pi(i)$ satisfies p, and $\pi(j)$ is the first state after $\pi(i)$ that satisfies q.*

In the following, we assume that the ending state o satisfies no formula of Prop, that is, $\forall p \in \mathit{Prop}.o \not\models p$.

Now, given a trace π and a proposition interval $[p, q]$, we denote with $\pi \Downarrow [p, q]$ the finite sequence of state intervals of π, written as $I_0 \cdot I_1 \cdots I_{m-1}$, that satisfy $[p, q]$ in the sense above described, that is, $\forall 0 \leq i < m.\pi \downarrow I_i \models [p, q]$.

Given $p \in \mathit{Prop}$, a finite trace π of length n, and $k \geq 0$, $\pi \downarrow_k p$ is the first state of π that occurs after (including) $\pi(k)$ and that satisfies p, if it exists, or symbol ∞, otherwise.

Given a finite trace π, and two state formulae p, q, we denote with $\pi \Downarrow [p, q]$ the sequence of state intervals determined by p, q.

Thus, two state formulae $p, q \in \mathit{Prop}$ determine a sequence of state intervals $\pi \Downarrow [p, q] = I_1 \cdots I_n$ in π that satisfy $[p, q]$. We can extend this definition to executions e of π as $e[\pi] \Downarrow [p, q] = e[\pi] \downarrow I_1 \cdots e[\pi] \downarrow I_n$.

The following definition states when an execution e of a trace π satisfies an interval expression Φ such as $\mathit{diff}\_c \leq K$.

Definition 3. *Let Φ and $[p, q]$ be an interval expression and a state proposition interval, respectively. Let e be an execution of a finite trace π. Then*

1. *We say that $e[\pi]$ satisfies Φ on the time intervals determined by $[p, q]$, and denote it as $e[\pi] \models [[\Phi]]_{[p,q]}$ iff $e[\pi] \Downarrow [p, q] = T_1 \cdots T_n$ with $n > 0$ and $\Phi(T_1)$ holds.*
2. *We say that $e[\pi]$ satisfies $\exists\Phi$ on the time intervals determined by $[p, q]$, and denote it as $e[\pi] \models \exists[[\Phi]]_{[p,q]}$ iff $e[\pi] \Downarrow [p, q] = T_1 \cdots_I n$ with $n \geq 0$ and $\exists 1 \leq i \leq n.\Phi(T_i)$ holds.*

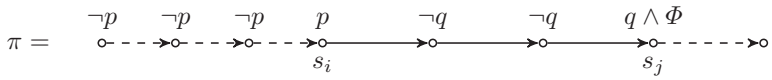
3. *We say that* $e[\pi]$ *satisfies* $\forall\Phi$ *on the time intervals determined by* $[p,q]$*, and denote it as* $e[\pi] \models \forall[[\Phi]]_{[p,q]}$ *iff* $e[\pi] \Downarrow [p,q] = T_1 \cdots T_n$ *with* $n \geq 0$ *and* $\forall 1 \leq i \leq n.\Phi(T_i)$ *holds.*

That is, an execution e of trace π satisfies formula (1) $[[\Phi]]_{[p,q]}$ iff the first time interval determined by $\pi \Downarrow [p,q]$ and e satisfies Φ, (2) $\forall[[\Phi]]_{[p,q]}$ iff all the time intervals determined of $\pi \Downarrow [p,q]$ and e satisfy Φ; $\exists[[\Phi]]_{[p,q]}$ iff a state interval exists in the sequence $e[\pi] \Downarrow [p,q]$ that satisfies Φ. Recall that if $I = [i,j]$ is a state interval of π, $e(\pi)(I)$ is the time interval $[e(\pi)(s_i), e(\pi)(s_j)]$. For instance, if $\Phi = diff\_c \leq K$, $[[\Phi]]_{[swifi,ewifi]}$ establishes that the time interval determined by the first state interval on which $[swifi, ewifi]$ holds must satisfy Φ.

The interval properties presented may be translated into LTL formulas to be analyzed by a model checker. As an example, we next explain how formula $[[\Phi]]_{[p,q]}$ is translated.

Assume that $\Phi(p,q)$ is defined as $\Phi(p,q) \equiv p \wedge (\neg q\, U\, (q \wedge \Phi))$ Intuitively, $\Phi(p,q)$ is the LTL representation of property: *"p holds on the current state, q will be true in a future state and, at that moment, the time interval determined by p and q will satisfy* Φ*"*.

The LTL specification of $[[\Phi]]_{[p,q]}$ is $[[\Phi]]_{[p,q]} \equiv (\neg p)\ U\ \Phi(p,q)$. The intended meaning of this formula is as follows. We first search for the first state (s_i) on which p holds, then we search for the first state following s_i on which q holds (s_j). These two states s_i and s_j determine a time interval. If Φ is true on this interval, then formula $[[\Phi]]_{[p,q]}$ holds. Otherwise, that is, if it is not possible to find either s_i or s_j, or if the time interval does not satisfy Φ, the formula is false. The following sequence shows a trace that satisfies $[[\Phi]]_{[p,q]}$.



5 Implementation for Android devices

This sections describes the implementation of our proposed approach for analyzing mobile applications. Although this architecture can be adapted for different mobile operating systems, here we have focused on the implementation for Android devices. Figure 4 outlines the main steps and components of this implementation, which can be divided into two groups: model-based test generator, and runtime verification.

5.1 Model-based test generator

The first half of the process consists of the generation and execution of test cases, starting from a model of the user behavior. The individual steps and components are shown on the left half of Figure 4. Firstly, the developer creates a model of the user behavior for the applications being analyzed, using the language described in Section 3.2. UIAutomatorViewer tool [11] is used to bind each view machine

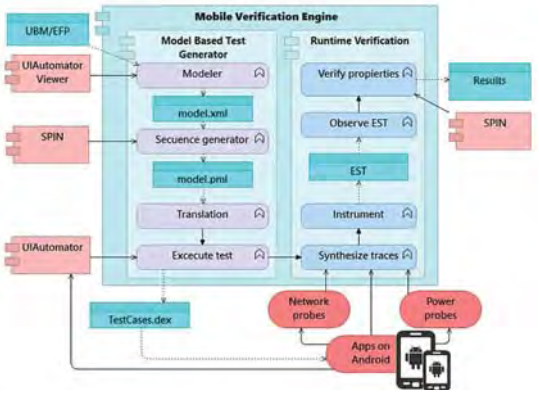


Fig. 4. Test generation and verification implementation for Android

of this model with the actual controls in a Android screen, producing an XML representation. We then enrich this XML file, adding new attributes to bind the controls with the labels on the transitions.

Next, the model is explored exhaustively to generate all possible sequences of user actions. As commented above, each sequence corresponds to a test case that can be executed on the device. For this step, we take advantage of the Spin model checker [12], a powerful tool for model analysis and validation. Spin can be used to verify the correctness of concurrent software systems modeled using the Promela specification language. In our case, we use Spin to perform an exhaustive exploration of the application model, by translating the XML representation into a Promela specification. Each device is represented by a Promela process that models all the state machines contained in that device. While the application model is composed of nested state machines, the Promela code for a device consists of a single loop, where each branch corresponds to a transition in the model. A global variable per device is used to track its current state, and decide which transitions can be taken next.

This specification is explored depth-first by Spin, recording each transition taken. When a valid end-state is reached, the sequence of transitions taken during the exploration contain the user actions to perform a single test case. If more than one transition can be taken at one point, Spin will first explore one of them, and later return to explore the rest. In this way, the set of all possible test cases will be generated.

In the translation step, each sequence produced by Spin is translated into a Java program that can be deployed on an Android device to execute a test case using the UIAutomator API from Android.

Listing 1.1 shows part of a class that implements one of the test cases generated from the example shown in Figure 2. Each method corresponds to a user action in the test case, which is carried out using the UIAutomator API. For system events, the method would implement the appropriate wait condition,

```
1  // prev  next S0 on SearchView
2  public void TestSpotifysetTitlePopular21() throws UiObjectNotFoundException
3  {
4    UiObject control = new UiObject(new UiSelector()
5                                    .className("android.widget.EditText")
6                                    .index(0).textContains("Search"));
7    control.setText("on top of the world");
8    Log.v("ANDROIDMVE","CONTROL-setTitlePopular");
9  }
10 // prev S0 next S0 on SearchView
11 public void TestSpotifyclicsong22() throws UiObjectNotFoundException
12 {
13   UiObject control = new UiObject(new UiSelector()
14                                   .className("android.widget.LinearLayout")
15                                   .index(1));
16   control.click();
17   Log.v("ANDROIDMVE","CONTROL-clicsong");
18 }
```

Listing 1.1. Extract of ANDROID test case translated to JAVA

blocking until the system event is received. The first method finds a text field named "Search" (line 4) and enters a song title (line 7). The second method clicks on the first search result (line 16. Both methods write the transition they just executed to the system log (lines 8 and 17).

Finally, the test is executed on the device using UIAUTOMATOR. Each step in the test case is in sequence, using the timing information from the user behavior model, if any. If a test case spans multiple devices, each one is executed concurrently. In this case, system events may be used to synchronize the devices on specific points, such as the reception of a message.

5.2 Runtime verification

The second half of the process is the analysis of a test case execution, shown on the right half of Figure 4. This part involves extracting an execution trace and analyzing it to verify the given extra-functional properties.

While the test case is being executed on the ANDROID device, runtime information is extracted from several sources, both from inside the device and from external probes. One of the sources is the LOGCAT [13], which is the place where all the logging information produced by the operating system and the running applications is collected. While this source is mostly unstructured and unfiltered, relevant information may be found here and extracted for the execution trace. For instance, our test cases log information about the progress of the execution of a test case there, as shown on Listing 1.1.

Since ANDROID applications are regular JAVA programs at their core, the Java Debugger Information (JDI) [14] can be used to extract fine-grained information. JDI is the same API used by JAVA debuggers to step through the code during its execution and show low-level information, such as the current values of program variables, or the execution of methods.

We have also implemented two useful sources for extra-functional properties: network traffic and energy consumption. For the former, we use the well-known TCPDUMP command-line tool on the device. TCPDUMP captures packet information from the network interfaces of a device, filters it, and provides it for further

analysis. The runtime verification engine collects this information live from the device, and includes it in the execute trace.

Energy information is obtained using an external power analyzer: a N6705B unit from Keysight Technologies. This instrument is connected to the ANDROID device in place of the battery, and provides it with power while also measuring energy consumption at the same time. The instrument can be controlled remotely using its SCPI interface, e.g. to query energy measurements periodically.

The runtime verification engine collects the information from all these sources, filters out irrelevant parts, and combines it to construct the enriched standardized trace (EST). This trace is a sequence of discrete states, which contains the values of the observed variables at certain time instants. These states usually correspond to relevant events during the execution of a test case, such as a new user action, or the execution of a particular JAVA method. In addition, external continuous variables, such as energy measurements, are incorporated trace using the timestamp of each state, as described in Section 4.

Finally, an observer analyzes the enriched trace to verify the extra-functional properties provided by the user. The observer analyzes the trace on the fly, while the test that produces it is still running, and produces a verdict as soon as it is available. The observer is implemented using the SPIN model checker, which can handle the LTL representation of the EFPs, as described on section 4. Instead of analyzing a PROMELA specification that models a system, the observer has to analyze the execution trace from a real system. To achieve this, we use a PROMELA specification with embedded C code that translates the enriched standardized trace into SPIN states on the fly [15]. Listing 1.2 shows a simplified fragment of the PROMELA specification for one of our case studies.

The PROMELA specification contains global variables for each of the relevant variables in the EST. The core of this specification is a loop (lines 20 to 35) that reads and reconstructs an EST from an external source. Each iteration of the loop fetches a state from the trace (line 27) and updates SPIN's global state accordingly (line 32), in an atomic step. If SPIN has to backtrack during the exploration of the trace, the loop can also restore previously visited states correctly. From SPIN's point of view, each new iteration leads to a new state to be explored. To analyze the EFPs, the corresponding LTL formula is negated and translated into a *never claim automata*.

To deal with continuous variables and interval properties, we rely on additional global variables and helper functions. The current value of a continuous variable c is stored in a floating point variable `c`. These variables are updated on each iteration of the core loop, like any other. Our example has one continuous variable, called `energy` (line 3). In order to evaluate an interval formula Φ is evaluated in an interval $[t_1, t_2]$, the initial and final values of any continuous variable c, i.e. $c(t_1)$ and $c(t_2)$, must be available. While the former is already available in the global variable `c`, we need a new global variable `c_t1` for the latter. These variables are updated automatically with the so-called update functions, i.e. C functions that compute the values of variables that are derived from the EST.

```
1  c_state "short _interval" "Global"
2  c_state "short testStep" "Global"
3  c_state "double energy" "Global"
4  c_state "double energy_t1" "Global"
5
6  c_code{
7      void update_interval(struct state* newState) {
8          if (!(now.testStep == START) && (newState->testStep == START))
9              newState->_interval = 1;
10         else if (!(now.testStep == END) && (newState->testStep == END)) {
11             newState->_interval = 0;
12     }
13     void update_energy_t1(struct state* newState) {
14         if (!now._interval && newState->_interval)
15             newState->energy_t1 = newState->energy;
16     }
17 }
18
19 init {
20     do
21     :: (running) -> c_code {
22             now.currentState++;
23             if (now.currentState > lastState) {
24                 if (!wasRunning) {
25                     now.running = 0;
26                 } else {
27                     readNewState();
28                     lastState++;
29                     callUpdateFunctions();
30                 }
31             } /* else: backtracked */
32             updateSpinStateFromStateStack();
33         }
34     :: (!running) -> break
35     od
36 }
```

Listing 1.2. Fragment of PROMELA observer for EST analysis

Line 13 shows the update function for the `energy_t1` variable (line 4). This function only updates the values of `energy_t1` at the start of a new interval. To detect this, we introduce another variable `_interval`, automatically updated with another function (line 7). This function encodes the start and end conditions of an interval formula. Both update functions are executed inside the loop (line 29), after the variables for the next state have been retrieved, in the same atomic step.

6 Conclusion and future work

The paper presented the foundations for a complete platform to verify extra-functional properties of Android mobile applications at runtime. The method is based on to formal languages to specify a) the interactions user-application as the input to automatically generate test cases, and b) the expected extra functional properties that the executions of the app should satisfy. Both, test generation and verification are performed with model checking. The paper also presents a first implementation and evaluation with realistic applications.

There are some points where improvements are required. One of them is the need to get better synchronization of the measurements and informations obtained at runtime. Enhancements in this context will make it possible to specify richer properties. A second improvement is the implementation of some automatic process to assist the programmer to build the formal model of the user-application. Both issues are part of current and future work.

An additional open research line is the study of how to adapt out approach to other mobile operating systems, like iOS or Windows Mobile. Most of the components could be reused with minor or no changes, like the language for EFPs and the mechanisms to generate test cases. However some of them require additional study, like the modeling language to describe the user interactions or the control of the executions in the smartphone.

References

1. Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. of the 2013 ACM SIGPLAN Int. Conf. on Object Oriented Prog. System Languages*, OOPSLA '13, pages 641–660, 2013.
2. Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proc. of the 36th Int. Conf. on Software Engineering*, ICSE 2014, pages 1013–1024, 2014.
3. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
4. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, USA, 1999.
5. Klaus Havelund. Implementing runtime monitors. Extended abstract., 2012. Invited talk, 2nd TORRENTS Workshop.
6. Ana Rosario Espada, María-Mar Gallardo, and Damián Adalid. Dragonfly : Encapsulating android for instrumentation. In *Proceedings of the XIII Jornadas de Programación y Lenguajes (PROLE 2013)*, 2013.
7. UML 2.4.1 Superstructure Specification. Technical report, Object Management Group (OMG), August.
8. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
9. Ana Rosario Espada, María-Mar Gallardo, Alberto Salmerón, and Pedro Merino. Using model checking to generate test cases for android applications. In *Proc. Tenth Workshop on Model Based Testing*, volume 180 of *Electronic Proceedings in Theoretical Computer Science*, pages 7–21. Open Publishing Association, 2015.
10. Zhou Chaochen and Michael R. Hansen. *Duration Calculus - A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004.
11. Android Open Source Project. UIAutomatorViewer.
12. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
13. Android Open Source Project. logcat.
14. Oracle. Java Debug Interface.
15. Alberto Salmerón and Pedro Merino. Integrating model checking and simulation for protocol optimization. *Simulation*, 91(1):3–25, 2015.