Hindawi
Mobile Information Systems
Volume 2017, Article ID 2012696, 14 pages
https://doi.org/10.1155/2017/2012696

Research Article

Performance Analysis of Spotify® for Android with Model-Based Testing

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

Departamento Lenguajes y Ciencias de la Computación, E.T.S.I. Informática, Universidad de Málaga, Andalucía Tech, Málaga, Spain

Correspondence should be addressed to Pedro Merino; pedro@lcc.uma.es

Received 24 September 2016; Revised 20 December 2016; Accepted 15 January 2017; Published 16 February 2017

Academic Editor: Porfirio Tramontana

This paper presents the foundations and the real use of a tool to automatically detect anomalies in Internet traffic produced by mobile applications. In particular, our MVE tool is focused on analyzing the impact that user interactions have on the traffic produced and received by the smartphones. To make the analysis exhaustive with regard to the potential user behaviors, we follow a model-based approach to automatically generate test cases to be executed on the smartphones. In addition, we make use of a specification language to define traffic patterns to be compared with the actual traffic in the device. MVE also includes monitoring and verification support to detect executions that do not fit the patterns. In these cases, the developer will obtain detailed information on the user actions that produce the anomaly in order to improve the application. To validate the approach, the paper presents an experimental study with the well-known Spotify app for Android, in which we detected some interesting behaviors. For instance, some HTTP connections do not end successfully due to timeout errors from the remote Spotify service.

1. Introduction

Nowadays, there are millions of applications available in mobile app stores, such as Google® Play Store, which are supported by hundreds of thousands of development teams around the globe. According to CISCO, the monthly global mobile data traffic will surpass 24.3 exabytes by 2019 [1] as smartphones have become one of the most popular ways to access Internet services. Mobile app developers are constantly confronted with a competitive environment where they have to satisfy the demand for more and better services, which frequently leads to a surge in mobile data traffic requirements. The increase in the complexity of modern mobile software, in addition to shortened release deadlines, has led to many optimization challenges. For instance, current initiatives towards 5G networks will produce new networking scenarios where 4G applications should be tested and optimized for ever-changing resources and technologies.

Mobile network traffic is heavily influenced by user interaction with the application. User behavior may trigger runtime errors that may not be identified during development stages [2], as the synergic effects of user interactions and the communication channel conditions are difficult to predict. For this reason, software verification and performance analysis are key steps in the development stages to assure the quality of applications. Analyzing the correctness and performance of mobile apps taking into account all possibilities of user behavior is a difficult task. In consequence, the quantification of this effect is valuable for all actors in the app's development and operation chain: app developers, mobile phone manufacturers, mobile network operators, and end users.

In this paper, we contribute to enhancing the quality of the mobile software analysis process by focusing on the network traffic produced by the applications. In order to do so, we employ a combination of model-driven exhaustive generation of user behavior test cases and software and hardware probes intended to extract runtime information from the applications, the smartphone itself, and network equipment.

This heterogeneous information is synchronized and gathered together in a single rich trace which can then be analyzed on-the-fly with techniques rooted in formal methods [3, 4].

In particular, we employ two specification languages. On the one hand we use UML state machines to model the potential interactions of the user with the application, including the typical concepts in the design of the application like screens or buttons and the user's actions. This model of user interactions is the input to the exhaustive test case generation. On the other hand, we make use of a specification language [5] to define extra-functional properties that correlate specific events in the applications (like push or swipe a button) with specific patterns for network traffic in the device. In addition, we implement monitoring and checking support to automatically detect whether the pattern is satisfied in the execution corresponding to a test case. In case of a mismatch between the real traffic and the expected behavior pattern, the developer is informed about the anomaly with enough data to locate the source of the problem.

It is important to note that the model-driven approach for simulating user behavior can be used to generate a large number of experiments, and the results from these analyses can serve as feedback to drive experiment generation, for example, to look for errors or optimize certain parameters. Furthermore, we intend to provide a solution where the user does not need to be an expert in this particular field, as we propose the use of a simple notation and a pattern catalog, which can be used to analyze the usual key performance indicators (KPIs).

MVE (Mobile Verification Engine) (tool and examples available at http://www.morse.uma.es/tools/mve), the current tool implementing this approach, is oriented to Android OS applications and has been validated with the Spotify app for Android as the case study. As is well-known, Spotify is one of the most successful music streaming services worldwide. Almost sixty million people use Spotify services and half of them are using the mobile application. Such a large number of concurrent users tend to show typical network traffic patterns related to the number of open sessions per hour or the duration of the connection. We have evaluated our tool in the controlled testbed PeformNetworks (formerly known as PerformLTE [6]), which includes all the components to create a real mobile network and the required monitoring capabilities.

Compared with our previous work reported in [4, 5], this paper presents three main novelties: (1) we focus on network traffic, and we introduce statistical variables in the extra-functional properties, while in our previous work we only focused on energy measurements, where only real values provided by external equipment was considered and no statistics were required; (2) we present MVE as an integrated tool with all the steps that in previous work were only partially implemented but not integrated, and (3) we use a new case of study, Spotify for Android, which has a major impact on mobile traffic today.

The rest of the paper is organized as follows. In the following section, we provide related work to highlight our contributions with respect to the state of the art. Section 3 describes the proposed approach to performance analysis with model-based testing, including some background on the main techniques used. Section 4 presents the case study Spotify app for Android, detailing the setup and steps to run the experiments, and a discussion of the results. In Section 5, we present Conclusions and Future Work.

2. Related Work

In this section, we summarize the main related works existing in the literature which apply model-based techniques to test mobile apps and/or that deal with the problem of the analysis of extra-functional properties on these devices. A complete survey of the techniques and tools for testing mobile applications may be found in [7].

The generation of test cases can be addressed with random techniques, which has the advantage of not requiring knowledge of the application under test. This random-behavior generation strategy is supported by tools like Monkey [8]. Clearly, the main drawback of these techniques is that the test cases generated can be very unrealistic, which can degrade the effectiveness of the testing process. To solve this, other tools such as Monkeyrunner [9], Robotium [10], and Troyd [11] employ scripts of predefined sequences of user interactions, but these scripts cannot be automatically constructed.

Other approaches such as Swift-Hand [12] make use of learning machine techniques to automatically generate input sequences trying to visit unexplored behaviors. Static analysis is also used as an underlying technique to automatically extract models of the app behavior. This is the case of the tools Automatic Android App Explorer (A3E) [13] and FicFinder [14]. In particular, A3E uses a static dataflow analysis of app bytecode to construct a high-level control flow graph that captures legal transitions between activities (i.e., app screens) and that is later explored with a depth-first strategy called targeted exploration. In contrast, FicFinder is concerned with the analysis of compatibility issues in Android apps. To this end, the tool carries out a static analysis of the app source code making use of the so-called API-Context pairs, which are used to detect when the apps call API functions incorrectly.

Other tools such as AndroidRipper [15] or, more recently, MobiGUITAR [16] are based on the automatic construction of state machines from the GUI components of the app under test. The state machine generated by AndroidRipper is a stateless model which is insufficient to analyze state-sensitive properties such as those related to the state-based life cycle of Android activities. On the contrary, MobiGUITAR constructs a more complex state machine from the app. It starts at an initial state and explores, using a breadth-first traversal, the new events that can be fired. This procedure results in a tree which is reduced to a graph using a notion of state equivalence. Maybe MobiGUITAR is the closest to our approach. However, there exist some significant differences. For instance, since the construction of state machines in our tool is guided by the tester, we do not need to remove unrealistic test cases. In addition, our approach allows generating test cases for several applications that interact using Android intents, while the complexity of

the runtime based modeling process for MobiGUITAR and Swift-Hand makes them more suitable for single applications.

There are other works that consider performance and energy consumption prediction as part of a model-driven approach, using app models at design time for this purpose, instead of for test case generation. For instance, the work in [17] uses a model of the app to drive the analysis. However, the authors can predict energy consumption with abstract interpretation and compare the results with the real measurements. The SPOT framework [18] implements a full model-driven software engineering process to estimate mobile software power consumption and performance. The main point in SPOT is the model of the full software architecture supporting the app and the generation of the app code, including the emulation of the mobile device. Since this proposal, many other papers have addressed the use of model-driven development to test Android apps (see [19] for a survey).

Other works use mathematical models to predict energy consumption. Instead of modeling the app or the user, in [20], the authors build a model of how typical components in the smartphone, like CPU or 3G/LTE radio, consume energy. Then they collect the logs of the application running in the device to make an estimation of energy consumption based on the model. A similar approach is described in [21], where the authors compute profiles of energy consumption for typical resources used in the app. However, the authors use a simulation tool instead of a real device to predict power consumption. Compared to our work, this is a different way of using models for analysis because the authors do not use the model to generate potential behaviors. Actually, our approach can be combined with the work in [20, 21], using our tool to generate automatically the app behaviors where their estimation methods can be applied. The idea of modeling the expected power consumption by the app is also developed in [22], using a model of the application instead of the actual source code. Both models can be compared using model checking.

Apart from the previously described model-based approaches, many commercial and academic tools have been developed to analyze and classify the traffic produced by apps in mobile phones without specific models of the app. For instance, AntMonitor [23] is a powerful tool for monitoring all connected apps in an Android device so as to produce statistics. The tool is able to detect how each app contributes to the total network traffic. However, since it collects too fine-grained mobile measures, it cannot be easily used to relate the measured traffic with specific events or user interactions.

Likewise, NetworkProfiler [24] is a tool oriented to help cellular operators identify the traffic in their networks when transported over HTTP/HTTPS. NetworkProfiler uses device emulators and machine learning techniques to establish the fingerprint of an app by collecting information about the hosts to which it connects and by subsequently constructing a state machine that represents the patterns of data (such as, e.g., some strings in the URL query that identify the apps) sent over the HTTP connection. However, since the manual exploration of apps under test will not cover all behaviors, NetworkProfiler randomly generates user actions to interact with the apps.

In contrast, ProfileDroid [25] is designed to systematically profile apps in order to discover inconsistencies or surprising behaviors. It is based on a multilayer analysis of the apps, considering static analysis of bytecodes, user interactions, calls to operating system, and network traffic. Nevertheless, this particular tool requires a real user interacting with the mobile, although the sequence of interactions can be recorded to be replayed in a different scenario.

Regarding traffic analysis, ProfileDroid is the closest proposal to our work. We could use ProfileDroid to generate the reference patterns that we employ to analyze the actual behavior of the apps. However, MVE has novel contributions compared with ProfileDroid, such as (a) the controlled coverage of the user interactions for one or several apps due to the model-based approach for test generation; (b) the ability to automatically find execution traces that violate the expected behavior of the app in terms of their effect over the Internet traffic; (c) the inclusion of time in the models, so we can test realistic situations where the time between user interactions is relevant; and (d) the ability to combine models of several apps running in parallel.

3. Analysis of Applications with Model-Based Testing

Our approach to analyze of mobile applications using model-based testing, represented in Figure 1, can be divided into two main steps: (a) the test case generation and subsequent execution and (b) the runtime verification tasks. In the first stage, executable test cases will be automatically generated from a *model* of the possible user interactions over the application under test. Then, each test case generated will be executed on a real mobile device. In the second stage, the execution of test cases will be analyzed to check whether they behave correctly according to user-defined properties.

The first stage can also be divided into three substeps: user behavior modeling, test case generation, and test case execution. In the first substep, the user provides a model of the application (or applications) to be analyzed. The model is constructed from the point of view of a user interacting with the application. Modeling the user behavior is crucial to generate test cases that correspond with realistic uses of the application, instead of generating random inputs. In the next substep, test cases are automatically generated from this model using the capacity for automatic exploration of Spin model checker [26]. Each test case produced by the model checker is then converted into executable Java code for Android devices. In the last substep, these test cases are executed one by one in an actual mobile device.

In the second stage, the test case executions will be analyzed according to properties given by the user. This stage is divided into two substeps: trace synthesis and property verification. A trace is an ordered sequence of states, that is, data gathered from the test case execution at different time instants. The trace is built from data provided by multiple sources at runtime, such as the actions performed by the

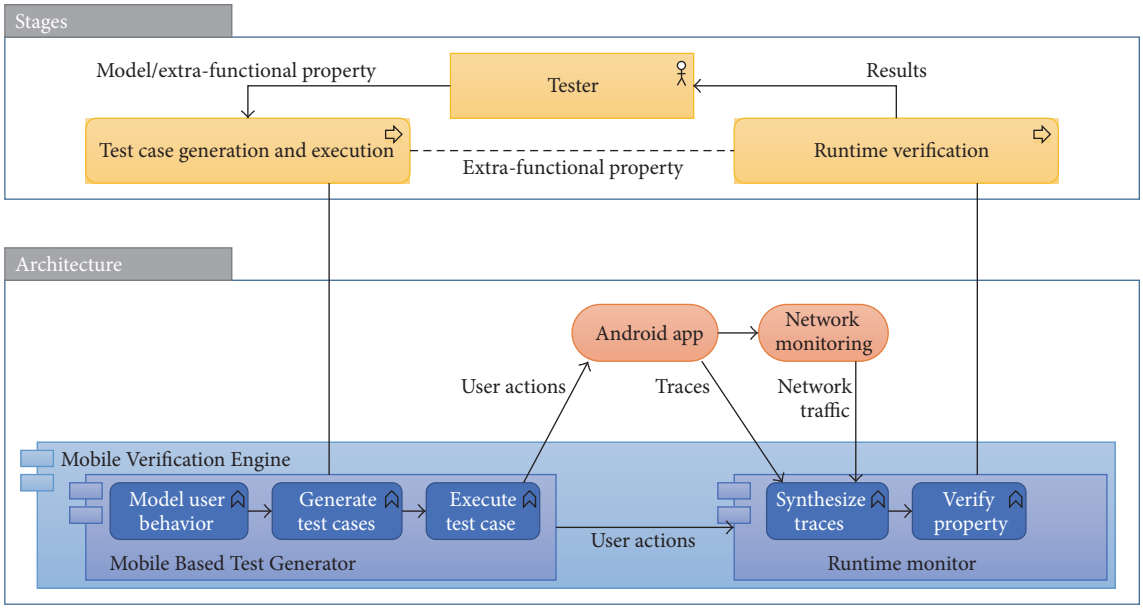


Figure 1: Architecture overview.

user, energy consumption measured by an external power analyzer, or network traffic statistics. In the last substep, each of these traces is analyzed at runtime by the Spin model checker to check the extra-functional properties given by the user; for example, in our case, to check whether network traffic follows a given expected pattern. The main components of this architecture are further detailed in the following sections.

3.1. Modeling User Behavior. The *model* of expected user behavior is used as the starting point for generating test cases. This model describes possible interaction paths of the user in each of the application "screens." For instance, in the inbox screen of an e-mail application, a user may tap one of the e-mail entries to open it or tap the "compose" button to write a new e-mail. Each of these paths will lead to a new screen with new possibilities.

As an example, Figure 2 shows three screens extracted from the Spotify Android app. Some interactions, such as swiping through a list, happen within the same screen, while others, such as tapping the back button, change the current screen. In the example shown in Figure 2, the "Playlist" screen is deeper in the hierarchy than the "Main" screen and is accessed by tapping on an item of the horizontal swipe-list at the top of the screen. Then, a further action of tapping the "Shuffle play" button will lead to the "Now Playing" screen where the song reproduction controls are displayed.

These interactions can be modeled using *state machines*, where each state machine represents the possible user behaviors in a given screen of a mobile application. The edges are labeled with the user actions performed on the elements present on the screen when a transition is taken. The supported user actions include tapping, swiping, and entering a text in a field. In the latter case, the model must also provide the values that can be entered or a pattern to generate them automatically. The user actions in the transitions, which are abstract, must be matched with concrete UI controls present in the screen of the Android app.

These state machines are both nested, to hierarchically organize them into screens, apps, and devices, and composed, to represent navigation between screens and application. This loose composition is key to represent some typical patterns found in mobile applications. For instance, the same screen may be reached from two different places, and navigating back should take the user to the correct previous screen.

Figure 3 shows a simplified example of a user behavior model for an e-mail application, using a UML-like notation. The figure shows three state machines: `Inbox`, `E-mail`, and `Compose`. The transitions between states are labeled with the user action that would be performed when taking that transition; for example, `/tapCompose` corresponds to tapping the "compose" button while the current state machine is `Inbox`. Observe that transition `/tapCompose` jumps to state machine Compose. This state machine can also be reached by tapping "reply."

The state machines can be executed by starting at the initial state and following the available transitions at each state. Some states have more than one outgoing transition meaning that there will be more than one possible execution sequence that passes that state. State machines are connected through the so-called connection states. When entering a connection state, the execution will continue on another state machine. When this second state machine reaches a final state, the execution will proceed with the state following the connection state in the first state machine.

The user behavior model is written in XML, following an XML Schema. A UML model of this schema is shown in Figure 4. A user behavior model contains the definition of a number of applications. Each application is composed of several views, which correspond to its screens, and each view

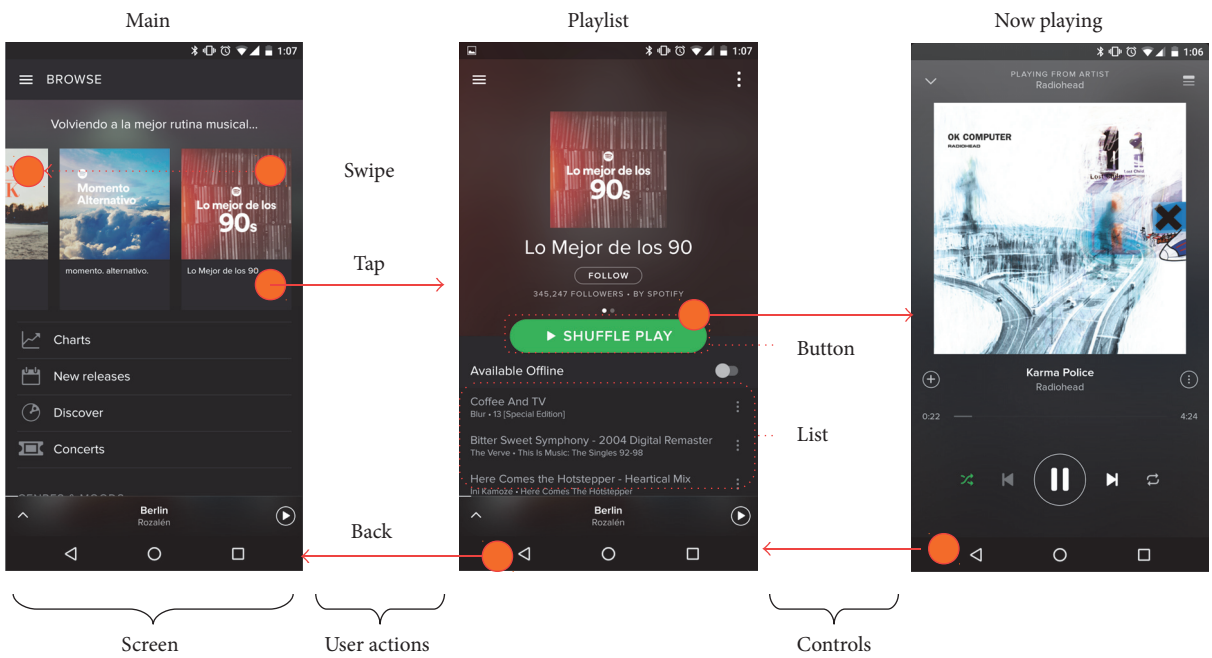


FIGURE 2: Example of Spotify screens and UI elements.

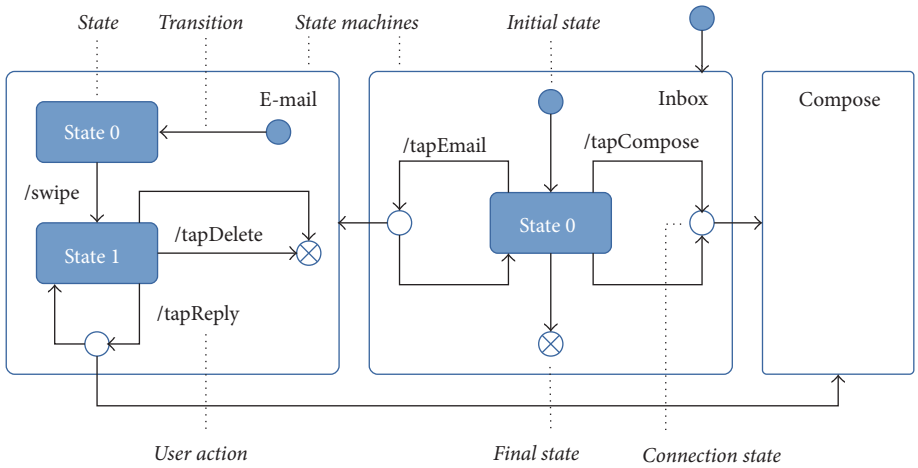


FIGURE 3: Fragment of user behavior model.

may define more than one state machine. Simple transitions are defined between states of the same state machine, while complex transitions define connection states, where the execution flow continues in another state machine. Such transitions can continue on a concrete state machine or in any of the state machines contained in a view. The *time* attribute in a transition indicates the time required to execute it, which must be simulated when executing the test case in a real device.

3.2. Test Case Generation and Execution. As mentioned above, to generate the test cases we use the Spin model checker [26]. Model checking is a formal technique traditionally used to check the correctness of nondeterministic concurrent systems with regard to some given temporal property, such as absence of deadlocks. Given a model of the system to be analyzed and a desirable property to be checked on the system, model checkers work, in general, by automatically and exhaustively exploring all the possible executions of the system searching for traces that violate the property (which are called counterexamples). In the case of Spin, the language for modeling the systems is Promela and properties are written in the temporal logic LTL. The strength of model checking technique may be its capability of providing counterexamples when the tool finds a model behavior that does not match the property. Counterexamples greatly facilitate debugging systems. In contrast, the weakness of model checking is the well-known state explosion problem that occurs when the system to be analyzed is too big to be stored in memory. There exist a very large number of

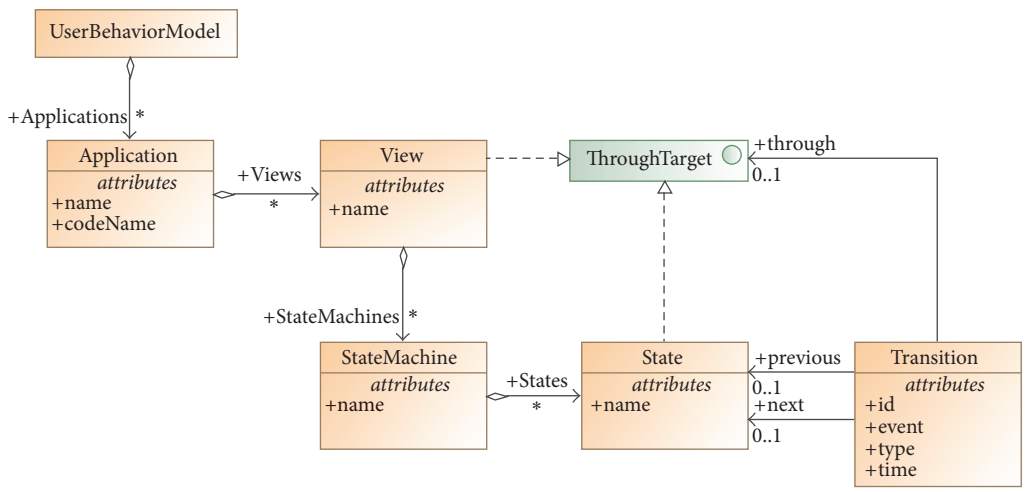


Figure 4: UML model of XML Schema for user behavior models.

approaches in the literature to palliate the effect of the state explosion problem, but they are beyond the scope of this paper.

Our approach uses the exhaustive exploration algorithms integrated into Spin to generate all the possible test cases defined by a user behavior model or a controlled portion of the test cases if there are not enough computational resources. Controlling the selection is now done limiting the length of the test cases. A test case is defined as a sequence of user actions encountered in the transitions taken while exploring the model starting at the initial state. To this end, the user behavior model is first translated into a Promela specification, which is then explored exhaustively by Spin. This translation process is outlined with an example in Section 4.1. When a final state is found, or the maximum sequence length is reached, Spin writes the generated sequence of user actions (the test case) to an XML file.

Each of the generated test cases is then translated into a Java program which uses the UiAutomator API (Application Programming Interface), an Android extension of the JUnit library. Our approach uses these Java programs to automate the user actions defined in a test case and perform them on real Android devices. The test is compiled to an Android application format in the form of a .dex file that is uploaded to the corresponding device. The test is then executed running the UiAutomator command line tool on the Android device, with the uploaded file as a parameter. Test case generation and execution summarized here is described in more detail in [4].

3.3. Trace Synthesis. The target device (or devices) and the additional measurement equipment are monitored with the aim of extracting relevant information (with respect to the property to be analyzed) about the execution of test cases in real-time. The devices can be instrumented to take data from different sources. One of these is the Android system-wide log, known as logcat, where applications and the operating system dump logging information. Another source is the JDI (Java Debugger Interface) API [27], the same API used by Java debuggers, which provides information at the source code level. We have used tcpdump [28], a well-known command line packet analyzer, to extract traffic information from the devices. The filtered packets captured by tcpdump can be processed in order to get relevant variables, such as the number of TCP packets received and sent through HTTP connections. We have also used an external power analyzer connected to the smartphones to quantify the energy consumed during the execution of the tests [5].

In consequence, the execution of each test case gives rise to a new timed trace where all these relevant data are gathered during execution. This trace is composed of a sequence of states, each of which contains the values of the observed variables at relevant time instants, for example, when a certain user action was performed. In consequence, the values of continuous variables, such as power consumption, may be observed over the time intervals determined by the time instants when some events of interest have occurred.

3.4. Extra-Functional Properties. The last step is to verify the set of properties given by the user. These properties are questions that the developer asks about the execution of the test cases, with the aim of finding bugs or anomalies in the app, and which can be answered by analyzing the traces. For this task, we propose architecture based on observers [29], which analyze the synthesized traces checking different types of properties. For instance, an observer may check log events to validate that a test case has been executed properly on the device, while others may check more complex properties involving data extracted from the hardware probes.

The properties to be checked on traces can be classified as functional or extra-functional. Functional properties are concerned with the correct functionality of the app (i.e., the app does not block, or its life cycle is correct). In contrast, extra-functional properties are usually related to physical magnitudes such as speed or resource consumption, which do not only depend on the application code but also on the specific hardware characteristic of the platform on which the app is executing. For instance, a typical property involving a particular physical (also called continuous) variable could be

that it remains below some threshold during some time interval. To make extra-functional properties easier to describe, we have defined a language to express them in a user-friendly manner, using interval formulas.

An interval formula defines a behavior *pattern* regarding the values of some continuous variable that will be checked in certain intervals of the trace. An interval formula is of the form $Q[[pattern]]_{[p,q]}$, where Q may be the universal/existential quantifier $\forall/\exists$ or empty. *Interval condition* $[p, q]$ defines subtraces of the trace being tested whose initial and ending states satisfy conditions p and q, respectively. In this case, we say that the subtrace *satisfies* condition interval $[p, q]$. Conditions p and q are simple expressions over the variables of the trace referring, for instance, to the last user action executed. Finally, *pattern* is an expression to be checked on subtraces. Usually, *pattern* is a formula over the continuous variables whose value is evaluated at the first and last state of subtraces that satisfy the interval condition.

As explained before, the formulas will be evaluated over the traces generated by the execution of the test cases. If a formula evaluates to false, the trace will be provided as a counterexample, that is, a sequence of states that lead to a property violation. The evaluation of the formula over a trace depends on the quantifier used: "no quantifier"/$\forall$/$\exists$ means, respectively, that the *pattern* has to be held on the "first"/"all"/"some" subtrace satisfying $[p, q]$.

For instance, a typical trace might include the *testStep* variable with the last executed user action in the test case and variable Z that stores the value of some continuous magnitude (as the Internet streaming traffic). Then, we can express that the streaming traffic measured while a song is being downloaded is less than a threshold T as

$$\forall\, [[Z < T]]_{[testStep=actionA, testStep=actionB]}, \tag{1}$$

where $\forall$ quantifier means that the pattern will be checked for all subtraces satisfying the interval condition $[testStep = actionA, testStep = actionB]$ found in a trace.

The observer that evaluates these interval formulas has been implemented with Spin. Interval formulas are translated into a special Promela observer called never-claim automata. Spin then analyzes a Promela specification which reconstructs on-the-fly the trace observed during the execution of a test case. This Promela specification is checked against the never-claim automata, and any violations are reported back to the user.

4. Case Study: Analysis of Spotify Network Traffic

This section describes a case study performed on the Spotify Android app, a well-known music streaming service. This case study will look into the traffic patterns generated by the application when playing a series of songs in different scenarios. Some screen shots of MVE graphical user interface (GUI) are also introduced to help reproduce the experiment. More details on how to do it can be found on the MVE web page at http://www.morse.uma.es/tools/mve.

4.1. Modeling. The first step is to provide a user behavior model of the application. Figure 5 shows a simplified model for the Spotify app for Android that follows the state machine based structure described in Section 3.1. The model is composed of views, each one corresponding to an app screen.

The model shows four main views (screens) and their decomposition in state machines. Observe that user actions in transitions may be enriched with time stamps to indicate the duration of the transition. HomeView may call PrincipalView which, in turn, may order the cache to be cleared through the ConfigurationView. PrincipalView may also call SearchView to search for a song (which can be *popular* or *unpopular*). Within the SearchView, the SearchPopularStateMachine first enters the name of a well-known popular song to be searched. The names for the songs that can be entered in the corresponding search field are provided by the user as part of the model, from the Spotify trending list in Spain. Then, the song may be played for 300 seconds or paused for 10 seconds half-way through. The SearchNonPopularStateMachine is similar, but with a set of unpopular song names to be used in the search instead. The whole search/play/pause/play process can be repeated.

Figure 6 shows the GUI of MVE for creating and configuring experiments. In Figure 6(a), the user first uploads the user behavior model, written as an XML file. Then, each state machine defined in the model must be configured to match user actions in the model with actual controls present in the screen (see Figure 6(b)). This is done by using the UiAutomatorViewer tool from the Android SDK, which obtains an XML definition of the controls present in a screen. The user must upload the definition of each of the screens used in the model and then select which control corresponds to each user action. If the action involves entering a text, then the values to be used must also be introduced. When the configuration is completed, the test cases may be automatically generated.

The XML user behavior model is automatically translated into Promela to generate the test cases with Spin, as explained in Section 3.2. Listing 1 shows part of the Promela code for the model in Figure 5. Each device defined in the model is translated into a `proctype` (see line (6)), that is, a Promela process that can be executed concurrently with others. All the state machines contained in a device are "flattened" and translated into a single Promela do loop. Each branch of the loop corresponds to a transition in one of the state machines of the device. The guard of each branch checks the current state of the device (line (9)), stored as a global variable (line (4)). If the transition is taken, the body records it in a global variable (line (11)) and then updates the current state with the next state of that transition (line (12)). If the transition implies moving to a different state machine, additional data must be stored (line (18)) to ensure that navigating back from the new state machine returns to the proper state (line (23)).

Promela do loops are nondeterministic, this meaning that when several branches are enabled (ready to be executed), Spin may select any of them to continue the execution in *the simulation mode*. In contrast, using the *verification mode*, Spin will explore all the possibilities using a first-depth search algorithm. For the case of the example, this means that Spin,

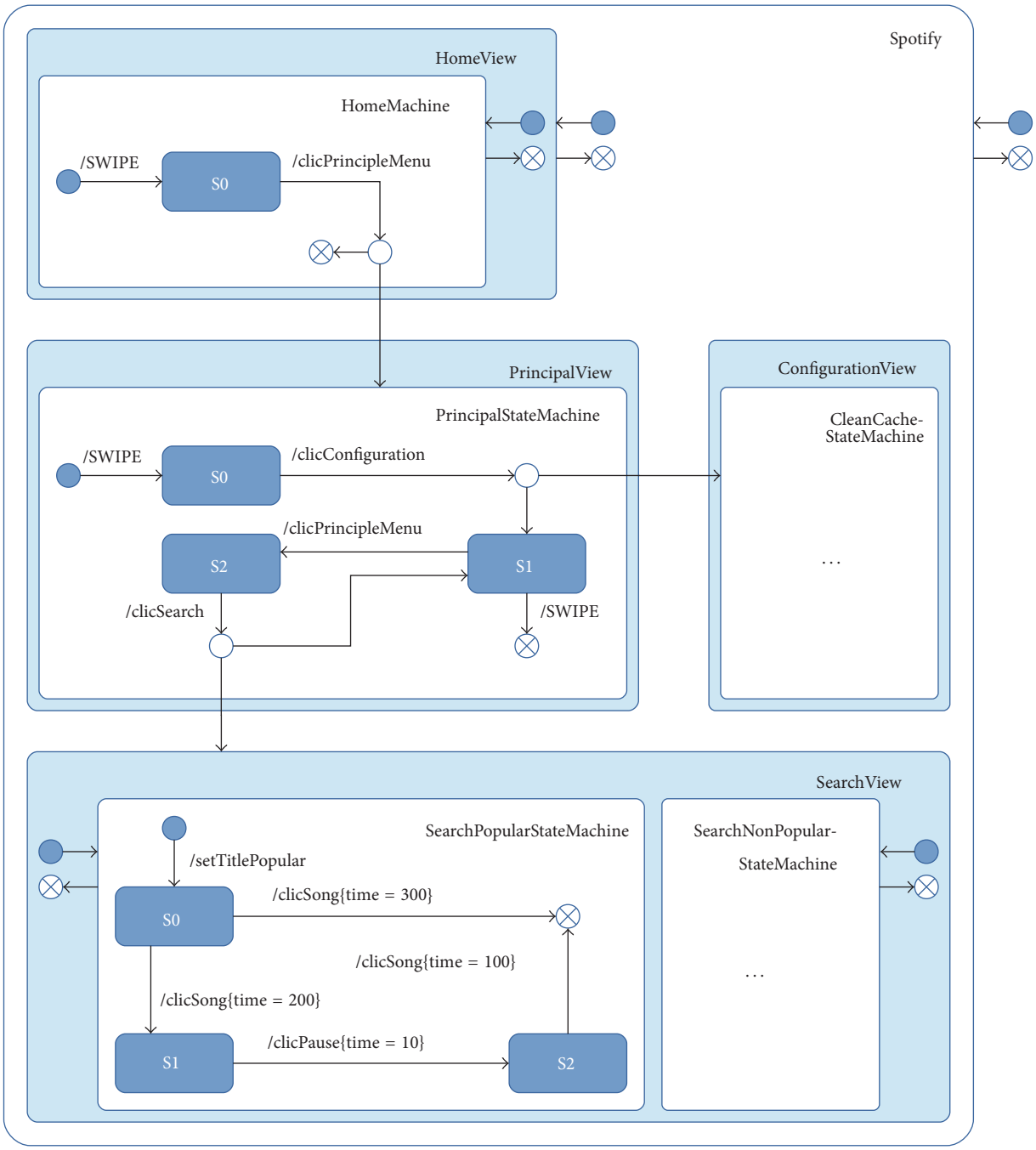
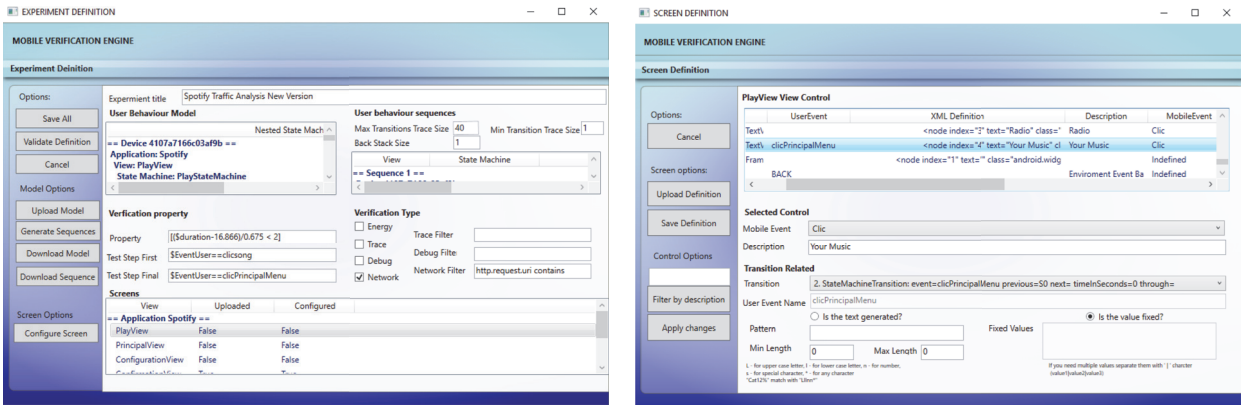


Figure 5: User behavior model for the Spotify app.

in verification mode, is able to generate all possible test cases defined by the model.

Each of the test cases generated by Spin is then translated into a Java program. Listing 2 shows part of the generated Java code for one test case from the Spotify model. Each user action in the test case is translated into a Java method that uses the UiAutomator API to perform the action in the mobile (see lines (6) and (16)). Additionally, each method writes metadata to the Android logcat, including which user action was executed, to make it available for the execution trace (line (7)).

4.2. Network Traffic Analysis. Once the test cases have been generated, we proceed with the network traffic analysis we want to perform on the Spotify app. We first studied the network traffic generated by the app. We found out that it uses HTTP connections to download songs from the Spotify servers, and identifies itself as "Spotify-Unknown" in



(a) User behavior model, test case generation, and verification property

(b) Screen and user actions

Figure 6: Configuring the analysis of the Spotify app with the GUI.

```
(1) typedefBackstack {mtype states [MAXB]; Transition trans [MAXB]; short index};
(2)
(3) #definecurBackstack devices [device]. backstack
(4) #definecurStatecurBackstack. states [curBackstack. index]
(5)
(6) proctype device_4107a7166c03af9b (int device) {
(7)      do
(8)      // Spotify - PlayView - PlayStateMachine
(9)      :: starting || curState == Spotify_PlayView_PlayStateMachine_init ->
(10)         pushToBackstack (device, Spotify_PlayView_PlayStateMachine_init);
(11)         transition (device, VIEW_PlayView, 1); // Swipe
(12)         curState = Spotify_PlayView_PlayStateMachine_S0
(13)     :: !starting && curState == Spotify_PlayView_PlayStateMachine_S0 ->
(14)         transition (device, VIEW_PlayView, 2); // clicPrincipalMenu
(15)         curState = Spotify_PrincipalView_PrincipalStateMachine_init
(16)     // Spotify - PrincipalView - PrincipalStateMachine
(17)     :: !starting && curState ==
              Spotify_PrincipalView_PrincipalStateMachine_init ->
(18)          pushToBackstack (device,
         Spotify_PrincipalView_PrincipalStateMachine_init);
(19)         transition (device, VIEW_PrincipalView, 1); // Swipe
(20)         curState = Spotify_PrincipalView_PrincipalStateMachine_S0
(21)     :: !starting && curState ==
              Spotify_PrincipalView_PrincipalStateMachine_end ->
(22)         popFromBackstack (device);
(23)         continueTransition_4107a7166c03af9b (device)
(24)      od;
(25) }
```

Listing 1: Extract of Promela specification for test case generation.

the User-Agent field. While Spotify uses HTTP for various purposes, song requests could be identified by GET requests with URLs following the format: "/audio/<id>." Each song is downloaded over a series of HTTP GET requests sent right after the user presses play, each fetching a fragment of the song. The following discussion considers only network data packets from these audio streams.

We introduce an extra-functional property to analyze the streaming traffic generated by a media-streaming app by checking the following interval property on the test cases:

$$Z = \frac{|r - \mu|}{\sigma} \qquad (2)$$
$$\forall\, [[Z < 2]]_{[testStep=clickSong, testStep=clicPrincipalMenu]}\,.$$

```
(1) // Transition 16: previous next S0 on view PrincipalView
(2) public void TestSpotifyclicSearch16 () throws UiObjectNotFoundException {
(3)     UiObject control = new UiObject (new UiSelector ().
(4)             className (" android. widget.TextView ")
(5)             .index (0).textContains (" Search "));
(6)     control. click ();
(7)     Log.v(" DRACODROID ", " CONTROL - clicSearch: " + reportDate);
(8)     //...
(9) }
(10)
(11) // Transition 17: previous next S0 on view SearchView
(12) public void TestSpotifysetTitlePopular17 () throws UiObjectNotFoundException{
(13)     UiObject control = new UiObject (new UiSelector ().
(14)             className (" android. widget.EditText ")
(15)             .index (0).textContains (" Search "));
(16)     control.setText (" Can't Hold Us ");
(17)     Log.v(" DRACODROID ", " CONTROL - setTitlePopular: Can't Hold Us "
(18)             + reportDate);
(19)     //...
(20) }
(21)
(22) // Transition 18: previous S0 next S0 on view SearchView
(23) public void TestSpotifyclicsong18 () //...
(24)
(25) // Transition 19: previous S1 next S0 on view PrincipalView
(26) public void TestSpotifyclicPrincipalMenu19 () //...
(27)
(28) // Transition 20: previous S1 next S0 on view PrincipalView
(29) public void TestSpotifyclicSearch20 () //...
(30)
(31) // Transition 21: previous S1 next S0 on view SearchView
(32) public void TestSpotifysetTitlePopular21 () //...
```

Listing 2: Extract of test cases translated into a Java program.

Here, Z is the unsigned standard score of a direct measurement of a given network traffic variable r that follows the normal distribution, μ is the known mean value of the distribution of r-measurements for all songs that exhibit a typical behavior, and σ is the standard deviation of the same distribution. The interval is defined between *clickSong*, which is the user action performed in the test case that starts playing a song, and *clicPrincipalMenu*, which represents a user starting a new song title search marking the end of the test scenario. The interval formula in (2) verifies whether the continuous variable rate of the given network traffic variable lies between a band around the mean in a normal distribution and a width of two standard deviations, that is, a significant confidence interval of 95.45% of the songs that exhibit a typical behavior. These events and variables can be identified and provided by the runtime monitor; specifically, these measurements may be taken at regular intervals using the tcpdump application remotely on an Android mobile and therefore be included in the enriched execution trace.

The following network traffic features were monitored during the playback: the number of packets sent and received ($PkgS$ and $PkgR$, resp.), the number of bytes sent and received (BS and BR, resp.), and the duration of the connection in milliseconds (D). Using these features, we established three network traffic assessment variables: $r1$ as the quotient between $PkgS$ versus $PkgR$ ($r1 = PkgS/PkgR$), $r2$ as the quotient between BS and BR ($r2 = BS/BR$), and $r3$ as D ($r3 = D$). The mean and standard deviation of $r1$, $r2$, and $r3$ (see Table 1) were empirically determined for song playbacks that exhibited a suitable network behavior and used as the reference for the verification of extra-functional properties.

Table 1: Mean and standard deviation of network traffic variables.

Variable	Mean	Std. deviation
$r1$	0.252	0.048
$r2$	0.010	0.002
$r3$	16.866	0.675

We considered that a song playback interval demonstrated an abnormal behavior when the app started to download a song but the connection prematurely closed and was thus forced to reopen a new healthy connection in order to download the whole song. Subsequently, an observer

Table 2: Numerical results for test case generation.

Configuration	Results			
Max. trans.	Number of test cases	Time	Number of states	Memory
50	199	0.02 s	7,072	129.8 MB
60	815	0.08 s	28,763	133.2 MB
100	226,029	25.3 s	795,9210	1,724.0 MB

detected abnormal behaviors by analyzing $r1$, $r2$, and $r3$ network traffic variables using the following extra-functional property:

$$\forall\,[[(Z1 < 2) \vee (Z2 < 2) \vee (Z3 < 2)]]_{[testStep=clickSong, testStep=clicPrincipalMenu]} \quad (3)$$

which is a special case of the extra-functional property presented in (2) and where $Z1$, $Z2$, and $Z3$ represent the unsigned standard scores of $r1$, $r2$, and $r3$, respectively. This extra-functional property, along with the appropriate tcpdump filter to get only Spotify audio packets, is configured in the GUI screen shown in Figure 6(a).

4.3. Results. We ran test case generation and verification of properties in (2) and (3) using MVE running on a 64-bit-operating system PC, featuring an Intel(R) Core(TM) i5-3337U CPU processor at 1.8 GHz and 6 GB of Ram. The test cases for Spotify Android were executed on a Samsung Galaxy Tab 10.1 running Android OS v. 4.1.2 (Firmware version 3.0.31-805288) connected to Internet via WiFi (throughput at 31.6 Mb/s). The total execution time of the experiment was 8.3 hours.

Table 2 shows numerical results related to test case generation from the XML model with three different values to limit the maximum depth allowed for the model exploration (transitions). For each one, we present numbers for transitions, test cases, processing time, and states and memory usage by Spin. The results show that a higher number of transitions significantly increased the execution time, number of test cases, states, and memory usage. Therefore, this is a key value to evaluate the generated test cases to diminish the redundancy between behaviors by finding and removing unnecessary nested state machines in the source user behavior model.

Figure 7 shows a graphical representation of a trace fragment from one of the test case executions. The trace is composed of discrete states, whose boundaries are determined by events, such as user actions in this case study. Each state contains the current value of its variables, such as the number of packets sent and received, as described in the previous section. This trace fragment corresponds to one execution of the SearchPopularStateMachine. The network traffic variables are filled with zeros (due to traffic filtering with tcpdump), until *testStep* is equals to *clicPrincipalMenu*. In this state the song has already been downloaded and played, and the variables contain the data for this song.

The overall results for network traffic verification with the extra-functional properties presented in (2) and (3) are listed in Table 3. Within this context, our tool monitored the traces to obtain values for the three variables $Z1$, $Z2$, and $Z3$. The normality of the measurements of these variables was evaluated using the Shapiro-Wilk test, confirming the normality of distribution in all cases, as the null hypothesis could not be rejected for $\alpha = 0.05$. Therefore, it was possible to assess the network traffic variables by establishing their respective unsigned standard scores using the extra-functional properties indicated in (2) and (3). We compare the performances of the assessment of the extra-functional property described in (2) for each monitored variable separately (denoted as $Z1$, $Z2$, and $Z3$ to simplify the presentation of (2)) and the performance of the special extra-functional property case presented in (3), which joins the three monitored variables into a single statement (simplified as $Z1 \vee Z2 \vee Z3$). The two first rows of Table 3 contains the number of trace intervals determined by [$testStep = clickSong, testStep = clicPrincipalMenu$] that behaves correctly/incorrectly with respect to the corresponding formula. The results in Table 3 show that taking into account all monitored variables at the same time provided the best performance (sensitivity = specificity = accuracy = 100%), while considering each monitored variable separately provided a lower but still highly accurate outcome.

Table 3: Results of the verification process applied over the Spotify for Android app.

Statistic	$Z1$	$Z2$	$Z3$	$Z1 \vee Z2 \vee Z3$
Positives	53	52	54	56
Negatives	13	13	13	13
False negatives	3	4	2	0
Sensitivity	95%	93%	96%	100%
Specificity	100%	100%	100%	100%
Accuracy	96%	95%	97%	100%

A graphical timeline representation of the variables $Z1$, $Z2$, and $Z3$ measured during the execution of the experiment is shown in Figure 8, where it is possible to see that abnormal network traffic behavior correlated to large peaks of unsigned standard scores for all variables. These are the traces reported by MVE as violations of the property in (3) (negatives in the table). When we analyzed these traces we noticed that some HTTP connections in the traces obtained did not end successfully for various reasons. In most cases the problem was caused by timeout errors from the remote Spotify service. The third row of Table 3 shows the number of interval property violations in which this timeout was not detected which could be interpreted as a false error.

The results can be presented to the developer with several levels of detail. For example, Figure 9 shows three levels of information for the first anomaly (8th song) in Figure 8. First, the song Demons was selected as part of a test case, as shown in the extract from the logging at the top of the figure. The clicSong action produces a first failed `GET` request (pink background in the Wireshark screenshot), followed by a number of successful `GET` requests (cyan background). If we check the internal TCP traffic due to the first (failed) `GET`,

testStep	PkgS	PkgR	BS	BR	D
clicSearch	0	0	0	0	0
setTitlePopular	0	0	0	0	0
clicSong	0	0	0	0	0
clicPrincipalMenu	358	1675	13384	2497325	230

Figure 7: Fragment of execution trace.

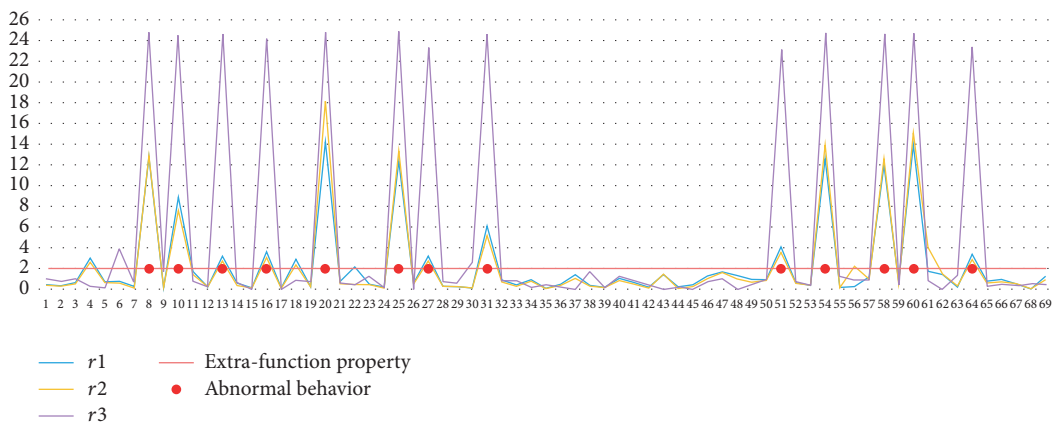


Figure 8: Timeline representation of the unsigned standard score of the quotient between the number of TCP packets sent versus received ($r1$), the quotient of bytes sent versus received ($r2$), and the duration of the connection in seconds ($r3$) obtained during the verification process of the Spotify for Android app.

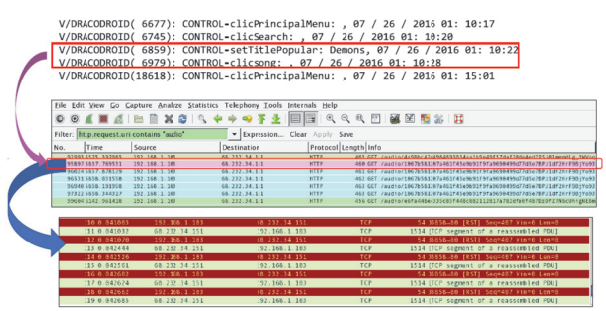


Figure 9: Different levels of information provided to developers to track errors.

then we can see that the client side closed the connection with the RESET option. As far as the RESET procedure is initiated by the Spotify program, the developer can discover the real problem detected in the connection.

5. Conclusions and Future Work

In this paper, we have presented a novel tool designed to identify abnormal network traffic behaviors in a multimedia application and how different user interactions may lead to unexpected traffic patterns. It allows generating a large amount of test cases that can be executed and measured with proper automation. These test cases represent realistic user behaviors, instead of random interactions, which may reveal unforeseen consequences to users. Furthermore, the automated execution and analysis provides detailed information when deviations from the expected results are detected, so that a postmortem analysis can be performed.

The applicability of the approach has been demonstrated with a case study based on a multimedia application. However, other types of applications can also benefit from a model-based testing solution like the one presented in this paper. On the one hand, the automatic generation of test cases from models is well suited for applications with complex but decomposable behaviors, such as social network applications. The composition of state machines enables building models from smaller pieces that can produce realistic behaviors that were not considered by the developers. This is can also be applied to games with discrete behaviors, with distinguishable screens or stages, such as online gambling games. Each stage can be implemented in a different state machine with the legal actions that can be performed. A stage change would be modeled as a transition to a different state machine.

On the other hand, the analysis of network traffic can benefit any application where developers can describe the expected patterns in response to certain user interactions. For instance, a social network application may be analyzed to check that an adequate amount of data is downloaded when entering a news stream or photo album. Our approach works best when the expected patterns can be set in intervals defined by user interactions or by events that can be observed by our MVE tool. Thanks to the automatic generation of test cases, these patterns can be checked over a large set of user behaviors, to verify whether interactions that were not considered by the developers can lead to unexpected traffic patterns.

While all these examples are of applications that communicate with a server, applications with network traffic between devices, such as VoIP clients, can also be analyzed. Our modeling language can be used to describe applications that have interactions across different devices. The test cases will be executed in the devices in parallel, with synchronization actions to ensure that they are executed in the expected sequence.

However, there are some points that can be further improved in future research. First, we consider that it is important to improve the synchronization of the measurements and additional information obtained at runtime. Enhancements in this context will provide the means to specify richer properties in the model. Additionally, we will consider implementing some automatic processes intended to assist the developers in building the user behavior model. Applications with more complex graphical interfaces, such as 3D games, are not currently supported and would require a different approach for interface automation. However, the same modeling language could be used, using a different type of binding to UI elements.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work is partially supported by Grants P11-TIC-07659 (Regional Government of Andalusia), TIN2015-67083-R (Spanish Ministry of Economy and Competitiveness), and European projects Fed4FIRE and FLEX under the European Union's Seventh Framework Programme (FP7).

References

[1] Cisco, *Visual Networking Index Traffic and Service Adoption Forecasts*, Cisco, 2016.

[2] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA '13)*, pp. 67–77, ACM, Lugano, Switzerland, July 2013.

[3] A. R. Espada, M. M. Gallardo, and D. Adalid, "A runtime verication framework for android applications," in *XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD '13)*, 2013.

[4] A. R. Espada, M. M. Gallardo, A. Salmerón, and P. Merino, "Using model checking to generate test cases for android applications," in *Proceedings of the 10th Workshop on Model Based Testing. Electronic Proceedings in Theoretical Computer Science*, N. Pakulin, A. K. Petrenko, and B. H. Schlingloff, Eds., vol. 180, pp. 7–21, Open Publishing Association, 2015.

[5] A. R. Espada, M. Mar Gallardo, A. Salmerón, and P. Merino, "Runtime verification of expected energy consumption in smartphones," in *Model Checking Software*, vol. 9232 of *Lecture Notes in Computer Science*, pp. 132–149, Springer International Publishing, 2015.

[6] A. Díaz-Zayas, C. A. García-Pérez, Á. M. Recio-Pérez, and P. Merino-Gómez, "PerformLTE: a testbed for LTE testing in the future internet," in *Wired/Wireless Internet Communications*, vol. 9071 of *Lecture Notes in Computer Science*, pp. 46–59, Springer International, Cham, Switzerland, 2015.

[7] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[8] Monkey, https://developer.android.com/studio/test/monkey.html.

[9] Monkeyrunner, https://developer.android.com/studio/test/monkeyrunner/index.html.

[10] Robotium, https://code.google.com/p/robotium/.

[11] J. Jeon and J. S. Foster, "Troyd: integration testing for android," 2012, https://github.com/plum-umd/troyd.

[12] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[13] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, pp. 641–660, ACM, Indianapolis, Ind, USA, 2013.

[14] L. Wei, Y. Liu, and S. C. Cheung, "Taming android fragmentation: characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, pp. 226–237, ACM, Singapore, September 2016.

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, pp. 258–261, Essen, Germany, September 2012.

[16] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[17] C. Wilke, S. Gotz, J. Reimann, and U. Aÿmann, "Vision paper: towards modelbased energy testing," in *Model Driven Engineering Languages and Systems*, J. Whittle, T. Clark, and T. Kühne, Eds., vol. 6981 of *Lecture Notes in Computer Science*, pp. 480–489, Springer, Berlin, Germany, 2011.

[18] C. Thompson, D. Schmidt, H. Turner, and J. White, "Analyzing mobile application software power consumption via model-driven engineering," in *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, pp. 101–113, March 2011.

[19] E. Umuhoza and M. Brambilla, "Model driven development approaches for mobile applications: a survey," in *Mobile Web and Intelligent Information Systems*, vol. 9847 of *Lecture Notes in Computer Science*, pp. 93–107, Springer International Publishing, Cham, Germeny, 2016.

[20] T. Kamiyama, H. Inamura, and K. Ohta, "A model-based energy profiler using online logging for Android applications," in *Proceedings of the 7th International Conference on Mobile Computing and Ubiquitous Networking (ICMU '14)*, pp. 7–13, January 2014.

[21] F. Willnecker, A. Brunnert, and H. Krcmar, "Model-based energy consumption prediction for mobile applications," in *Proceedings of the 28th International Conference on Informatics for Environmental Protection: ICT for Energy Effieciency (EnviroInfo '14)*, pp. 747–752, Oldenburg, Germany, September 2014.

[22] S. Nakajima, "Model checking of energy consumption behavior," in *Complex Systems Design & Management Asia: Designing Smart Cities: Proceedings of the First Asia—Pacic Conference on Complex Systems Design & Management, CSD&M Asia 2014*, M. A. Cardin, D. Krob, C. P. Lui, H. Y. Tan, and K. Wood, Eds., pp. 3–14, Springer, Cham, Switzerland, 2015.

[23] A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langho, and A. Markopoulou, "AntMonitor: network traffic monitoring and real-time prevention of privacy leaks in mobile devices," in *Proceedings of the Workshop on Wireless of the Students, by the Students, and for the Students (S3 '15)*, pp. 25–27, ACM, Paris, France, September 2015.

[24] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Network profiler: towards automatic ngerprinting of android apps," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM '13)*, pp. 809–817, April 2013.

[25] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Proledroid: multi-layer proling of android applications," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (Mobicom '12)*, pp. 137–148, ACM, Istanbul, Turkey, August 2012.

[26] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 2003.

[27] Oracle: Java debug interface, http://docs.oracle.com/javase/6/docs/jdk/api/jpda/jdi/index.html.

[28] V. Jacobson, C. Leres, and S. McCanne, *The tcpdump Manual Page*, Lawrence Berkeley Laboratory, Berkeley, Calif, USA, 1989, http://www.tcpdump.org/manpages/tcpdump.1.html.

[29] J. O. Blech, Y. Falcone, and K. Becker, "Towards certified runtime verification," in *Formal Methods and Software Engineering*, pp. 494–509, Springer, Berlin, Germany, 2012.